

Сверхбыстрый Python

Эффективные техники
для работы с большими
наборами данных



Тиаго Антао

Тиаго Антао

Сверхбыстрый Python

Эффективные техники для работы
с большими наборами данных

Fast Python

HIGHT PERFORMANCE TECHNIQUES
FOR LARGE DATASETS

TIAGO RODRIGES ANTÃO



MANNING
Shelter Island

Сверхбыстрый Python

ЭФФЕКТИВНЫЕ ТЕХНИКИ ДЛЯ РАБОТЫ
С БОЛЬШИМИ НАБОРАМИ ДАННЫХ

ТИАГО АНТАО



Москва, 2023

УДК 004.438Python:004.6

ББК 32.973.22

A72

Тиаго Антао

A72 **Сверхбыстрый Python. Эффективные техники для работы с большими наборами данных / пер. с англ. А. Ю. Гинько. – М.: ДМК Пресс, 2023. – 370 с.: ил.**

ISBN 978-5-93700-226-6

Данная книга предлагает уникальные техники ускорения выполнения кода на Python с акцентом на большие данные. Вы узнаете, как оптимизировать работу со встроенными структурами данных за счет конкурентного выполнения, а также научитесь сокращать объем занимаемой данными памяти без ущерба для их точности. Ознакомившись с тщательно проработанными примерами, вы узнаете, как добиться большей производительности популярных библиотек, таких как NumPy и pandas, и как эффективно обрабатывать и хранить данные. В книге используется целостный подход к повышению эффективности решений, так что вы научитесь оптимизировать и масштабировать целые системы – начиная от кода и заканчивая архитектурой.

Издание предназначено для разработчиков Python, знакомых с основами языка и принципами конкурентных вычислений.

УДК 004.438Python:004.6

ББК 32.973.22

DMK Press 2023. Authorized translation of the English edition ©2023 Manning Publications. This translation is published and sold by permission of Manning Publications, the owner of all rights to publish and sell the same.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN (анг.) 978-1-61729-793-9

ISBN (рус.) 978-5-93700-226-6

© 2023 by Manning Publications Co.

© Оформление, издание, перевод,
ДМК Пресс, 2023

Оглавление

Предисловие от издательства	11
Предисловие	13
Благодарности	15
О книге	16
Об авторе	22
О переводчике	23
Об изображении на обложке	24

ЧАСТЬ I. ФУНДАМЕНТАЛЬНЫЕ ПОДХОДЫ..... 25

1 *Острая нехватка производительности при обработке данных* 27

1.1. Насколько велик всемирный потоп данных?	29
1.2. Современные вычислительные архитектуры и высокопроизводительные вычисления	33
1.2.1. Изменения в архитектуре компьютеров	33
1.2.2. Изменения в архитектуре сети	36
1.2.3. Облако	38
1.3. Работа с ограничениями языка Python	38
1.3.1. Глобальная блокировка интерпретатора	40
1.4. Возможные решения	41
Заключение	44

2 *Извлечение максимума возможного из встроенных средств Python* 46

2.1. Профилирование приложений с операциями ввода-вывода и вычислениями	48
2.1.1. Загрузка данных и поиск минимальной температуры	48
2.1.2. Встроенный в Python модуль профилирования	50
2.1.3. Использование локального кеша для снижения сетевой нагрузки	51
2.2. Профилирование кода для обнаружения проблем с производительностью	53
2.2.1. Визуализация профилировочной информации	54
2.2.2. Профилирование с детализацией до строк	55
2.2.3. Профилирование кода: выводы	57
2.3. Оптимизация работы базовых структур данных Python: списки, множества и словари	58
2.3.1. Быстродействие поиска в списке	59
2.3.2. Поиск с использованием множеств	60
2.3.3. Вычислительная сложность списков, множеств и словарей в Python	61

2.4. В поисках избыточного выделения памяти.....	63
2.4.1. По минному полю выделения памяти в Python	64
2.4.2. Выделение памяти для альтернативных представлений ...	67
2.4.3. Использование массивов в качестве компактной альтернативы спискам	69
2.4.4. Систематизирование новых знаний: оценка объема памяти, занимаемой объектом	71
2.4.5. Оценка занимаемой объектами памяти в Python: выводы.....	72
2.5. Использование ленивых вычислений и генераторов для работы с большими данными	73
2.5.1. Использование генераторов вместо обычных функций.....	73
Заключение	75

3 Конкурентность, параллелизм и асинхронная обработка 77

3.1. Написание шаблона асинхронного сервера.....	81
3.1.1. Разработка шаблона для взаимодействия с клиентами	83
3.1.2. Программирование с сопрограммами	85
3.1.3. Передача сложных данных от простого синхронного клиента.....	87
3.1.4. Альтернативные способы передачи данных между процессами	88
3.1.5. Асинхронное программирование: выводы	89
3.2. Реализация базового движка MapReduce.....	89
3.2.1. Описание фреймворка MapReduce	89
3.2.2. Разработка простейшего тестового сценария	91
3.2.3. Первая реализация фреймворка MapReduce	92
3.3. Реализация конкурентной версии фреймворка MapReduce.....	93
3.3.1. Использование модуля concurrent.futures для реализации многопоточного сервера	93
3.3.2. Асинхронное выполнение с использованием будущих объектов	95
3.3.3. Глобальная блокировка интерпретатора и многопоточность.....	98
3.4. Реализация фреймворка MapReduce с использованием библиотеки multiprocessing.....	99
3.4.1. Решение на основе модуля concurrent.futures	100
3.4.2. Решение на основе модуля multiprocessing	101
3.4.3. Отслеживание прогресса при использовании модуля multiprocessing	103
3.4.4. Передача данных порциями	105
3.5. Собираем все воедино: асинхронный многопоточный и многопроцессный сервер MapReduce.....	109
3.5.1. Архитектура высокопроизводительного решения	109
3.5.2. Создание надежной версии сервера.....	113
Заключение	115

4 *Высокопроизводительный NumPy* 117

4.1. Библиотека NumPy с точки зрения производительности	119
4.1.1. Копии и представления существующих массивов.....	119
4.1.2. Внутреннее устройство представлений NumPy	125
4.1.3. Эффективное использование представлений	131
4.2. Программирование на основе массивов	133
4.2.1. Отправная точка.....	135
4.2.2. Транслирование в NumPy.....	135
4.2.3. Применение приемов программирования на основе массивов	138
4.2.4. Векторизуем сознание	141
4.3. Оптимизация внутренней архитектуры NumPy	145
4.3.1. Обзор зависимостей в NumPy	146
4.3.2. Настройка NumPy в дистрибутиве Python	148
4.3.3. Потоки в NumPy	149
Заключение	151

ЧАСТЬ II. АППАРАТНОЕ ОБЕСПЕЧЕНИЕ..... 153

5 *Реализация критически важного кода с помощью Cython* ... 155

5.1. Обзор техник для эффективной реализации кода	156
5.2. Беглый обзор расширения Cython	158
5.2.1. Наивная реализация в Cython	159
5.2.2. Использование аннотаций типов в Cython для повышения производительности	162
5.2.3. Как аннотации типов влияют на производительность.....	163
5.2.4. Типизация возвращаемых из функции значений.....	166
5.3. Профилирование кода на Cython	167
5.3.1. Использование встроенной инфраструктуры профилирования Python	168
5.3.2. Использование line_profiler.....	169
5.4. Оптимизация доступа к массивам в Cython с помощью memoryview.....	173
5.4.1. Использование представлений памяти	173
5.4.2. Избавление от всех взаимодействий с Python	175
5.5. Написание обобщенных универсальных функций NumPy на Cython.....	177
5.6. Продвинутая работа с массивами в Cython	179
5.6.1. Обход ограничений GIL по запуску нескольких потоков одновременно	182
5.6.2. Базовый анализ производительности.....	186
5.6.3. Космические войны в Quadlife	187
5.7. Параллелизм с Cython	189
Заклучение	190

6 *Иерархия памяти, хранение данных и работа с сетью* 192

6.1. Как современная архитектура аппаратных средств влияет на эффективность кода Python	194
--	-----

6.1.1. Неожданное влияние современной архитектуры на производительность	195
6.1.2. Влияние кеша процессора на эффективность алгоритма.....	196
6.1.3. Современные устройства постоянного хранения	198
6.2. Эффективное хранение данных при помощи Blosc	199
6.2.1. Сжимаем данные, экономим время	199
6.2.2. Операции чтения (буферы памяти)	201
6.2.3. Влияние алгоритма сжатия на эффективность хранения	202
6.2.4. Использование сведений о представлении данных для повышения эффективности сжатия.....	203
6.3. Ускорение NumPy с помощью NumExpr	204
6.3.1. Быстрая обработка выражений	205
6.3.2. Влияние архитектуры аппаратных средств на результаты.....	206
6.3.3. Когда не стоит использовать библиотеку NumExpr	207
6.4. Производительность при использовании локальных сетей	208
6.4.1. Причины неэффективности вызовов REST.....	209
6.4.2. Наивный клиент на основе UDP и msgpack	209
6.4.3. Сервер на основе UDP	211
6.4.4. Безопасность на клиенте с помощью тайм-аутов	212
6.4.5. Прочие предпосылки для оптимизации сетевых вычислений	214
Заключение	214

ЧАСТЬ III. ПРИЛОЖЕНИЯ И БИБЛИОТЕКИ ДЛЯ СОВРЕМЕННОЙ ОБРАБОТКИ ДАННЫХ..... 217

7 *Высокопроизводительный pandas и Apache Arrow* 219

7.1. Оптимизация памяти и времени при загрузке данных.....	220
7.1.1. Сжатые и несжатые данные.....	221
7.1.2. Определение типов данных колонок.....	222
7.1.3. Эффект изменения точности типа данных.....	226
7.1.4. Кодирование и снижение объема данных.....	227
7.2. Техники для повышения скорости анализа данных	230
7.2.1. Использование индексирования для ускорения доступа к данным	231
7.2.2. Техники перемещения по строкам	232
7.3. Взаимодействие pandas с NumPy, Cython и NumExpr.....	235
7.3.1. Явное использование NumPy	236
7.3.2. Pandas поверх NumExpr	237
7.3.3. Cython и pandas	239
7.4. Чтение данных в pandas с помощью Arrow	241
7.4.1. Взаимодействие между pandas и Apache Arrow	241
7.4.2. Чтение из файла CSV	243
7.4.3. Анализ данных в Arrow	246

7.5. Использование механизма взаимодействий в Argo для делегирования задач более эффективным языкам и системам	247
7.5.1. Предпосылки архитектуры межязыкового взаимодействия Argo	247
7.5.2. Операции с нулевым копированием с использованием сервера Plasma от Argo	249
Заключение	254

8 *Хранение больших данных*..... 256

8.1. Универсальный интерфейс для доступа к файлам: fsspec	257
8.1.1. Использование fsspec для поиска файлов в репозитории GitHub	258
8.1.2. Использование fsspec для поиска zip-файлов.....	260
8.1.3. Доступ к файлам с использованием библиотеки fsspec	260
8.1.4. Использование цепочки URL для обращения к разным файловым системам	261
8.1.5. Замена реализации файловой системы	262
8.1.6. Взаимодействие с PyArrow	262
8.2. Parquet: эффективный формат хранения колоночных данных.....	263
8.2.1. Исследование метаданных Parquet.....	264
8.2.2. Кодирование колонок в Parquet.....	266
8.2.3. Секционирование наборов данных.....	269
8.3. Работа с наборами данных, не помещающимися в памяти, по-старому.....	271
8.3.1. Отображение в памяти с помощью NumPy.....	271
8.3.2. Порционирование данных при чтении и записи в датафрейм	273
8.4. Использование Zarr для хранения больших массивов.....	276
8.4.1. Знакомство с внутренней структурой формата Zarr	277
8.4.2. Хранение массивов в Zarr.....	279
8.4.3. Создание нового массива	282
8.4.4. Параллельное чтение и запись массивов в Zarr	284
Заклучение	286

ЧАСТЬ IV. ПРОДВИНУТЫЕ ВОЗМОЖНОСТИ 289

9 *Анализ данных с использованием графического процессора*.....291

9.1. Предпосылки для использования вычислительных мощностей GPU.....	294
9.1.1. Преимущества использования графического процессора	294
9.1.2. Связь между центральным и графическим процессорами.....	297
9.1.3. Внутренняя архитектура графического процессора	298
9.1.4. Архитектура программного обеспечения	299
9.2. Использование компилятора Numba для генерации кода под GPU	300

9.2.1. Программное обеспечение для работы с GPU в Python	301
9.2.2. Основы программирования для GPU с помощью Numba	302
9.2.3. Создание генератора Мандельброта с помощью графического процессора	306
9.2.4. Создание генератора Мандельброта с помощью NumPy	309
9.3. Анализ производительности кода для GPU:	
приложение с использованием CuPy	310
9.3.1. Библиотеки для анализа данных на базе GPU	310
9.3.2. Использование CuPy – версии библиотеки NumPy для GPU	311
9.3.3. Базовое взаимодействие с CuPy	311
9.3.4. Создание генератора Мандельброта с помощью Numba	313
9.3.5. Создание генератора Мандельброта с помощью CUDA C....	315
9.3.6. Средства профилирования кода для GPU	317
Заключение	320
10 Анализ больших данных с использованием библиотеки Dask	321
10.1. Знакомство с моделью выполнения Dask.....	323
10.1.1. Шаблон pandas для сравнения	324
10.1.2. Решение на основе датафреймов Dask.....	326
10.2. Вычислительная стоимость операций Dask	327
10.2.1. Секционирование данных для обработки	328
10.2.2. Сохранение промежуточных вычислений	330
10.2.3. Реализации алгоритмов при работе с распределенными датафреймами	331
10.2.4. Рассекционирование данных	334
10.2.5. Хранение распределенных датафреймов.....	337
10.3. Использование распределенного планировщика Dask	338
10.3.1. Архитектура dask.distributed	340
10.3.2. Запуск кода с помощью dask.distributed.....	344
10.3.3. Работа с наборами данных, превышающими по объему доступную память.....	350
Заключение	351
Приложение А. Настройка окружения	353
A.1. Установка Anaconda Python.....	354
A.2. Установка дистрибутива Python	355
A.3. Использование Docker.....	355
A.4. Вопросы, касающиеся аппаратного обеспечения.....	355
Приложение Б. Использование Numba для создания эффективного низкоуровневого кода	357
B.1. Создание оптимизированного кода с помощью Numba	359
B.2. Написание параллельных функций в Numba	362
B.3. Написание кода с использованием NumPy в Numba	362
Предметный указатель	365

Предисловие от издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в основном тексте или программном коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить следующие издания этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательство «ДМК Пресс» очень серьезно относится к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Предисловие

Несколько лет назад один из процессов на основе Python, которым занималась моя команда разработчиков, вдруг наглухо подвис. Он продолжал нагружать процессор, но не завершался. Это был один из критически важных процессов для компании, и нам необходимо было срочно разобраться с возникшей ситуацией. Мы взглянули на алгоритм и не обнаружили каких-то серьезных проблем. Да там и не было ничего сложного. После нескольких часов работы мы поняли, что узким местом является процесс поиска в довольно объемном списке. Заменив список на множество, мы решили проблему. По сути, мы изменили структуру для хранения данных, тем самым снизив время поиска с нескольких часов до миллисекунд.

Это было какое-то прозрение, которое заставило меня задуматься о том, что:

- проблема была пустяковой, но с помощью нее мы выявили, что в процессе разработки совершенно не заботились об эффективности своего кода. К примеру, если бы мы в своей работе пользовались профайлером, то смогли бы обнаруживать подобные утечки за минуты, а не тратить на это часы;
- в выигрыше в итоге остались все: мы одновременно сократили время выполнения процесса и снизили объем используемых ресурсов памяти. Да, зачастую в подобных ситуациях мы вынуждены идти на компромиссы, но здесь мы имели дело с беспроеигрышной партией;
- в глобальном смысле также никто не остался в накладе. Во-первых, ускорение процесса пошло на пользу компании, а во-вторых, решение проблемы позволило снизить процессорное время, что положительно сказывается на потреблении электричества и экологии в целом;

- хотя в этом конкретном случае о большой экономии ресурсов речи не шло, кому-то наверняка приходится ежедневно сталкиваться с подобными ситуациями.

В итоге я решил посвятить свое время написанию книги, с помощью которой смогу транслировать свои озарения другим программистам. Моя главная миссия – помочь бывалым программистам на языке Python разрабатывать и внедрять более эффективный код и легко распознавать ситуации, в которых можно и нужно идти на компромиссы. При этом я постарался подойти к проблеме комплексно, рассмотрев как базовый Python, так и основные библиотеки и уделив внимание алгоритмам, архитектуре современного аппаратного обеспечения и эффективности процессоров и способов хранения данных. Надеюсь, книга, которую вы держите в руках, поможет вам более уверенно подходить к решению проблем, связанных с производительностью, в экосистеме Python.

Благодарности

Я бы хотел поблагодарить моего редактора Фрэнсис Лефковиц (Frances Lefkowitz) за ее безграничное терпение. Также мне бы хотелось сказать спасибо дочери и жене, которые прощали мое отсутствие дома в процессе написания книги. Кроме того, я безмерно благодарен выпускающей команде издательства Manning за то, что книга увидела свет.

Хочу перечислить и всех рецензентов, принимавших участие в проверке и вычитке книги, это: Абхилаш Бабу Йотиндра Бабу (Abhilash Babu Jyothendra Babu), Андреа Смит (Andrea Smith), Бисванат Чаудхури (Biswanath Chowdhury), Брайан Гринер (Brian Griner), Брайан С. Коул (Brian S Cole), Дэн Шейх (Dan Sheikh), Дана Робинсон (Dana Robinson), Дэниель Васкес (Daniel Vasquez), Дэвид Паккуд (David Paccoud), Дэвид Пачке (David Patschke), Гжегож Мика (Grzegorz Mika), Джеймс Лиу (James Liu), Йенс Кристиан Б. Мэдсен (Jens Christian B. Madsen), Джереми Чен (Jeremy Chen), Кальян Редди (Kalyan Reddy), Лоренсо Де Леон (Lorenzo De Leon), Ману Сарина (Manu Sareena), Ник Пипенбреер (Nik Piepenbreier), Ноа Флинн (Noah Flynn), Ор Голан (Or Golan), Пауло Нуин (Paulo Nuin), Пера Т. Афшар (Pegah T. Afshar), Ричард Вон (Richard Vaughan), Рууд Гийсен (Ruud Gijzen), Шашанк Каланити (Shashank Kalanithi), Симеон Лейзерсон (Simeon Leyzerzon), Симоне Сгуазца (Simone Sguazza), Срирам Махарла (Sriram Macharla), Срути Шивакумар (Sruti Shivakumar), Стив Лав (Steve Love), Волтер Александер Мата Лопес (Walter Alexander Mata López), Уильям Джамир Силва (William Jamir Silva) и Се Йикуан (Xie Yikuan). Ваши предложения помогли мне сделать эту книгу лучше.

О книге

Целью создания этой книги в первую очередь была помощь программистам в написании высокоэффективных приложений в рамках экосистемы Python. Под эффективностью приложений я прежде всего подразумеваю снижение процессорного времени, а также расходования памяти и сетевых ресурсов при их выполнении.

В книге мы будем затрагивать все аспекты разработки приложений, так или иначе относящиеся к производительности. При этом мы не будем ограничивать себя оптимизацией только в базовом функционале Python, а рассмотрим приемы эффективного использования популярных библиотек, таких как NumPy и pandas. Кроме того, поскольку Python не всегда способен похвастаться быстродействием, мы также при необходимости будем обращаться за помощью к более эффективному расширению языка под названием Cython. Помимо этого, мы затронем вопросы влияния аппаратного обеспечения на эффективность кода. В частности, проанализируем взаимосвязь между современной архитектурой компьютеров и быстродействием выполняемых алгоритмов. Также мы изучим воздействие на производительность конфигурации сети и рассмотрим возможность использования вычислительных ресурсов графического процессора (GPU) для быстрого анализа данных.

Для кого эта книга

Эта книга рассчитана на программистов с определенным опытом. Читая содержание книги, вы должны быть более или менее знакомы с большинством упоминающихся в нем технологий. А если вам довелось поработать с какими-то из них, вообще прекрасно. За исключением разделов, посвященных библиотекам ввода-вывода и вычислениям с помощью GPU, мы не будем сильно вдаваться в описание базовых вещей, а будем полагаться на то, что вы их и так знаете.

Если в настоящее время вы пишете код и думаете о том, как сделать его максимально эффективным, эта книга точно для вас.

И все же для извлечения максимальной пользы из данной книги вы должны обладать хотя бы двухлетним опытом разработки на языке Python, знать его основные управляющие структуры и понимать, как обращаться со списками, множествами и словарями. У вас также желательно должен быть опыт работы с популярными библиотеками Python, такими как `os`, `sys`, `pickle` и `multiprocessing`. Кроме того, чтобы воспользоваться всеми преимуществами показанных в этой книге техник, вы должны неплохо ориентироваться в таких популярных пакетах, как NumPy с его массивами и pandas с датафреймами.

Было бы здорово, если бы вы обладали некоторыми знаниями, пусть и не практическими, в области оптимизации кода на Python с привлечением сторонних языков программирования наподобие C или Rust или с использованием других подходов, включающих задействование расширения Cython или компилятора Numba. Практические наработки в области библиотек ввода-вывода в Python также помогут вам в освоении материала этой книги. Поскольку эти библиотеки не так широко освещаются в литературе, мы начнем с самого начала и познакомимся с таким форматом, как Apache Parquet, и пакетом Zart.

Кроме того, вам необходимо знать основные команды для работы с терминалом Linux (или MacOS). Если у вас Windows, установите любую оболочку на основе Unix или заручитесь необходимыми знаниями для работы с командной строкой или оболочкой PowerShell. Ну и, конечно, без установленного на компьютере интерпретатора Python вам будет не обойтись.

При необходимости я буду давать определенные рекомендации по работе с облачными ресурсами, но доступ к облаку или какие-то особые знания в этой области при чтении книги вам не понадобятся. Если вы заинтересованы в работе с облаком, вы можете узнать все необходимое о приобретении и настройке своей среды у вашего провайдера.

Хотя мы не подразумеваем каких-то особых углубленных знаний с вашей стороны в области оптимизации кода, базовые понятия о скорости выполнения алгоритмов вам не помешают. Например, вы должны понимать, что алгоритмы, масштабирующиеся с ростом объема данных линейно, лучше тех, что масштабируются экспоненциально. Что касается оптимизации вычислений при помощи графического процессора, в этой области вам не потребуются никаких предварительных знаний.

Организация книги

Главы в этой книге по большей части независимы друг от друга, и вы можете перепрыгивать между ними по своему желанию. Несмотря на это, книга состоит из четырех частей.

Часть I (главы 1–4). Фундаментальные подходы. Здесь будет в основном вводный материал:

- в главе 1 мы сформулируем для себя проблему и определимся с тем, зачем именно нужно повышать эффективность в области вычислений и хранения информации. Также в этой главе мы представим целостный подход, применяемый в книге, и обеспечим вас необходимой навигацией;
- глава 2 будет посвящена оптимизации в базовом Python. Мы также коснемся вопросов повышения производительности при использовании структур данных языка Python и поговорим о профилировании кода, выделении памяти и техниках, связанных с ленивыми (отложенными) вычислениями;
- в главе 3 мы будем обсуждать конкурентность и параллелизм в Python, а также узнаем, как можно наиболее эффективно использовать многопроцессную обработку и многопоточность. Одновременно мы затронем вопросы ограничений параллельной обработки при использовании потоков. В этой главе также будет рассмотрена асинхронность как эффективный способ обработки множества конкурентных запросов с низкой нагрузкой, что типично при работе с веб-службами;
- в главе 4 познакомимся с библиотекой NumPy, позволяющей эффективно работать с многомерными массивами. NumPy лежит в основе всех современных техник обработки данных, что делает эту библиотеку одной из ключевых в Python. В этой главе мы затронем специфичные для NumPy техники, позволяющие создавать более эффективный код, такие как представления, транслирование и векторизация.

Часть II (главы 5–6). Аппаратное обеспечение. Здесь мы сосредоточимся на извлечении максимума производительности из имеющихся аппаратных и сетевых ресурсов:

- в главе 5 познакомимся с расширением языка Python под названием Cython, позволяющим создавать гораздо более эффективный код. Python – это интерпретируемый высокоуровневый язык, а значит, от него не стоит ожидать оптимизации под конкретное аппаратное обеспечение. В то же время есть языки программирования, такие как C или Rust, которые способны оптимизировать код под «железо». Cython принадлежит к тому же подмножеству языков, оставаясь при этом в тесном родстве с языком Python. В то же время он позволяет компилировать код Python в C. Для создания эффективного кода на Cython необходимо придерживаться определенных правил в отношении реализации. И в этой главе мы о них подробно поговорим;
- в главе 6 речь пойдет о связи между архитектурой современного аппаратного обеспечения и реализацией кода на Python.

С учетом нюансов архитектуры «железа» наиболее производительным может оказаться код, который внешне не выглядит эффективным. К примеру, иногда работа со сжатыми данными может оказаться более эффективной в сравнении с несжатыми даже с учетом накладных расходов, связанных с их распаковкой. В этой главе мы также поговорим о том, как на реализацию кода на Python влияют архитектура центрального процессора, память, хранилище и сеть. Попутно мы познакомимся с библиотекой NumExpr, позволяющей повысить эффективность кода NumPy за счет использования характерных свойств архитектуры аппаратного обеспечения.

Часть III (главы 7–8). Приложения и библиотеки для современной обработки данных. Как ясно из названия, в этой части книги мы поговорим о распространенных приложениях и библиотеках, позволяющих оптимизировать процесс обработки данных:

- в главе 7 поработаем с библиотекой pandas, позволяющей максимально эффективно оперировать с табличными данными в виде датафреймов в Python. Мы рассмотрим различные техники оптимизации кода, связанные с этой библиотекой. В отличие от большинства глав этой книги в этой главе мы будем возвращаться к тому, что уже упоминалось ранее. Библиотека pandas работает поверх NumPy, так что вспомним кое-что из главы 4 и разработаем методы оптимизации pandas, связанные с пакетом NumPy. Кроме того, мы посмотрим, как можно повысить эффективность pandas с применением библиотеки NumExpr и расширения Cython. Наконец, познакомимся с Arrow – библиотекой, которая, помимо остального функционала, может быть использована для повышения эффективности обработки датафреймов в pandas;
- глава 8 будет целиком посвящена вопросам хранения данных. Мы рассмотрим библиотеку Parquet, позволяющую эффективно обрабатывать колоночные данные, и Zarr, служащую для обработки очень больших данных в виде массивов на диске. Мы также начнем разговор о том, как справляться с наборами данных, объем которых превышает доступные ресурсы памяти.

Часть IV (главы 9–10). Продвинутые возможности. В заключительной части книги мы поговорим о двух не связанных друг с другом темах: работе с графическим процессором (GPU) и использовании библиотеки Dask:

- в главе 9 научимся использовать ресурсы GPU при обработке больших данных. Мы обратим внимание на то, что архитектура графического процессора, предполагающая наличие большого количества простых ядер, хорошо подходит для решения актуальных задач из области науки о данных. Мы опробуем два различных подхода к использованию ресурсов GPU. Сначала поговорим о существующих библиотеках с возможно-

стями, схожими с уже известными вам, таких как CuPy – версии библиотеки NumPy для работы с графическим процессором. А затем узнаем, как писать код на Python, который будет выполняться с использованием ресурсов GPU;

- в главе 10 мы познакомимся с библиотекой Dask, позволяющей писать параллельный масштабируемый код с использованием ресурсов множества машин, как локальных, так и облачных. При всей своей кажущейся сложности эта библиотека предлагает интерфейс, очень похожий на уже знакомые вам библиотеки NumPy и pandas.

Книга также содержит два приложения:

- в приложении А содержатся инструкции для установки и настройки программного обеспечения, необходимого для проверки примеров из этой книги;
- в приложении Б обсуждается компилятор Numba, являющийся альтернативой расширению Cython в области генерирования эффективного низкоуровневого кода. Cython и Numba – это два основных инструмента для создания производительного кода на Python. Для решения насущных задач лично я рекомендую использовать Numba. Почему же я посвятил Cython целую главу, а компилятор Numba оставил для приложения? Причина в том, что главной целью этой книги является помощь в написании эффективного кода в рамках экосистемы Python, а расширение Cython, несмотря на дополнительные сложности, позволяет копнуть глубже в понимании того, что происходит на самом деле.

О сопроводительном коде

В данной книге содержится масса примеров исходного кода как в отдельных листингах, так и внутри обычного текста. В обоих случаях исходный код будет написан моноширинным шрифтом для его выделения на фоне описательного текста.

Стоит отметить, что в большинстве случаев мы вынуждены были переформатировать исходный код, добавив переносы строк и отступы для лучшей читаемости на страницах книги. Также мы удалили из кода некоторые комментарии, которые дублируются рядом с помощью аннотаций. Таких специальных аннотаций к коду будет достаточно много – с помощью них мы постарались выделить особо важные моменты в листингах.

Вы можете загрузить исполняемые примеры кода из онлайн-версии книги, находящейся по адресу <https://livebook.manning.com/book/fast-python>. В полном виде исходные коды собраны на GitHub по адресу <https://github.com/tiagoantao/python-performance>, а также на сайте издательств <https://www.manning.com> и <https://dmkpress.com>. При обнаружении ошибок или изменении требований исполь-

зованных библиотек мы будем обновлять содержимое исходных кодов. Таким образом, в текущем виде код в репозитории может отличаться от присутствующего в книге. Сопроводительные материалы в репозитории организованы по главам.

В целом приведенный в книге код был серьезно адаптирован для нужд печати. К примеру, на практике я являюсь ярким сторонником длинных и понятных имен переменных, но в связи с ограничениями книги такие имена здесь не подходят. Я сделал все, чтобы сохранить выразительность имен переменных и соблюсти все требования стандартов в Python, таких как PEP8, но читаемость кода в книге была поставлена мной во главу угла. То же самое верно и для аннотаций типов: я бы с удовольствием их использовал, но они нарушают читаемость кода.

В большинстве случаев код, приведенный в этой книге, будет работать со стандартным интерпретатором Python. Иногда для выполнения кода потребуется IPython, особенно когда речь пойдет об анализе производительности. Также вы можете использовать Jupyter Notebook.

Инструкции по установке и настройке необходимого программного обеспечения можно найти в приложении А. Если в какой-то главе или разделе потребуется установить дополнительные программы или библиотеки, об этом будет упомянуто отдельно.

Программное и аппаратное обеспечение

Исходный код, приведенный в книге, вы можете запускать на любой операционной системе. В то же время в большинстве случаев рабочие проекты развертываются в среде Linux, так что эта операционная система может считаться предпочтительной. На MacOS X тоже никаких проблем с адаптацией не возникнет. Что касается Windows, я рекомендую установить подсистему Linux для Windows (WSL – Windows Subsystem for Linux).

Альтернативой операционным системам может считаться запуск кода в Docker. Вы можете использовать образы Docker, содержащиеся в репозитории. С помощью Docker вы можете создать окружение Linux в виде контейнера для запуска приведенного в книге кода.

В качестве минимальных требований для запуска кода я бы порекомендовал конфигурацию с 16 Гб памяти и 150 Гб свободного дискового пространства. В главе 9 мы будем обсуждать темы, связанные с использованием вычислительных ресурсов графического процессора, и для проверки кода вам потребуется GPU от NVIDIA с микроархитектурой Pascal. Большинство GPU, выпущенных компаниями за последние пять лет, будут отвечать этому требованию. Все остальные инструкции по установке и настройке программного и аппаратного обеспечения можно найти в приложении А.

Об авторе



Тиаго Антао (Tiago Rodrigues Antão) обладает степенью бакалавра технических наук в области информатики и докторской степенью в области биоинформатики. В настоящее время работает в сфере биотехнологии. В своей работе Тиаго на постоянной основе использует язык Python и все его популярные библиотеки для выполнения научных расчетов и анализа данных. Для оптимизации критических алгоритмов исполь-

зует низкоуровневые языки программирования C и Rust. В настоящее время Тиаго занимается разработкой инфраструктуры на базе Amazon AWS, но на протяжении большей части карьеры использовал локальные вычислительные кластеры.

Помимо плодотворной работы в индустрии, Тиаго прошел две постдокторантуры по анализу данных в Кембриджском и Оксфордском университетах. В качестве ученого-исследователя при университете Монтаны он с нуля создал научно-вычислительную инфраструктуру для анализа биологических данных.

Тиаго является одним из создателей популярного набора модулей *Biopython*, написанных на Python, а также автором книги «Bioinformatics with Python Cookbook», увидевшей свет в 2022 году уже в третьем издании. Кроме того, Тиаго написал большое количество важных научных докладов и статей по биоинформатике.

О переводчике



Александр Гинько, обладающий богатым опытом работы в сфере ИТ и более десяти лет посвятивший переводам книг и статей на самые разные темы, в последние годы специализируется на переводе книг в области бизнес-аналитики и программирования для издательства «ДМК Пресс» по направлениям Python, SQL, Power BI, DAX, Excel, Power Query, Tableau, R... На данный момент в активе Александра уже более 20 книг, включая одну автор-

скую, и он продолжает плодотворно работать над переводом новых.

Помимо перевода книг, Александр ведет свой канал в Telegram (https://t.me/alexanderginko_books), на котором вы можете из первых уст получить ответы на все интересующие вас вопросы об уже переведенных книгах, находящихся в работе и запланированных на будущее. Также на канале можно найти промокоды на все книги Александра для покупки книг на сайте издательства «ДМК Пресс» с большими скидками.

Об изображении на обложке

Картина на обложке книги носит название «Представительница буржуазии из Пассо» (Bourgeoise de Passeau) и принадлежит коллекции художника Жака Грассе де Сэйнт-Совера (Jacques Grasset de Saint-Sauveur). Впервые картина была показана в 1797 году.

В те времена очень легко было по одежде определить местожителство, род занятий и статус человека. Издательство Manning традиционно оформляет обложки книг по компьютерной тематике шедеврами мирового искусства, отдавая дань богатому разнообразию региональных культур прошлых веков.

Часть I

Фундаментальные подходы

В первой части книги мы поговорим об основных подходах и предпосылках в отношении эффективности кода, написанного на языке Python. Мы пройдемся по популярным библиотекам языка и основным структурам данных, а также посмотрим, как в Python (без использования внешних пакетов) реализованы техники параллельных вычислений. Отдельная глава будет посвящена оптимизации вычислений с использованием библиотеки NumPy. И хотя теоретически библиотека NumPy является внешней, ее роль в современных технологиях обработки данных столь велика, что мы включили ее в эту часть книги наряду с другими фундаментальными подходами к вычислениям.

1

Острая нехватка производительности при обработке данных

В этой главе мы обсудим следующие темы:

- способы борьбы с экспоненциальным ростом объемов данных;
- сравнение традиционных и современных вычислительных архитектур;
- роль и недостатки языка Python в современном анализе данных;
- техники для реализации эффективных вычислительных решений на Python.

В настоящее время данные собираются в невероятном количестве, на огромных скоростях и из самых разнообразных источников. Они собираются даже безотносительно к тому, будут использоваться в ближайшем будущем или нет, есть ли место для их хранения, время и мощности для обработки, анализа и изучения. Еще до того, как аналитики решат, как использовать эти данные, а разработчики и управленцы поймут, как на их основе создавать новые служ-

бы и продукты, инженеры-программисты должны найти способы для их хранения и обработки. И сейчас больше, чем когда-либо, им необходимо пытаться использовать любые возможности для повышения эффективности процесса сбора данных и оптимизации их хранения.

В этой книге я попытался собрать все известные мне стратегии по оптимизации обработки и хранения данных – лично я всеми ими пользуюсь в своей ежедневной работе. Просто выделить больше машин для обработки информации зачастую бывает невозможно, да это и не помогает. Таким образом, мы будем больше говорить об эффективных средствах, имеющихся в нашем распоряжении, как то: оптимизация кода, адаптация под архитектуру программного и аппаратного обеспечения и, конечно, нюансы языка, библиотек и экосистемы Python в целом.

Python стал наиболее предпочтительным языком или некой связующей прослойкой для выполнения всей рутинной работы в условиях этого нескончаемого потока или даже потопа данных. Популярность Python в науке о данных и инженерии данных обусловила рост его востребованности во всех остальных сферах, что сделало этот язык, по данным различных исследований, одним из трех наиболее популярных языков программирования. Язык Python имеет целый ряд преимуществ, но без недостатков применительно к обработке больших данных в нем не обошлось. В частности, это касается вопросов скорости обработки данных. К счастью, существует множество подходов для устранения этих недостатков, которые способны значительно повысить эффективность языка Python при обработке больших данных.

Но, прежде чем решать проблемы, необходимо точно и четко их проговорить, чем мы по большей части и будем заниматься в данной главе. Мы поговорим о последствиях наводнения всех сфер нашей жизни данными и обрисуем проблемы, с которыми нам, как инженерам, приходится сталкиваться при обработке этого бесконечного потока. После этого обсудим роль аппаратного обеспечения, сети и архитектуры облачных ресурсов, чтобы понять, что прежние подходы с увеличением рабочей частоты центрального процессора больше не работают. Затем мы проговорим проблемы, с которыми сталкивается язык Python при обработке больших объемов данных, включая управление потоками и глобальную блокировку интерпретатора, применяемую в CPython. И только после осознания того, что для повышения эффективности кода на Python нам необходимы новые пути, мы представим решения, которые будем реализовывать на протяжении всей книги.

1.1. Насколько велик всемирный потоп данных?

Возможно, вы слышали о существовании двух законов, носящих имена Мура (Moore) и Эдхольма (Edholm), которые в совокупности рисуют очень драматичную картину, связанную с экспоненциальным ростом собираемых данных и отставанием индустрии по обработке этих самых данных. Так, *закон Эдхольма* гласит, что объем собираемых данных с помощью средств телекоммуникации удваивается каждые 18 месяцев. В то же время *закон Мура* утверждает, что количество транзисторов, размещаемых на кристалле интегральной схемы, удваивается каждые 24 месяца. С целью упрощения мы можем применить закон Эдхольма ко всем собираемым данным, а закон Мура интерпретировать как индикатор доступных ресурсов вычислительного оборудования. Если совместить два этих закона, мы получим запаздывание развития технологий относительно роста объема данных для обработки и хранения на полгода. Поскольку экспоненциальный рост трудно описать словами, лучше будет взглянуть на график, показанный на рис. 1.1.

Ситуацию на этом графике можно описать как борьбу между тем, что нам нужно проанализировать (закон Эдхольма), и тем, с помощью чего мы собираемся проводить анализ (закон Мура). При этом на графике перспектива показана даже в более оптимистичном свете по сравнению с реальностью. Почему? Узнаем в главе 6, когда будем рассматривать закон Мура применительно к современной архитектуре центральных процессоров.

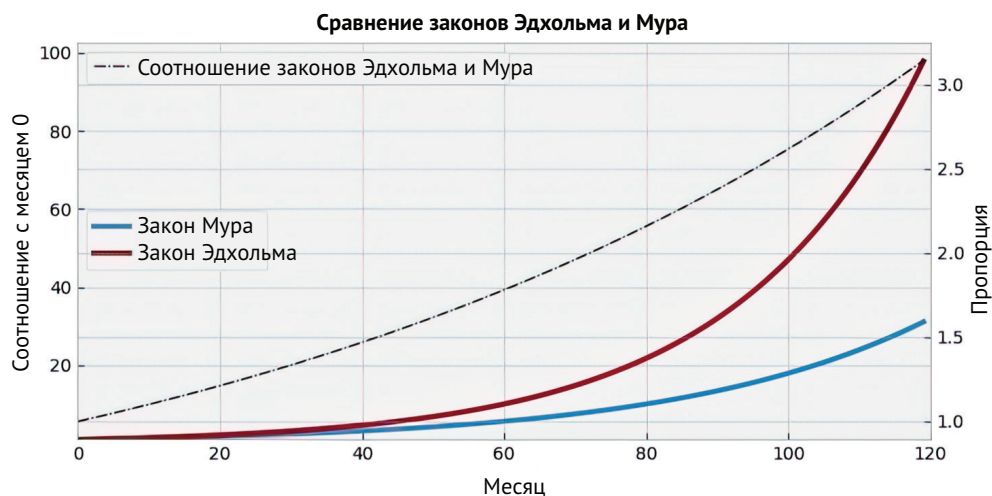


Рис. 1.1. Соотношение законов Эдхольма и Мура предвещает постоянно увеличивающееся отставание вычислительных возможностей от доступного объема данных

Чтобы лучше прочувствовать рост объема данных, взглянем еще на один пример, связанный с интернет-трафиком, с помощью которого косвенно можно оценить объем собираемых данных. На графике, показанном на рис. 1.2, видно, что объем трафика с годами неминуемо растет и практически повторяет линию закона Эдхольма.

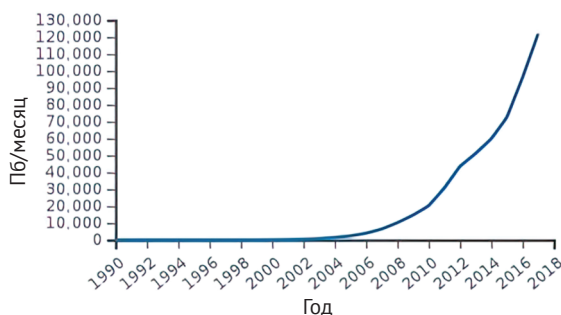


Рис. 1.2. Рост интернет-трафика с течением лет, измеренный в петабайтах (Пб) в месяц (источник: https://en.wikipedia.org/wiki/Internet_traffic)

Кроме того, по данным сайта [uschamberfoundation.org](https://www.uschamberfoundation.org), 90 % данных, сгенерированных человечеством, пришлось на последние два года (<https://www.uschamberfoundation.org/bhq/big-data-and-what-it-means>). Можно спорить о том, как соотносится ценность этих данных с их объемом. Но суть в том, что сгенерированные и собранные данные нужно как-то обрабатывать, а это требует ресурсов.

При этом не только объем данных становится препятствием для их эффективной обработки. Способ представления данных за последнее время также претерпел серьезные изменения. Согласно некоторым прогнозам, к 2025 году порядка 80 % всех данных станут неструктурированными (<https://mitsloan.mit.edu/ideas-made-to-matter/tapping-power-unstructured-data>). Позже в этой книге мы поговорим об этом подробно, а сейчас лишь скажем, что для обработки неструктурированных данных требуется гораздо больше вычислительных ресурсов.

Как же мы в настоящее время справляемся с растущими объемами данных? Да по большей части никак. Если верить The Guardian, 99 % собранной информации так и не были проанализированы (<https://www.theguardian.com/news/datablog/2012/dec/19/big-data-study-digital-universe-global-volume>). И частично это происходит из-за недостатка вычислительных средств, с помощью которых можно обрабатывать и анализировать имеющиеся данные.

Рост объема данных и соответствующий запрос на увеличение вычислительных мощностей приводят к пагубной мантре, то и дело повторяемой то тут, то там: «Если к вам пришло больше данных, просто выделите для их обработки больше серверов». По разным причинам такой подход в большинстве случаев уже не рабо-

тает. Вместо этого при необходимости повысить производительность существующей системы необходимо в первую очередь обратить внимание на ее архитектуру и реализацию, а также найти места для применения оптимизации. Я уже сбился со счета, сколько раз мне удавалось значительно повысить производительность решения путем логического анализа кода и его оптимизации.

Здесь очень важно понимать, что связь между ростом объемов данных и сложностью инфраструктуры для их анализа имеет нелинейный характер. Решение подобных проблем потребует от вас, как от разработчика, больше времени и изобретательности, чем от машин. И это справедливо не только для облачных решений, но и для локальных кластеров и даже для одной машины. Следующие примеры помогут вам понять, о чем именно я говорю:

- *ваше решение было рассчитано на один компьютер, но вдруг потребовалось больше машин для обработки данных.* Добавление компьютеров в вычислительную систему потребует от вас настройки распределения рабочей нагрузки между машинами и отслеживания корректности разделения данных между ними при хранении. Также вам может понадобиться распределенная файловая система с отдельным сервером, который нужно будет добавить к списку машин. Иными словами, поддерживать и обслуживать целую серверную ферму или даже просто облако будет намного сложнее, чем один компьютер;
- *ваше решение уместилось в памяти, но с ростом данных памяти компьютера стало не хватать.* Вариант с хранением дополнительных данных на диске обычно требует серьезного вмешательства в код. К тому же сам код при этом неизбежно усложнится. К примеру, если основная база данных теперь будет располагаться на диске, вам, скорее всего, придется реализовывать подсистему управления кешем. Также может понадобиться реализация одновременного чтения из разных процессов. Или, что еще хуже, одновременной записи;
- *вы используете базу данных SQL и в какой-то момент достигли уровня предельной пропускной способности сервера.* Если вопрос только в чтении данных, можете обойти это ограничение путем создания нескольких реплик для чтения. Но что делать, если проблема с записью? Вы можете настроить *шардирование* (sharding), заключающееся в расположении разных сегментов базы данных на разных физических серверах. А можете полностью сменить технологию хранения данных, перейдя на вариант NoSQL;
- *если вы оказались в плену облачной системы, реализованной с использованием собственных технологий провайдера, вы в какой-то момент можете обнаружить, что вопросы масштабирования ресурсов лежат больше в плоскости маркетинга, нежели технологических реалий.* За-

частую при достижении какого-либо ограничения единственным реальным вариантом является смена используемой технологии, что требует времени, денег и немалых затрат энергии.

Я полагаю, что этих примеров достаточно, чтобы понять, что увеличение вычислительных мощностей далеко не всегда может быть связано с добавлением новых серверов. Это гораздо более сложная составная проблема. Даже такое «простое» улучшение, как внедрение параллельных вычислений на одной машине, может тянуть за собой все известные проблемы параллельных вычислений, такие как *гонки данных* (races), состоящие в одновременном доступе разных потоков к одной и той же ячейке памяти, *взаимные блокировки* (deadlocks) и т. д. Таким образом, подобные методы оптимизации могут быть связаны с дополнительными сложностями, проблемами с надежностью и высокой стоимостью.

Наконец, если даже предположить, что мы можем масштабировать нашу инфраструктуру линейно (а мы не можем!), перед нами встали бы вопросы этического и экологического свойства. Дело в том, что прогнозы отводят на «цунами данных» порядка 20 % всего мирового энергопотребления (<https://www.theguardian.com/environment/2017/dec/11/tsunami-of-data-could-consume-fifth-global-electricity-by-2025>), и о проблемах с промышленными отходами при расширении аппаратных ресурсов также забывать не стоит.

Хорошие новости состоят в том, что использование более эффективных вычислительных решений при обработке больших данных может позволить существенно снизить затраты на расчеты, требования к архитектуре аппаратных средств и хранению данных, энергопотребление, а также срок внедрения готового продукта. А бонусом является то, что зачастую эффективные вычислительные решения не потребуют от вас никаких дополнительных вложений средств и времени. К примеру, грамотное использование структур данных в Python может кардинально снизить время вычисления и не потребует существенных изменений в исходном коде.

С другой стороны, многие решения, которые мы будем рассматривать в этой книге, потребуют определенных затрат на разработку и будут достаточно сложными. При оценке *своих* данных и перспективы роста их объема вам необходимо принять решение касательного вектора оптимизации, поскольку единого универсального рецепта здесь просто нет и быть не может. Есть лишь одно общее правило, которого стоит придерживаться, и звучит оно так: если какое-то решение подходит Netflix, Google, Amazon, Apple или Facebook, это еще не значит, что оно подходит вам, – если, конечно, вы не работаете в одной из этих компаний.

Объемы данных, которыми оперирует большинство из нас, несопоставимы с объемами крупнейших технологических компаний.

Да, данных у всех много, и справляться с ними непросто, но их все равно на несколько порядков меньше, чем у гигантов рынка. И предполагать, что решения, оптимально подходящие этим компаниям, так же хорошо подойдут и вам, мне кажется, довольно самонадеянно. Скорее всего, в наших случаях лучше отработают гораздо более простые решения.

Как видите, в этом обновленном мире с экстремальными ростами объемов данных и сложности алгоритмов необходимо искать новые продвинутые техники для осуществления вычислений и хранения, не допуская серьезного роста стоимости решений. Не поймите меня неправильно: иногда просто необходимо расширять инфраструктуру. Но при построении архитектуры и разработке своих решений всегда нужно думать о производительности вне зависимости от выбранной техники.

1.2. Современные вычислительные архитектуры и высокопроизводительные вычисления

Создание высокоэффективных решений происходит не на пустом месте. Первое, с чем вам необходимо определиться, – это характер проблемы: какую именно задачу вы решаете? Также важное значение имеет *вычислительная архитектура* (computing architecture), в рамках которой будет работать ваше решение. Выбор вычислительной архитектуры напрямую влияет на техники, используемые при оптимизации, так что этому вопросу стоит уделить особое внимание. В этом разделе мы укажем на основные проблемы, связанные с выбором архитектуры, которые могут оказывать влияние на реализацию ваших решений.

1.2.1. Изменения в архитектуре компьютеров

Современная архитектура компьютеров претерпевает серьезные изменения. Во-первых, мы видим, что вычислительные мощности *центральных процессоров (CPU)* в последнее время полагаются в большей степени на количество параллельных ядер, а не на тактовую частоту, как это было в недавнем прошлом. Кроме того, новые компьютеры оснащаются современными *графическими процессорами (GPU)*, роль которых изменилась – если раньше они предназначались исключительно для графической обработки, то сегодня они могут использоваться и для более общих вычислений. По сути, многие эффективные реализации алгоритмов, связанных с искусственным интеллектом, рассчитаны именно на графические процессоры. К сожалению – по крайней мере, для нас, – графические процессоры обладают совершенно иной архитектурой по сравнению с центральными процессорами: они состоят из ты-

сяч простых ядер, рассчитанных на одновременное выполнение множества однотипных «простых» операций. Модель управления памятью в них тоже серьезно отличается. Все эти отличия обуславливают кардинальную разницу в подходах к программированию для CPU и GPU.

Для понимания того, как можно использовать графический процессор с целью обработки данных, необходимо хорошо знать его предназначение и архитектуру. Графический процессор, как ясно из названия, изначально предназначался для обработки графических сигналов. Самыми требовательными приложениями с точки зрения такого рода вычислений являются игры. А чем в основном занимаются игры и графические приложения? Они постоянно обновляют миллионы пикселей на экране. Именно поэтому графические процессоры снабжаются огромным количеством небольших ядер, которые и занимаются этой рутинной работой. Таким образом, вы очень легко можете встретить GPU с тысячами ядер, тогда как в CPU количество вычислительных модулей обычно не превышает десяти. Конечно, архитектура ядер графического процессора намного проще, чем у центрального процессора, и обычно эти ядра выполняют один и тот же код. Именно это делает графические процессоры наиболее пригодными для выполнения большого количества однотипных операций вроде обновления пикселей.

Осознав вычислительную мощь, которой с годами были наделены графические процессоры, инженеры сделали попытку использовать их в вычислениях иного типа – так появилась техника, получившая название *общие вычисления на графических процессорах* (general-purpose computing on graphics processing units – *GPGPU*). Оказалось, что архитектура GPU идеально подходит для решения огромного количества параллельных задач. А многие алгоритмы искусственного интеллекта, как те, что базируются на нейронных сетях, построенны как раз на параллельных вычислениях. Итог – полная гармония.

К сожалению, отличия между архитектурами GPU и CPU не ограничиваются количеством ядер и их сложностью. Память графических процессоров, особенно это касается наиболее мощных из них, отделена от основной памяти. Таким образом, появляется задержка, обусловленная передачей данных между модулями памяти. Вот вам и вторая проблема, которую стоит учитывать при программировании в расчете на GPU.

В главе 9 мы подробно рассмотрим причины, по которым программировать под GPU на Python гораздо сложнее и не так практично, как под CPU. Но не стоит отчаиваться – существует немало приемов эффективного использования вычислительной мощи графического процессора из Python.

Изменения в архитектуре CPU за последнее время были не столь заметными по сравнению с GPU, зато все преимущества от них

можно полноценно использовать в Python. Производители центральных процессоров в последние годы сменили вектор развития вычислительных мощностей. Они последовали за законами физики и сделали упор на параллельные вычисления, а не на увеличение тактовой частоты. Закон Мура, который мы упоминали ранее, зачастую приводится как утверждение о том, что частота процессоров увеличивается раз в 24 месяца. На самом же деле речь идет об удвоении количества транзисторов, размещаемых на кристалле интегральной схемы. Линейная связь между количеством транзисторов и рабочей частотой процессора была разорвана больше десяти лет назад, и с тех пор частота вышла на плато. А с учетом того, что объем данных и сложность алгоритмов продолжали расти, ситуация стала щекотливой. Первым делом производители процессоров обратили взгляд на параллелизм: больше процессоров в компьютерах, больше ядер в процессорах, многопоточность... Таким образом, скорость последовательной обработки задач с тех времен почти не изменилась, а основной упор был сделан на параллельные вычисления. И это неминуемо отразилось на подходах к программированию – по сути, изменилась вся парадигма. В прежние времена скорость выполнения программ сама по себе увеличивалась при смене центрального процессора. Сейчас же быстроедействие напрямую зависит от того, насколько эффективно программист реализовал принципы параллельных вычислений при написании решения.

В целом за последние годы произошло достаточно много изменений в отношении принципов программирования под современные процессоры, и в главе 6 вы увидите, что некоторые из них не столь очевидны на первый взгляд. Например, несмотря на стагнацию частоты процессора в последние годы, по скорости CPU по-прежнему значительно превосходит *оперативную память* (RAM). Если бы не существовало *кеша* (CPU cache), процессоры большую часть времени проводили бы в ожидании отклика от оперативной памяти. Этим объясняется тот факт, что иногда бывает быстрее работать со сжатыми данными, чем с распакованными, даже с учетом накладных расходов. Почему так происходит? Если вы можете поместить блок сжатых данных в кеш CPU, то такты процессора, которые иначе простаивали бы в ожидании отклика от памяти, могут быть использованы для распаковки данных, тогда как оставшиеся такты могут заниматься вычислениями. Похожая аргументация работает и в пользу использования *сжатой файловой системы* (compressed file system) в противовес обычной. Применение этому аспекту находится и в мире Python. К примеру, изменив одно булево значение, отвечающее за внутреннее представление массивов NumPy, вы можете воспользоваться всеми преимуществами, связанными с кешем, и значительно ускорить вычисления в библиотеке NumPy. Разные типы памяти, такие как кеш процес-

сора, оперативная память, локальный диск и сетевое хранилище, характеризуются своим объемом и *временем доступа* (access time), которые мы свели в табл. 1.1. Здесь важна не абсолютная точность показателей, а порядок значений как в отношении объема, так и в плане времени доступа.

Таблица 1.1. Иерархия устройств памяти с объемами и временами доступа для вымышленной, но вполне реалистичной современной машины

Тип	Объем	Время доступа
Центральный процессор		
Кеш первого уровня (L1 cache)	256 Кб	2 нс
Кеш второго уровня (L2 cache)	1 Мб	5 нс
Кеш третьего уровня (L3 cache)	6 Мб	30 нс
Оперативная память		
DIMM	8 Гб	100 нс
Вторичная память		
SSD	256 Гб	50 мкс
HDD	2 Тб	5 мс
Третичная память		
Сервер сетевого доступа (Network Access Server – NAS)	100 Тб	Зависит от сети
Облачный ресурс	1 Пб	Зависит от провайдера

В табл. 1.1 мы также включили *третичную память* (tertiary storage), которая физически находится за пределами нашего устройства. В этой области также произошли изменения, о которых мы поговорим в следующем разделе.

1.2.2. Изменения в архитектуре сети

В высокопроизводительных вычислительных системах *сеть* (network) используется как для хранения данных, так и (в особенности) для их обработки путем увеличения вычислительных мощностей. И даже если нам очень хочется обходиться ресурсами одного компьютера, зачастую это бывает невозможно, и требуется привлечение дополнительного вычислительного кластера. Оптимизация приложений для выполнения в распределенной среде – будь то в облаке или на локальном кластере – будет частью нашего путешествия в область высокоэффективных решений.

Использование множества компьютеров для расчетов и внешнего хранилища ведет к новому классу проблем распределенных вычис-

лений, связанных с топологией сетей, распределенным хранением данных и управлением сетевыми процессами. В этой связи можно привести множество примеров. Какой смысл в использовании REST API в службах, требующих высокой производительности и низких задержек? Как можно справиться с проблемами, связанными с использованием удаленных файловых систем, или хотя бы нивелировать их?

Мы попытаемся оптимизировать использование *сетевого стека* (network stack), но для этого необходимо понимать, как устроены все его уровни, показанные на рис. 1.3. За пределами сети у нас есть код на Python с использованием различных библиотек, в котором делается выбор относительно использования этих уровней. На вершине сетевого стека обычно располагается протокол передачи данных *HTTPS* с входной информацией в формате JSON.

Хотя для большинства приложений этот выбор будет приемлемым, существуют и более эффективные варианты передачи данных в отношении скорости и задержек. К примеру, вы можете передавать входные данные запроса в двоичном виде, а не в формате JSON. Кроме того, вы можете заменить транспортный протокол HTTP на *TCP-socket* (TCP socket). Существуют и более радикальные альтернативы по замене транспортного слоя с протоколом TCP: большинство интернет-приложений используют протокол TCP, хотя есть и редкие исключения, прибегающие к помощи *DNS* и *DHCP*, построенных на базе протокола *UDP*. Протокол TCP отличается высокой надежностью, но за эту надежность приходится платить быстродействием. Иногда можно воспользоваться протоколом UDP с гораздо меньшими накладными расходами, если надежность не играет ключевой роли.

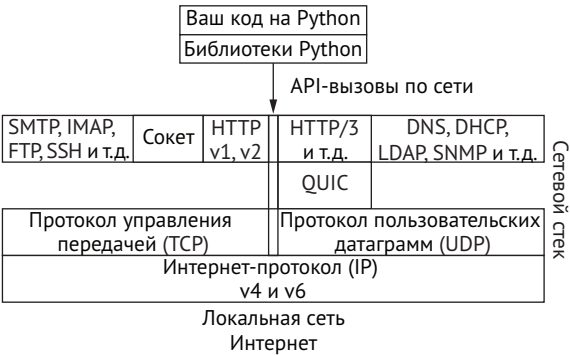


Рис. 1.3. Вызовы API по сети. Понимание существующих альтернатив в отношении передачи информации по сети может позволить существенно увеличить быстродействие приложения

Под транспортными протоколами располагается *интернет-протокол* (Internet protocol – IP) и физическая инфраструктура. Выбор физической инфраструктуры бывает очень важен при проектиро-

вании решения. К примеру, если ваша локальная сеть отличается повышенной надежностью, выбор в пользу протокола UDP, способного терять пакеты, может быть более оправданным по сравнению с ненадежной сетью.

1.2.3. Облако

В прошлом большинство процессов по обработке данных реализовывались с расчетом на одну машину или локальный кластер в рамках одной организации. Сейчас большое распространение получили *облачные* (cloud) инфраструктуры, в которых все серверы являются виртуальными, а поддержка осуществляется извне. А когда речь идет о *бессерверных вычислениях* (serverless computing), мы даже не взаимодействуем с серверами напрямую.

Облако – это не только про дополнительные компьютеры и сетевые хранилища. Это еще и про установку собственных расширений по управлению хранением данных и вычислительными ресурсами, и эти расширения также влияют на производительность. Кроме того, виртуальные компьютеры могут помешать выполнению некоторых оптимизаций в области центрального процессора. К примеру, на локальной машине вы можете разработать решение, которое будет использовать кеш, тогда как на виртуальной вы не можете быть уверены в том, что содержимое вашего кеша не вытесняется другой виртуальной машиной, работающей параллельно. Как в подобном окружении поддерживать эффективность алгоритмов? Помимо этого, в среде с применением облачных ресурсов стоимостная модель вычислений будет совершенно иной. Время здесь – это в буквальном смысле деньги, а значит, быстроедействие решений будет играть еще большую роль.

Также в облачных средах зачастую реализованы собственные подсистемы вычислений и хранения данных, что обуславливает наличие специфических API и свое поведение. Каждое из таких обособленных технических решений имеет свою производительность, и с этим необходимо считаться. Таким образом, помимо проблем, характерных для кластерных реализаций, облачные решения будут страдать от недостатков, присущих только им. Теперь, когда мы поговорили об архитектурных возможностях и ограничениях при разработке приложений, давайте рассмотрим преимущества и недостатки решений на языке Python в отношении производительности.

1.3. Работа с ограничениями языка Python

В наше время язык Python повсеместно используется для обработки данных. Как и у любого языка программирования, у него есть свои достоинства и недостатки. Для использования Python существует масса причин, но в этой книге мы по большей части сосре-

доточимся на ограничениях языка в отношении высокоэффективной обработки данных.

Давайте не будем приукрашивать реальность: Python просто не предназначен для выполнения высокопроизводительных операций по работе с данными. Если бы значение имели только быстроедействие и параллелизм, никто бы не использовал Python в своей работе. Но у этого языка богатейший арсенал библиотек для анализа данных, великолепная документация и поразительно отзывчивое сообщество. Именно это заставляет нас использовать Python, а отнюдь не его непревзойденная скорость.

Есть одно распространенное высказывание: «Нет медленных языков программирования, есть их медленные реализации». Полагая, вы позволите мне с ним не согласиться. Было бы несправедливо спрашивать у человека, реализующего интерпретатор динамического высокоуровневого языка программирования, коим является Python (или, скажем, JavaScript), почему его детище не может сравниться в скорости с низкоуровневыми языками C, C++, Rust или Go.

Такие вещи, как динамическая типизация и сбор мусора, не могут не сказываться на быстродействии приложений. Но сегодня это нормально: в наши дни время, затраченное на работу программистом, иногда ценится больше, чем время вычислений. Однако не стоит делать вид, что проблем нет, – просто при работе с более декларативными и динамическими языками программирования вы будете вынуждены платить определенную цену в виде быстрогодействия и расходования памяти. Это компромисс.

Несмотря на все вышесказанное, медленные реализации языков тоже существуют. Как узнать, где на шкале быстродействия находится флагманская реализация *CPython*, которую вы наверняка сами используете? Провести полноценный анализ будет непросто, но вы можете поставить простой опыт: напишите функцию перемножения матриц и замерьте время ее выполнения. Затем сделайте то же самое в другой реализации языка, например в *PyPy*. После этого перепишите код на языке JavaScript (сопоставимый выбор по сравнению с Python, поскольку этот язык также является динамическим; несправедливо было бы сравнивать Python с C) и снова замерьте время.

Внимание, спойлер: *CPython* не покажет каких-то потрясающих результатов. Стоит признать, что мы имеем дело не с самым быстрым языком, и в его флагманской реализации производительность также не ставится во главу угла. Хорошая новость состоит в том, что большинство из связанных с этим проблем преодолимы. Для Python написано большое количество расширений и библиотек, с помощью которых при желании можно минимизировать многие неудобства. Таким образом, вы сможете продолжить писать код на языке Python, который будет выполняться достаточно

быстро и расходовать мало памяти. Просто нужно будет выполнять определенные требования и помнить об особенностях языка.

ПРИМЕЧАНИЕ. По большей части, когда мы будем говорить в этой книге про Python, мы будем подразумевать интерпретатор CPython. Обо всех исключениях из этого правила будет сообщаться дополнительно.

С учетом проблем языка Python с быстроедействием одного только изменения кода иногда нам будет недостаточно. В таких случаях мы будем прибегать к переписыванию фрагментов кода на низкоуровневых языках или хотя бы пользоваться специальными пометками, которые позволят транслировать код в низкоуровневый при помощи специальной утилиты. Эти фрагменты обычно будут небольшими, и из-за них нет смысла отказываться от использования языка Python. Даже после этой адаптации порядка 90 % кода может быть написано на Python. Именно так устроены популярные научные библиотеки вроде NumPy, scikit-learn и SciPy – наиболее требовательные участки кода в них реализованы на языках C или Fortran.

1.3.1. Глобальная блокировка интерпретатора

Практически ни одна дискуссия о языке Python не обходится без упоминания пресловутой *глобальной блокировки интерпретатора* (Global Interpreter Lock – *GIL*). Что же это за зверь такой? Несмотря на заявленную концепцию потоков в Python, в реализации CPython присутствует глобальная блокировка интерпретатора, препятствующая запуску нескольких потоков на исполнение одновременно.

В других интерпретаторах Python, таких как *Jython* и *IronPython*, глобальная блокировка интерпретатора отсутствует, что позволяет использовать многоядерную архитектуру современных процессоров. Но интерпретатор CPython по-прежнему считается эталоном, и именно под него пишутся все основные библиотеки. К тому же реализации *Jython* и *IronPython* зависят от установки JVM и .NET соответственно. В общем, интерпретатор CPython с его массивным набором библиотек на сегодняшний день считается реализацией Python по умолчанию. В этой книге мы коротко остановимся на нескольких интерпретаторах языка, главным образом на PyPy, но на практике CPython – безусловный номер один.

Для понимания того, как можно обойти глобальную блокировку интерпретатора, полезно будет вспомнить о разнице между *конкурентностью* (concurrency) и *параллелизмом* (parallelism). Конкурентность, как вы можете помнить, относится к ситуации с возможным пересечением выполнения нескольких задач во времени, в то же время разные задачи не могут выполняться одновременно. Но они вполне могут чередоваться. Параллелизм подразумевает одновременное выполне-

ние разных задач. Получается, в Python мы можем реализовать конкурентность, а о параллелизме можно забыть? Или как?

Конкурентность сама по себе очень полезна – даже без применения принципов параллелизма. Лучшим примером этого высказывания является платформа *Node.js* на базе JavaScript, ставшая едва ли не основным инструментом для реализации бэкенда на веб-серверах. Зачастую серверная часть веб-служб организована так, что большую часть времени происходит ожидание операций ввода-вывода, и в это время ожидающий поток добровольно передает управление другим потокам, в которых могут производиться вычисления. В современных реализациях Python присутствуют эти асинхронные возможности, и мы будем подробно о них говорить.

Но вернемся к главному вопросу: серьезно ли глобальная блокировка интерпретатора сказывается на быстродействии? Удивительно, но в большинстве случаев ответ будет отрицательным. И на то есть две причины:

- требовательные к быстродействию участки кода могут быть переписаны с использованием низкоуровневых языков программирования, как мы уже говорили ранее;
- в Python реализованы механизмы для низкоуровневых языков по отказу от блокировки.

Это означает, что, переписав часть кода с использованием низкоуровневого языка программирования, вы можете проинструктировать Python выполнять другие потоки параллельно с вашей обновленной реализацией фрагмента кода. Вы должны отказываться от использования глобальной блокировки интерпретатора только тогда, когда это безопасно, – например, если вы не выполняете запись в объекты, которые могут использоваться другими потоками.

Что касается *многопроцессной обработки* (multiprocessing), т. е. запуска нескольких процессов одновременно, то на нее глобальная блокировка интерпретатора не влияет, поскольку она распространяется только на потоки. Это позволяет реализовывать параллельные решения даже с использованием базового Python.

Таким образом, глобальная блокировка интерпретатора только на словах является большой проблемой в области быстродействия решений. На практике же она редко становится источником проблем, которые невозможно решить. Подробнее мы поговорим об этом в главе 3.

1.4. Возможные решения

Эта книга посвящена оптимизации решений на языке Python, но программный код существует не в вакууме. Вы можете добиться

существенного прогресса с области быстрогодействия кода, только если будете принимать во внимание все требования к данным и алгоритмам, а также учитывать существующую вычислительную архитектуру. Поскольку невозможно осветить в одной книге все аспекты, связанные с архитектурой и алгоритмами, я в первую очередь помогу вам понять роль архитектуры центрального и графического процессоров, устройств хранения информации, сетевых протоколов и других составляющих, показанных на рис. 1.4, чтобы вы могли учитывать их при написании оптимального кода на Python. Эта книга должна помочь вам в оценке преимуществ и недостатков архитектуры вашего оборудования, будь то персональный компьютер, компьютер с современным графическим процессором, локальный кластер или облачное окружение, и написании оптимального кода с учетом всех этих аспектов.

Среда аппаратного обеспечения

Локально / в облаке / гибрид		
Вычисления	Хранение	Сеть
Компьютер	Кеш ЦПУ	Топология
Виртуальная машина	Оперативная	Протоколы
Экземпляр облака	память	Скорость
Бессерверные	Файловая система	Задержка
вычисления	SQL	
	NoSQL	
	Облако	
	Сетевое хранилище	

Рис. 1.4. При выборе оптимальных решений необходимо учитывать все слои архитектуры аппаратного обеспечения

Цель этой книги – показать вам все разнообразие решений по оптимизации и дать понять, когда и какие из них стоит использовать, чтобы прирост эффективности был максимальным с учетом особенностей вашего программного и аппаратного окружения. Мы рассмотрим большое количество примеров, так что вы сами сможете узнать, какие решения оптимальны для вашей конкретной среды. Нет никакого смысла применять все подходы одновременно, как нет и строго установленного порядка, в котором они должны быть применены. Каждый подход обладает своими плюсами и минусами, и мы практически всегда будем иметь дело с компромиссами. Если вы понимаете, какая система есть у вас в распоряжении и каковы перспективы для ее улучшения, вы можете выбрать, во что вкладывать свое время и ресурсы. Чтобы помочь вам сделать правильный выбор, в табл. 1.2 мы перечислили техники, которые будем описывать в главах этой книги, и области их применения.

Таблица 1.2. Назначение глав книги

Предмет исследования	Область использования	Глава
Получение максимальной производительности от интерпретатора Python	Интерпретатор Python	2. Извлечение максимума возможного из встроенных средств Python
Понимание внутреннего функционала Python для извлечения максимальной вычислительной мощности вашего компьютера	Интерпретатор Python	3. Конкурентность, параллелизм и асинхронная обработка
Оптимизация работы одной из базовых библиотек Python	Библиотеки Python	4. Высокопроизводительный NumPy
Исследование низкоуровневых языков программирования, когда одного Python оказывается недостаточно	Библиотеки Python	5. Реализация критически важных фрагментов кода на Cython
Понимание роли аппаратного обеспечения в деле повышения вычислительной мощности	Аппаратное обеспечение	6. Иерархия памяти, хранение и работа в сети
Извлечение максимума возможного из табличных данных	Библиотеки Python	7. Высокопроизводительный pandas и Apache Arrow
Повышение эффективности хранения данных с использованием современных библиотек Python	Библиотеки Python	8. Хранение больших данных
Применение графического процессора для повышения эффективности вычислений в Python	Аппаратное обеспечение	9. Анализ данных с использованием графического процессора (GPU)
Работа с приложениями, требующими более одного компьютера для обработки данных	Библиотеки Python и аппаратное обеспечение	10. Анализ больших данных с помощью Dask

В этой таблице много всего написано, так что позвольте мне акцентировать ваше внимание на некоторых ключевых практических аспектах. Прочитав эту книгу, вы сможете взглянуть на код Python и понять, как можно оптимизировать участки кода, связанные с использованием встроенных структур данных и алгоритмов. Вы без труда сможете дать рекомендации по замене используемых структур на более подходящие. Например, вам будет более очевиден переход со списков на множества для ускорения процедуры поиска или использование массивов без объектов вместо списков объектов. Также вы сможете взглянуть на используемый алгоритм, страдающий от недостатка быстродействия, и (1) выполнить его профилирование для выявления узких мест, а также (2) предложить приемлемый способ по его оптимизации.

Как было заявлено, в этой книге мы пройдемся по наиболее популярным библиотекам Python для обработки и анализа данных, таким как pandas и NumPy, и рассмотрим способы их оптимизации. Но все, что касается высокоуровневых библиотек, мы охватить не сможем по вполне понятным причинам. Например, мы не будем говорить про оптимизацию использования библиотеки TensorFlow, но при этом обсудим техники, которые позволят повысить эффективность лежащих в ее основе алгоритмов.

Далее рассмотрим различные источники данных и выясним, какими недостатками они обладают с точки зрения эффективной обработки и хранения. Затем выполним обработку данных таким образом, чтобы вся нужная нам информация осталась, а шаблоны доступа к ней стали гораздо более оптимальными. Наконец, мы познакомимся с фреймворком Dask, позволяющим разрабатывать параллельные решения, способные масштабироваться от одной машины до огромного кластера из локальных или облачных серверов.

Это не столько книга рецептов, сколько введение в способы мышления относительно процесса оптимизации и методов исследования возможных улучшений. Подходы, рассматриваемые в книге, по большей части должны успешно выдерживать любые изменения в отношении аппаратного и программного обеспечения, сетевого окружения и даже самих данных. При этом техники, о которых мы будем говорить, будут подходить далеко не для всех ситуаций, и чтение этой книги от корки до корки позволит вам полноценно оценить все нюансы вашего конкретного случая и выбрать наиболее подходящий метод.

ПРИМЕЧАНИЕ. Настройка программного обеспечения. Перед тем как продолжить читать книгу, ознакомьтесь с приложением А, в котором находится инструкция по установке всего необходимого программного обеспечения для запуска кода из сопроводительных материалов. Сами материалы можно найти на странице книги или на GitHub по адресу <https://github.com/tiangantao/python-performance>.

Заключение

- Старое доброе клише гласит: данных много, и нам необходимо повышать эффективность их обработки, чтобы сохранить возможность воспользоваться наиболее ценными сведениями.
- Чем выше сложность алгоритма, тем выше требования к вычислительным мощностям для его выполнения, и мы стараемся сделать все возможное для смягчения этого эффекта.

- Вычислительная архитектура обладает высоким разнообразием, и с появлением облачных ресурсов этот эффект стал еще более заметным. Кроме того, современные графические процессоры предлагают свои вычислительные мощности для обработки данных, но их архитектура кардинально отличается от архитектуры центральных процессоров, что необходимо учитывать при построении решений.
- Python – прекрасный язык для анализа и обработки данных, существующий в рамках своей экосистемы с большим разнообразием библиотек и фреймворков. Однако в своем базовом виде этот язык не отличается высоким быстродействием. Мы сделаем все, чтобы нивелировать эту особенность Python и использовать для обработки данных самые эффективные алгоритмы.
- В этой книге мы будем обсуждать достаточно сложные проблемы, но все они решаемы. Главная цель книги состоит в том, чтобы познакомить вас со всем богатством альтернативных решений и научить выбирать из имеющихся вариантов наиболее подходящий с учетом особенностей вашей ситуации.

2

Извлечение максимума возможного из встроенных средств Python

В этой главе мы обсудим следующие темы:

- профилирование кода с целью поиска узких мест;
- повышение эффективности работы встроенных структур данных Python;
- проблемы с эффективностью при выделении памяти для типовых структур данных;
- использование ленивых вычислений при обработке больших объемов данных.

Существует немало инструментов и библиотек для написания более эффективного кода на языке Python. Но, перед тем как погрузиться во внешние средства оптимизации, давайте внимательнее присмотримся к имеющимся в базовом Python возможностям для повышения быстродействия кода как в отношении операций ввода-вывода, так и в плане скорости вычислений. На самом деле большинство, но, увы, не все, проблемы с быстродействием в Python

можно решить, если найти правильный подход к возможностям и ограничениям языка.

С целью демонстрации решения по оптимизации кода в базовом Python мы обратимся к гипотетической, хотя и вполне реалистичной задаче. Представьте, что вы инженер данных, и вам поручено подготовить данные для анализа климатических данных. Источником для вас будет база данных *Integrated Surface Database* (ISD) на сайте *Национального управления океанических и атмосферных исследований США* (National Oceanic and Atmospheric Administration – NOAA), располагающаяся по адресу <https://www.ncei.noaa.gov/products/land-based-station/integrated-surface-database>. При этом у вас достаточно сжатые сроки, а в своей работе вы будете ограничены по большей степени базовыми средствами Python. Что касается расширения вычислительных возможностей, на это можете даже не рассчитывать из-за ограниченного бюджета проекта. Данные начнут поступать через месяц, и вы планируете использовать это время для оптимизации своего кода. Ваша задача состоит в поиске узких мест в коде и их устранении.

Первое, что вам необходимо сделать, – это выполнить *профилирование* (profile) существующего кода, занимающегося загрузкой данных. Вы знаете, что код, который уже написан, не отличается быстродействием, но, перед тем как приступить к его оптимизации, вам нужно определить, где именно находятся узкие места. Профилирование – это очень важный процесс на этапе разработки кода, позволяющий вовремя выявить и исправить его слабости. Альтернативный способ, заключающийся в оптимизации кода наугад, здесь не подойдет, поскольку зачастую узкие места кода будут скрываться в совершенно непредсказуемых фрагментах.

Оптимизация кода базового языка Python – довольно очевидная вещь, но иногда именно она позволяет решить большую часть проблем с быстродействием программного кода, и поэтому вы не можете игнорировать ее важность. В этой главе мы узнаем, какие возможности Python предлагает, что называется, из коробки для разработки более эффективного кода. Начнем мы с использования нескольких инструментов для профилирования кода, помогающих при выявлении узких мест. Затем переключимся на базовые структуры данных в Python, такие как списки, множества и словари. Наша задача будет прежней – повысить эффективность программного кода, в том числе за счет оптимального выделения памяти для новых структур данных. В заключительном разделе главы мы рассмотрим так называемые ленивые (отложенные) техники программирования, зачастую позволяющие оптимизировать поток данных внутри приложения.

В этой главе мы не будем пользоваться никакими сторонними библиотеками при написании кода, но прибегнем к помощи пары

внешних инструментов в процессе его оптимизации и осуществления доступа к данным. Для визуализации данных о профилировании кода воспользуемся библиотекой `Snakeviz`, а для выполнения построчного профилирования – модулем `line_profiler`. Наконец, мы никуда не денемся от использования библиотеки `requests` для загрузки данных из интернета.

Если вы используете `Docker`, в образе по умолчанию будет установлено все, что вам нужно. Последовав инструкциям из раздела `Anaconda Python` в приложении А, вы будете готовы к запуску кода. Давайте начнем процесс профилирования с загрузки исходных файлов с метеорологических станций и изучения данных о температуре по каждой станции.

2.1. Профилирование приложений с операциями ввода-вывода и вычислениями

Для начала нам необходимо загрузить данные о работе метеорологической станции и найти минимальную зафиксированную температуру за указанный год. Данные на сайте NOAA располагаются в формате файлов CSV: один за год и отдельные файлы по станциям. К примеру, в файле по адресу <https://www.ncei.noaa.gov/data/global-hourly/access/2021/01494099999.csv> хранится вся информация о станции с идентификатором 01494099999 за 2021 год. Помимо прочих показателей, в файле есть информация о температуре воздуха и давлении, замеры по которым могут проводиться несколько раз в день.

Давайте напишем код для загрузки данных для набора станций за выбранный интервал из нескольких лет. После загрузки данных выберем информацию о минимальной температуре по каждой станции.

2.1.1. Загрузка данных и поиск минимальной температуры

Наша программа будет запускаться по-простому – из командной строки, а на вход она будет принимать список станций через запятую и интервал лет через дефис. Код, приведенный ниже, можно найти в файле `02-python/sec1-io-cpu/load.py` в сопроводительных материалах:

```
import collections
import csv
import datetime
import sys
```

```
import requests
```

```
stations = sys.argv[1].split(",")
years = [int(year) for year in sys.argv[2].split("-")]
```

```
start_year = years[0]
end_year = years[1]
```

Для простоты восприятия кода мы воспользуемся библиотекой `requests` для получения файлов. Ниже показан код для загрузки данных с сервера:

```
TEMPLATE_URL = "https://www.ncel.noaa.gov/data/global-hourly/access/{year}/\n➡ {station}.csv"\nTEMPLATE_FILE = "station_{station}_{year}.csv"
```

```
def download_data(station, year):\n    my_url = TEMPLATE_URL.format(station=station, year=year)\n    req = requests.get(my_url)\n    if req.status_code != 200:\n        return # не найден\n    w = open(TEMPLATE_FILE.format(station=station, year=year), "wt")\n    w.write(req.text)\n    w.close()
```

Библиотека `requests` облегчает доступ к веб-контенту

```
def download_all_data(stations, start_year, end_year):\n    for station in stations:\n        for year in range(start_year, end_year + 1):\n            download_data(station, year)
```

Этот код записывает на диск файлы для всех запрашиваемых станций по всем годам из заданного интервала. Теперь давайте извлечем все значения температуры из одного файла, как показано ниже:

```
def get_file_temperatures(file_name):\n    with open(file_name, "rt") as f:\n        reader = csv.reader(f)\n        header = next(reader)\n        for row in reader:\n            station = row[header.index("STATION")]\n            # date = datetime.datetime.fromisoformat(row[header.index('DATE')])\n            tmp = row[header.index("TMP")]\n            temperature, status = tmp.split(",")\n            if status != "1":\n                continue\n            temperature = int(temperature) / 10\n            yield temperature
```

Игнорируем строки, для которых данные недоступны

Формат поля с температурой включает в себя подполе со статусом информации

Теперь извлечем все показания температуры и найдем минимумы:

```
def get_all_temperatures(stations, start_year, end_year):\n    temperatures = collections.defaultdict(list)\n    for station in stations:\n        for year in range(start_year, end_year + 1):\n            for temperature in get_file_temperatures(\n➡ TEMPLATE_FILE.format(station=station, year=year)):\n                temperatures[station].append(temperature)\n    return temperatures
```

```
def get_min_temperatures(all_temperatures):
    return {station: min(temperatures) for station, temperatures in
    ➤ all_temperatures.items()}
```

Соединим все вместе: загрузим данные, получим все показания температур, вычислим минимум для каждой станции и выведем результаты:

```
download_all_data(stations, start_year, end_year)
all_temperatures = get_all_temperatures(stations, start_year, end_year)
min_temperatures = get_min_temperatures(all_temperatures)
print(min_temperatures)
```

Для того чтобы проанализировать данные по станциям с идентификаторами 01044099999 и 02293099999 за 2021 год, необходимо запустить код следующим образом:

```
python load.py 01044099999,02293099999 2021-2021
```

Вывод должен быть следующим:

```
{'01044099999': -10.0, '02293099999': -27.6}
```

Ну а теперь начинается самое интересное. Нам необходимо в постоянном режиме загружать большое количество информации по множеству станций за разные годы, и для этого нужно максимально ускорить работу кода, насколько это возможно. Первым шагом в деле повышения эффективности кода является его *профилирование* (profiling) с целью поиска узких мест. Воспользуемся встроенными средствами профилирования в Python.

2.1.2. Встроенный в Python модуль профилирования

На пути повышения производительности кода первым делом нам нужно определиться с тем, что именно мешает его быстрому выполнению. Для начала узнаем время, необходимое для выполнения каждой функции, запустив наш код посредством модуля *cProfile*. Этот встроенный в Python модуль позволяет извлечь профилировочную информацию. Особо отметим, что вам необходимо использовать именно модуль *cProfile*, а не *profile*, поскольку последний работает на порядок медленнее и может применяться, например, при создании собственных инструментов профилирования.

Запустить процесс профилирования можно следующим образом:

```
python -m cProfile -s cumulative load.py 01044099999,02293099999 2021-
2021 > profile.txt
```

Если помните, флаг *-m* позволяет запустить модуль в Python, так что в данном случае мы выполняем именно модуль *cProfile*. Это рекомендованный модуль в Python для сбора информации о профилировании. В данном случае мы запрашиваем статистику, упорядоченную по накопительному времени выполнения функций. Простей-

ший способ использования этого модуля состоит в передаче ему нашего скрипта. Итог выполнения профилирования показан ниже:

375402 function calls (370670 primitive calls) in 3.061 seconds

Ordered by: cumulative time

Основная итоговая информация приведена в первой строке: количество вызовов функций и общее время выполнения

nccalls	totttime	percall	cumtime	percall	filename:lineno(function)
158/1	0.000	0.000	3.061	3.061	{built-in method builtins.exec}
1	0.000	0.000	3.061	3.061	load.py:1(<module>)
1	0.001	0.001	2.768	2.768	load.py:27(download_all_data)
2	0.001	0.000	2.766	1.383	load.py:17(download_data)
2	0.000	0.000	2.714	1.357	api.py:64(get)
2	0.000	0.000	2.714	1.357	api.py:16(request)
2	0.000	0.000	2.710	1.355	sessions.py:470(request)
2	0.000	0.000	2.704	1.352	sessions.py:626(send)
3015	0.017	0.000	1.857	0.001	socket.py:690(readinto)
3015	0.017	0.000	1.829	0.001	ssl.py:1230(recv_into)
[...]					
1	0.000	0.000	0.000	0.000	load.py:58(get_min_temperatures)

Вычислительные затраты нашего кода (а вычисления происходят в функции `get_min_temperatures`) пренебрежимо малы

Вывод, как мы и просили, был упорядочен в соответствии с накопительным временем выполнения функций. Также мы видим информацию о количестве вызовов функций. К примеру, функция `download_all_data` вызывается всего один раз, загружая все нужные нам данные, но ее накопительное время вызова очень близко к общему времени выполнения скрипта. В выводе вы видите две колонки с именем `percall`. В первой выводится время выполнения функции, не считая времени вложенных вызовов. Во второй эти вызовы включаются. В случае с функцией `download_all_data` видно, что большая часть времени расходуется как раз на внутренние вызовы функций.

Если в вашем скрипте, как в рассматриваемом случае, задействуется большое количество *операций ввода-вывода* (I/O), именно они могут стать камнем преткновения в отношении общего времени выполнения. У нас в коде присутствуют как сетевые операции ввода-вывода (загрузка данных с сайта NOAA), так и дисковые (запись файлов на диск). Сетевые операции могут очень сильно варьироваться по скорости даже между отдельными вызовами, поскольку они зависят от множества факторов. А раз так, давайте постараемся минимизировать нагрузку на сеть в нашем приложении.

2.1.3. Использование локального кеша для снижения сетевой нагрузки

Для уменьшения количества сетевых соединений мы будем повторно использовать данные из файла, загруженного ранее. Это будет

наш локальный кеш. Оставим наш код как есть, за исключением функции `download_all_data` (обновленный скрипт можно найти по адресу `02-python/sec1-io-cpu/load_cache.py`):

```
import os
def download_all_data(stations, start_year, end_year):
    for station in stations:
        for year in range(start_year, end_year + 1):
            if not os.path.exists(TEMPLATE_FILE.format(
                ➔ station=station, year=year)):
                download_data(station, year)
```

Проверяем существование файла и загружаем новый файл только в случае его отсутствия

Первый запуск обновленного кода займет примерно столько же времени, сколько и раньше, зато повторный запуск не потребует повторной загрузки файлов. В моем случае время выполнения кода между запусками снизилось с 2,8 до 0,26 с – более чем на порядок. Помните, что время загрузки файлов в вашем приложении будет напрямую зависеть от скорости текущего сетевого соединения и варьироваться вместе с ней. Еще одна причина для использования локального кеша – предсказуемое время выполнения:

```
python -m cProfile -s cumulative load_cache.py 01044099999,02293099999
➔ 2021-2021 > profile_cache.txt
```

На этот раз вывод окажется совершенно иным, что показано ниже:

```
299938 function calls (295246 primitive calls) in 0.260 seconds
```

```
Ordered by: cumulative time
```

```
ncalls tottime percall cumtime percall filename:lineno(function)
156/1 0.000 0.000 0.260 0.260 {built-in method builtins.exec}
  1 0.000 0.000 0.260 0.260 load_cache.py:1(<module>)
  1 0.008 0.008 0.166 0.166 load_cache.py:51(
➔ get_all_temperatures)
33650 0.137 0.000 0.156 0.000 load_cache.py:36(
➔ get_file_temperatures)
[...]
  1 0.000 0.000 0.001 0.001 load_cache.py:60(
➔ get_min_temperatures)
```

И хотя общее время выполнения кода снизилось на порядок, операции ввода-вывода по-прежнему возглавляют список. Правда, на этот раз речь идет не о сетевых операциях, а об операциях доступа к диску. Причина этого в том, что мы в своем коде не производим никаких сложных вычислений.

ПРЕДУПРЕЖДЕНИЕ. Локальный кеш может помочь повысить быстродействие приложения в разы, что мы видели на этом примере. Но в то же время наличие кеша зачастую является проблемой и источником ошибок в данных. В нашем случае мы имеем дело с архивными данными, которые не меняются задним чис-

лом с течением времени, но это не всегда и не у всех так. Иногда данные могут меняться, и в этом случае вы должны отслеживать эти изменения, чтобы вовремя подгрузить обновленные сведения. В других главах этой книги мы также иногда будем касаться вопросов кеширования данных.

Теперь давайте рассмотрим ситуацию, в которой ограничивающим фактором являются вычислительные ресурсы центрального процессора.

2.2. Профилирование кода для обнаружения проблем с производительностью

Перейдем к случаю, в котором узким местом является вычислительная мощность центрального процессора. Мы возьмем все метеорологические станции из базы данных NOAA и вычислим расстояниями между ними. Сложность этого алгоритма составляет n^2 .

В репозитории располагается файл 02-python/sec2-cpu/locations.csv со всеми географическими координатами станций, а сам код, приведенный ниже, можно найти в файле 02-python/sec2-cpu/distance_cache.py:

```
import csv
import math

def get_locations():
    with open("locations.csv", "rt") as f:
        reader = csv.reader(f)
        header = next(reader)
        for row in reader:
            station = row[header.index("STATION")]
            lat = float(row[header.index("LATITUDE")])
            lon = float(row[header.index("LONGITUDE")])
            yield station, (lat, lon)

def get_distance(p1, p2):
    lat1, lon1 = p1
    lat2, lon2 = p2

    lat_dist = math.radians(lat2 - lat1)
    lon_dist = math.radians(lon2 - lon1)
    a=(
        math.sin(lat_dist / 2) * math.sin(lat_dist / 2) +
        math.cos(math.radians(lat1)) * math.cos(math.radians(lat2)) *
        math.sin(lon_dist / 2) * math.sin(lon_dist / 2)
    )
    c = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))
    earth_radius = 6371
    dist = earth_radius * c

    return dist
```

Здесь вычисляется расстояние между двумя станциями

```
def get_distances(stations, locations):
    distances = {}
    for first_i in range(len(stations) - 1):
        first_station = stations[first_i]
        first_location = locations[first_station]
        for second_i in range(first_i, len(stations)):
            second_station = stations[second_i]
            second_location = locations[second_station]
            distances[(first_station, second_station)] = get_distance(
                first_location, second_location)
    return distances

locations = {station: (lat, lon) for station, (lat, lon) in get_
locations()}
stations = sorted(locations.keys())
distances = get_distances(stations, locations)
```

Поскольку мы перебираем все комбинации станций, сложность алгоритма будет составлять n^2

Этот код будет выполняться очень долго. Кроме того, для его выполнения будет выделено чересчур много памяти. Если у вас на компьютере не так много памяти, ограничьте количество обрабатываемых станций. А пока давайте воспользуемся инфраструктурой Python для выполнения профилирования и обнаружения узких мест.

2.2.1. Визуализация профилировочной информации

Сейчас мы снова будем использовать инструменты для профилирования кода и поиска узких мест, но на этот раз прибегнем к помощи внешнего визуального инструмента *SnakeViz*, располагающегося по адресу <https://jiffyclub.github.io/snakeviz>.

Для начала сохраним информацию о ходе профилирования, выполнив следующую инструкцию:

```
python -m cProfile -o distance_cache.prof distance_cache.py
```

Параметр `-o` указывает на то, что профилировочная информация должна быть сохранена в указанном файле.

ПРИМЕЧАНИЕ. Python также предлагает модуль *pstats* для анализа профилировочной информации, сохраненной на диске. Вы можете воспользоваться инструкцией `python -m pstats distance_cache.prof`, после чего откроется интерфейс командной строки с анализом производительности кода. Больше об этом модуле вы можете узнать из документации Python или в разделе профилирования главы 5.

Мы для анализа профилировочной информации будем использовать веб-инструмент визуализации под названием *SnakeViz*. Для этого достаточно написать инструкцию `snakeviz distance_cache.prof`. В результате откроется интерактивное окно браузера, показанное на рис. 2.1.

Знакомство с интерфейсом SnakeViz

Вы можете самостоятельно поработать с интерфейсом инструмента SnakeViz, чтобы получше с ним познакомиться. К примеру, можно сменить стиль Icicle на Sunburst – более привлекательный, но с меньшим количеством информации. Также допустимо упорядочивать таблицу, выведенную в нижней части окна, и менять параметры визуализации в левой части. Вы можете щелкать по цветным блокам и возвращаться к исходному виду путем нажатия на кнопку **Call Stack** и выбора нулевой записи.

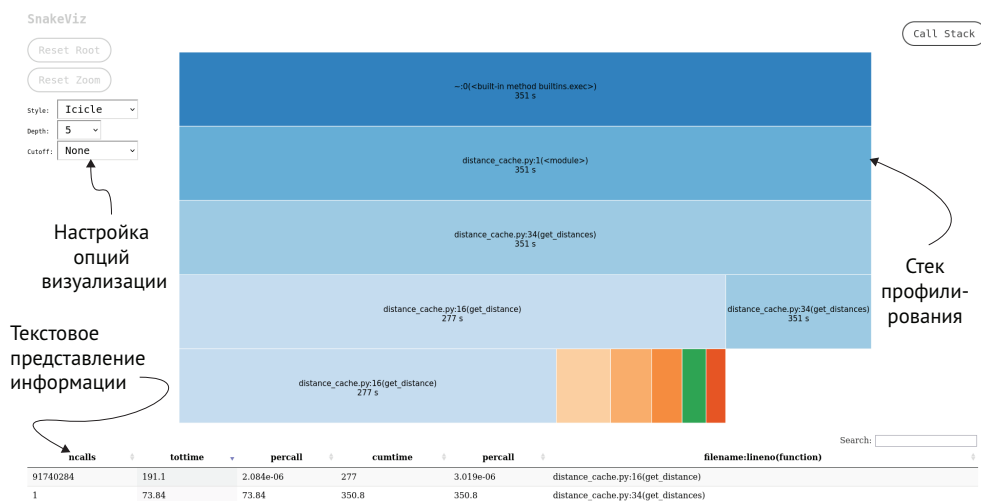


Рис. 2.1. Использование SnakeViz для анализа профилировочной информации

Большую часть времени, как и ожидалось, заняло выполнение функции `get_distance`, но какие именно вычисления стали узким местом? Мы можем лишь предположить, что все дело в математических операциях, но встроенные средства профилирования Python не позволяют углубиться в детали. Максимум, что мы можем получить, – это обобщенную информацию по каждой тригонометрической функции. Да, мы провели какое-то время в функции `math.sin`, но мы вызывали ее несколько раз, и какой из них отнял больше времени? Похоже, придется воспользоваться профайлером с возможностью анализировать конкретные строки кода.

2.2.2. Профилирование с детализацией до строк

Встроенные инструменты профилирования кода позволили нам взглянуть на картину в целом, но не дали возможности опуститься до максимального уровня детализации и проанализировать конкретные строки кода на предмет скорости выполнения.

Для понимания того, как распределяется нагрузка внутри функции `get_distance`, мы воспользуемся пакетом `line_profiler`, доступным по адресу https://github.com/pyutils/line_profiler. Он очень прост в применении – вам будет достаточно добавить декоратор к вызову функции `get_distance`, как показано ниже:

```
@profile
def get_distance(p1, p2):
```

Заметьте, что мы не импортировали никаких пакетов. Все дело в том, что далее мы будем использовать скрипт `kernprof` из пакета `line_profiler`, который обо всем позаботится. Давайте запустим профайлер по строкам для нашего кода следующим образом:

```
kernprof -l lprofile_distance_cache.py
```

Будьте готовы к тому, что во время профилирования наш код замедлится в разы. Дайте ему поработать около минуты, после чего остановите выполнение программы (если этого не сделать, `kernprof` может работать долгие часы). После прерывания процесса у вас будет в распоряжении записанная профилировочная информация. Результаты вы сможете посмотреть с помощью следующей инструкции:

```
python -m line_profiler lprofile_distance_cache.py.lprof
```

Если вы взглянете на листинг 2.1 с выводом, то увидите, что многие вызовы длятся довольно долго. Вероятно, придется оптимизировать наш код. В данный момент мы говорим лишь о процессе профилирования, так что на этом остановимся, но далее в этой главе мы займемся улучшением этого кода. Если вам интересно, как можно оптимизировать приведенный здесь скрипт, обратитесь к главе 6, посвященной расширению Cython, или к приложению Б, в котором рассказывается о компиляторе Numba, поскольку с их помощью можно серьезно ускорить работу этого фрагмента кода.

Листинг 2.1. Вывод пакета `line_profiler` для нашего кода

Timer unit: 1e-06 s					
Total time: 619.401 s					
File: lprofile_distance_cache.py					
Function: get_distance at line 16					
Line #	Hits	Time	Per Hit	% Time	Line Contents
=====					
16					@profile
17					def get_distance(p1, p2):
18	84753141	36675975.0	0.4	5.9	lat1, lon1 = p1
19	84753141	35140326.0	0.4	5.7	lat2, lon2 = p2

Информация, которую мы получаем для каждой строки профилирования. Здесь вы видите количество вызовов каждой строки, суммарное время выполнения строки, время на каждый вызов и процент времени, проведенного при выполнении этой строки

Общее время выполнения кода

```

20
21  84753141  39451843.0      0.5      6.4      lat_dist = math.
    ➔ radians(lat2 - lat1)
22  84753141  38480853.0      0.5      6.2      lon_dist = math.
    ➔ radians(lon2 - lon1)
23  84753141  28281163.0      0.3      4.6      a = (
24  169506282  84658529.0      0.5      13.7      math.sin(
    ➔ lat_dist / 2) *
    ➔ math.sin(lat_dist / 2) +
25  254259423  118542280.0      0.5      19.1      math.cos(
    ➔ math.radians(lat1)) *
    ➔ math.cos(math.radians(
    ➔ lat2)) *
26  169506282  81240276.0      0.5      13.1      math.sin(
    ➔ lon_dist / 2) *
    ➔ math.sin(lon_dist / 2)
27                                          )
28  84753141  65457056.0      0.8      10.6      c = 2 * math.
    ➔ atan2(math.sqrt(a),
    ➔ math.sqrt(1 - a))
29  84753141  29816074.0      0.4      4.8      earth_radius =
    ➔ 6371
30  84753141  33769542.0      0.4      5.5      dist =
    ➔ earth_radius * c
31
32  84753141  27886650.0      0.3      4.5      return dist

```

Полагаю, вывод этого профайлера интуитивно более понятен по сравнению с выводом встроенных средств профилирования.

2.2.3. Профилирование кода: выводы

Как видите, использование встроенных средств профилирования кода отлично подходит в качестве первой неотложной меры по устранению проблем с быстродействием. К тому же эти средства работают ощутимо быстрее по сравнению с профилированием на уровне строк кода. В то же время детализация до строк дает больше информации, в первую очередь потому, что встроенные инструменты не позволяют внедряться в выполнение функций. Вместо этого они лишь рассчитывают нарастающие итоги по ним и показывают, какое время было потрачено на внутренние вызовы. В отдельных случаях мы можем узнать, принадлежит ли внутренний вызов другой функции, но чаще всего это невозможно. И общая стратегия профилирования кода должна все это учитывать.

Мы предлагаем подходить к процессу профилирования комплексно. Сначала необходимо воспользоваться встроенным в Python модулем `cProfile` – он работает достаточно быстро и при этом дает достаточное количество базовой информации. Если вам этого не хватает, можете прибегнуть к помощи профилирования с детализацией до строк. Здесь вам уже придется пожертвовать

скоростью ради информативности вывода. Помните, что пока мы просто занимались тем, что искали узкие места в коде. Оптимизацией кода мы займемся в следующих главах. К сожалению, иногда будет недостаточно просто поменять фрагменты кода местами, а придется полностью перерабатывать концепцию приложений, и об этом мы также поговорим, когда придет время.

Другие инструменты профилирования

При профилировании кода вы также можете воспользоваться другими инструментами, и раздел, посвященный этому процессу, был бы неполным без упоминания популярного модуля *timeit*. Возможно, это самый распространенный способ простого профилирования кода для новичков, и в интернете вы найдете массу примеров его использования. Проще всего воспользоваться модулем *timeit* можно при работе с IPython или Jupyter Notebook. Просто добавьте инструкцию `%timeit` к строке кода, скорость выполнения которой хотите замерить, как показано ниже:

```
In [1]: %timeit list(range(1000000))
27.4 ms ± 72.5 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
In [2]: %timeit range(1000000)
189 ns ± 22.6 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

В результатах вывода вы увидите время, измеренное на нескольких вызовах профилируемой функции. Эта инструкция сама решает, сколько раз прогонять код, и оповещает вас о своем решении. В показанном примере мы можем наблюдать разницу в скорости выполнения инструкций `range(1000000)` и `list(range(1000000))`. В данном случае ленивая версия функции `range` показала результат, на два порядка превышающий по скорости вариант с преобразованием диапазона в список.

Больше информации о модуле *timeit* вы сможете найти в документации, но в большинстве случаев инструкции `%timeit` в IPython будет вполне достаточно для использования базового функционала. Что касается нас, то на протяжении большей части книги мы будем пользоваться стандартным интерпретатором Python. Подробно о магических командах вроде `%timeit` вы можете почитать по адресу <https://ipython.readthedocs.io/en/stable/interactive/magics.html>.

Теперь, когда мы познакомились с принципами и инструментами профилирования кода, давайте перейдем к другой, не менее важной теме, касающейся оптимизации использования структур данных в Python.

2.3. Оптимизация работы базовых структур данных Python: списки, множества и словари

Итак, приступим к выявлению узких мест при использовании базовых структур данных Python и перепишем несколько фрагментов

кода с целью их оптимизации. Для демонстрации мы продолжим работать с данными о температурах с сайта NOAA, но на этот раз попытаемся определить, встречалось ли указанное значение температуры на станции в заданный временной период.

Для загрузки данных мы воспользуемся тем же кодом, что и раньше, и дополним его новыми инструкциями. Исходный код для этого раздела содержится в файле 02-python/sec3-basic-ds/exists_temperature.py. В данном случае нас будет интересовать информация по станции с идентификатором 01044099999 за период с 2005 по 2021 год:

```
stations = ['01044099999']
start_year = 2005
end_year = 2021
download_all_data(stations, start_year, end_year)
all_temperatures = get_all_temperatures(stations, start_year, end_year)
first_all_temperatures = all_temperatures[stations[0]]
```

В переменной `first_all_temperatures` содержится *список* (list) всех температур, зафиксированных на выбранной метеорологической станции. Базовую статистику можно получить при помощи инструкции `print(len(first_all_temperatures), max(first_all_temperatures), min(first_all_temperatures))`. Судя по выводу, мы имеем дело с 141 082 показаниями с максимальным значением температуры, равным 27,0 °C, и минимальным –16,0 °C.

2.3.1. Быстродействие поиска в списке

Проверить, находится ли искомое значение температуры в списке, можно при помощи оператора `in`. Давайте посмотрим, сколько времени займет поиск значения –10,7 в нашем списке, выполнив следующую инструкцию:

```
%timeit (-10.7 in first_all_temperatures)
```

На моем компьютере вывод оказался таким, как показано ниже:

```
313 µs ± 6.39 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

Теперь давайте выполним поиск значения, которого заведомо нет в списке:

```
%timeit (-100 in first_all_temperatures))
```

Результат у меня получился следующий:

```
2.87 ms ± 20.3 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Это примерно на один порядок медленнее по сравнению с поиском значения –10,7.

Почему мы получили такое ухудшение скорости поиска во втором случае? Ответ очевиден – потому что оператору `in` пришлось проходить в поисках указанной температуры весь список от нача-

ла до конца. В худшем случае (с которым мы и имеем дело) поиск будет охватывать весь список целиком. Для небольших списков такая процедура поиска от начала и до конца списка не будет губительно сказываться на производительности. Но с ростом количества элементов в списке общее время поиска в нем будет стремительно увеличиваться.

Пока нам не с чем сравнивать, но все мы понимаем, что цифры в районе нескольких миллисекунд, и даже микросекунд, мало кого могут впечатлить. Очевидно, что существуют способы существенно увеличить скорость поиска.

2.3.2. Поиск с использованием множеств

Давайте посмотрим, сможем ли мы добиться улучшений при переходе от списков к *множествам* (set). Преобразуем наш *упорядоченный список* (ordered list) в множество и попробуем выполнить поиск:

```
set_first_all_temperatures = set(first_all_temperatures)
```

```
%timeit (-10.7 in set_first_all_temperatures)
```

```
%timeit (-100 in set_first_all_temperatures)
```

Результаты получились следующие:

62.1 ns \pm 3.27 ns per loop (mean \pm std. dev. of 7 runs, 10,000,000 loops each)

26.6 ns \pm 0.115 ns per loop (mean \pm std. dev. of 7 runs, 10,000,000 loops each)

Как понимаете, это на несколько порядков быстрее по сравнению с предыдущим поиском! Что же позволило достичь такого прогресса? Есть две основные причины: одна связана с размером множества, а вторая – с вычислительной сложностью используемых алгоритмов. О сложности алгоритмов мы поговорим в следующем разделе, а здесь коснемся размера множества.

Вы помните, что в нашем исходном списке было 141 082 значения. Но в случае с множеством все повторяющиеся значения схлопываются вместе, а в нашем списке дубликатов было предостаточно. Таким образом, если взглянуть на размер итогового множества с помощью инструкции `print(len(set_first_all_temperatures))`, вы увидите, что в нем находится всего 400 значений, что в 350 раз меньше размера исходного списка. Неудивительно, что так увеличилась скорость поиска.

Получается, всякий раз, когда в вашем списке присутствуют повторяющиеся элементы, вы можете преобразовать его в множество, в котором дубликатов не может быть по определению, что позволит в разы ускорить поиск. Но есть и гораздо более важное отличие между реализациями списков и множеств в языке Python.

2.3.3. Вычислительная сложность списков, множеств и словарей в Python

Существенное ускорение процедуры поиска заданного значения, которое мы наблюдали в предыдущем разделе, по большей части можно объяснить снижением размера исходного перечня значений. В связи с этим возникает логичный вопрос: а что было бы, если бы в исходном списке не было дублирующихся значений? Давайте узнаем. Для этого мы можем воспользоваться функцией *range*, которая гарантирует отсутствие дублей в источнике:

```
a_list_range = list(range(100000))
a_set_range = set(a_list_range)
```

```
%timeit 50000 in a_list_range
%timeit 50000 in a_set_range
%timeit 500000 in a_list_range
%timeit 500000 in a_set_range
```

Итак, мы получили два одинаковых диапазона из 100 000 значений (от 0 до 99 999), реализованных в виде списка (*a_list_range*) и множества (*a_set_range*). В обеих структурах данных мы выполнили поиск значений 50 000 (присутствует в нашем диапазоне) и 500 000 (отсутствует в диапазоне). Полученные результаты приведены ниже:

```
455 µs ± 2.68 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
40.1 ns ± 0.115 ns per loop (mean ± std. dev. of 7 runs,
⇒ 10,000,000 loops each)
936 µs ± 9.37 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
28.1 ns ± 0.107 ns per loop (mean ± std. dev. of 7 runs,
⇒ 10,000,000 loops each)
```

Как видите, поиск в множестве по-прежнему вне конкуренции. Все дело в том, что в языке Python (а точнее, в реализации CPython) множества реализованы в виде *хеши* (hash). В результате скорость поиска в множестве приравнивается к скорости поиска в хеше. Реализации *хеш-функций* (hash functions) бывают самыми разными, и при их создании разработчики сталкиваются с многими проблемами. Но если сравнивать поиск в списке и поиск в множестве, можно сказать, что при работе с множеством результат будет достигаться за одно и то же время вне зависимости от того, сколько элементов в нем будет содержаться: 10 или 10 млн. Это не до конца верно, но для лучшего понимания того, почему поиск в множестве всегда будет выполняться быстрее, чем в списке, вполне годится.

Также полезно помнить о том, что множества обычно реализуются в виде *словарей* (dictionary) без значений. Этим объясняется, почему поиск по ключам словаря будет выполняться за время, со-

поставимое с поиском в множестве. Но не стоит воспринимать множества и словари как панацею от всех болезней. К примеру, если вам необходимо будет выполнить поиск диапазона, упорядоченный список подойдет гораздо лучше. В нем вы можете найти нижнюю границу диапазона, а затем двигаться вверх до обнаружения первого значения, выходящего за верхнюю границу, что и будет сигналом к остановке поиска. В случае с множеством или словарем вам придется проверять на вхождение в искомый диапазон каждый элемент. Таким образом, если вы точно знаете, какое значение ищете, словарь подойдет идеально. А при поиске интервала этот выбор будет не самым оптимальным – гораздо лучше с этим справится упорядоченный список и *алгоритм двоичного поиска* (bisection algorithm).

Списки в языке Python используются повсеместно, несмотря на то что иногда другие структуры данных справились бы гораздо лучше. И все же не стоит умалять заслуг списка – одной из базовых структур в Python, обладающей весомым перечнем преимуществ. Мы лишь подчеркиваем, что для каждой задачи необходимо выбирать наиболее подходящий способ и средства ее решения.

СОВЕТ. Будьте очень осторожны при использовании оператора `in` со списками, которые могут разрастаться до больших размеров. Разбирая разные исходные коды на Python, понимаешь, что этот оператор для поиска элемента в списке использует абсолютно все (то же касается и метода *index* в списках). Это не проблема, если список небольшой и никогда таковым не станет, но с объемными списками работать таким образом может быть весьма неэффективно.

На самом деле на практике эта проблема зачастую проявляется из-за разницы в наполнении данными проекта на стадии разработки и готового рабочего проекта. В результате разработчик тестирует программный код на небольших объемах данных, и с ними все работает весело и задорно. Но при развертывании проекта в рабочей среде данные и их объем могут значительно измениться, что приведет к серьезным проблемам с быстродействием.

В этой связи было бы полезно время от времени тестировать систему на стадии разработки на данных, приближенных к реальным. При этом данную проверку можно выполнять на разных этапах: от модульного тестирования до сквозного. И не стоит использовать этот аспект в качестве аргумента против использования оператора `in` со списками. Просто ко всему нужно подходить с умом и учитывать возможные различия между тестовыми данными и рабочими.

Кстати, для выполнения операций, связанных с поиском, существует и более подходящая структура данных по сравнению со списками, множествами и словарями. Это *деревья* (trees). Но в этой

главе мы говорим о структурах, присутствующих в базовом Python, а деревьев там нет.

Выбор наиболее подходящих алгоритмов и структур данных для решения разных задач – это тема для отдельных книг и предмет длительных дискуссий на курсах по информатике. Но смысл не в том, чтобы отчаянно дискутировать, а в том, чтобы научиться искать оптимальные решения для каждого отдельного случая. В этой книге мы сосредоточимся на Python, но есть и другие книги, целиком посвященные выбору алгоритмов и структур данных. Одна из них так и называется «Структуры данных и алгоритмы в Python» (Data Structures and Algorithms in Python) авторов Майкла Т. Гудрича (Michael T. Goodrich), Роберто Таммассии (Roberto Tamassia) и Майкла Х. Голдвассера (Michael H. Goldwasser), она была выпущена издательством Wiley в 2013 году.

Еще один полезный ресурс располагается по адресу <https://wiki.python.org/moin/TimeComplexity>. Здесь собраны в одном месте *временные сложности* (time complexity) для всех операций с использующимися в Python встроенными структурами данных.

До сих пор в этой главе мы говорили о быстродействии алгоритмов. Но это не единственный аспект, на который нужно обращать внимание при работе с большими наборами данных. И в следующем разделе мы обратимся еще к одному важному фактору, касающемуся выделения памяти.

2.4. В поисках избыточного выделения памяти

Процесс выделения *памяти* (memory) во время работы приложения может стать ключевым фактором при оценке его производительности, и дело не только в том, что памяти может не хватить. Эффективное распределение ресурсов памяти может позволить запускать больше процессов, работающих параллельно на одной машине. И, что еще более важно, разумное использование памяти может позволить использовать алгоритмы, работающие с данными, помещающимися в памяти.

Давайте вернемся к уже знакомому нам сценарию с базой данных с сайта NOAA и посмотрим, как можно уменьшить количество обращений к диску. Для этого мы начнем с изучения содержимого файлов с данными. Наша цель – загрузить несколько файлов и посмотреть на статистику распределения символов.

```
def download_all_data(stations, start_year, end_year):
    for station in stations:
        for year in range(start_year, end_year + 1):
            if not os.path.exists(TEMPLATE_FILE.format(
                ➡ station=station, year=year)):
                download_data(station, year)
```

```
def get_all_files(stations, start_year, end_year):
    all_files = collections.defaultdict(list)
    for station in stations:
        for year in range(start_year, end_year + 1):
            f = open(TEMPLATE_FILE.format(station=station, year=year), 'rb')
            content = list(f.read())
            all_files[station].append(content)
            f.close()
    return all_files

stations = ['01044099999']
start_year = 2005
end_year = 2021
download_all_data(stations, start_year, end_year)
all_files = get_all_files(stations, start_year, end_year)
```

В переменной `all_files` теперь хранится словарь, в элементах которого находится содержимое всех файлов, относящихся к станции. Давайте посмотрим, что происходит с памятью.

2.4.1. По минному полю выделения памяти в Python

В Python присутствует полезная функция `getsizeof` в модуле `sys`, которая, судя по ее названию, должна возвращать объем памяти, занимаемый объектом. Мы можем узнать, сколько памяти занимает наш словарь, воспользовавшись приведенным ниже кодом:

```
print(sys.getsizeof(all_files))
print(sys.getsizeof(all_files.values()))
print(sys.getsizeof(list(all_files.values())))
```

Результат будет таким:

```
240
40
64
```

Скорее всего, вы ожидали не такого вывода от функции `getsizeof`. Размер наших файлов на диске исчисляется мегабайтами, так что цифры, не дотягивающие даже до 1 Кб, выглядят действительно подозрительно. Фактически функция `getsizeof` возвращает размер контейнеров (первый – это словарь, второй – итератор, а третий – список) *без* учета содержимого. Так что при оценке занимаемой памяти мы должны учитывать две составляющие: объем контейнера и объем его содержимого.

ПРИМЕЧАНИЕ. С реализацией функции `getsizeof` на самом деле никаких проблем нет. Просто ее назначение несколько отличается от ожиданий доверчивых разработчиков, полагающих, что функция с таким названием просто обязана выводить полный

объем занимаемой объектом и всем его содержимым памяти. Но если вы обратитесь к официальной документации, то найдете там даже рекурсивную реализацию этой функции, использование которой решает большинство проблем. Мы же используем странности поведения функции `sizeof` как затравку для разговора о том, как в реализации CPython организован процесс выделения памяти.

Давайте начнем с извлечения некоторой базовой информации о данных, собранной для нашей станции:

```
station_content = all_files[stations[0]]
print(len(station_content))
print(sys.getsizeof(station_content))
```

Вывод будет следующим:

```
17
248
```

В нашем словаре есть только один элемент, соответствующий нашей станции. Он содержит список из 17 элементов. Сам список занимает 248 байт, но помните, что сюда не включается его содержимое. Теперь давайте взглянем на размер первого элемента:

```
print(len(station_content[0]))
print(sys.getsizeof(station_content[0]))
print(type(station_content[0]))
```

В первой строке вы увидите число 1 303 981, что соответствует размеру файла. Что касается функции `sizeof`, она показала число 10 431 904, что примерно в восемь раз больше размера файла. Почему в восемь? Потому что каждый входящий элемент в этом объекте – это указатель на символ, а каждый указатель занимает 8 байт. Пока выглядит не очень. У нас есть огромная структура данных, и мы еще даже не учитывали само ее содержимое. Давайте опустимся на уровень символов:

```
print(sys.getsizeof(station_content[0][0]))
print(type(station_content[0][0]))
```

Разница колоссальная. На выводе мы увидим число 28 и тип `int`. Получается, что каждый символ, который должен занимать 1 байт, на самом деле занимает в 28 раз больше. В результате мы имеем 10 431 904 байта для хранения списка плюс $28 * 1\,303\,981 = 36\,511\,468$. В итоге получаем 46 943 372 байта. Это в 36 раз больше, чем весит исходный файл! К счастью, ситуация не столь плачевная, как кажется на первый взгляд, но мы можем

ее улучшить. Начнем с того, что Python (или, скорее, CPython) довольно грамотно работает с выделением памяти.

Надо признать, что интерпретатор CPython умеет оптимизировать процесс выделения памяти для объектов, а наши предположения о том, как на самом деле происходит распределение памяти, были весьма наивными. Давайте посчитаем объем только внутреннего содержимого, но вместо простого прохода по целочисленным значениям в нашем массиве мы будем следить за тем, чтобы одно и то же значение не учитывалось дважды. Если объект в Python используется многократно, он получает один и тот же идентификатор, который можно извлечь при помощи функции `id`. Таким образом, при обнаружении одинаковых идентификаторов объектов мы можем выделять память лишь единожды:

```
single_file_data = station_content[0]
all_ids = set()
for entry in single_file_data:
    all_ids.add(id(entry))
print(len(all_ids))
```

Функция `id` позволяет получить
уникальный идентификатор
объекта

Здесь мы собираем в одно множество уникальные идентификаторы для всех чисел, которые встречаются в нашем списке. В CPython именно так работает выделение памяти. Этот интерпретатор достаточно умен, чтобы выявить повторное использование содержимого. Если вы помните, символы ASCII представляются в виде целочисленных значений от 0 до 127, а длина нашего множества с уникальными идентификаторами встреченных в исходном списке элементов равна 46.

Что ж, можно выдохнуть – CPython оказался весьма прозрачным в деле выделения памяти. В результате для хранения наших данных нам понадобилось всего 10 431 904 байта – именно столько потребовала инфраструктура списка. Обратите внимание, что в нашем примере файл содержал лишь 46 уникальных символов, и с таким небольшим массивом данных Python справился отлично. Но не думайте, что так будет всегда, – здесь все зависит от индивидуальных данных.

Кеширование и повторное использование объектов в Python

Python делает все возможное, чтобы по максимуму повторно использовать объекты, но не стоит на это слишком полагаться. Во-первых, многое зависит от конкретной реализации языка. Интерпретатор CPython в этом отношении сильно отличается от других реализаций Python.

Кроме того, даже в случае с CPython никто не гарантирует одинакового поведения интерпретатора от версии к версии. Что работает в одной версии, может не работать в другой.

Наконец, даже в случае с фиксированной версией действия интерпретатора могут быть не столь очевидными. Давайте рассмотрим код в версии Python 3.7.3 (в других версиях он может работать иначе):

```
s1 = 'a' * 2
s2 = 'a' * 2
s=2
s3 = 'a' * s
s4 = 'a' * s
print(id(s1))
print(id(s2))
print(id(s3))
print(id(s4))
print(s1 == s4)
```

Здесь мы получим строку aa как результат двукратного повторения a

Здесь мы получим строку aa как результат s-кратного повторения a, где s = 2

Эти строки идентичны по содержанию

Результат будет таким:

```
140002256425568
140002256425568
140002256425904
140002256425960
True
```

При задании мультипликатора строки в виде переменной распределитель ресурсов не может определить, что содержимое двух строк одинаковое, даже если у них одна длина. Если такой простой пример так работает, то что говорить о более сложных случаях? Конечно, вы можете воспользоваться знаниями о том, как именно работает распределитель ресурсов, и проконтролировать используемую версию Python – в этом есть смысл. Но будьте осторожны и не слишком доверяйте своим ожиданиям.

Мы использовали представление файла на основе списка чисел. А что, если рассмотреть другие представления?

2.4.2. Выделение памяти для альтернативных представлений

Рассмотрим некоторые альтернативные представления данных в файлах. Какие-то из них покажут себя лучше, какие-то – хуже. Здесь очень важно понимать, какую стоимость придется заплатить за каждый вариант. Вместо целочисленных значений для представления исходных символов мы можем использовать строки единичной длины, как показано ниже:

```
single_file_str_list = [chr(i) for i in single_file_data]
```

Такой подход будет еще хуже того, что мы использовали ранее. Взгляните на объем памяти, занимаемый каждым символом:

```
print(sys.getsizeof(single_file_str_list[0]))
```


В случае с целочисленными значениями мы получили результат 28 байт, а здесь – целых 50! Это явный шаг назад, так что не стоит идти в этом направлении.

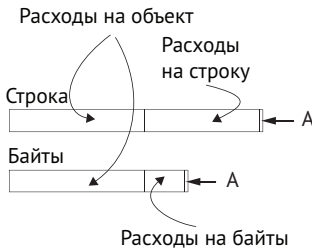


Рис. 2.2. Накладные расходы для строк и массива байтов

Объекты в Python тянут за собой довольно большие накладные расходы, и это имеет значение при работе с множеством мелких объектов. Почему маленькие числа занимают 28 байт, а один символ – 50? Дело в том, что каждый объект в Python подразумевает обязательные накладные расходы в 24 байта, к которым необходимо добавить накладные расходы для конкретного типа объекта, которые варьируются в зависимости от типа. Как мы уже видели и как показано на рис. 2.2, для строковых значений эти расходы больше, чем для массива байтов.

Внутреннее представление строк и чисел

В Python очень эффективно реализовано внутреннее представление строк, которое может меняться в зависимости от обстоятельств, а это может сбивать с толку в отношении ожиданий, связанных с выделением памяти:

```
from sys import getsizeof
getsizeof('')
getsizeof('c')
getsizeof('c' * 10000)
getsizeof('ç' * 10000)
getsizeof('ç')
getsizeof('☺')
getsizeof('☺' * 10000)
```

Вывод будет следующий:

```
49
50
10049
10073
74
80
40076
```

Как видите, пустая строка заняла 49 байт, строка с символом с – 50 байт, а повторение этого символа 10 000 раз потребовало 10 049 байт. Пока все нормально. Что касается символа ç с диакритическим знаком, на его хранение понадобилось уже 74 байта, а на его повторение 10 000 раз – 10 073 байта. Если вы немного смущены, посмотрите, сколько места требуется для хранения настоящего смущения, выраженного на лице смайлика в последних двух строках. Для одного смайлика Python выделил 80 байт, а для 10 000 смайликов – аж 40 076 байт.

В Python три строки представлены в кодировке Unicode, но есть нюанс. Внутреннее представление реализовано и оптимизировано в виде функции от строки, которую необходимо представить. За подробностями вы можете обратиться к стандарту *PEP 393 – Гибкие представления строк* (PEP 393 – Flexible String Representation). Для хранения символов из кодового набора Latin-1 (надмножество ASCII) в Python используется 1 байт (символ с с диакритическим знаком как раз входит в этот набор), а для других символов может потребоваться до 4 байт, как в случае с нашим смущенным смайликом. Все это приводит к большим сложностям при подсчете истинных размеров строк. Хранение целочисленных значений также оптимизировано. Точность является произвольной, но для *целых чисел со знаком* (signed integer), уместящихся в 30 бит, потребуется минимальный объем памяти, равный 28 байт. Нулевое значение при этом является исключением – для его хранения необходимо всего 24 байта, что, как вы помните, совпадает с минимальными накладными расходами для объектов в CPython.

Для файлов существует и более очевидное представление – вместо списка строк, состоящих из одного символа, можно использовать строку с целым файлом, как показано ниже:

```
single_file_str = ''.join(single_file_str_list)
print(sys.getsizeof(single_file_str))
```

Размер этого объекта будет равен 1 304 030 байт, что составляет из объема файла и накладных расходов на хранение строки. Хотя это простое и очевидное решение, мы еще поработаем с контейнерами для последовательностей байтов, поскольку их можно оптимизировать.

2.4.3. Использование массивов в качестве компактной альтернативы спискам

Посмотрим, насколько более эффективным с точки зрения использования памяти может быть альтернативный контейнер в виде *массива* (array). Вспомним нашу предыдущую реализацию функции `get_all_files` и немного ее изменим, как показано ниже:

```
def get_all_files_clean(stations, start_year, end_year):
    all_files = collections.defaultdict(list)
    for station in stations:
        for year in range(start_year, end_year + 1):
            f = open(TEMPLATE_FILE.format(station=station, year=year), 'rb')
            content = f.read()
            all_files[station].append(content)
            f.close()
    return all_files
```

← В исходном варианте функции было `content = list(f.read())`

Исходная строка `content = list(f.read())` здесь была преобразована в `content = f.read()`, чтобы избежать преобразования данных в список. В результате мы получили массив байтов. Давайте посмотрим на размер объекта:

```
all_files_clean = get_all_files_clean(stations, start_year, end_year)
single_file_data = all_files_clean[stations[0]][0]
print(type(single_file_data))
print(sys.getsizeof(single_file_data))
```

На выводе мы увидим, что тип данных стал `bytes`, а полный размер объекта, включая данные, – 1 304 014 байт.

Массивы имеют фиксированный размер и могут содержать только объекты одного типа. Поэтому их представление можно сделать намного более компактным: они могут храниться с накладными расходами объекта. Если помните, в случае с целочисленными значениями нам приходилось выделять по 28 байт для данных, фактически занимающих 1 байт.

Выделение памяти для списков

При создании списка Python выделяет дополнительное место в памяти для будущих вставок, так что списки на самом деле могут занимать больше места в памяти, чем вы ожидаете. Из плюсов то, что добавление элементов происходит быстро, поскольку не требуется тратить время на выделение памяти, если заранее выделенное место не закончилось. А из минусов – дополнительный расход памяти. Как правило, такое поведение интерпретатора не влияет на быстродействие приложения, если вы не используете большое количество мелких списков. Здесь вступает в силу закон присутствия множества крошечных объектов, который применительно к спискам может быть еще более губительным. Во всех остальных случаях вы не должны заметить падения производительности.

Большая часть кода, отвечающего за управление массивами, содержится в модуле *array*. За исключением этой главы мы не будем использовать этот модуль, а в качестве альтернативы воспользуемся функционалом библиотеки NumPy, во многом превосходящей модуль *array*. Да и здесь мы в основном говорим не о самом этом модуле, а о способах избежать излишнего расходования ресурсов памяти.

На этом этапе у вас уже должно быть общее представление о том, как в Python выделяется память для объектов. Теперь давайте научимся рассчитывать, какой именно объем памяти выделен для объекта.

2.4.4. Систематизирование новых знаний: оценка объема памяти, занимаемой объектом

Сейчас вы уже знаете в общих чертах, как в Python работает процесс выделения памяти. Теперь пришло время собрать все знания воедино и написать общую функцию, которая поможет понять, сколько места в памяти занимает объект и все его содержимое.

Функция, приведенная ниже, возвращает как размер самих объектов, так и информацию о контейнерах. Если вы внимательно рассмотрите код, то увидите, что здесь мы работаем с идентификаторами объектов, подсчитываем контейнеры, включая сопоставительные объекты вроде словарей, в которых необходимо отслеживать и ключи, и значения, а также управляем строками и массивами.

Вычисление объема памяти, занимаемой объектами, – то еще минное поле, а для внешних объектов силами одного Python с этой задачей просто не справиться. В листинге 2.2, показанном ниже, мы сделали все, чтобы не учитывать дважды повторяющиеся объекты и контейнеры/итераторы. Исходный код можно найти в файле 02-python/sec4-memory/compute_allocation.py в сопроводительных материалах.

Листинг 2.2. Вычисление размера объекта в Python

```
from array import array
from collections.abc import Iterable, Mapping
from sys import getsizeof
from types import GeneratorType

def compute_allocation(obj) -> int:
    my_ids = set([id(obj)])
    to_compute = [obj]
    allocation_size = 0
    container_allocation = 0
    while len(to_compute) > 0:
        obj_to_check = to_compute.pop()
        allocation_size += getsizeof(obj_to_check)
        if type(obj_to_check) == str:
            continue
        if type(obj_to_check) == array:
            continue
        elif isinstance(obj_to_check, GeneratorType):
            continue
        elif isinstance(obj_to_check, Mapping):
            container_allocation += getsizeof(obj_to_check)
            for ikey, ivalue in obj_to_check.items():
                if id(ikey) not in my_ids:
                    my_ids.add(id(ikey))
                    to_compute.append(id(ikey))
```

Нам нужно сохранить идентификаторы объектов во избежание двойного подсчета

Также мы будем учитывать память, занятую контейнерами, такими как списки или словари

Строки и массивы – это итерируемые объекты, возвращающие размер своего содержимого. А мы не хотим дважды учитывать содержимое

Мы будем игнорировать содержимое генераторов

Для сопоставительных объектов нам нужно считать и ключи, и значения

```

        if id(ivalue) not in my_ids:
            my_ids.add(ivalue)
            to_compute.append(id(ivalue))
    elif isinstance(obj_to_check, Iterable):
        container_allocation += getsizeof(obj_to_check)
        for inner in obj_to_check:
            if id(inner) not in my_ids:
                my_ids.add(id(inner))
                to_compute.append(inner)
return allocation_size, allocation_size - container_allocation

```

← Для остальных итераторов нам нужно проверить их размер

Для вычисления объема занимаемой памяти мы воспользовались итеративным подходом. Этот алгоритм неплохо смотрелся бы в рекурсивной версии, но из-за отсутствия в Python надлежащей оптимизации хвостовых вызовов и рекурсивных функций в целом мы будем использовать итерации.

Подсчет памяти, занимаемой объектами из внешних библиотек, реализованных с помощью низкоуровневых языков программирования, таких как C или Rust, будет в основном зависеть от реализации и доступности нужной нам информации. Подробно об этом вы можете узнать из документации к соответствующим библиотекам и объектам.

ПРЕДУПРЕЖДЕНИЕ. Также для оценки занимаемой объектами памяти вы можете воспользоваться специальными библиотеками профилирования памяти. У меня есть некоторый опыт использования сторонних средств в этой области, но не могу сказать, что они меня сильно порадовали, что неудивительно с учетом сложности подсчета выделенной памяти в Python. Вы можете попробовать эти библиотеки, но будьте с ними осторожны.

Существуют и другие, более низкоуровневые способы расчета занимаемой памяти в Python, но о них мы поговорим тогда, когда будем обсуждать библиотеку NumPy. В данной главе мы не затрагиваем темы, связанные со сторонними библиотеками.

2.4.5. Оценка занимаемой объектами памяти в Python: выводы

Расчет занимаемого объектами места в памяти в языке Python – дело не такое простое, как может показаться. Функция `sys.getsizeof`, несмотря на свое название, не дает полной информации об объеме выделенной объекту памяти, и, чтобы учесть все возможные нюансы, необходимо изрядно потрудиться. Более того, в общем случае эта задача вообще не решается, поскольку библиотеки, написанные на низкоуровневых языках программирования, могут не предоставлять для подсчета всей необходимой информации.

У недостаточно гибкой подсистемы распределения памяти есть пара положительных побочных моментов. Первый из них состоит в том, что вы можете запускать больше параллельных процессов

в случае, если потребление памяти является сдерживающим фактором, как это иногда бывает. Второй предполагает использование алгоритмов, работающих с данными, помещающимися в памяти, которые отличаются более высокой эффективностью по сравнению с алгоритмами, использующими дисковое пространство для доступа к данным.

2.5. Использование ленивых вычислений и генераторов для работы с большими данными

Теперь давайте переключим внимание на функциональную возможность, получившую повсеместное распространение в Python 3, а именно *ленивые* (lazy), или отложенные, вычисления. Эта концепция позволяет отложить выполнение расчетов до момента, когда вам действительно понадобятся данные. Она бывает действительно полезна при работе с большими данными, поскольку зачастую вам вообще нет необходимости выполнять какие-то вычисления (и нести связанные с ними накладные расходы в отношении памяти) или они могут быть разнесены по времени. Если вы используете в своей работе генераторы, вы уже имеете дело с отложенными вычислениями. Python 3 стал гораздо более ленивым по сравнению с Python 2, поскольку в нем такие функции, как `range`, `map` и `zip`, стали работать в отложенном режиме. Такой режим позволяет обрабатывать большие блоки данных с гораздо меньшими требованиями к объему задействованной памяти и легко создавать *конвейеры данных* (data pipelines) внутри программного кода.

2.5.1. Использование генераторов вместо обычных функций

Давайте вернемся к исходному коду из первого раздела этой главы:

```
def get_file_temperatures(file_name):
    with open(file_name, "rt") as f:
        reader = csv.reader(f)
        header = next(reader)
        for row in reader:
            station = row[header.index("STATION")]
            # date = datetime.datetime.fromisoformat(row[header.
            index('DATE')])
            tmp = row[header.index("TMP")]
            temperature, status = tmp.split(",")
            if status != "1":
                continue
            temperature = int(temperature) / 10
            yield temperature
```

Ключевое слово `yield` в определении функции говорит о том, что это генератор

Наша функция `get_file_temperatures` представляет собой *генератор* (generator) – обратите внимание на ключевое слово *yield*. Давайте его запустим:

```
temperatures = get_file_temperatures(TEMPLATE_FILE.format(
    ➡ station="01044099999", year=2021))
```

```
print(type(temperatures))
print(sys.getsizeof(temperatures))
```

На экран будет выведен тип `generator` и размер структуры, равный 112. В реальности же практически ничего не произошло, поскольку мы имеем дело с ленивыми вычислениями. Сам код начнет выполняться только тогда, когда мы запустим итерации по созданному генератору, как показано ниже:

```
for temperature in temperatures:
    print(temperature)
```

На каждой итерации цикла `for` будет запускаться код генератора для извлечения следующего значения

Этот подход обеспечивает сразу несколько преимуществ. Первое и важнейшее из них состоит в том, что вам не понадобится выделять место в памяти для хранения всех значений температур, поскольку они будут извлекаться из генератора по очереди. Сравните это со списком, все значения которого размещаются в памяти компьютера одновременно. И это преимущество тем важнее, чем более объемную структуру данных возвращает функция, – вам может просто не хватить памяти для размещения всех ее элементов.

Второе преимущество связано с тем, что иногда нам могут быть не нужны все данные, возвращаемые функцией. Таким образом, *жадный* (eager) подход вместо ленивого может привести к бесполезным вычислениям, требующим ресурсов. Представьте, что вы пишете функцию, определяющую, встречаются ли в наших данных значения температуры ниже нуля. В этом случае вам нет необходимости перебирать все значения – достаточно будет дождаться первого вхождения отрицательной температуры.

При этом вы всегда при необходимости можете реализовать жадную версию генератора следующим образом:

```
temperatures = list(temperatures)
```

В этом случае вы утратите все присущие генератору преимущества, но иногда это может быть полезно. К примеру, если для вычисления всех значений требуется не так много времени и они займут приемлемое место в памяти, то вы можете воспользоваться жадной версией генератора, особенно при необходимости многократного обращения к его элементам.

ПРИМЕЧАНИЕ. Одно из главных отличий между Python 2 и Python 3 состоит в том, что многие встроенные структуры в новой версии стали ленивыми. К примеру, в нашем случае функции `zip`, `map` и `filter` вели бы себя в Python 2 совершенно иначе.

Генераторы могут быть использованы для уменьшения объема памяти, занимаемого рассчитываемыми данными, а иногда и для снижения времени вычисления. Таким образом, при написании кода, возвращающего последовательность результатов, необходимо всегда задаваться вопросом о том, есть ли смысл преобразовать обрабатывающую функцию в генератор.

Заключение

- Определение узких мест в отношении производительности решений – не такое простое дело, чтобы опираться исключительно на теорию и догадки. Первым делом необходимо выполнить профилирование программного кода для выяснения мест с повышенной нагрузкой. Интуиция не лучший помощник при определении проблем с производительностью, прагматичный практический подход почти всегда будет давать ей фору.
- Встроенные в Python механизмы профилирования бывают очень полезны, но иногда их результаты оказываются не самыми простыми для восприятия и интерпретирования. В этих случаях вам помогут визуальные средства профилирования кода, такие как SnakeViz.
- Кроме того, встроенные механизмы профилирования зачастую не могут ответить на вопрос о том, где конкретно находится узкое место в программном коде. Для более точного определения проблемных областей в коде можно воспользоваться инструментами вроде `line_profiler`, которые, однако, серьезно замедляют выполнение исходного кода в момент сбора статистики.
- Хотя первым делом при оптимизации мы обращаем внимание на производительность центрального процессора, не меньшее, а иногда и большее значение имеет процесс управления памятью. К примеру, решения, в которых не проведена оптимизация памяти и которые вследствие этого требуют применения алгоритмов для работы с данными, выходящими за границы доступной памяти, могут быть заменены вариантами, предусматривающими размещение всех данных в памяти, что позволит существенно сократить время выполнения.
- Python предоставляет разработчикам структуры данных, которые нужно уметь правильно использовать с точки зрения произ-

водительности. К примеру, поиск элементов в неупорядоченном списке может оказаться крайне неэффективным. Вы должны хорошо понимать разницу в скорости выполнения доступных операций применительно к различным базовым структурам данных. Работа с этими структурами производится практически в любой программе на языке Python, и умение эффективно с ними обращаться может быть первым и одним из наиболее важных шагов по повышению производительности ваших решений.

- Для написания эффективного кода на Python критически важно иметь представление о вычислительной сложности алгоритмов в *нотации O большое* (Big-O notation). При этом вы должны время от времени обновлять для себя эту информацию, поскольку Python не стоит на месте, и реализации алгоритмов могут меняться, а вместе с ними и их сложность.
- Ленивые техники программирования позволяют разрабатывать решения, допускающие использование меньших объемов памяти. Кроме того, иногда с их помощью можно существенно сократить объем вычислений.
- Приемы и методы, описанные в этой главе, могут применяться в решениях Python совместно с техниками, о которых мы будем говорить в следующих главах книги.

3

Конкурентность, параллелизм и асинхронная обработка

В этой главе мы обсудим следующие темы:

- применение асинхронной обработки для разработки приложений с уменьшенным временем ожидания;
- многопоточность в Python и ее ограничения при написании параллельных приложений;
- применение многопроцессности при написании приложений для извлечения максимума из многоядерной архитектуры процессоров.

Современные архитектуры *центральных процессоров* (ЦП – CPU) позволяют нескольким последовательным программам выполняться одновременно, что значительно повышает быстродействие. Фактически таким способом можно увеличить скорость выполнения приложения в число раз, равное количеству доступных обрабатывающих модулей или *ядер ЦП* (CPU core). Плохой новостью является то, что для реализации подобных решений необходимо сделать код параллельным, а базовый Python к такому повороту событий

не слишком подготовлен. Большая часть кода, написанного на Python, является последовательной, что не позволяет интерпретатору использовать все доступные ресурсы центрального процессора. Кроме того, как вы узнаете далее, интерпретатор Python вообще не предназначен для выполнения параллельных вычислений. Проще говоря, код на Python, который мы пишем, не может использовать все преимущества современного аппаратного обеспечения и всегда выполняется медленнее, чем теоретически мог бы. Наша задача – помочь Python использовать все имеющиеся в его распоряжении ресурсы.

Эта глава как раз будет посвящена этой теме. Начнем мы с тем, которые могут быть у вас на слуху, но при этом обладают собственной уникальной реализацией в Python. Мы обсудим конкурентность, многопоточность и параллелизм в терминах, принятых в Python, а также уделим особое внимание ограничениям, свойственным для многопоточного программирования в этой среде.

Кроме того, мы затронем вопросы, связанные с методами асинхронного программирования, позволяющие эффективно обслуживать конкурентные запросы без необходимости реализовывать параллельные решения. Асинхронное программирование – далеко не новинка, и оно очень популярно в среде JavaScript/Node.js. В Python эта парадигма была стандартизована совсем недавно, результатом чего стало появление модулей, способствующих реализации этих принципов.

В этой главе мы представим, что работаем в крупной компании, специализирующейся на разработке программного обеспечения. Нам поставили задачу разработать сверхбыстрый фреймворк MapReduce. Все данные должны храниться в памяти, а обработка выполняться на одном компьютере. Более того, ваша служба должна одновременно обрабатывать запросы от разных клиентов, большую часть которых будут составлять боты. Для реализации этого проекта мы в целях ускорения обработки запросов воспользуемся техниками конкурентного и параллельного программирования, включая многопоточность и многопроцессность. Кроме того, мы используем асинхронные принципы программирования для параллельного приема запросов от разных пользователей.

Разделим поставленную задачу на две части. В первом разделе главы мы реализуем сервер, способный обрабатывать множество запросов одновременно. После этого создадим сам фреймворк MapReduce, чему посвятим большую часть главы после раздела 3.1. При этом мы рассмотрим три различных способа реализации фреймворка: последовательный, многопоточный и многопроцессный. Это позволит нам увидеть в работе все три варианта и отметить для себя их преимущества, недостатки, компромиссы и ограничения. В заключительном разделе главы мы соединим вместе

сервер и фреймворк MapReduce, что должно привести к пониманию того, как разрабатывать решения, эффективно сочетающие разные технологии и подходы.

Для лучшего понимания организации главы взгляните на рис. 3.1, на котором изображен визуальный маршрут нашего путешествия. На нем мы отобразили подходы, о которых будем говорить, а также связи между ними. В верхней левой части каждого блока показан номер раздела, в котором будет обсуждаться тот или иной аспект.

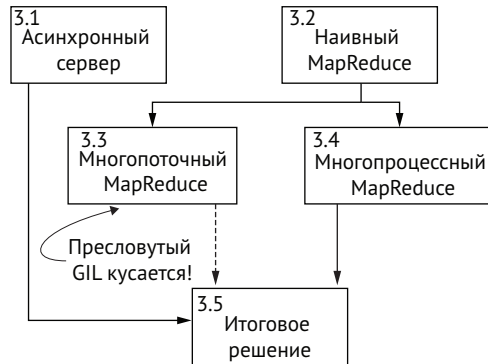


Рис. 3.1. Маршрут путешествия

Последовательная обработка, конкурентность и параллелизм

Перед тем как погрузиться в работу, давайте определимся с терминами *последовательная обработка*, *конкурентность* и *параллелизм*. Несмотря на то что они относятся к разряду базовых терминов, многие опытные разработчики до сих пор путают их, так что будет лучше, если мы заранее расставим точки над *i* и определимся с общей терминологией.

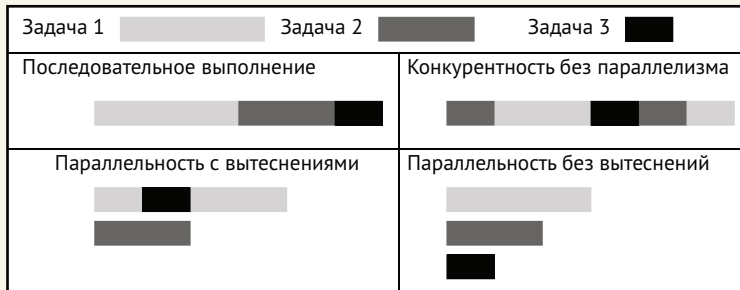
Легче всего будет объяснить концепцию *параллелизма* (parallelism). В этом случае задачи выполняются на самом деле параллельно и одновременно. *Конкурентные* (concurrent) задачи могут выполняться по-разному: параллельно или последовательно, в зависимости от языка программирования и операционной системы. Таким образом, все параллельные задачи являются конкурентными, но не наоборот.

Термин *последовательный* (sequential) можно использовать двояко. Во-первых, он может говорить о том, что некая последовательность задач должна выполняться в строго заданной очередности. Например, чтобы что-то записать в компьютер, вам необходимо сначала его включить, при этом порядок действий или их последовательность определяется самими действиями. Вторая задача может быть выполнена только по завершении первой.

Однако иногда слово *последовательный* употребляется применительно к ограничениям, накладываемым системой на выполнение задач. К примеру, в один момент времени только один человек может пройти через металлодетектор в аэропорту, даже если чисто физически места там хватило бы и двоим.

Наконец, существует концепция *вытеснения* (preemption), которая вступает в действие, когда одна задача в принудительном порядке прерывается для передачи выполнения другой задаче. Это имеет отношение к политике расписаний задач и требует наличия соответствующего программного или аппаратного обеспечения, именуемого *планировщиком* (scheduler).

Альтернативой *вытесняющей многозадачности* (preemptive multitasking) является *кооперативная многозадачность* (cooperative multitasking): в этом случае ваш код сам несет ответственность за оповещения системы о том, когда он может добровольно прерваться и передать управление другой задаче. Взгляните на рисунок ниже, чтобы лучше понять эти концепции.



Концептуальные модели последовательного, конкурентного и параллельного выполнения. Последовательное выполнение возникает тогда, когда все задачи запускаются по очереди и никогда не прерываются. Конкурентное выполнение без параллелизма предполагает возможность прерывания задач и их последующего возобновления. Параллелизм возникает в случае одновременности выполнения нескольких задач. Однако даже в случае параллельного выполнения зачастую мы имеем дело с вытесняющей многозадачностью, поскольку количество задач может превышать количество доступных ядер процессора. В идеале нам бы хотелось, чтобы было наоборот, – в этом случае все задачи выполнялись бы параллельно без необходимости прерывать их.

ПРИМЕЧАНИЕ. Мы не будем описывать базовые возможности многопоточности и многопроцессности в Python. Если вам необходимо заполнить эти пробелы, можете обратиться к соответствующей литературе, включая книгу «Asynсio и конкурентное программирование на Python» (Python Concurrency with Asynсio) от Мэттью Фаулера (<https://dmkpress.com/catalog/computer/programming/python/978-5-93700-166-5/>).

3.1. Написание шаблона асинхронного сервера

Тогда как главной нашей задачей является создание фреймворка MapReduce для обработки конкурентных запросов, начнем мы с написания серверной части, принимающей запросы, т. е. предоставляющей соответствующий интерфейс клиентам. В оставшейся части главы мы займемся процессом обработки запросов, а в этом разделе напишем сервер, который будет регистрировать подключения от всех клиентов и получать запросы MapReduce (данные и код). В процессе мы увидим, какую пользу способно принести асинхронное программирование при написании подобных серверов, даже без использования параллелизма.

Эпоха асинхронного программирования

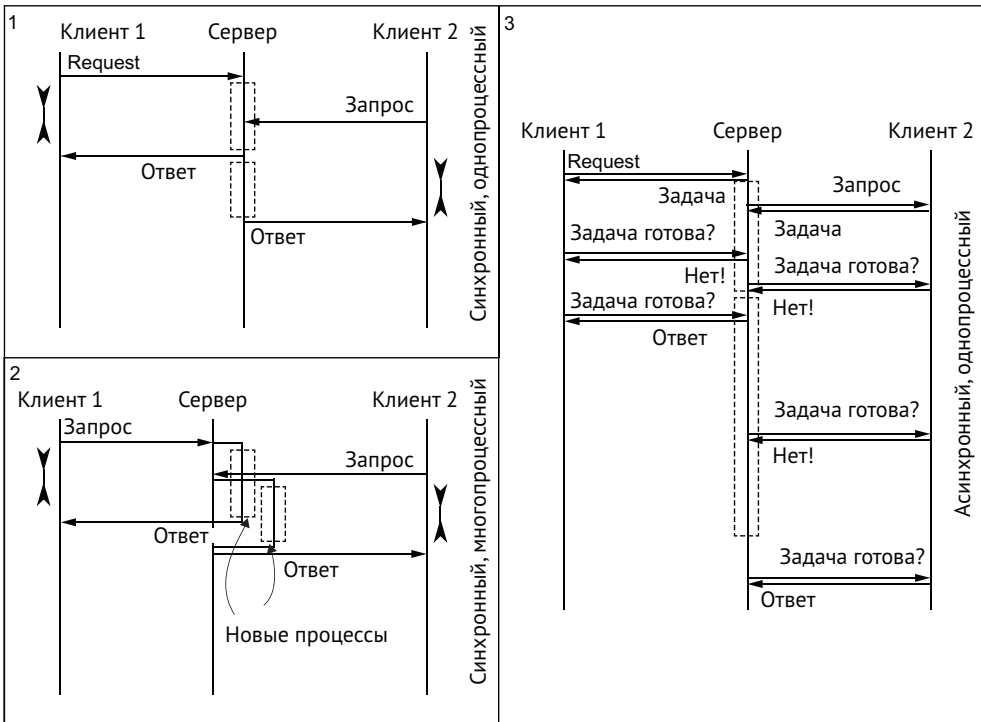
Принципы асинхронного программирования приобрели наибольшую популярность в мире JavaScript, и особенно при написании серверов на платформе Node.js. Эта парадигма идеально подходит при наличии большого количества медленных потоков ввода-вывода, нуждающихся в отслеживании. Наиболее показательным является пример веб-сервера, обрабатывающего множество мелких пакетов ограниченного размера, затрачивая на это минимум времени (обычно время исчисляется миллисекундами). В то же время асинхронная модель может пригодиться нам при написании конкурентных и параллельных приложений в чистом виде. Более того, как мы увидим в этой главе, асинхронный подход может быть использован при реализации более традиционных сценариев по анализу данных.

Чтобы было ясно, термин *асинхронность* никак не связан с понятиями «однопоточность», «многопоточность» или «многопроцессность». Асинхронные системы могут быть построены на основе любого из этих трех принципов.

Для начала давайте сформулируем одну из главных проблем, относящихся к синхронной обработке, чтобы в дальнейшем можно было сравнить синхронный подход с асинхронным. *Синхронное программирование* (synchronous programming) наиболее часто применяется в Python, и разработчики, пишущие на этом языке, первым делом обращаются именно к этой технологии. Однако синхронная (и однопоточная) версия сервера заблокирует систему во время ожидания ввода от клиента. А поскольку от открытия соединения клиентом до отправки запроса на сервер может пройти и 1 мс, и 1 час, это будет означать, что все остальные клиенты все это время вынуждены будут простаивать в ожидании. Здесь существует три возможных решения, показанных на рис. 3.2.

- 1 Осуществляется блокировка, как показано в блоке 1. Это означает, что во время обработки одного подключения все остальные задачи (включая другие подключения) ставятся на паузу. Такая блокировка прочих событий недопустима.

- 2 Здесь у нас реализовано многопоточное или многопроцессное решение, в котором один поток или процесс запускается для обработки первого запроса, как показано в блоке 2 на рис. 3.2. В результате главный процесс освобождается для приема других входящих запросов. Здесь возможна реализация однопоточного решения, которое может оказаться более легковесным в случаях, когда у нас есть много каналов ввода-вывода с небольшим количеством данных.
- 3 Наконец, при возникновении блокирующего вызова можно как-то освободить контроль исполнения, чтобы во время поступления данных могли выполняться другие фрагменты кода, как показано в блоке 3 на рис. 3.2. Это пример асинхронной обработки с одним потоком, и именно его мы будем здесь рассматривать.



Легенда

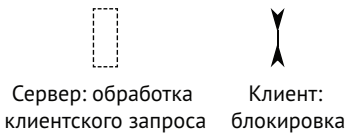


Рис. 3.2. Примеры архитектур с использованием синхронной однопроцессной/однопоточной, синхронной многопроцессной и асинхронной однопроцессной реализации сервера

В то же время существует масса альтернатив трем показанным на рисунке реализациям. К примеру, в конце главы мы рассмотрим решение, являющееся сочетанием второго и третьего подходов. С целью ускорения второго решения зачастую создается пул предварительно запущенных процессов. Для третьего сценария мы предполагаем, что вычислительные задачи могут быть прерваны (позже в этой главе мы ослабим это требование). Как вы можете знать, в Python многопоточный код обычно (хотя и не всегда) выполняется не параллельно. Позже поговорим об этом подробнее. При работе над проектом MapReduce мы обсудим все эти решения и характерные для них проблемы. При этом будем следовать плану, показанному на рис. 3.1. Первым делом мы реализуем простое наивное решение без всякого параллелизма¹. После этого опробуем решение на основе потоков, которое для наших нужд окажется недостаточно эффективным. Далее реализуем многопроцессное решение, которое позволит повысить быстродействие приложения. В заключение мы объединим сетевой интерфейс, разработанный в этом разделе, с многопроцессным решением и найдем применение многопоточности, хотя и не в области параллелизма.

СОВЕТ. Как и всегда, наше решение будет лишь одним из множества возможных, и ему есть масса альтернатив. И даже если бы оно было лучшим из возможных (а это не так), мы все равно шли бы на определенные компромиссы с целью улучшения восприятия кода. Разные задачи требуют разных решений, и лучший вариант зависит от ваших требований.

Мы хотим, чтобы вы извлекли из прочитанного не строгий набор правил, а техники и подходы, которые помогут реализовать оптимальное решение для ваших конкретных условий.

Что ж, давайте вернемся к нашей идее асинхронного однопоточного и однопроцессного сервера.

3.1.1. Разработка шаблона для взаимодействия с клиентами

Наш сервер будет работать по *протоколу TCP* на порту 1936. Исходный код можно найти в сопроводительных материалах в файле 03-concurrency/sec1-async/serve.py. Рассмотрим верхний уровень шаблона, обрабатывающий клиентские запросы:

¹ Несмотря на то что первое реализованное нами решение с использованием асинхронного подхода для нашего случая кажется слишком наивным, оно отлично подойдет в других ситуациях. К примеру, оно вполне годится для большинства веб-серверов, что видно на примере Node.js. Здесь, как и везде, лучшее решение зависит от специфики задачи.


```

import asyncio
import pickle

results = {}

async def submit_job(reader, writer):
    job_id = max(list(results.keys()) + [0]) + 1
    writer.write(job_id.to_bytes(4, 'little'))
    results[job_id] = job_id * 3

async def get_results(reader, writer):
    job_id = int.from_bytes(await reader.read(4), 'little')
    pickle.dump(results.get(job_id, None), writer)

async def accept_requests(reader, writer):
    op = await reader.read(1)
    if op[0] == 0:
        await submit_job(reader, writer)
    elif op[0] == 1:
        await get_results(reader, writer)

async def main():
    server = await asyncio.start_server(
        accept_requests, '127.0.0.1', 1936)
    async with server:
        await server.serve_forever()

asyncio.run(main())

```

Мы воспользовались библиотекой `asyncio`

Все наши функции помечены ключевым словом `async`

Эти строки могут заблокировать весь код вокруг

Мы используем метод `start_server` из библиотеки `asyncio` для вызова функции `accept_requests` для каждого подключения. Наш сервер будет прослушивать порт 1936 на локальном хосте 127.0.0.1

Ключевое слово `async` может быть использовано совместно с `with`, чтобы сделать операцию неблокирующей

Это точка входа в наше приложение: здесь запускается функция `main`

Бесконечное обслуживание подключений к серверу

Пока это лишь шаблон кода для сервера. В последнем разделе этой главы мы дополним его, когда будем собирать все воедино. Но и на данном этапе нам есть что обсудить. Главный вопрос состоит в том, зачем вообще это так сложно реализовывать? Почему бы не написать все синхронно, как обычно?

Основная причина в том, что функции чтения и записи по сети (третье сверху примечание в коде) могут длиться неопределенно долго. Кроме того, скорость работы сетевых интерфейсов на несколько порядков уступает быстродействию центрального процессора, и если мы позволим этим строкам блокировать наш код, то приложение станет работать очень медленно, а новые пользователи вынуждены будут бесконечно ждать своей очереди для подключения.

Показанная здесь инфраструктура Python, включающая ключевые слова `async`, `await` и модуль `asyncio`, призвана предотвратить ситуации, когда блокирующие вызовы мешают выполняться остальным элементам кода, не зависящим от них, в однопоточном приложении.

3.1.2. Программирование с сопрограммами

Асинхронные функции, показанные в предыдущем разделе, которые объявляются при помощи ключевых слов *async def*, называются *сoproграммами*, или *корутинами* (coroutine). Сопрограммы представляют собой функции, способные добровольно отдавать контроль выполнения. Другая часть системы, именуемая *управляющей программой* (executor), координирует все сопрограммы, запуская их согласно определенной политике.

При вызове сопрограммы из другой сопрограммы с использованием ключевого слова *await* вы, по сути, говорите Python, что в данный момент управление может быть передано в другое место. Такое поведение называется *кооперативной многозадачностью* (cooperative scheduling), поскольку передача контроля выполнения происходит добровольно и должна быть явным образом прописана в коде сопрограммы.

Сравните схему, основанную на сопрограммах, с традиционными потоковыми механизмами, принятыми в большинстве операционных систем. Там потоки вытесняются принудительно и не обладают контролем над своим выполнением. Это называется *вытесняющей многозадачностью* (preemptive scheduling). Обычно в многопоточном коде нет нужды явно указывать, где код может отдать контроль выполнения, поскольку его вытеснение будет выполнено принудительно. Потоки в Python в этом отношении очень похожи на потоки операционной системы. В то же время добровольное вытеснение работает только в асинхронном коде.

В качестве типичного примера можно привести программу, ожидающую неких данных по сети и записывающую их на диск (обычные операции ввода-вывода). Работа программы может проходить так, как показано ниже. Заметьте, что выполнение программы здесь происходит последовательно, так что в потоках нет никакой необходимости.

- 1 В основной программе с помощью асинхронного диспетчера запускаются две сопрограммы: одна служит для ожидания соединений, а вторая – для записи данных на диск.
- 2 Управляющая программа останавливает выбор (возможно, случайным образом) на сопрограмме ожидания соединений, и она запускается.
- 3 Сетевая сопрограмма настраивает процесс прослушивания сети. После этого она переходит в режим ожидания подключений. В данный момент никаких подключений нет, в связи с чем сопрограмма *добровольно* отдает управление управляющей программе, чтобы она могла запустить что-то еще.
- 4 Управляющая программа запускает сопрограмму записи на диск.

- 5 Сопрограмма начинает процесс записи на диск. При этом запись осуществляется гораздо медленнее в сравнении с работой центрального процессора, поэтому и эта сопрограмма *добровольно* отдает управление управляющей программе, чтобы она могла позаниматься чем-то еще.
- 6 Управляющая программа вновь возвращается к сопрограмме прослушивания сети.
- 7 Подключений по-прежнему нет, так что сопрограмма передает управление обратно.
- 8 Управляющая программа переходит к сопрограмме записи на диск.
- 9 Сопрограмма записи на диск завершает свою работу.
- 10 Управляющая программа запускает сопрограмму прослушивания сети в бесконечном режиме, поскольку других задач не осталось. Если сопрограмма вернет управление, диспетчер снова ее запустит.
- 11 Сетевая сопрограмма в конце концов отвечает на подключение клиента или завершается по тайм-ауту.
- 12 Управляющая программа завершает работу и передает управление основной программе.

В этом, а также в заключительном разделе главы мы рассмотрим примеры различных сопрограмм, все из которых будут объявлены при помощи ключевых слов `async def`. Но давайте проведем небольшую проверку на следующем поднаборе строк исходного кода:

```
import asyncio

async def accept_requests(reader, writer):
    op = await reader.read(1)
    #...

result = accept_requests(None, None)
print(type(result))
```

Давайте посмотрим, что здесь нам даст ключевое слово `async` в определении функции. Без этого ключевого слова вы вполне были бы вправе ожидать возникновения *исключения* (`exception`) в строке кода `reader.read()`, поскольку в качестве аргумента `reader` мы передали `None`. Но показанный выше вызов функции `accept_requests` не приводит к запуску функции, а просто возвращает сопрограмму, которую мы и создали при помощи ключевых слов `async def`.

Ключевое слово `await` говорит Python, что сопрограмма `accept_requests` может быть в этот момент «заморожена», а вместо нее могут выполняться другие задачи. Таким образом, во время ожидания данных от объекта `reader` Python может заниматься другими делами, пока данные не поступят. Если сопрограммы своим поведением напоминают вам генераторы, о которых мы говорили в главе 2, значит, вы на верном пути.

3.1.3. Передача сложных данных от простого синхронного клиента

Для взаимодействия с нашим сервером мы напишем простой синхронный клиент. Он будет служить примером типичного кода на Python, и этого будет вполне достаточно для подключающейся к нашему серверу программы. Что более важно, на этом примере мы продемонстрируем процедуру обмена данными и кодом между процессами. И если исходный код сервера мы будем дорабатывать в дальнейшем, то клиента изменения не коснутся.

Наш клиент будет осуществлять передачу как кода (исходный код может быть найден в файле 03-concurrency/sec1-async/client.py), так и данных, после чего перейдет в режим ожидания ответа от сервера:

```
import marshal
import pickle
import socket
from time import sleep

def my_funs():
    def mapper(v):
        return v, 1

    def reducer(my_args):
        v, obs = my_args
        return v, sum(obs)

    return mapper, reducer

def do_request(my_funs, data):
    conn = socket.create_connection(('127.0.0.1', 1936))
    conn.send(b'\x00')
    my_code = marshal.dumps(my_funs.__code__)
    conn.send(len(my_code).to_bytes(4, 'little'))
    conn.send(my_code)
    my_data = pickle.dumps(data)
    conn.send(len(my_data).to_bytes(4, 'little'))
    conn.send(my_data)
    job_id = int.from_bytes(conn.recv(4), 'little')
    conn.close()

    print(f'Getting data from job_id {job_id}')
    result = None
    while result is None:
        conn = socket.create_connection(('127.0.0.1', 1936))
        conn.send(b'\x01')
        conn.send(job_id.to_bytes(4, 'little'))
        result_size = int.from_bytes(conn.recv(4), 'little')
        result = pickle.loads(conn.recv(result_size))
        conn.close()
```

Библиотека `marshal` используется для передачи кода

Библиотека `pickle` используется для передачи большинства высокоуровневых структур данных в Python

Наши функции объявлены внутри другой функции, которая их возвращает

Создаем сетевое подключение

Создаем представление нашего кода в виде байтов

Получаем `job_id` и сами заботимся о кодировке

Будем поддерживать соединение, пока не получим результат

```

    sleep(1)
    print(f'Result is {result}')
```

```

if __name__ == '__main__':
    do_request(my_funs, 'Python rocks. Python is great'.split(' '))
```

Здесь также есть что обсудить. Начнем с кода, отвечающего за сетевое подключение. Мы создаем подключение по протоколу TCP с помощью интерфейса сокетов, принятого в Python, и используем соответствующий API для отправки и получения данных. При этом все вызовы являются потенциально блокирующими, что вполне приемлемо для нашего клиента.

Возможно, самой важной частью представленного выше кода являются фрагменты передачи данных. В Python для *сериализации данных* с целью их последующей передачи между процессами в большинстве случаев используется модуль *pickle*. Однако этот модуль не может использоваться для передачи кода. Мы для этой цели выбрали библиотеку *marshal*. Также мы воспользовались методом *to_bytes* объекта *int* в напоминание о том, что можем сами выполнять необходимую кодировку в критически важных ситуациях, например когда нам нужно, чтобы решение было одновременно компактным и быстрым. Ни одним из этих достоинств *pickle* похвастаться не может. Конечно, в данном сценарии мы много внимания уделили кодированию и декодированию. Вернемся к этому вопросу при обсуждении операций ввода-вывода.

Наш код для передачи мы разместили внутри функции *my_funs*, которая его возвращает. В качестве альтернативы мы могли бы использовать объекты. Для выполнения этого кода откройте терминал и запустите сервер следующим образом:

```
python server.py
```

После этого похожим образом запустите клиент:

```
python client.py
```

Результат будет следующий:

```

Getting data from job_id 1
Result is [Number between 1 and 4]
```

3.1.4. Альтернативные способы передачи данных между процессами

Более распространенным подходом для клиент-серверной коммуникации является интерфейс REST, использующий протокол HTTPS, но он не очень хорошо подходит для описания лежащих в его основе концепций. В главе 6 мы затронем вопросы, связанные

с альтернативными подходами к сетевым коммуникациям. Так или иначе любая реалистичная реализация в этом случае потребовала бы хотя бы минимального шифрования.

3.1.5. Асинхронное программирование: выводы

Асинхронный подход может оказаться достаточно эффективным при обработке большого количества одновременных запросов от пользователей. Для повышения быстродействия подобных систем должны выполняться два основных условия. Первое состоит в том, что коммуникация с внешними процессами должна быть ограничена. Второе заключается в минимальном процессорном времени для обработки каждого поступающего запроса. Поскольку оба эти условия вполне применимы к большинству веб-серверов, эта технология может быть достаточно эффективно использована при создании веб-приложений.

В этом разделе мы обсудили лишь самые основы асинхронного программирования, и, если хотите погрузиться в эту тему глубже, вам необходимо более подробно изучить функционал асинхронного подхода в Python. Я советую познакомиться с такими важными аспектами асинхронного программирования, как *асинхронные итераторы* (asynchronous iterators – `async for`) и *менеджеры контекста* (context managers – `async with`). Помимо этого, существует масса библиотек для применения асинхронного подхода, одной из которых является *aiohttp*, представляющая собой асинхронный аналог стандартной библиотеки `requests` для обмена данными по протоколу HTTP.

3.2. Реализация базового движка MapReduce

Теперь давайте вернемся к главной цели, которую мы поставили себе в начале главы, – написать фреймворк MapReduce. В первом разделе мы позаботились о коммуникационной инфраструктуре вокруг нашего фреймворка, а теперь займемся созданием самого решения. Для начала мы построим базовую конструкцию сценария, после чего разовьем ее и напишем более эффективные версии с точки зрения вычислений.

3.2.1. Описание фреймворка MapReduce

Давайте начнем с описания будущего фреймворка MapReduce и всех входящих в него компонентов. С точки зрения теории вычисления в этом фреймворке должны быть разделены на две части, представляющие собой *map* и *reduce*. Рассмотрим типичное применение подобного фреймворка для подсчета количества слов в тексте. Позаимствуем следующую пару строк из оригинала пьесы Уильяма Шекспира «Буря» (The Tempest): «*I am a fool. To weep at what I am glad of*». На рис. 3.3 показана схема фреймворка MapReduce

с такими входными данными. При этом, помимо функций *map* и *reduce*, этот фреймворк на практике может включать и другие операции. К примеру, результаты, полученные из функции *map*, должны быть определенным образом *сведены* (shuffle) перед отправкой в функцию *reduce*. Смысл сведения состоит в том, что, если в функцию *reduce* направить два экземпляра слова *at*, результат подсчета его вхождений окажется некорректным.

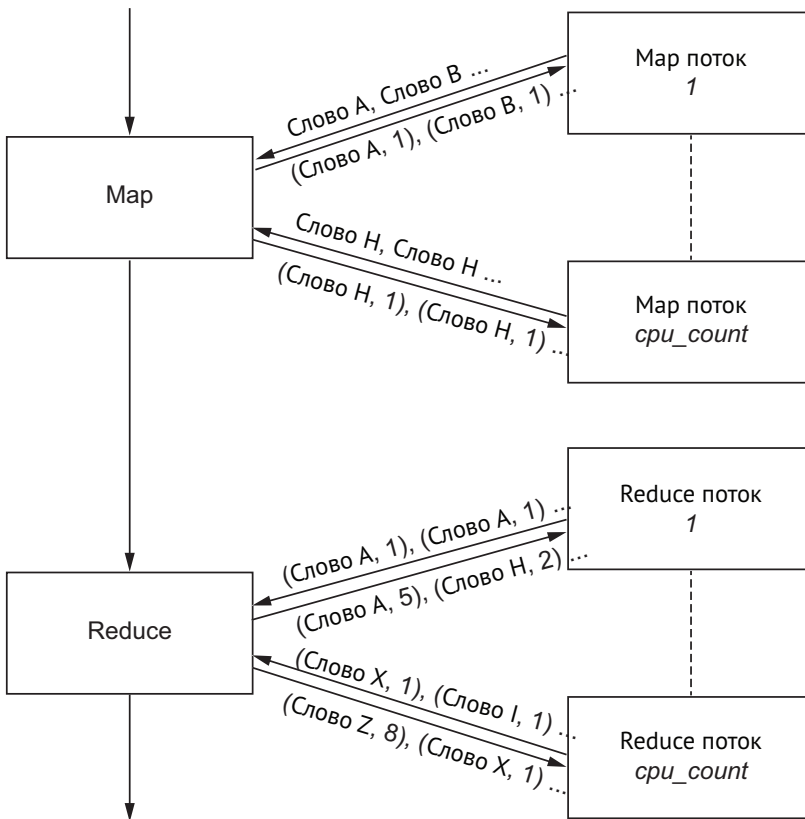


Рис. 3.3. Пример использования фреймворка MapReduce для подсчета количества слов. Обычно подобные фреймворки предполагают наличие нескольких процессов или потоков, в рамках которых реализуются функции *map* и *reduce*. Зачастую эти действия выполняются на разных компьютерах в распределенной системе

Подсчет количества слов может быть реализован при помощи функции *map*, которая будет возвращать запись для каждого слова с количеством, равным единице, и функции *reduce*, суммирующей количества вхождений во всех записях для одного и того же слова. Таким образом, функция *map* может возвращать следующие записи:

```
I, 1  
am, 1  
a, 1  
fool, 1  
To, 1  
weep, 1  
at, 1  
what, 1  
I, 1  
am, 1  
glad, 1  
of, 1
```

А функция `reduce` преобразует их в следующий вид с количествами вхождений слов:

```
I, 2  
a, 1  
fool, 1  
To, 1  
weep, 1  
at, 1  
what, 1  
am, 2  
glad, 1  
of, 1
```

Где-то между этими процессами нам необходимо дополнительно преобразовать входные данные таким образом, чтобы каждое уникальное слово обрабатывалось функцией `reduce` лишь раз. К примеру, если бы слово *am*, присутствующее в исходном наборе дважды, проходило через функцию `reduce` два раза, мы бы получили два его вхождения с количеством, равным единице, а не одно вхождение с количеством два. В нашем сервере эта дополнительная функция (`shuffle`) будет встроена, так что пользователю не нужно будет ее дополнительно нам предоставлять.

3.2.2. Разработка простейшего тестового сценария

Помните, что мы самостоятельно разрабатываем фреймворк MapReduce. И хотя мы сами им пользоваться не будем, необходимо как-то его протестировать. Для этого мы обратимся к простейшему использованию механизма MapReduce – подсчету количества слов в тексте. В дальнейшем наш фреймворк может быть использован для самых разных задач, но с целью тестирования идеально подойдет процедура подсчета слов.

Пользовательский код для использования нашего фреймворка может быть очень простым, как показано ниже. Помните, что мы

не должны ограничиваться исключительно таким использованием фреймворка, это просто пример для проверки его работоспособности:

Мы намеренно используем функциональную нотацию, поскольку механизм MapReduce проистекает из функциональной парадигмы. Если вы следуете рекомендациям по написанию кода на Python PEP 8, контроль синтаксиса выдаст предупреждение, поскольку одно из правил PEP 8 гласит: «Всегда используйте ключевое слово `def` вместо присвоения выражения с использованием `lambda` переменной непосредственно». Как именно это предупреждение будет выглядеть, зависит от используемого вами инструмента контроля синтаксиса. Только вы решаете, следовать ли приведенной выше нотации или соблюдать правила PEP 8. Во втором случае обработчик будет выглядеть так: `def emitter(word)`. Мы будем придерживаться своего подхода при тестировании нашего фреймворка.

```
emitter = lambda word: (word, 1)
counter = lambda emitted: (emitted[0], sum(emitted[1]))
```

3.2.3. Первая реализация фреймворка MapReduce

Помните, что две строчки кода, приведенные выше, должен будет ввести пользователь вашего фреймворка. Сейчас мы займемся реализацией фреймворка MapReduce, который сможет подсчитывать количество слов и делать много других полезных вещей. Начнем с минимального работающего прототипа, а в процессе написания главы улучшим его с применением многопоточности, параллелизма и асинхронности. Первую версию фреймворка можно найти в файле `03-concurrency/sec2-naive/naive_server.py`:

```
from collections import defaultdict

def map_reduce_ultra_naive(my_input, mapper, reducer):
    map_results = map(mapper, my_input)

    shuffler = defaultdict(list)
    for key, value in map_results:
        shuffler[key].append(value)

    return map(reducer, shuffler.items())
```

Использовать этот инструмент можно следующим образом:

```
words = 'Python is great Python rocks'.split(' ')
list(map_reduce_ultra_naive(words, emitter, counter))
```

Функция `list` форсирует распаковку ленивой структуры `map` (если вы еще не знакомы с семантикой ленивых выражений, обратитесь к главе 2), в результате чего мы получим следующий список:

```
[('Python', 2), ('is', 1), ('great', 1), ('rocks', 1)]
```

Полученная нами первая реализация фреймворка получилась довольно простой с точки зрения концепции, и она не учитывает один важнейший аспект, состоящий в том, что функции в механизме MapReduce должны выполняться параллельно. В следующих разделах мы изрядно поработаем над этим и реализуем эффективное решение с применением принципов параллелизма.

3.3. Реализация конкурентной версии фреймворка MapReduce

Теперь давайте сделаем еще одну попытку и реализуем конкурентную версию фреймворка – на этот раз с использованием многопоточности. Воспользуемся многопоточным диспетчером из модуля `concurrent.futures` для управления задачами MapReduce. Мы хотим, чтобы наша реализация была не только конкурентной, но и параллельной, т. е. использовала все доступные вычислительные мощности. По крайней мере, это есть в наших планах.

3.3.1. Использование модуля `concurrent.futures` для реализации многопоточного сервера

Начнем с высокоуровневого модуля `concurrent.futures`, являющегося более декларативным по сравнению с распространенными библиотеками `threading` и `multiprocessing`. Упомянутые библиотеки лежат в основе низкоуровневых конкурентных интерфейсов в Python, и в следующем разделе мы воспользуемся модулем `multiprocessing`, позволяющим более тонко управлять ресурсами центрального процессора.

Ниже приведена обновленная версия фреймворка, код которой находится в файле `03-concurrency/sec3-thread/threaded_mapreduce_sync.py`:

```
from collections import defaultdict
from concurrent.futures import ThreadPoolExecutor as Executor

def map_reduce_still_naive(my_input, mapper, reducer):
    with Executor() as executor:
        map_results = executor.map(mapper, my_input)

        distributor = defaultdict(list)
        for key, value in map_results:
            distributor[key].append(value)

        results = executor.map(reducer, distributor.items())
    return results
```

Мы будем использовать многопоточный диспетчер (executor) из модуля `concurrent.futures`

Диспетчер может использоваться в качестве менеджера контекста

У диспетчера есть функция `map` с блокирующим поведением

Мы воспользуемся очень простой процедурой объединения данных

Наша функция принимает на вход исходные данные, а также функции `mapper` и `reducer`. Диспетчер из модуля `concurrent.futures` ответственен за управление потоками, хотя мы можем и явно указать, какое количество потоков хотели бы задействовать. Если не указать этот параметр, по умолчанию будет использоваться значение, возвращаемое методом `os.cpu_count()`. Актуальное количество потоков может варьироваться в зависимости от версии Python. На рис. 3.4 представлена схема работы программы.

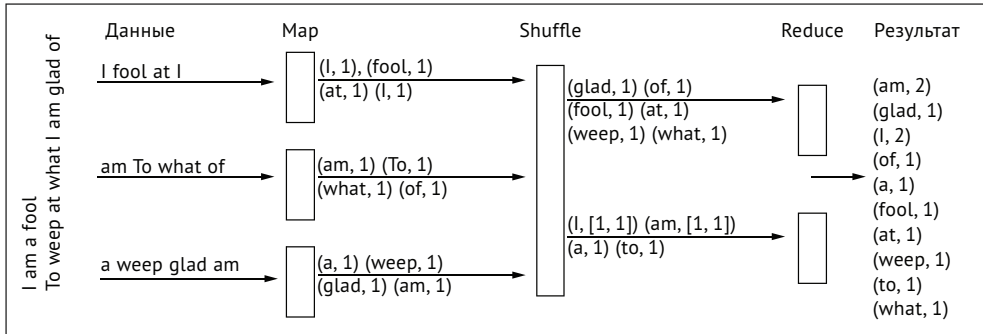


Рис. 3.4. Многопоточный запуск фреймворка MapReduce

Помните, что мы должны убедиться в том, что результаты для одного и того же объекта (в нашем случае слова) отправляются в правильную функцию `reduce`. Мы реализовали простейшую версию в виде словаря `distributor`, в котором находится по одной записи для каждого слова.

В нашей версии кода может возникать перерасход памяти по причине того, что обработчик `shuffle` должен хранить все данные в памяти, хоть и в компактном виде. С целью упрощения понимания кода мы закроем на это глаза.

При использовании модуля `concurrent.futures` управление *обработчиками* (`worker`) происходит в режиме черного ящика. В связи с этим мы даже не знаем, как именно здесь выполняется и на что направлена оптимизация. Если же мы хотим полностью контролировать происходящее и настраивать обработчики по собственному желанию, нам придется воспользоваться библиотекой `threading`, что мы и сделаем в следующем разделе¹.

Чтобы проверить работу второй версии фреймворка, выполним следующий код:

```
words = 'Python is great Python rocks'.split(' ')
print(list(map_reduce_still_naive(words, emitter, counter)))
```

¹ Также вы можете самостоятельно реализовать диспетчер `concurrent.futures`, но в этом случае вам все равно необходимо знать, как работают лежащие в основе этого модуля библиотеки `threading` и `multiprocessing`.

Вывод будет таким же, как и в предыдущем примере.

У нашего последнего решения есть одна проблема, и заключается она в том, что при его запуске мы лишаемся возможности взаимодействовать с внешней программой. Фактически при запуске инструкции `executor.map` мы вынуждены ждать окончания выполнения задачи. Это не так важно, когда мы имеем дело с пятью словами, но во время обработки объемных текстов вам бы наверняка хотелось получать какую-то обратную связь. К примеру, мы могли бы создать строку с прогрессом, в которой отображался бы процент выполнения задачи. Это потребует несколько иного подхода к реализации сценария.

3.3.2. Асинхронное выполнение с использованием будущих объектов

Давайте для начала напомним часть кода, отвечающую за функцию `map`, чтобы понять, что происходит. Этот код находится в файле `03-concurrency/sec3-thread/threaded_mapreduce.py`:

```
from collections import defaultdict
from concurrent.futures import ThreadPoolExecutor as Executor

def async_map(executor, mapper, data):
    futures = []
    for datum in data:
        futures.append(executor.submit(mapper, datum))
    return futures

def map_less_naive(executor, my_input, mapper):
    map_results = async_map(executor, mapper, my_input)
    return map_results
```

Здесь мы используем метод `submit` вместо `map`

Если метод `map` ожидает окончания выполнения задачи в блокирующем режиме, то `submit` – нет. Скоро мы увидим, что это означает на практике.

Давайте для начала изменим нашу функцию для мэппинга, чтобы мы могли контролировать происходящее:

```
from time import sleep

def emitter(word):
    sleep(10)
    return word, 1
```

Функцию `sleep` мы добавили для искусственного замедления работы кода – это поможет нам отслеживать, что происходит. Попробуем использовать нашу функцию `map` как есть:

```
with Executor(max_workers=4) as executor:
    maps = map_less_naive(executor, words, emitter)
    print(maps[-1])
```

Если вывести на экран последний элемент полученного списка, результат вас немало удивит:

```
<Future at 0x7fca334e0e50 state=pending>
```

Как видите, мы не получили ожидаемый кортеж ('rocks', 1), а вместо него на экран вывелась информация о *будущем объекте*, или *объекте future*. Будущие объекты представляют собой ожидаемые результаты, статус которых может быть проверен в любой момент. Это можно сделать следующим образом:

```
with Executor(max_workers=4) as executor:
    maps = map_less_naive(executor, words, emitter)
    not_done = 1
    while not_done > 0:
        not_done = 0
        for fut in maps:
            not_done += 1 if not fut.done() else 0
        sleep(1)
        print(f'Still not finalized: {not_done}')
```

Мы выделили лишь четыре обработчика для пяти задач, чтобы можно было отслеживать прогресс

Проверяем статусы будущих объектов

Будем выводить статус, пока есть незавершенные задачи

Ненадолго засыпаем, чтобы избежать появления большого количества текста

Если мы запустим этот код, то получим несколько сообщений с текстом `Still not finalized...`. За первые 10 с вы обычно увидите пять сообщений, а после этого еще одно. Поскольку мы отрядили на выполнение процедуры четыре обработчика, первые четыре задачи завершатся за 10 с, после чего сможет быть запущена пятая. С учетом того, что мы имеем дело с конкурентным кодом, ситуация может меняться от запуска к запуску, и потоки могут вытесняться друг другом по-разному. Заранее это не определено.

Осталось собрать последний элемент в этой мозаике, который появится в финальной версии многопоточного решения. Нам нужно как-то научиться оповещать вызывающую программу о процессе работы. Мы это сделаем путем передачи на вход *функции обратного вызова* (callback function), которая будет запускаться при возникновении важного события. В нашем случае важным событием является процесс отслеживания выполнения программы. Реализуем это следующим образом:

```
def report_progress(futures, tag, callback):
    not_done = 1
    done = 0
```

Функция `report_progress` требует передачи функции обратного вызова, которая будет запускаться каждые полсекунды и выводить статистическую информацию о работе программы

```

while not_done > 0:
    not_done = 0
    done = 0
    for fut in futures:
        if fut.done():
            done += 1
        else:
            not_done += 1
    sleep(0.5)
    if not_done > 0 and callback:
        callback(tag, done, not_done)

def map_reduce_less_naive(my_input, mapper, reducer, callback=None):
    with Executor(max_workers=2) as executor:
        futures = async_map(executor, mapper, my_input)
        report_progress(futures, 'map', callback)
        map_results = map(lambda f: f.result(), futures)
        distributor = defaultdict(list)
        for key, value in map_results:
            distributor[key].append(value)

        futures = async_map(executor, reducer, distributor.items())
        report_progress(futures, 'reduce', callback)
        results = map(lambda f: f.result(), futures)
    return results

```

Отслежи-
ваем ин-
формацию
для задач
с типом
map

Поскольку результаты фак-
тически являются будущими
объектами, нам необходимо
извлечь из них данные

Поскольку результаты фактически являются будущими
объектами, нам необходимо извлечь из них данные

Отслеживаем
информацию
для задач с ти-
пом reduce

Итак, каждые полсекунды, пока выполняются функции `map` и `reduce`, происходит вызов переданной пользователем функции обратного вызова. Эта функция может быть как очень простой, так и достаточно сложной. Единственное условие – выполнять она должна довольно быстро, поскольку ее все вокруг будут ждать. Для нашего примера с подсчетом количества слов функция обратного вызова может быть очень простой:

```

def reporter(tag, done, not_done):
    print(f'Operation {tag}: {done}/{done+not_done}')

```

Обратите внимание, что сигнатура функции обратного вызова не является произвольной, а должна соответствовать шаблону вызова из функции `report_progress`, в котором прописана передача тега, а также количества выполненных и невыполненных задач.

Давайте запустим наше приложение следующим образом:

```

words = 'Python is great Python rocks'.split(' ')
results = map_reduce_less_naive(words, emitter, counter, reporter)

```

В результате вы увидите несколько строк со статусами выполнения задач, после чего будет отображен результат, как показано ниже:

```
Operation map: 3/5
Operation reduce: 0/4
('is', 1)
('great', 1)
('rocks', 1)
('Python', 2)
```

Было бы совсем нетрудно, к примеру, использовать возвращаемое значение в качестве индикатора для отмены выполнения фреймворка MapReduce. Это могло бы нам позволить изменить семантику функции обратного вызова и применять ее для прерывания процесса.

К сожалению, приведенное выше решение является конкурентным, но не параллельным. Причина в том, что интерпретатор Python (или скорее CPython) позволяет одновременно работать только одному потоку. Это ограничение получило название *глобальная блокировка интерпретатора* (Global Interpreter Lock – *GIL*). Давайте выделим целый раздел для обсуждения этой концепции и посмотрим, как *GIL* ведет себя в отношении потоков.

3.3.3. Глобальная блокировка интерпретатора и многопоточность

Поскольку интерпретатор CPython оперирует потоками операционной системы, которые являются вытесняющими, глобальная блокировка интерпретатора (*GIL*) накладывает на них соответствующее ограничение таким образом, что в любой момент времени может выполняться только один поток. В результате вы можете писать многопоточные приложения и запускать их на многоядерных процессорах, но при этом не получите параллелизма. Хуже того, накладные расходы, связанные с передачей управления между потоками, на многоядерной архитектуре могут оказаться весьма большими из-за противоречий между *GIL*, не позволяющей одновременно запускаться более чем одному потоку, и центральным процессором с операционной системой, оптимизированными для таких операций.

Данная книга включает в себя раздел, посвященный многопоточному программированию, поскольку без него она была бы просто неполноценной. Но, положив руку на сердце, если вы хотите реализовывать высокоэффективные сценарии, потоки в Python редко вам смогут прийти на помощь.

Что касается *GIL*, то связанные с ней проблемы переоценены. Факт в том, что, если вам нужно написать быстрый и эффектив-

ный код на потоках, Python в любом случае не подойдет из-за своего быстрого действия. Виной тому и особенности реализации интерпретатора CPython, и динамическая природа языка в целом. Когда дело дойдет до критически важных участков кода, вы все равно отдадите предпочтение языкам программирования низкого уровня, таким как C или Rust, или встроенным в Python подсистемам вроде Cython или Numba, о которых мы поговорим позже.

GIL предоставляет пару лазеек для низкоуровневых языков – таким образом, при реализации фрагментов кода с их помощью вы сможете обойти связанные с блокировкой потоков ограничения и использовать параллелизм на полную мощность. Именно так работают библиотеки NumPy, SciPy и scikit-learn. Они задействуют многопоточный код, написанный на C или Fortran, и тем самым могут обходить ограничения GIL. Так что у вас есть возможность инициировать параллельные вычисления в Python. Другое дело, что сам код при этом будет написан не на чистом Python.

В то же время у вас есть вполне легальная возможность писать параллельный код на Python, но для этого вам придется воспользоваться модулем multiprocessing, как будет описано далее.

PyPy

Тогда как CPython является стандартной реализацией языка Python, существуют и другие реализации, такие как *IronPython* и *Jython* для .NET и JVM соответственно. Также отдельно стоит упомянуть реализацию языка PyPy, которая представляет собой не интерпретатор, а динамический компилятор, или JIT-компилятор (just-in-time compiler). PyPy нельзя рассматривать как упрощенную замену интерпретатора CPython, поскольку многие его библиотеки не работают с PyPy напрямую. В то же время в случаях, когда нужные вам библиотеки обладают совместимостью с PyPy, этот компилятор может показывать более высокие результаты, несмотря на наличие GIL. В этой книге мы в основном будем работать с интерпретатором CPython, но в отдельных случаях компилятор PyPy может оказаться более эффективной альтернативой.

Добавлю также, что, если вы путаете названия компилятора PyPy и пакетного репозитория PyPI, знайте, что вы не одиноки.

3.4. Реализация фреймворка MapReduce с использованием библиотеки multiprocessing

Из-за действия глобальной блокировки интерпретатора реализованная нами многопоточная версия фреймворка на самом деле не работает параллельно. Чтобы изменить это, мы можем пойти двумя путями: переписать наш код с использованием низкоуровневых языков вроде C или Rust или, что мы и сделаем в этом разделе,

ле, обратиться к библиотеке `multiprocessing`, чтобы задействовать всю имеющуюся в нашем распоряжении мощность центрального процессора. Решения с использованием языков низкого уровня будут обсуждаться в следующих главах книги.

3.4.1. Решение на основе модуля `concurrent.futures`

В теории решение с использованием модуля `concurrent.futures` должно быть очень простым. Сам модуль устроен так, чтобы вы могли максимально быстро и легко заменить импорт `ThreadPoolExecutor` на `ProcessPoolExecutor` (код из этого раздела можно найти в файле с именем `03-concurrency/sec4-multiprocess/futures_mapreduce.py`):

```
from concurrent.futures import ProcessPoolExecutor as Executor
```

Если вы замените на эту строку соответствующую строку в нашем асинхронном коде из предыдущего раздела, вы заметите, что программа начнет подтормаживать в части `reduce`. Давайте разбираться. Для этого мы немного усовершенствуем нашу функцию `report_progress`, написанную ранее:

```
def report_progress(futures, tag, callback):
    not_done = 1
    done = 0
    while not_done > 0:
        not_done = 0
        done = 0
        for fut in futures:
            if fut.done():
                done += 1
            else:
                not_done += 1
        sleep(0.5)
        if callback:
            callback(tag, done, not_done)
```

Мы добавили всего два вывода на экран. Если запустим код снова, то получим следующий вывод:

```
<Future at 0x7f1ffff104c0 state=finished raised PicklingError>
Can't pickle <function <lambda> at 0x7f2000131ca0>: attribute lookup
<lambda> on __main__ failed
```

Получается, что лямбды (а наша функция `counter` реализована в виде лямбды) не могут обрабатываться при помощи модуля `pickle`. А в библиотеке `multiprocessing` коммуникация между процессами происходит именно при помощи модуля `pickle`. Таким образом, мы не можем передать нашу функцию `counter` в подпроцесс в ее нынешнем виде. Придется переписать ее с использованием ключевого слова `def`:

```
def counter(emitted):  
    return emitted[0], sum(emitted[1])
```

Это решит нашу текущую проблему, но в общем смысле вы не можете просто так взять и перейти с многопоточного решения на многопроцессное. Какие изменения вас еще могут ждать? Это вы узнаете уже в следующем разделе.

Проблемы с передачей кода и данных при использовании модуля `multiprocessing`

Как мы успели увидеть, передача лямбда-функций между процессами с использованием модуля `pickle` в конфигурации, принятой по умолчанию, невозможна. Если вы хотите делать это, то вынуждены будете реализовывать собственный протокол.

Если модуль `pickle` не может вам помочь, придется пользоваться обходными путями, поскольку библиотека `multiprocessing` целиком и полностью полагается на него при передаче данных. Свои методы могут включать в себя объекты из сторонних библиотек, особенно если они реализованы не на Python.

Файловые указатели, подключения к базам данных и сокеты – все эти объекты вы не сможете передавать между процессами, если не позаботитесь об этом отдельно. При работе с потоками вы сможете совместно пользоваться этими объектами, если убедитесь, что они потокобезопасны. Еще одной проблемой при работе с модулем `pickle` является его скорость. Если у вас предусмотрена множественная передача данных внутри программы, это может свести на нет все преимущества параллелизма.

Использование примитивов Python с целью осуществления коммуникации идеально подходит для приложений с низкой гранулярностью и небольшими объемами передаваемых данных. Если в вашем сценарии предполагается наличие высоких накладных расходов, связанных с передачей информации, все преимущества использования модуля `multiprocessing` могут быть нивелированы.

3.4.2. Решение на основе модуля `multiprocessing`

Модуль `concurrent.futures` обеспечивает достаточно простой интерфейс для осуществления конкурентных вычислений. Он может продемонстрировать высокую производительность для большого числа разнообразных задач. В то же время за простоту использования этого модуля приходится платить потерей контроля за происходящим в коде. В каком порядке выполняются будущие объекты? Хотя мы сами задаем максимальное количество обработчиков, сколько из них будут в действительности доступны в конкретный момент времени? Будут ли процессы использоваться повторно, или для каждой задачи будут создаваться новые? При использовании модуля `concurrent.futures` все эти вопросы решаются самим диспетчером, и мы никак не можем повлиять на ход событий.

В нашем случае мы бы хотели произвести некоторые дополнительные действия для повышения быстродействия кода. К примеру, было бы неплохо создать все процессы заранее или держать их активными даже в отсутствие задач. Дело в том, что создание и уничтожение процессов при поступлении задач связаны с дополнительными накладными расходами, которыми лучше заняться в свободное от обработки запросов время. Начнем с самостоятельного создания пула процессов и пока не будем отслеживать происходящие процессы в реальном времени. Исходный код можно найти в файле `03-concurrency/sec4-multiprocess/mp_mapreduce_0.py`:

```
from collections import defaultdict
import multiprocessing as mp
```

← Импортируем модуль multiprocessing

```
def map_reduce(my_input, mapper, reducer):
    with mp.Pool(2) as pool:
        map_results = pool.map(mapper, my_input)
        distributor = defaultdict(list)
        for key, value in map_results:
            distributor[key].append(value)
        results = pool.map(reducer, distributor.items())
    return results
```

← Создаем пул из двух процессов

← В пуле мы можем использовать синхронную функцию map

Код получился очень простым и понятным. Единственная новинка здесь состоит в создании объекта *Pool*. Пул создается каждый раз при запросе на выполнение операции MapReduce и не сохраняется между вызовами. Таким образом, мы платим цену создания пула каждый раз при использовании фреймворка.

CPU_count против sched_getaffinity при определении размера пула

В показанном выше коде мы явным образом указали количество процессов в пуле. В большинстве случаев вы будете пользоваться специальными функциями, результат которых будет зависеть от конкретных вычислительных мощностей. По умолчанию пул использует в качестве входного параметра функцию *os.cpu_count*, название которой может сбивать с толку. На самом деле она возвращает не количество вычислительных блоков процессора, а количество гиперпотоков (hyperthread).

Более правдоподобным аналогом этой функции может служить выражение `len(os.sched_getaffinity())`, поскольку оно возвращает количество доступных в данный момент ядер. При этом в вашем компьютере может быть и больше ядер, но операционная система, контейнер или виртуальная машина могут вносить для вас определенные ограничения.

ПРЕДУПРЕЖДЕНИЕ. Метод `Pool.map` является жадным, тогда как функция `map` в языке Python – ленивой. Таким образом, следующие две строки кода нельзя считать семантически эквивалентными:

```
map(fun, data)
Pool.map(fun, data)
```

Первая возвращает управление сразу, не выполняя функцию `fun`. Ее жадным эквивалентом является выражение `list(map(fun, data))`. Обычно при разработке кода вы можете заменять `Pool.map` на `map`, поскольку отладку легче проводить, когда все выполняется в одном процессе. Но такой подход не всегда оправдан. В модуле `multiprocessing` также присутствует ленивая версия функции с именем `imap` и асинхронная версия `map_async`.

3.4.3. Отслеживание прогресса при использовании модуля `multiprocessing`

Так вышло, что метод `map_async` не поддерживает отслеживание состояния. У него есть поддержка функции обратного вызова, но она вызывается только по готовности результатов. Нам же необходим больший уровень гранулярности: в идеале было бы неплохо вызывать функцию по готовности каждого элемента в итераторе. Именно это нам нужно для отслеживания прогресса.

Давайте немного изменим код для поддержки такого поведения. Как мы уже сказали, у нас есть в распоряжении метод `Pool.map_async`, но, к сожалению, функция обратного вызова для него срабатывает только в конце цикла выполнений, а нам этого недостаточно. Нам нужно больше контроля за ситуацией. Код, приведенный ниже, можно найти в файле `03-concurrency/sec4-multiprocess/mp_mapreduce.py`:

```
def async_map(pool, mapper, data):
    async_returns = []
    for datum in data:
        async_returns.append(pool.apply_async(
            mapper, (datum, )))
    return async_returns

def map_reduce(pool, my_input, mapper, reducer, callback=None):
    map_returns = async_map(pool, mapper, my_input)
    report_progress(map_returns, 'map', callback)
    map_results = [ret.get() for ret in map_returns]
    distributor = defaultdict(list)
    for key, value in map_results:
        distributor[key].append(value)
    returns = async_map(pool, reducer, distributor.items())
    results = [ret.get() for ret in returns]
    return results
```

Мы используем метод `Pool.apply_async` для запуска каждой отдельной задачи

Обратите внимание, что параметр в функцию передается в виде кортежа

Получение результатов из асинхронных объектов путем использования метода `get`

Этот код не сильно отличается от решения с использованием модуля `concurrent.futures` – настолько, что у вас может появиться резонный вопрос, а стоит ли недостаток гибкости модуля `concurrent.futures` этих дополнительных упрощений.

Наша функция `map_reduce` теперь принимает пул в качестве параметра, что позволяет использовать его повторно. Обычно это дает прирост производительности в сравнении с необходимостью создавать новые процессы для каждой операции. В нашем примере мы не заметим этих дополнительных накладных расходов, но в более сложных сценариях инициализация каждого процесса может отнимать немало времени и ресурсов.

Для использования написанного кода нам необходимо предварительно создать пул процессов, как показано ниже:

Закрываем
пул

```
pool = mp.Pool()
results = map_reduce(pool, words, emitter, counter, reporter)
pool.close()
pool.join() ← Ожидаем завершения всех процессов
```

Мы могли бы использовать пул в виде менеджера контекста, но показали, что очистка пула включает в себя не только закрытие всех процессов, но и ожидание их завершения с помощью метода `join`. Более жесткой альтернативой методу `close` является метод `terminate`, который привел бы к немедленному завершению процессов без ожидания окончания их работы.

Недостаточное и избыточное выделение ресурсов процессора

При создании пула мы обычно используем значение для количества процессов по умолчанию, равное `os.cpu_count()`. Однако бывают ситуации, когда лучше выделять меньше ресурсов, чем имеется в вашем распоряжении. Более того, иногда стоит выделять и больше ресурсов.

Наиболее частой причиной для недостаточного выделения ресурсов является выполнение операций ввода-вывода. Слишком большое количество таких задач может навредить системе, и в какой-то момент общее число процессов ввода-вывода может превысить порог, с которым можно комфортно работать. Это особенно применимо к дисковым операциям ввода-вывода.

Если процессы занимают много памяти, вам также стоит задуматься о снижении выделяемых для них ресурсов, чтобы не уронить производительность по причине использования кеша памяти. В худшем случае, когда обнаружится недостаток памяти, операционная система может даже начать «прибивать» процессы, мешающие ей нормально функционировать.

Типичной ситуацией, когда можно избыточно выделить ресурсы, является процесс ожидания сети. Обычно это означает, что процессы боль-

шую часть времени заняты не будут, а значит, и ресурсы процессора будут свободны.

Парадокс состоит в том, что избыточное выделение ресурсов иногда может быть полезно для некоторых операций, связанных с процессором, например когда в течение операции процессорные мощности используются не постоянно, а ярко выраженными пиками, или когда длительное время занимает сам процесс подготовки к запуску операции.

Функция отслеживания прогресса `report_progress` осталась практически без изменений: она вызывает функцию обратного вызова периодически для отслеживания состояния прогресса. При этом вызов `Future.done` был заменен на `AsyncResult.ready`:

```
def report_progress(map_returns, tag, callback):
    done = 0
    num_jobs = len(map_returns)
    while num_jobs > done:
        done = 0
        for ret in map_returns:
            if ret.ready():
                done += 1
        sleep(0.5)
        if callback:
            callback(tag, done, num_jobs - done)
```

Теперь вы можете запустить измененный код, и он будет работать. Но достаточно ли быстро?

3.4.4. Передача данных порциями

Чтобы ответить на вопрос о быстродействии предложенного решения, нужно сравнить его с чем-то. Как мы уже видели в предыдущей главе и еще не раз убедимся в этой книге в дальнейшем, *порционирование* (chunking) данных при записи на диск может значительно повысить эффективность решения. Подойдет ли подобная техника для сокращения вычислительных расходов и межпроцессной коммуникации?

Для ответа на этот вопрос внесем небольшое изменение в архитектуру нашего фреймворка MapReduce. Добавим фазу *разделения* (splitting) данных на начальной фазе, чтобы дальше данные передавались не в виде единого целого, а по частям. На рис. 3.5 показан новый шаг, ответственный за порционирование данных.

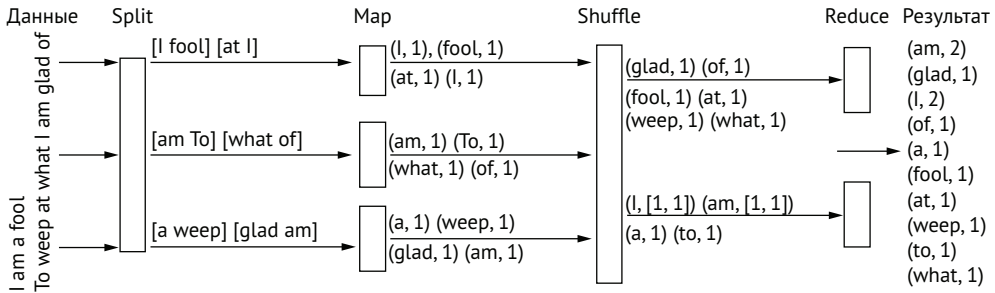


Рис. 3.5. Фреймворк map_reduce с разбиением данных (в нашем случае порционированием)

В нашем случае операция порционирования данных оказалась очень простой, тогда как в продвинутых фреймворках MapReduce на этом этапе может выполняться большой пласт оптимизационных работ. Взглянем на код, передающий порционированные данные на обработку и собирающий их обратно (код находится в файле 03-concurrency/sec4-multiprocess/chunk_mp_mapreduce.py):

```
def chunk(my_iter, chunk_size):
    chunk_list = []
    for elem in my_iter:
        chunk_list.append(elem)
        if len(chunk_list) == chunk_size:
            yield chunk_list
            chunk_list = []
    if len(chunk_list) > 0:
        yield chunk_list

def chunk_runner(fun, data):
    ret = []
    for datum in data:
        ret.append(fun(datum))
    return ret

def chunked_async_map(pool, mapper, data, chunk_size):
    async_returns = []
    for data_part in chunk(data, chunk_size):
        async_returns.append(pool.apply_async(
            chunk_runner, (mapper, data_part)))
    return async_returns
```

Порционирующий генератор, разбивающий итератор на списки заданного размера

Эта функция запускается в процессе пула для распаковки порционированного списка

Адаптируем функцию для передачи операций в пул посредством промежуточного диспетчера распаковки

Здесь мы вызываем функцию порционирования

Запускаем промежуточную функцию, а не окончательную

Функция `chunked_async_map` выполняет распределение задач в пуле. В ней происходит обращение к генератору `chunk`, разбивающему исходные данные на порции размером `chunk_size`. Обратите внимание, что мы больше не вызываем целевую функцию напрямую.

мую. Первым делом запускается функция `chunk_gunner`, проходящая по порции данных и вызывающая нужную функцию, переданную в виде параметра `fun`.

Вы могли бы подумать, что можно было реализовать генератор `chunk` более простым способом, например как показано ниже:

```
def chunk0(my_list, chunk_size):
    for i in range(0, len(my_list), chunk_size):
        yield my_list[i:i + chunk_size]
```

Проблема с такой реализацией состоит в том, что она требует наличия `len(my_list)`, что ограничивает тип данных входного потока списком. Но итератор можно сделать ленивым, чтобы он не занимал много места в памяти и, возможно, требовал меньше ресурсов центрального процессора.

Теперь пришло время изменить основную функцию фреймворка MapReduce:

```
def map_reduce(
    pool, my_input, mapper, reducer, chunk_size, callback=None):
    map_returns = chunked_async_map(pool, mapper, my_input, chunk_size)
    report_progress(map_returns, 'map', callback)
    map_results = []
    for ret in map_returns:
        map_results.extend(ret.get())  ← Использовали метод extend
    distributor = defaultdict(list)    ← вместо append
    for key, value in map_results:
        distributor[key].append(value)
    returns = chunked_async_map(
        pool, reducer, distributor.items(), chunk_size)  ←
    report_progress(returns, 'reduce', callback)
    results = []
    for ret in returns:
        results.extend(ret.get())
    return results
```

Добавили размер порции данных в виде параметра chunk_size

Добавили размер порции данных в виде параметра

Единственной загвоздкой стало то, что в результате каждого вычисления нам возвращается не элемент, а список элементов. В связи с этим мы перешли на использование метода `extend` вместо `append`.

Для проверки скорости выполнения мы воспользуемся текстом романа Льва Толстого «Анна Каренина», доступным на сайте [gutenberg.org](http://gutenberg.org/files/1399/1399-0.txt) по адресу <http://gutenberg.org/files/1399/1399-0.txt>. Ниже приведен вызывающий код:


```

words = [word
          for word in map(lambda x: x.strip().rstrip(),
                          ' '.join(open('text.txt', 'rt', encoding='utf-8').
readlines()).split(' '))
          if word != '']

chunk_size = int(sys.argv[1])
pool = mp.Pool()

counts = map_reduce(pool, words, emitter, counter, chunk_size, reporter)
pool.close()
pool.join()

for count in sorted(counts, key=lambda x: x[1]):
    print(count)

```

← Считываем весь текст в список

← Размер порции берем из параметра командной строки

← Распечатываем количество вхождений в порядке возрастания

Я прогнал этот текст со следующими размерами порций: 1, 10, 100, 1000 и 10 000. Результаты в секундах показаны в табл. 3.1.

Таблица 3.1. Время выполнения для разных размеров порций

Размер порции	Время (с)
1	114,2
10	12,3
100	4,3
1000	3,1
10 000	3,1

Числа в таблице говорят сами за себя: порционирование данных помогло значительно повысить быстродействие нашего фреймворка. Вообще, порционирование данных является очень эффективным приемом оптимизации процессов, и мы еще не раз будем обращаться к нему в следующих главах.

СОВЕТ. Если вы пользуетесь методом `map` объекта `Pool`, то можете не тратить лишние силы на реализацию порционирования данных – такая возможность встроена в этот метод при помощи параметра `chunksize`. То же самое касается методов `map_async` и `imap`. При использовании других библиотек параллельного программирования обязательно интересуйтесь, реализовано ли в них порционирование. Зачастую вам не придется писать реализацию самостоятельно.

Разделяемая память

Альтернативой показанному здесь решению на основе неявной передачи сообщений может быть система с *разделяемой памятью* (shared memory). Модели с общей памятью, реализованные во встроенных библиотеках Python, использовать бывает небезопасно, и это не секрет. Именно поэтому мы не будем говорить о них в этой книге. Если вам необходимо использовать решение на основе разделяемой памяти, вероятно, вам в любом случае придется переписывать его на более низкоуровневом языке. Позже в этой книге при обсуждении интеграции языков программирования низкого уровня с Python мы вернемся к теме разделяемой памяти.

3.5. Собираем все воедино: асинхронный многопоточный и многопроцессный сервер MapReduce

В этой главе мы опробовали различные подходы и их комбинации с использованием принципов параллелизма, многопоточности, а также синхронности и асинхронности. Теперь попытаемся выбрать наиболее оптимальное сочетание стратегий и подходов, объединив их в максимально быстрое решение для нашего фреймворка MapReduce. Давайте вспомним вводные сведения для нашей задачи: все данные хранятся в памяти, все действия должны выполняться на одном компьютере, и наша система должна обрабатывать запросы от множества клиентов, включая ботов. В заключительном разделе главы мы наконец разработаем финальное решение в виде многопроцессного фреймворка MapReduce с асинхронным сервером TCP, обслуживающим множество входящих запросов.

Мы уже создали два фрагмента, представляющие реализацию фреймворка MapReduce с возможностью порционирования данных и клиента, который мы написали еще в разделе 3.1. Эти фрагменты можно использовать как есть. Как написать остальные составляющие решения и собрать их вместе, обсудим далее в этом разделе.

3.5.1. Архитектура высокопроизводительного решения

При создании архитектуры приложения мы будем опираться на схему, показанную на рис. 3.6. Клиентская часть, принимающая запросы, будет реализована в асинхронном виде. Задачи будут отправляться в другой поток посредством очереди. Этот поток будет нести ответственность за управление многопроцессным пулом MapReduce.

Сервер TCP мы реализуем внутри асинхронного цикла. Будет у нас и второй поток, в зону ответственности которого будет входить только управление пулом процессов MapReduce.

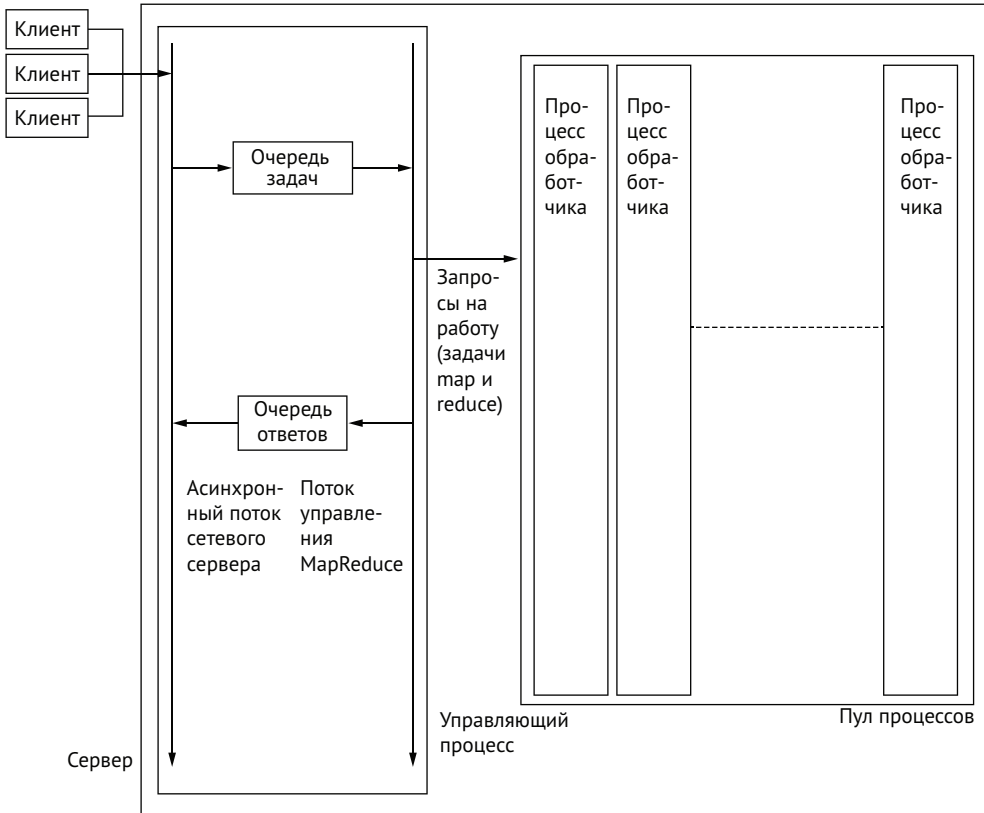


Рис. 3.6. Итоговая архитектура сервера MapReduce

Коммуникация между потоками будет осуществляться с помощью объекта *Queue* из модуля *queue*. В точке входа будет создаваться асинхронный сервер и поток, управляющий пулом MapReduce. Код можно найти в файле `03-concurrency/sec5-all/server.py`:

```
import asyncio
import marshal
import multiprocessing as mp
import pickle
from queue import Empty, Queue
import threading
import types

import chunk_mp_mapreduce as mr

work_queue = Queue()
results_queue = Queue()
results = {}
```

```
def worker():
    pool = mp.Pool()
```

Эта функция вызывается
в отдельном потоке

Пул создается внутри
потока обработчика

```

while True:
    job_id, code, data = work_queue.get()
    func = types.FunctionType(code, globals(), 'mapper_and_reducer')
    mapper, reducer = func()
    counts = mr.map_reduce(pool, data, mapper, reducer, 100, mr.reporter)
    results_queue.put((job_id, counts))
pool.close()
pool.join()

async def main():
    server = await asyncio.start_server(accept_requests, '127.0.0.1', 1936)
    worker_thread = threading.Thread(target=worker)
    worker_thread.start()
    async with server:
        await server.serve_forever()

asyncio.run(main())

```

Поток обработчика ожидает какую-то работу

Результаты собираются в очередь ответов

Запуск потока

Поток готов и направлен на соответствующую функцию-обработчик

В нашей точке входа – функции `main` – происходит подготовка асинхронной инфраструктуры, а также создание и запуск потока, который будет управлять пулом MapReduce, реализованным в функции `worker`. В функции `worker` создается пул процессов и происходит обработка запросов от асинхронного сервера. Коммуникация осуществляется при помощи очереди, работающей по принципу FIFO. Модуль `queue` обеспечивает синхронизированность очередей при помощи механизмов блокировки, чтобы потоки не нарушили консистентность данных. У нас есть две очереди: одна хранит полученные задания (работу), а вторая – возвращаемые результаты. Все операции внутри функции `worker` являются блокирующими, поскольку при инициализации приложения нам нечего обрабатывать, – клиентские запросы управляются асинхронной частью программы.

ПРИМЕЧАНИЕ. Очереди также отлично подходят для коммуникации при использовании многопроцессности вместо многопоточности. В модуле `multiprocessing` существует специальный класс `Queue`, созданный с этой целью, поскольку межпроцессная коммуникация выполняется сложнее, чем межпоточная. Некоторая часть работы здесь перекладывается на пользователя. В очередях могут храниться только объекты, которые могут сериализовываться при помощи модуля `pickle`. Кроме того, использование этого модуля и сам характер межпроцессных взаимодействий могут негативно сказаться на быстродействии решения.

Давайте начнем с отправки задач (работ). Асинхронная часть кода выглядит так:

```

async def submit_job(job_id, reader, writer):
    writer.write(job_id.to_bytes(4, 'little'))
    writer.close()
    code_size = int.from_bytes(await reader.read(4), 'little')
    my_code = marshal.loads(await reader.read(code_size))
    data_size = int.from_bytes(await reader.read(4), 'little')
    data = pickle.loads(await reader.read(data_size))
    work_queue.put_nowait((job_id, my_code, data))

```

Пишем данные в очередь работ без блокировки

Наконец мы дошли до функции, которая делает что-то полезное. В функции `submit_job` происходит отправка работы в очередь `work_queue`, которая впоследствии будет подхвачена потоком с функцией `worker`. Мы воспользовались методом `put_nowait`, чтобы избежать возможных блокировок при сохранении результатов. В нашем случае этого быть не должно, поскольку наша очередь была создана без каких-либо ограничений на размер. Но может так случиться, что в будущем здесь могут использоваться очереди с определенными ограничениями, что нужно учитывать в реализации.

Оставшаяся часть асинхронного кода выглядит так, как показано ниже:

```

def get_results_queue():
    while results_queue.qsize() > 0:
        try:
            job_id, data = results_queue.get_nowait()
            results[job_id] = data
        except Empty:
            return

```

Проверяем размер очереди, чтобы понять, есть ли входящие данные

Исключение на случай пустой очереди

Читаем ответы из очереди результатов в неблокирующем режиме

```

async def get_results(reader, writer):
    get_results_queue()
    job_id = int.from_bytes(await reader.read(4), 'little')
    data = pickle.dumps(None)
    if job_id in results:
        data = pickle.dumps(results[job_id])
        del results[job_id]
    writer.write(len(data).to_bytes(4, 'little'))
    writer.write(data)

async def accept_requests(reader, writer, job_id=[0]):
    op = await reader.read(1)
    if op[0] == 0:
        await submit_job(job_id[0], reader, writer)
        job_id[0] += 1
    elif op[0] == 1:
        await get_results(reader, writer)

```

Функция `accept_requests` осталась точно такой же, как и в первом разделе, и здесь мы привели ее лишь для полноты записи.

В функции `get_results` появилась новая строчка в начале – с вызовом функции `get_results_queue`, ответственной за проверку окончания работы фреймворка MapReduce и сбор результатов в словарь `results`. Стоит упомянуть, что проверка размера очереди с помощью метода `qsize` является приблизительной, так что нам необходимо учесть возможность отсутствия элементов в очереди и избежать блокировок при ожидании сообщений.

Блокировка и низкоуровневая синхронизация с многопоточностью и многопроцессностью

Избегайте использования низкоуровневых примитивов для осуществления блокировок. Существует немало примитивов синхронизации, поддерживаемых библиотеками `threading` и `multiprocessing`, включая блокировки и семафоры, а также некоторые другие. Но дело в том, что, если вам необходимо использовать эти низкоуровневые конструкции, вероятно, пришло время задуматься о переписывании всего кода на языке более низкого уровня. Мы обсудим подобные механизмы далее в этой книге, когда будем говорить о реализации фрагментов кода вне стандартного Python.

Распространенным, хотя и не связанным с производительностью напрямую примитивом для осуществления коммуникаций в многопроцессном окружении является *Pipe*, позволяющий взаимодействовать с внешними приложениями с использованием стандартных каналов ввода и вывода.

3.5.2. Создание надежной версии сервера

До сих пор мы практически не обращали внимания на возможные ошибки и неожиданный ввод в нашем приложении. Сейчас попробуем сделать наш фреймворк более устойчивым к отказам и непредвиденным ситуациям. Конечно, это скажется на объеме кода. Мы убедимся, что при остановке сервера наш асинхронный сервер также будет аккуратно остановлен, что подразумевает закрытие потока обработчика и пула.

В функции `main` мы предусмотрим следующие моменты:

- проверим ввод клиента на предмет прерывания, обычно вызываемого сочетанием клавиш **Ctrl+C**, и выполним необходимую очистку;
- поскольку наш асинхронный сервер теперь может быть приостановлен, нужно перехватить и это событие;
- озаботимся наличием системы оповещения рабочего потока о необходимости произвести очистку.

Ниже приведены важные фрагменты реализации надежной версии нашего сервера. Полный код можно найти в файле 03-concurrency/sec5-all/server_robust.py):

```
import signal
from time import sleep as sync_sleep

def handle_interrupt_signal(server):
    server.close()
    while server.is_serving():
        sync_sleep(0.1)

def init_worker():
    signal.signal(signal.SIGINT, signal.SIG_IGN)

async def main():
    server = await asyncio.start_server(accept_requests, '127.0.0.1', 1936)
    mp_pool = mp.Pool(initializer=init_worker)
    loop = asyncio.get_running_loop()
    loop.add_signal_handler(signal.SIGINT, partial(
        handle_interrupt_signal, server=server))
    worker_thread = threading.Thread(target=partial(worker, pool=mp_pool))
    worker_thread.start()
    async with server:
        try:
            await server.serve_forever()
        except asyncio.exceptions.CancelledError:
            print('Server cancelled')
            work_queue.put((-1, -1, -1))
            worker_thread.join()
    mp_pool.close()
    mp_pool.join()
    print('Bye Bye!')
```

Определяем обработчик сигналов для прерываний

Запрашиваем остановку сервера

Ждем, когда сервер обработает все запросы

Игнорируем сигнал прерывания, чтобы убедиться, что он не распространяется на пул

Обеспечиваем инициализацию пула процессов

Добавляем обработчик сигналов к нашему асинхронному серверу

Перехватываем прерывания и информируем пользователя

Посылаем значение -1, которое будет интерпретировано обработчиком как завершение работы

Ожидаем окончания работы всех потоков

Если вы взглянете на функцию `main`, то обнаружите, что теперь пул процессов создается именно здесь для повышения эффективности, а каждый процесс инициализируется с помощью отдельной функции `init_worker`. Причина в том, что мы не хотим, чтобы пул прерывался по нажатию сочетания клавиш **Ctrl+C**, а сигнал распространяется на весь пул. С этой целью мы воспользовались библиотекой `signal` и с ее помощью проинструктировали каждый процесс в пуле игнорировать (`signal.SIG_IGN`) сигнал прерывания (`signal.SIGINT`).

Нам необходимо, чтобы сигнал прерывания перехватывал наш главный поток и обрабатывал его надлежащим образом. Поскольку у нас есть потребность контролировать асинхронный код из сигнала, придется воспользоваться методом `add_signal_handler` объекта `loop`. На вход методу нужно передать объект сервера, и мы делаем это с помощью функции `partial`. Обработчик

`handle_interrupt_signal` останавливает сервер и дожидается, когда он обработает все запросы, поскольку остановка сервера может занять какое-то время.

Теперь при запуске асинхронного сервера мы должны учитывать возможные прерывания и перехватывать нужные исключения. Наконец, нам необходимо позаботиться об очистке отслеживающего потока. Поскольку сигнал передается только главному потоку, придется сделать это с помощью какого-то механизма коммуникации. Мы просто решили послать работу с `job_id`, равным `-1`.

Управление ошибками и исключениями в многопоточном и многопроцессном коде

Отладка кода с множеством потоков или процессов бывает серьезно затруднена даже при условии использования простейших моделей межпроцессорного взаимодействия. Мы здесь не стали углубляться в эту тему и предположили для простоты, что архитектура у нас выстроена исправно. Если вы используете конкурентные вычисления в своих решениях, то должны озаботиться хорошей системой логирования, которая в случае чего поможет с обнаружением ошибок. Вы всеми путями должны пытаться убедиться в том, что проблемы не связаны с конкурентностью. Для этого бывает полезно запустить проблемный код вместо многопоточного или многопроцессного окружения в одном потоке или процессе. Допустим, вы можете временно заменить инструкцию `multiprocessing.Pool.map` на простое выражение `list(map)`.

Также нам необходимо обеспечить явное закрытие потока обработчика, как показано ниже:

```
def worker(pool):
    while True:
        job_id, code, data = work_queue.get()
        if job_id == -1:
            break
        func = types.FunctionType(code, globals(), 'mapper_and_reducer')
        mapper, reducer = func()
        counts = mr.map_reduce(pool, data, mapper, reducer, 100,
                               mr.reporter)
        results_queue.put((job_id, counts))
    print('Worker thread terminating')
```

Покидаем цикл, если значение переменной `job_id` равно `-1`

Заключение

- Асинхронный подход может оказаться чрезвычайно эффективным при необходимости одновременно обрабатывать множество запросов и условии, что расходы на коммуникацию и объемы вычислений будут достаточно малы. Эти требования хорошо сочетаются с работой веб-серверов.

- Python – язык медленный, и это, в частности, касается его флагманской реализации. Этот факт лишний раз говорит в пользу использования принципов параллельного программирования.
- Работа с потоками в Python по умолчанию не лучшим образом сказывается на производительности создаваемых решений. *Глобальная блокировка интерпретатора* (Global Interpreter Lock – GIL) подразумевает, что в любой момент времени активным может быть лишь один поток. В то же время другие реализации Python, в числе которых IronPython, не используют GIL, а значит, многопоточный код в них может работать параллельно.
- Принципы многопоточного программирования могут оказаться довольно полезными в процессе разработки архитектуры приложений. Не сбрасывайте их со счетов, даже если они не ведут к повышению производительности напрямую. Существуют и другие области вне рамок данной книги, где они могут пригодиться.
- С помощью модуля `multiprocessing` вы можете воспользоваться всеми ресурсами центрального процессора в Python, даже не прибегая к использованию сторонних техник.
- Как правило, рекомендуется сохранять степень гранулирования вычислений и не злоупотреблять коммуникациями, которые способны замедлить работу вашего решения. При осуществлении межпроцессной коммуникации важно быть уверенным в том, что связанные с ней накладные расходы не станут узким местом в отношении производительности приложения.
- Избегайте использования разделяемой памяти и низкоуровневых блокировок при разработке параллельного кода. Если без них вам не обойтись, лучше будет отдать предпочтение последовательному коду, написанному на языке низкого уровня. Отладка параллельных решений со сложными шаблонами межпроцессной коммуникации может оказаться чрезвычайно сложной, поскольку обмен сообщениями в таких системах по большей части недетерминирован.

Высокопроизводительный NumPy

В этой главе мы обсудим следующие темы:

- исследование библиотеки NumPy с точки зрения производительности;
- использование представлений массивов NumPy для повышения вычислительной эффективности и экономии памяти;
- программирование на основе массивов;
- настройка NumPy для повышения эффективности вычислений.

Очень трудно переоценить важность библиотеки NumPy при анализе данных в Python. Думаю, название этой книги могло выглядеть и так: «Сверхскоростной Python с NumPy». С каким бы стеком вы ни работали, от NumPy вы никуда не денетесь. Используете pandas? NumPy! А может, scikit-learn? Все равно NumPy! Dask? NumPy! SciPy? NumPy! Matplotlib? NumPy! TensorFlow? NumPy! Если вы занимаетесь анализом данных в Python, почти любой ваш ответ на вопрос об использовании библиотек будет включать NumPy.

NumPy представляет собой библиотеку в составе Python для работы с *многомерными массивами* (multidimensional array), такими

как матрицы, у которых есть два измерения, и эффективного манипулирования этими массивами. Работа с массивами в NumPy производится действительно быстро и эффективно по причине того, что функциональная основа этой библиотеки написана на таких низкоуровневых языках программирования, как Fortran и C. Многие задачи, связанные с анализом данных, могут быть смоделированы с применением многомерных массивов, и именно поэтому библиотека NumPy получила столь широкое распространение в этой области.

С учетом такой популярности NumPy в среде анализа данных с помощью Python мы будем затрагивать эту библиотеку при обсуждении различных тем и в других главах, а именно:

- в главе 5 мы обсудим тему векторизации функций с использованием расширения Cython;
- в главе 6 поговорим о внутренней организации памяти в массивах;
- также в главе 6 мы коснемся темы использования библиотеки NumExpr для быстрого вычисления выражений;
- в главах 8 и 10 поговорим об использовании массивов, размер которых превышает объем доступной памяти, а также об эффективном хранении массивов;
- наконец, в главе 9 мы затронем тему использования ресурсов графического процессора (GPU) при обработке массивов.

Начнем эту главу с напоминания о том, что из себя представляет библиотека NumPy. Хотя при написании книги я предполагал, что вы знаете, как работать с NumPy, зачастую бывает, что программисты на Python не взаимодействуют с этой библиотекой напрямую. К примеру, вы можете активно работать с пакетами pandas или Matplotlib и при этом не выполнять никаких действий с объектами NumPy явным образом. Мы пройдемся по азам NumPy с точки зрения производительности. Если вы чувствуете, что вам нужны более базовые сведения об этой библиотеке, вы всегда можете воспользоваться официальной инструкцией по адресу <https://numpy.org/devdocs/user/quickstart.html>. Также на этом сайте есть неплохой перечень образовательных ресурсов: <https://numpy.org/learn>.

После беглого знакомства с основами библиотеки мы поговорим о технике программирования на основе массивов, в которой действия применяются одновременно более чем к одному атомарному значению. Это очень эффективный и в то же время элегантный подход к написанию программного кода. В заключительной части главы мы заглянем во внутреннее устройство архитектуры NumPy и узнаем, какие настройки можно применить для повышения эффективности работы библиотеки.

4.1. Библиотека NumPy с точки зрения производительности

В этом разделе и далее в главе мы будем изучать ключевые концепции и техники библиотеки NumPy на практическом примере, связанном с разработкой базовых манипуляций с изображениями. В первом приближении изображения представляют собой двумерные массивы данных (так называемые матрицы), а значит, легко поддаются обработке с помощью библиотеки NumPy. Представьте, что вы разрабатываете программное обеспечение для обработки изображений. Напоминаем, что в этом разделе мы пройдемся по основам NumPy с точки зрения производительности, так что, даже если у вас есть опыт работы с этой библиотекой, вы легко можете открыть для себя что-то новое.

4.1.1. Копии и представления существующих массивов

Первое, что мы сделаем, это прочитаем изображение из файла и выполним его вращение. При этом мы будем пользоваться исключительно средствами библиотеки NumPy, а не *Pillow*, с помощью которой будем читать изображение. Кроме того, мы будем манипулировать как копией массива, так и его представлением, чтобы можно было сравнить эффективность обоих методов.

Представления (view) массивов используют в качестве источника данных уже заполненную исходным массивом область памяти, но при этом могут интерпретировать ее по-разному. Таким образом, с помощью представлений можно добиться гораздо большей производительности, однако, как мы увидим далее, воспользоваться ими можно далеко не всегда.

Начнем с загрузки изображения логотипа издательства Manning Publications и получения на его основании массива NumPy. Обратите внимание, что такие операции, как вращение изображения, можно представить просто в виде альтернативной интерпретации исходного массива, в которой колонки становятся строками, а строки – колонками. Именно так и работают представления массивов NumPy – они строятся на основе измененной интерпретации исходных данных. Давайте разделим процесс на отдельные шаги, чтобы вы могли досконально понять все происходящее. Код можно найти в файле 04-numpy/sec1-basics/image_processing.py:

```
import sys
import numpy as np
from PIL import Image
```

```
image = Image.open("../manning-logo.png").convert("L")
print("Image size:", image.size)
```

Метод `convert("L")` переводит изображение в черно-белый вид

```
width, height = image.size
image_arr = np.array(image)
print("Array shape, array type:", image_arr.shape, image_arr.dtype)
print("Array size * item size: ", image_arr.nbytes)
print("Array nbytes:", image_arr.nbytes)
print("sys.getsizeof:", sys.getsizeof(image_arr))
```

Мы воспользовались библиотекой Pillow для загрузки логотипа Manning, показанного на рис. 4.1, и перевода его в черно-белый вид. Каждый пиксель изображения представлен в виде *беззнакового байта* (unsigned byte), а размеры исходного изображения составляют 182×45. Вывод будет следующим:

```
Image size: (182, 45)
Array shape, array type: (45, 182) uint8
Array size * item size: 8190
Array nbytes: 8190
sys.getsizeof: 8302
```

После этого мы с помощью функции *np.array* получили массив, представляющий загруженное изображение. Такая возможность обусловлена не тем, что библиотека NumPy умеет работать с изображениями, а тем, что загруженный объект изображения реализует интерфейс *__array__ interface__*, который использует библиотека NumPy при создании представления массива.

На экран мы вывели значение свойства *shape* нашего массива, равное (45×182). Обратите внимание, что принятые для изображений нормы следования высоты за шириной в библиотеке NumPy нарушены в угоду математическим принципам, согласно которым необходимо указывать сначала количество строк, а затем количество столбцов. Этот нюанс имеет гораздо большее значение в сравнении с тем, как звучит, и вы прочувствуете это в следующих разделах, где мы будем обсуждать представления массивов. А всю полноту диссонанса вы ощутите только при чтении главы 6, в которой мы затронем вопросы представления данных в памяти.

Также мы вывели на экран тип данных, использующийся в массиве, – *uint8* (беззнаковое целое число из 8 бит, или 1 байта). Этого типа данных вполне достаточно для хранения информации о черно-белом изображении. Кроме того, мы попытались узнать, сколько места в памяти занимает наш массив. Это можно сделать, перемножив количество элементов массива на осях ($45 * 182 = 8190$) или воспользовавшись свойством *nbytes* напрямую.

Наконец, с помощью уже знакомой нам функции *getsizeof* мы вывели размер, занимаемый *объектом* массива в памяти. Он включает в себя размер массива (8190), а также накладные расходы Python и NumPy и метаданные (в сумме размер составил 8302 байта).

Итак, мы определили два размера массива: с накладными расходами и без них. Теперь давайте попробуем перевернуть изображение вверх ногами. Это можно сделать, создав копию массива или просто изменив интерпретацию исходных данных. Отличный способ познакомиться с представлениями массивов. После двух переворотов изображения мы закрасим черным цветом одну его половину:

```
flipped_from_view = np.flipud(image_arr)
flipped_from_copy = np.flipud(image_arr).copy()
image_arr[:, :width//2] = 0
removed = Image.fromarray(image_arr, "L")
image.save("image.png")
removed.save("removed.png")

flipped_from_view_image = Image.fromarray(flipped_from_view, "L")
flipped_from_view_image.save("flipped_view.png")
flipped_from_copy_image = Image.fromarray(flipped_from_copy, "L")
flipped_from_copy_image.save("flipped_copy.png")
```

Переворот изображения по вертикали с использованием представления массива

Берем перевернутое изображение и создаем его копию

На рис. 4.1 показаны все четыре сохраненных изображения. Массив `flipped_from_view` создан на основе представления массива `image_arr`. Это означает, что после изменения исходного массива (`image_arr[:, :width//2] = 0`) массив `flipped_from_view` также изменится, поскольку в качестве источника он использует исходный объект.



Рис. 4.1. Четыре изображения: исходный логотип (`image.png`), закрашенный логотип (`removed.png`), а также вертикальные отражения массива на основе копии (`flipped_copy.png`) и представления (`flipped_view.png`)



Представления используют в качестве источника базовый массив, а копии – нет. Именно поэтому изменение исходного массива `image_arr` никак не отразилось на массиве `flipped_from_copy`. Стоит отметить, что метод `fromarray` создает копию исходного массива, в связи с чем изображения `image.png` и `removed.png` у нас разные. Если бы этот метод возвращал представление массива, в файлах хранились бы одинаковые изображения.

На рис. 4.2 представлены структуры данных, лежащие в основе наших изображений. Заметьте, что исходное изображение было уничтожено, а в массиве `image_arr` хранится закрашенное изображение.

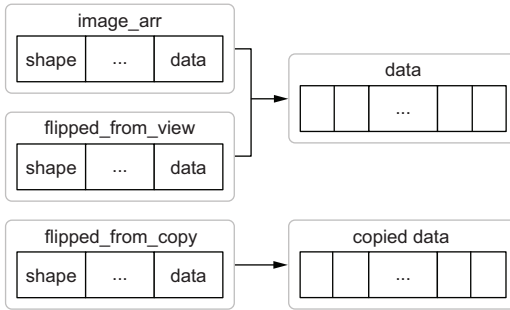


Рис. 4.2. В результате выполнения многих операций в NumPy возвращается новый объект, являющийся представлением исходного массива. Но иногда работа с представлениями может быть невозможна или нежелательна, и в таких случаях создаются копии массива

Бывают ситуации, когда работать с представлениями массивов нельзя или не хочется. К примеру, вам может быть не нужно закрашивать исходное изображение. В этом случае лучше будет создать копию, чтобы произведенные изменения не затронули исходный объект. Иногда же бывает невозможно получить данные в виде представления. Такой пример мы рассмотрим далее в этом разделе.

При работе с массивами вы всегда можете узнать, является ли один массив источником для другого, с помощью свойства `base`, как показано ниже:

```
print(flipped_from_copy.base, flipped_from_view.base)
print(flipped_from_view.base is image_arr)
print(flipped_from_view.base == image_arr)
```

Проверка на эквивалентность объектов

Проверка на поэлементное равенство элементов массивов

Вывод будет следующим:

```
None [[ 0 .... <long array>]]
True
[[ True True True ... True True True]]
```

Результат выражения `flipped_from_copy.base` интерпретируется как `None`, поскольку это самостоятельная отдельная копия. В то же время свойство `flipped_from_view.base` представляет собой матрицу. Эквивалентность двух объектов можно проверить при помощи оператора `is`. Но будьте осторожны: при использовании оператора двойного равенства (`==`) вы получите поэлементное сравнение двух массивов. Таким образом, если оператор `is` вернет значение `True`, то оператор `==` – массив значений `True`.

СОВЕТ. Не забывайте, что свойство `base` для любого представления возвращает не тот объект, на основе которого оно создано непосредственно, а первый объект в иерархии наследования. Таким образом, если `v2 = v1[:-1]`, а `v1 = arr[:-1]`, то оба выражения `v2.base is arr` и `v1.base is arr` вернут значение `True`, тогда как выражение `v2.base is v1` вернет `False`.

Как мы уже видели, объекты NumPy хранят целый набор метаданных в свойствах вроде `shape` и `dtype`. Исходный массив данных объекта хранится в поле `data` и представляет собой встроенный в Python объект `memoryview`. Класс `memoryview` позволяет разработчикам Python получить доступ к базовому функционалу для работы с блоками выделенной памяти, хранящими данные одного типа. Этот функционал включает в себя такие привычные операции, как индексирование, срезы и разделение памяти.

Существует возможность напрямую запросить, используют ли массивы NumPy одну и ту же область памяти, с помощью функции `np.shares_memory`. Это более общий способ по сравнению с представлениями, поскольку объекты `memoryview` могут одновременно принадлежать другим объектам или массивам без использования для их создания представлений:

```
print(np.shares_memory(image_arr, flipped_from_copy),  
      np.shares_memory(image_arr, flipped_from_view))
```

Функция `np.shares_memory` вернет значение `False` для аргументов `image_arr` и `flipped_from_copy`, поскольку мы здесь имеем дело с копией данных. В то же время для `image_arr` и `flipped_from_view` она вернет значение `True`. Обычно если свойство `base` у объектов общее, то они используют общую память. Но обратное не всегда верно: память может использоваться объектами совместно без единой основы.

СОВЕТ. Определение факта совместного использования памяти двумя массивами – задача не из легких. В сложных сценариях эта процедура может занимать немало времени, что лишает ее смысла. В таких ситуациях можно воспользоваться более быстрой функцией `may_share_memory`, которая строит догадку о том, используют ли массивы одну и ту же область памяти.

Вывод

Из этого раздела вы должны уяснить, что представления массивов зачастую бывает использовать гораздо более выгодно по сравнению с копиями. Это обусловлено двумя основными причинами. Первая состоит в том, что сам процесс копирования массива может отнимать немало времени и вычислительных ресурсов, тогда как создание представления сводится просто к получению новой ссылки на исходные данные. Вторая, и более важная, заключается в том, что при копировании массива вы фактически удваиваете используемую им память, что недопустимо при работе с большими данными. В целом же вам просто не стоит забывать, что при изменении представления будут затронуты все массивы, разделяющие с ним память.

Давайте прогоним небольшой пример, который поможет разобраться с эффективностью разных подходов. Мы будем создавать массив переменного размера и замерять время создания его копии и представления. Эти данные мы собрали в табл. 4.1:

```
import sys
import timeit
```

Если вы используете iPython,
помните, что вам доступна
инструкция %timeit

```
import numpy as np

for size in [1, 10, 100, 1000, 10000, 100000, 200000, 400000, 800000,
1000000]:
    print(size)
    my_array = np.arange(size, dtype=np.uint16)
    print(sys.getsizeof(my_array))
    print(my_array.data.nbytes)
    view_time = timeit.timeit(
        "my_array.view()",
        f"import numpy; my_array = numpy.arange({size})")
    print(view_time)
    copy_time = timeit.timeit(
        "my_array.copy()",
        f"import numpy; my_array = numpy.arange({size})")
    print(copy_time)
    copy_gc_time = timeit.timeit(
        "my_array.copy()",
        f"import numpy;
        import gc; gc.enable(); my_array = numpy.arange({size})")
    print(copy_gc_time)

print()
```

Таблица 4.1. Сравнение времени создания представления и копии массива

Размер массива	Объем массива в памяти (б)	Время создания представления	Время создания копии
1	2	0,171	0,281
10	20	0,137	0,259
100	200	0,139	0,286
1000	2000	0,162	0,502
10 000	20 000	0,142	2,275
100 000	200 000	0,138	31,257
200 000	400 000	0,152	67,005
400 000	800 000	0,144	354,287
800 000	1 600 000	0,177	547,843
1 000 000	2 000 000	0,142	729,966

Удвоенный объем памяти, требующийся для создания копии массива, не вызывает вопросов, поскольку мы делаем физический дубликат нашего объекта¹. Что касается времени, требуемого для создания представления и копии, то тут не все так очевидно. Мы могли бы предположить, что время, затрачиваемое на создание копии массива, должно увеличиваться линейно с ростом его объема. Но если взглянуть на таблицу, становится понятно, что зависимость здесь не линейная. В главе 6 мы еще вернемся к этому любопытному факту.

Вывод

Как видите, представления массивов в большинстве случаев обходятся более выгодно как в плане расходования памяти, так и с точки зрения времени создания. Таким образом, в общем случае вы должны всегда стремиться к решению своих задач при помощи представлений. Единственным недостатком представлений является то, что их, к сожалению, не всегда можно применять. Бывают случаи, когда достойной альтернативы созданию копии массива просто не существует. Во всех остальных ситуациях вы должны использовать всю мощь механизма создания представлений в NumPy. А для этого необходимо глубже понимать устройство представлений. Этому мы и посвятим следующий раздел.

4.1.2. Внутреннее устройство представлений NumPy

Чтобы использовать представления массивов предельно эффективно, для начала нужно понять, как они устроены. Гибкость представлений обусловлена по большей части двумя характеристиками метаданных. С первой из них – свойством `shape` – мы уже встречались. Второе важное свойство, с которым мы ближе познакомимся совсем скоро, – это `strides`. Но сначала давайте рассмотрим несколько примеров, иллюстрирующих свойства `stride` и `shape`.

Начнем с определения массива, содержащего значения `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]` в виде беззнаковых целых чисел по 4 байта. Узнаем его значения свойств `stride` и `shape`, после чего изменим размерность массива:

```
import numpy as np
linear = np.arange(10, dtype=np.uint32)
```

В памяти массив данных располагается последовательно и непрерывно. Это означает, что за нулем в памяти будет следовать единица, двойка и т. д. На данном этапе это звучит довольно очевидно, но скоро все изменится.

¹ Для массивов маленького размера это не совсем так, поскольку накладные расходы могут составлять приличную часть их объема. Но применительно к большим массивам, которые нас в основном и интересуют, расчеты будут очень близки к реальной картине, а 96 байт расходов, связанных с Python и NumPy, утонут в общей массе.

ПРЕДУПРЕЖДЕНИЕ. С целью облегчения донесения материала мы решили несколько упростить реальные принципы размещения данных в памяти, хотя на данный пример это не повлияет. Далее мы будем встречаться с массивами, которые размещаются в памяти непоследовательно. Тонкости размещения данных в памяти мы придержим до главы 6. Если вы впервые сталкиваетесь с обсуждением последовательности хранения массивов данных в памяти, советуем не переходить сразу к главе 6, а дойти до нее своим ходом.

Теперь давайте построим матрицу размером 2×5 с теми же данными. После этого создадим еще одно представление массива, на этот раз размером 5×2 , путем транспонирования матрицы 2×5 . Давайте попробуем разобраться в связях между исходным массивом и его представлениями:

```
m2x5 = linear.reshape((2, 5))
print(np.shares_memory(linear, m2x5))

print("2x5", m2x5.shape)
print("2x5 corners", m2x5[0, 0], m2x5[0, 4], m2x5[1, 0], m2x5[1, 4])

m5x2 = m2x5.T
print(np.shares_memory(m2x5, m5x2))
print("5x2", m5x2.shape)
print("5x2 corners", m5x2[0, 0], m5x2[0, 1], m5x2[4, 0], m5x2[4, 1])
```

Первое, что нам нужно сделать, это убедиться, что массивы совместно используют память, ссылаясь на один и тот же исходный набор данных. Это позволит нам понять, что мы имеем дело с представлениями массивов, а не с копиями. Применение функции `np.shares_memory` показало, что массивы `linear`, `m5x2` и `m2x5` действительно делят память, что можно видеть на рис. 4.3. Мы также вывели на экран угловые точки для каждого представления, чтобы нам были хорошо понятны их границы.

Главный вопрос состоит в том, как библиотеке NumPy найти нужные данные в памяти. Свойства `shape` должно быть достаточно, чтобы отличить одномерный набор данных от двумерного. Но как отличить два двумерных массива с разными размерностями, обращающихся к одной и той же памяти? Здесь на помощь приходит свойство `strides`:

```
print(«linear», linear.strides)
print(«2x5 strides», m2x5.strides)
print(«5x2 strides», m5x2.strides)
```

Результаты будут следующими: 4, (20, 4) и (4, 20).

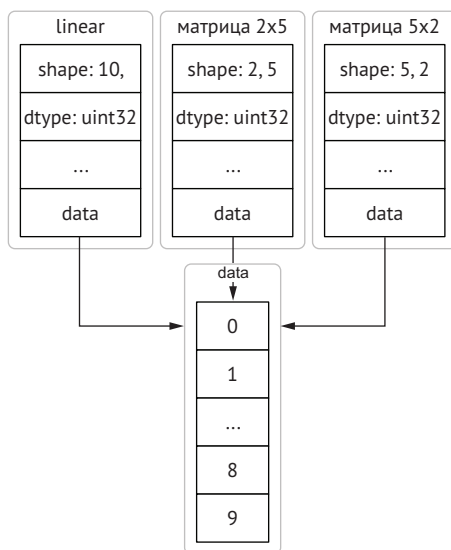


Рис. 4.3. Три массива обращаются к одним и тем же ячейкам памяти

Таким образом, свойство *strides* отвечает за то, на сколько байт необходимо продвинуться от текущего элемента в массиве для достижения следующего элемента в соответствующем измерении. Давайте проясним эту концепцию на примере трех описанных выше массивов.

Для массива *linear* свойство *strides* возвращает 4 или (4,). Это означает, что для перехода к следующему элементу в первом и единственном измерении вам необходимо сделать прыжок на 4 байта, что характеризует выбранный нами тип данных в массиве (пр. *uint32*). На рис. 4.4 показано, что при переходе к каждому следующему элементу в массиве вы фактически проходите расстояние, равное четырем байтам. Таким образом, для перехода к *i*-му элементу вам нужно сделать прыжок на *i**шаг, или *i**4 байта.

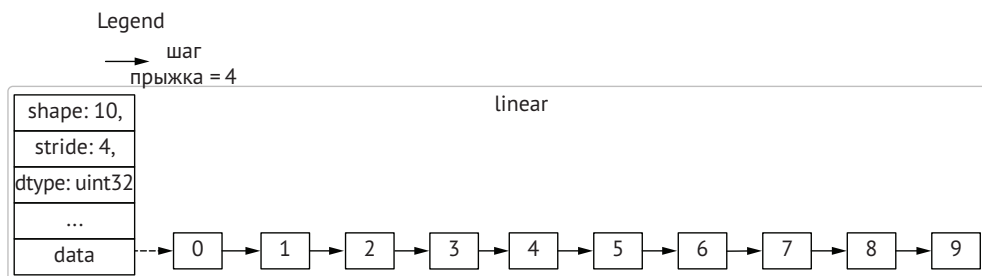


Рис. 4.4. Переход к соседнему элементу в одномерном массиве. Здесь у нас есть только один шаг, равный размеру типа данных

Все становится несколько сложнее при добавлении второго измерения. К примеру, если мы хотим в нашем массиве размером 2×5

добраться до элемента в соседней колонке, нам достаточно сделать один шаг, как показано на рис. 4.5. С учетом того, что каждый элемент у нас занимает 4 байта, мы должны сделать шаг длиной 4 байта. Но если нам необходимо добраться до следующей строки, то нам придется перепрыгивать текущий элемент и еще четыре, поскольку в строке у нас располагается пять элементов. В результате получаем пять размеров элемента, т. е. $5 * 4 = 20$. В такой интерпретации памяти доступ к элементу i, j можно получить с помощью выражения $\text{strides}[0]*i + \text{strides}[1]*j$ ($20*i + 4*j$).

Свойство `strides` для матрицы $m \times 5$ возвращает кортеж $(20, 4)$: это означает, что у нас есть два измерения, и для того, чтобы перепрыгнуть на следующую строку, вам необходимо переместиться вперед на 20 байт (пять колонок по 4 байта каждая). Следующее значение в строке при этом отстоит от текущего элемента всего на 4 байта.

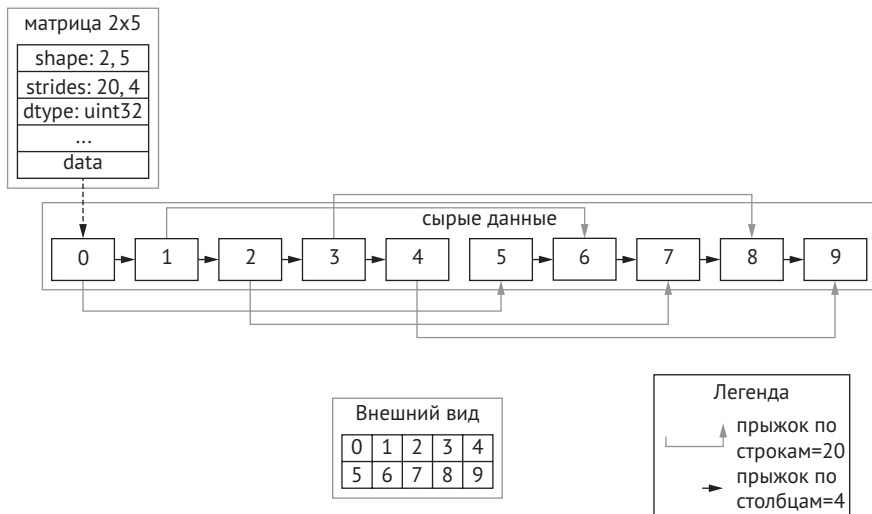


Рис. 4.5. Переход по элементам в матрице размером 2 5. Здесь у нас есть два шага: для перехода к элементам обоих измерений

Рисунок 4.6 должен еще больше прояснить ситуацию. Картина здесь может варьироваться в зависимости от внутреннего представления массива, о чем мы будем говорить в главе 6.

Многие операции NumPy по своей сути просто преобразовывают представления массивов. Например, обращение массива может быть отображено как представление:

```
back = linear[::-1]
print(«back», back.shape, back.strides, back[0], back[-1])
```

Обратите внимание, что шаг в данном случае стал равен -4 . NumPy может создавать представления с обратными проходами, как показано на рис. 4.7.

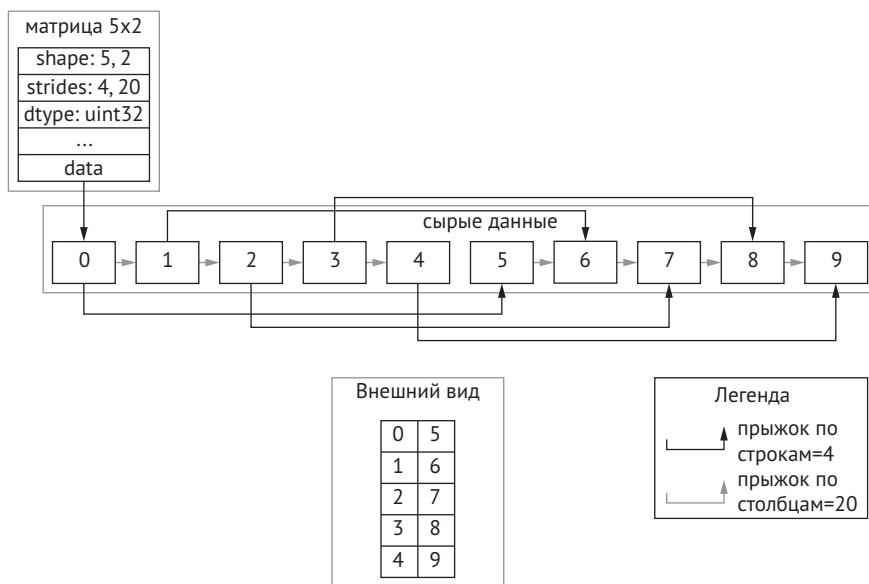


Рис. 4.6. Переход по элементам в матрице размером 2x5. Здесь у нас есть два шага: для перехода к элементам обоих измерений

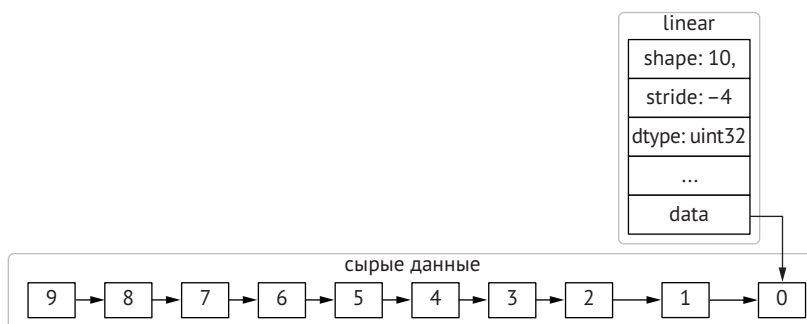


Рис. 4.7. Обращение одномерного массива приведет к появлению отрицательного шага

Похожий эффект можно наблюдать и при работе с двумерными массивами. Что станет с шагами массивов $m \times 5$ и $m \times 2$, если их обратить?

Воспроизводить результаты преобразований в виде представлений можно не всегда, поскольку здесь все зависит от возможности установить линейные связи между существующим и новым представлением. К примеру, возьмем матрицу размером 20×5 и оставим в ней каждую третью строку и каждый второй столбец, в результате получив матрицу размером 7×3 . После этого преобразуем полученную матрицу в одномерный массив. Все эти операции показаны ниже:

```

a100 = np.arange(100, dtype=np.uint8).reshape(20, 5)
a100_step_3_2 = a100[::3, ::2]
print(a100_step_3_2.shape, a100_step_3_2.strides)
print(np.shares_memory(a100, a100_step_3_2))
a100_step_3_2_linear = a100_step_3_2.reshape(21)
print(np.shares_memory(a100_step_3_2, a100_step_3_2_linear))

```

Матрица размером 7×3 может быть воспроизведена в виде представления, использующего общую память, со свойством `strides`, возвращающим кортеж (15, 2). Однако внешне гораздо более простой процесс преобразования полученной матрицы в одномерный массив приводит к утрате представления, в результате чего создается копия массива. Так называемая «прихотливая» индексация (fancy indexing) в NumPy всегда приводит к созданию копий массивов. Ниже приведен пример, в котором мы попеременно извлекаем элементы из первой и второй строки массива размером 2×5 :

```

m5x2 = np.arange(10).reshape(2, 5)
my_rows = [0, 1, 0, 1, 0]
my_cols = [0, 1, 2, 3, 4]
alternate = m5x2[my_rows, my_cols]

print(m5x2)
print(alternate)
print(np.shares_memory(m5x2, alternate))

```

Если вы подзабыли, как работает «прихотливая» индексация, напомним, что на вход мы принимаем списки индексов – по одному для каждого измерения в массиве – и возвращаем элементы, соответствующие этим индексам. В табл. 4.2 приведен исходный вид матрицы, из которой мы будем выбирать чередующиеся элементы.

Таблица 4.2. Исходная матрица

0	1	2	3	4
5	6	7	8	9

В массиве `alternate` у нас будут находиться элементы `[0 6 2 8 4]`, соответствующие перекрестным индексам переданных списков координат по строкам (`[0, 1, 0, 1, 0]`) и по столбцам (`[0, 1, 2, 3, 4]`). Функция `np.shares_memory` в этом случае вернет значение `False`.

Вывод

Первое, что нужно помнить, – это то, что работать с представлениями массивов может быть гораздо эффективнее по сравнению с копиями. Второе – что их можно применять так, как мы показали в этом разделе. Давайте применим полученные знания и навыки

при обработке изображений и увидим на практике достоинства и недостатки использования представлений.

4.1.3. Эффективное использование представлений

Итак, давайте попробуем поработать с нашим изображением с использованием операций, совместимых с представлениями массивов. И сейчас самое время вспомнить, почему мы говорим о работе с представлениями в отношении больших данных: при хранении объемных массивов может возникнуть дефицит памяти, и операции копирования могут негативно сказаться как на ее расходовании, так и на требующемся для их выполнения времени.

Начнем с зеркальных отражений изображения по вертикали и горизонтали:

```
import numpy as np
from PIL import Image

image = Image.open("../manning-logo.png").convert("L")
width, height = image.size
image_arr = np.array(image)
print("original array", image_arr.shape, image_arr.strides, image_arr.
dtype)
image.save("view_initial.png")

invert_rows_arr = image_arr[::-1, :]  ←
print("invert rows", invert_rows_arr.shape, invert_rows_arr.strides,
np.shares_memory(invert_rows_arr, image_arr))
Image.fromarray(invert_rows_arr).save("invert_x.png")

invert_cols_arr = image_arr[:, ::-1]
print("invert columns", invert_cols_arr.shape, invert_cols_arr.strides,
np.shares_memory(invert_cols_arr, image_arr))
Image.fromarray(invert_cols_arr).save("invert_y.png")
```

Мы можем зеркально отразить изображение путем обращения массива по одному из измерений. Если помните, в предыдущем разделе мы использовали функцию `flipud` для отражения изображения по горизонтали

На этом этапе вы должны уже очень легко читать такой код. Для массива, представляющего оригинальный логотип издательства Manning размером 182×24 , свойство `shape` вернет кортеж $(45, 182)$, а свойство `strides` – $(182, 1)$. Мы имеем дело с беззнаковыми целыми числами размером 1 байт, именно столько занимает у нас хранение информации об одном пикселе. При отражении изображения по горизонтали (т. е. по строкам) меняется только второе значение в кортеже `strides` – с 1 на -1 . Так же точно при отражении изображения по вертикали (т. е. по столбцам) меняется только первое значение со 182 на -182 .

Теперь давайте попробуем поворачивать наш логотип. При этом мы воспользуемся тремя разными подходами: изменением формы массива (`reshape`), транспонированием (`T`) и операций поворота

на 90° (*rot90*). Посмотрим, как внутренне представляются результаты этих трех преобразований:

```
view_swap_arr = image_arr.reshape(image_arr.shape[1],
                                   image_arr.shape[0])

print("view_swap", view_swap_arr.shape, view_swap_arr.strides)
Image.fromarray(view_swap_arr, "L").save("view_swap.png")

trans_arr = image_arr.T
print("transpose", trans_arr.shape, trans_arr.strides)
Image.fromarray(trans_arr, "L").save("transpose.png")

rot_arr = np.rot90(image_arr)
print("rot", rot_arr.shape, rot_arr.strides)
Image.fromarray(rot_arr, "L").save("rot90.png")
```

Такого же результата можно было достигнуть с использованием метода *swapaxes*

В результате применения всех указанных выше способов были созданы отдельные представления исходного массива.

Давайте еще применим срез, оставив на нашем логотипе только слово Manning. В результате этого также будет создано представление:

```
slice_arr = image_arr[15:, 77:]
print("slice_arr", slice_arr.shape, slice_arr.strides, np.shares_
      memory(slice_arr, image_arr))
Image.fromarray(slice_arr, "L").save("slice.png")
```

Значения свойств *shape* и *strides* для созданных объектов собраны в табл. 4.3.

Таблица 4.3. Значения свойств *shape* и *strides* для новых массивов

Объект	Shape	Strides
<i>view_swap_arr</i>	182 45	45 1
<i>trans_arr</i>	182 45	1 182
<i>rot_arr</i>	182 45	-1 182
<i>slice_arr</i>	30 105	182 1

Можете предположить, как будут выглядеть сохраненные на диске изображения? Попробуйте угадать и сравните свои догадки с рис. 4.8.

СОВЕТ. Для простоты объяснений мы ограничились манипуляциями с одно- и двумерными массивами данных, но все описанные механизмы будут работать с массивами NumPy любых размеров.

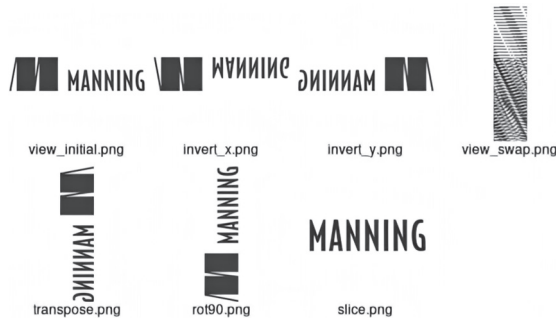


Рис. 4.8. Результаты прямых манипуляций с массивами применительно к логотипу Manning

ПРЕДУПРЕЖДЕНИЕ. При работе с библиотекой NumPy дозвоительно влиять на значения в кортежах `shape` и `strides` напрямую. В модуле `numpy.lib.stride_tricks` есть функция `as_strided`, принимающая исходный массив, желаемую форму и значения шагов и возвращающая результирующий преобразованный массив.

Уже тот факт, что эта функция располагается в модуле с именем `stride_tricks` (трюки со stride), должен вызвать определенные подозрения. Кроме того, страница с описанием функции начинается словами: «*Эта функция должна использоваться с предельной осторожностью*».

Проблема состоит в том, что вы по ошибке можете передать этой функции некорректные значения. Это может привести к редчайшему для Python случаю нарушения целостности памяти, когда функция получает доступ к нецелевым ячейкам памяти. В результате в лучшем случае программа может закрыться, а в худшем произойдет утечка информации, возможно, конфиденциальной.

Вывод

Использование представлений массивов способно существенно повысить эффективность работы с памятью и привести к значительной экономии времени. И хотя представления могут быть использованы не всегда, движок библиотеки NumPy обладает достаточной гибкостью, чтобы обеспечивать их применение почти в любых ситуациях с помощью полезных свойств `shape` и `strides`.

На данный момент это все о представлениях и копиях массивов NumPy с точки зрения производительности. Теперь давайте поговорим об эффективных приемах использования конструкций NumPy на практике.

4.2. Программирование на основе массивов

Модель *программирования на основе массивов* (array programming) опирается на принципы применения операций ко всем значениям

в массиве одновременно. Она активно используется в научном и высокоэффективном программировании и характеризуется двумя главными постулатами, один из которых базируется на более декларативном стиле написания выражений, а второй связан с максимально эффективным выполнением кода. В библиотеке NumPy принципы программирования на основе массивов используются повсеместно.

Необходимо проводить очень четкую линию между использованием *библиотеки массивов* (array library) и применением программирования на основе массивов. Сейчас мы разберем простой пример, который продемонстрирует эту разницу. Пример будет исключительно гипотетическим, поскольку мне трудно представить, что кто-то будет решать его без массивов.

ПРИМЕЧАНИЕ. Хотя этот конкретный сценарий вряд ли кто-то будет реализовывать без использования массивов, до сих пор очень часто можно встретить код NumPy, в котором не применяются заложенные в эту библиотеку мощнейшие концепции. В представленном ниже примере преимущества подхода с использованием массивов вполне очевидны, но зачастую библиотеку NumPy применяют далеко не самым эффективным образом, как мы очень скоро увидим. Вообще, одним из наиболее важных навыков разработчика является умение быстро и четко распознавать ситуации, в которых при использовании таких пакетов, как NumPy или pandas, можно заменить простой код на Python быстрыми и эффективными инструкциями с применением массивов.

Представьте, что вам необходимо рассчитать сумму значений двух векторов. Ниже приведена версия решения этой задачи в лоб, без применения программирования на основе массивов. Код можно найти в файле [04-numpy/sec3-vectorize/array_and_broadcasting.py](#):

```
import numpy as np

def sum_arrays(a, b): # Предположим, что наши массивы одного размера
    my_sum = np.empty(a.size, dtype=a.dtype)
    for i, (a1, b1) in enumerate(zip(np.nditer(a), np.nditer(b))):
        my_sum[i] = a1 + b1
    return my_sum.reshape(a.shape)
```

Мы намеренно написали код, проходящий в цикле по всем элементам массивов. Эта реализация таит в себе массу проблем, о которых мы поговорим далее. А сейчас давайте сравним ее с эквивалентным выражением на основе массивов:

```
a + b
```

Трудно спорить с тем, что эта версия вышла более короткой и декларативной, и уже одного этого должно быть достаточно для того,

чтобы отдать ей предпочтение. Но и с точки зрения производительности здесь есть масса интересного, о чем стоит поговорить.

Способ с массивами может оказаться на несколько порядков быстрее. Первый пример решения реализован на чистом Python, а значит, тянет за собой все ограничения этого языка. В то же время перегруженный оператор `+` из второго решения выполнится максимально быстро, насколько это возможно, поскольку он реализован не на Python, а на языках низкого уровня вроде C или Fortran, вероятно, с использованием векторизованных операций центрального процессора или даже задействованием ресурсов графического процессора.

Сейчас мы не будем углубляться в тему реализации перегрузки операторов. Достаточно просто понимать, что программирование на основе массивов всегда будет выигрывать в эффективности у традиционных приемов.

4.2.1. Отправная точка

Эффективный код может быть ясным и понятным. Миф о том, что высокопроизводительная программа должна быть обязательно запутана и непонятна, давно остался в прошлом. Таким образом, реализация кода с использованием массивов практически всегда будет одновременно и более эффективной, и более лаконичной.

Давайте ненадолго вернемся к предыдущему неуклюжему решению задачи на чистом Python, чтобы бегло познакомиться еще с одной концепцией NumPy, именуемой транслярованием. Эта концепция позволяет писать более эффективный и прозрачный код для работы с массивами.

4.2.2. Транслирование в NumPy

Для понимания концепции *транслирования*, или *бродкастинга* (broadcasting), начнем с более глубокого погружения в наше решение на Python для сложения элементов двух массивов. Повторим код, приведенный выше:

```
import numpy as np

def sum_arrays(a, b): # Предположим, что наши массивы одного размера
    my_sum = np.empty(a.size, dtype=a.dtype)
    for i, (a1, b1) in enumerate(zip(np.nditer(a), np.nditer(b))):
        my_sum[i] = a1 + b1
    return my_sum.reshape(a.shape)
```

Форму и тип данных элементов
берем из первого массива



Перед тем как обратиться к главной проблеме этого подхода, давайте отметим одну любопытную деталь, а именно вызов функции `np.empty` для создания нового массива. Эта функция выделяет память под массив, но не инициализирует его исходными значениями. Это может вам пригодиться при необходимости создавать

объемные массивы данных. Но вам ни в коем случае не стоит забывать о последующей инициализации массива значениями, иначе вы получите мусор на входе. Это не самый эффективный метод для работы с массивами, но иногда может оказаться полезным.

Основная проблема приведенного кода состоит в отсутствии всяческой проверки входных данных. В результате функция может принять на вход массивы самой разной формы, и в худшем случае эти массивы могут оказаться несовместимыми. Очевидно, простейшим способом решения этой задачи будет введение проверки на совместимость размеров массивов `a` и `b` и изменение размера выходного массива при необходимости. Конечно, это можно реализовать, но решение будет далеко не самым элегантным. Может дойти до того, что при необходимости увеличить на единицу каждое значение в массиве из 100 000 элементов вам придется писать следующий неуклюжий код:

```
array_100000 = np.arange(100000)
sum_arrays(array_100000, np.ones(array_100000.shape))
```

В результате мы выделим еще немало памяти под массив из единиц, и все для того, чтобы инкрементировать значения в исходном массиве. Это и здесь выглядит довольно дико, а что будет при работе с большими данными?

Интуитивно хочется написать что-то подобное: `sum_arrays(array_100000, 1)`. И при работе с NumPy это возможно! Только взгляните – следующий код прекрасно работает:

```
array_100000 = np.arange(100000)
array_100000 += 1
```

Как мы увидим позже, это не то же самое, что написать `array_100000 = array_100000 + 1`

В переменной `array_100000` хранится массив из 100 000 элементов, тогда как единица представляет собой скалярное атомарное значение. Таким образом, они обладают разными типами данных. Подобное допустимо в NumPy именно благодаря технологии транслирования, предстающей в виде набора правил, позволяющих NumPy применять операторы к массивам разных размеров.

Ниже мы приведем несколько примеров, которые помогут вам лучше понять правила транслирования массивов. Мы также сравним их с функциями, которые можно спутать с операторами транслирования. Начнем с одномерных массивов:

```
a = np.array([0, 20, 21, 9], dtype=np.uint8)
b = np.array([10, 2, 25, 5], dtype=np.uint8)

print("прибавляем единицу", a + 1)
print("умножаем на двойку", a * 2)
print("прибавляем вектор", a + [10, 2, 25, 5])
```

```
print("умножаем на вектор", a * [10, 2, 25, 5])
print("скалярное (внутреннее) произведение", a.dot(b))
print("matmul (внутреннее произведение)", a @ b)
```

Оператор `+` добавляет единицу ко всем элементам массива в первой строке вывода. В третьей строке производится поэлементное сложение двух массивов, что приводит к следующему результату: $[0 \ 20 \ 21 \ 9] + [10 \ 2 \ 25 \ 5] = [10 \ 22 \ 46 \ 14]$.

Обратите внимание, что оператор `*` работает с массивами похожим образом. Сначала мы все элементы массива удваиваем, а затем производим поэлементное перемножение массивов: $[0 \ 20 \ 21 \ 9] * [10 \ 2 \ 25 \ 5] = [0 \ 40 \ 525 \ 45]$.

Скалярное (внутреннее) произведение реализовано в NumPy с помощью функции `np.dot` и оператора `@` (`np.matmul`). Подробнее об операторе `@` мы поговорим позже. Теперь посмотрим на примеры вычислений с использованием правил транслирования:

```
x = np.array([[0, 20], [250, 500], [1, 2]], dtype=np.uint16)
y = np.array([[1, 10], [25, 5]], dtype=np.uint16)

print("прибавление матрицы к себе самой", x + x)
print("прибавление матрицы с одной строкой", x + [1, 2])
# print(x + [-1, -2, -3])
print("прибавление матрицы с одной колонкой с транспонированием", (x.T +
[-1, -2, -3]).T)
print("внутреннее произведение", np.inner(a, b))
print("перемножение матриц", x.dot(y))
# print(x.T.dot(y))

print("matmul", x @ y)

x[:, 0] = 0
print("транслирование присвоения", x)
```

Прибавление матрицы к себе самой привело к ожидаемому результату – все значения в ней удвоились. Также вы можете прибавить к матрице одномерный массив. В нем должно быть столько элементов, сколько колонок в исходной матрице, а вычисления будут выполнены построчно. В то же время вы не можете прибавить к матрице массив с одной колонкой. Легче всего – без использования операций копирования или медленных приемов традиционного программирования – можно сделать это путем транспонирования исходной матрицы (помните, что это очень быстрая операция) и затем обратного транспонирования полученного результата.

СОВЕТ. Операторы NumPy могут не соответствовать аналогичным математическим операторам в их привычном виде. Например, оператор `*` не выполняет математическое перемножение матриц. Эту функцию берет на себя `np.dot`.

Вывод

О концепции транслирования в NumPy можно говорить долго, здесь же мы лишь прошли по верхам и обсудили основные моменты с точки зрения производительности. Самое важное, что необходимо знать в этой связи, – это то, что операции транслирования по большей части являются *векторизованными* (vectorized), а это, как мы уже не раз говорили, положительно сказывается на их быстродействии. Теперь мы готовы вернуться к нашему примеру с обработкой изображений с новыми силами и знаниями в области программирования на основе массивов.

4.2.3. Применение приемов программирования на основе массивов

Давайте используем полученные навыки при манипулировании нашим изображением. Мы будем одновременно учиться применять более эффективные методики работы с массивами и обходить известные ловушки на нашем пути. Но не позволяйте этим ловушкам вынудить вас отказаться от использования этих приемов: программирование на основе массивов в любом случае является гораздо более эффективным по сравнению с типичным императивным программированием, базирующимся на циклах.

На этот раз мы попробуем сделать наше изображение более светлым. Помните, что мы используем 1 байт на каждый пиксель, для чего применяем опцию L при загрузке изображения. Таким образом, значения в каждой ячейке массива могут находиться в диапазоне от 0 до 255. Мы увеличим яркость изображения двумя различными способами: путем добавления пяти к значению каждого пикселя и с помощью удвоения каждого значения, как показано ниже:

```
import numpy as np
from PIL import Image

image = Image.open("../manning-logo.png").convert("L")
width, height = image.size
image_arr = np.array(image)

brighter_arr = image_arr + 5
Image.fromarray(brighter_arr).save("brighter.png")
brighter2_arr = image_arr * 2
Image.fromarray(brighter2_arr).save("brighter2.png")
```

В идеальном мире на этом наша задача должна была быть решена. Но взгляните на результаты, показанные на рис. 4.9. Здесь явно нужно что-то исправить.



Рис. 4.9. Осветление исходного изображения не привело к желаемым результатам

С удвоением значения все *вроде* неплохо, но с увеличением значений пикселей на 5 явно есть проблемы. Мы уже упоминали, что пиксели у нас представлены целочисленными беззнаковыми значениями в диапазоне от 0 до 255. И если к белому цвету (значение 255) прибавить 5, то мы столкнемся с переполнением типа данных, и желаемое число 260 превратится в 4, что приведет к закрашиванию пикселя практически черным цветом.

Но есть и более серьезная проблема, поскольку она незаметна на первый взгляд. Связана она уже с примером удвоения значений пикселей. Вроде внешне никаких проблем не видно, но они есть. Чтобы понять, что происходит, давайте выведем максимальные значения пикселей исходного изображения и изображения, хранящегося в переменной `brighter2`:

```
print(image_arr.max(), image_arr.dtype)
print(brighter2_arr.max(), brighter2_arr.dtype)
```

Для оригинального изображения максимальное значение составило 255 (чистый белый цвет), тогда как для изображения с удвоенными значениями пикселей результат получился 254. Как же так? Все дело в двоичной математике: поскольку число 255 в двоичной системе счисления представляется как `0x11111111` (8 бит, все по единице), а $2 * 255$ будет 510 или `0x11111110` (8 старших бит по единице и завершающий ноль), переполнение типа данных приведет к обрезке старшего бита. В результате мы получим `0x11111110`, а это 254. Как итог, изображение выглядит нормально, но цвета пикселей на нем некорректны.

ПРЕДУПРЕЖДЕНИЕ. Будьте особенно осторожны при выборе типов данных. Если вы не сильно ограничены объемом памяти или требованиями к скорости, всегда делайте определенный запас. Если же ограничения присутствуют, старайтесь избегать ситуаций, когда данные просто не будут уместиться в выделенный для них тип, как в нашем случае. Универсальное правило гласит о том, что типы данных должны быть выбраны так, чтобы вмещать любые тестовые значения даже в пограничных случаях.

Простейшим, пусть и не самым эффективным с точки зрения расходования памяти, будет решение с использованием более вместительных типов данных, как показано ниже:


```

brighter3_arr = image_arr.astype(np.uint16)
brighter3_arr = brighter3_arr * 2
print(brighter3_arr.max(), brighter3_arr.dtype)
brighter3_arr = np.minimum(brighter3_arr, 255) ←
print(brighter3_arr.max(), brighter3_arr.dtype)
brighter3_arr = brighter3_arr.astype(np.uint8)
print(brighter3_arr.max(), brighter3_arr.dtype)
Image.fromarray(brighter3_arr).save("brighter3.png")

```

Не путайте функции `minimum` и `min`. Функция `min` возвращает минимальное значение в массиве, а `minimum` выбирает меньшее значение с транслированием

Мы можем предположить, что цвет пикселя не может быть белее белого, так что значения, большие чем 255, мы можем смело приравнивать к 255. Начнем мы с приведения исходного массива к типу `np.uint16`, после чего умножим его значения на 2. Проверка максимального значения в этом случае выдаст ожидаемое число 510. Затем мы применим функцию `np.minimum` для поэлементного выбора наименьшего значения между текущим числом и 255. Это яркий пример создания копии массива с использованием операции транслирования. В результате все значения в новом массиве могут быть представлены с помощью однобайтового беззнакового типа данных. К этому типу мы и приводим массив, после чего максимальное значение отображается правильно. В следующем разделе мы рассмотрим более эффективный способ решить эту проблему¹.

Существуют и более эффективные способы приведения типов, например вы можете выполнить умножение и сразу вернуть тип данных `np.uint16`, о чем мы подробнее поговорим далее.

Наконец, давайте рассмотрим еще один вариант удвоения значений в массиве. Ранее мы использовали следующий синтаксис:

```

brighter3_arr = brighter3_arr * 2

```

В этом случае мы неявным образом создаем промежуточный массив, в котором будет храниться результат умножения. После этого новый массив переносится в переменную `brighter3_arr`. Но на протяжении короткого временного отрезка – пока сборщик мусора не активировался – в памяти будет содержаться два массива. Это может стать проблемой при работе с очень объемными данными как в плане расходования памяти, так и в отношении времени, требуемого для создания копии массива. Гораздо лучше использовать оператор умножения с присваиванием, как показано ниже:

```

brighter3_arr *= 2

```

¹ Есть одно простейшее решение представленной задачи, не слишком удачное с точки зрения педагогики. Догадались? Подсказка: попробуйте использовать функцию `maximum` вместо `minimum`.

В этом случае NumPy поймет, что вы планируете изменять исходный массив, и будет производить вычисления прямо на месте. Это означает, что вам не придется расходовать лишнюю память и тратить вычислительные ресурсы на создание и удаление временного массива. Итоговый результат при этом окажется таким же, хотя чисто практически эти подходы работают совершенно по-разному: $x = x * 2$ требует дополнительную память и время, а $x *= 2$ – нет, из-за чего всегда, когда это возможно, предпочтение должно отдаваться ему.

Вывод

Отчасти проблема применения библиотек вроде NumPy и pandas состоит в том, что разработчики зачастую просто не используют всех преимуществ, которые они дают. Но делают они это ненамеренно – все мы в той или иной степени склонны использовать «обычные» традиционные подходы к программированию, пока насильно не заставим себя попробовать что-то новое. Теперь давайте углубимся в принципы программирования на основе массивов еще больше, поскольку с их помощью вы можете строить потрясающие по своей эффективности решения.

4.2.4. Векторизуем сознание

Векторизованный код на *чистом* Python не превосходит в эффективности обычный код без применения принципов векторизации, о чем прямым текстом сказано в официальной документации к функции *np.vectorize*.

Но мы будем продолжать векторизовывать свое сознание и в следующих главах этой книги: когда будем говорить о расширении Cython, библиотеке pandas и векторизации вычислений с помощью центрального и даже графического процессора. Вполне вероятно, что вам будет гораздо проще разобраться с этой концепцией в поздних главах книги, если вы привыкнете к ней на «мелководье», – применительно к чистому Python и NumPy. Иными словами, все постулаты и принципы, которые мы озвучим в этой главе касательно векторизации, пригодятся вам в следующих главах книги.

Для лучшего понимания принципов векторизации и знакомства с *универсальными функциями* (universal functions) NumPy мы обратимся к уже знакомому примеру. В предыдущем разделе мы видели, что выражение `brighter2_arr = image_arr * 2` может приводить к переполнению типа данных. А что нужно сделать, чтобы это переполнение не возникало? Можно реализовать эту операцию в виде простой *векторизованной функции* (vectorized function) следующим образом:

```
def double_wo_overflow(v):
    return min(2 * v, 255)
```

Теперь мы векторизируем эту функцию и применим ее к нашему изображению, как показано ниже:

```
import numpy as np
from PIL import image
```

```
vec_double_wo_overflow = np.vectorize(
    double_wo_overflow, otypes=[np.uint8])
```

← Нам нужно указать выход-
ной тип данных

```
brighter_arr = vec_double_wo_overflow(image_arr)
print(brighter_arr.max(), brighter_arr.dtype)
Image.fromarray(brighter_arr).save("vec_brighter.png")
```

Функция `np.vectorize` принимает на вход обычную невекторизованную функцию и, как в данном случае, преобразовывает ее таким образом, чтобы она могла применяться ко всем скалярам в массиве. В результате мы применили нашу созданную функцию `vec_double_wo_overflow` к массиву с изображением, и она отработала для каждого его элемента.

СОВЕТ. Несмотря на то что функция `np.vectorize` реализована с использованием банального цикла `for`, теоретически она может быть вызвана параллельно на разных ядрах вашего процессора, что позволит повысить ее эффективность. Использование такого подхода совместно с параллельными вычислениями вплотную подводит нас к принципам работы графических процессоров. Если вы усвоите эти концепции здесь, вам будет гораздо легче справиться с более сложным материалом в главе 9, посвященной оптимизации с использованием ресурсов графического процессора.

В подтверждение наших слов о том, что этот подход сам по себе не ведет к ускорению кода, мы использовали инструкцию `%timeit`, которая выдала результат в районе нескольких миллисекунд. При этом операция умножения должна выполняться на порядок быстрее.

В защиту функции `np.vectorize` можно сказать, что она поддерживает правила транслирования массивов. Чтобы продемонстрировать это на примере, давайте поработаем с цветным изображением с сайта NASA, которое называется *St. Patrick's Aurora* и располагается по адресу https://images.nasa.gov/details-GSFC_20171208_Archive_e000760.

Начнем с чтения изображения, которое на этот раз будет иметь несколько иное представление.

```
import numpy as np
from PIL import Image

image = Image.open("../aurora.jpg")
width, height = image.size
image_arr = np.array(image)
print(image_arr.shape, image_arr.dtype)
```

Размер нашего изображения – 2040×1367. Поскольку мы имеем дело с цветным изображением, по умолчанию будет использоваться режим RGB с красным, зеленым и синим каналами, и для каждого будет выделен один беззнаковый байт. В результате каждый пиксель теперь будет занимать в памяти не 1 байт, а 3 байта. Таким образом, мы имеем дело с трехмерным массивом NumPy со значением свойства `shape` (2048, 1367, 3). Сделать цветное изображение черно-белым можно довольно просто – для этого достаточно рассчитать среднее значение для трех каналов, и этот показатель и будет интенсивностью серого:

```
def get_grayscale_color(row):
    mean = np.mean(row)
    return int(mean)

vec_get_grayscale_color = np.vectorize(get_grayscale_color,
    otypes=[np.uint8],
    signature="(n)->()")

grayscale_arr = vec_get_grayscale_color(image_arr)
print(grayscale_arr.max(), grayscale_arr.dtype, grayscale_arr.shape)
Image.fromarray(grayscale_arr).save("grayscale.png")
```

Среднее значение по трем цветовым каналам

Среднее значение будет дробным, а мы возвращаем целое значение

Мы переопределяем сигнатуру функции по умолчанию при ее векторизации

По умолчанию `np.vectorize` будет передавать в нашу функцию скаляр, но мы можем изменить это поведение, поправив сигнатуру функции для приема и возврата других типов данных. В нашем случае мы бы хотели, чтобы функция принимала на вход массив (т. е. три компонента) и возвращала скаляр, для чего использовали сигнатуру `(n)->()`. Результат запуска этого кода показан на рис. 4.10.

ПРЕДУПРЕЖДЕНИЕ. Стоит еще раз повторить, что здесь мы приводим примеры исключительно для демонстрации пользы от векторизации, тогда как сами решения могут быть неоптимальными. К примеру, в этом случае наиболее эффективным и, возможно, лучшим решением будет следующее:



Рис. 4.10. Результат работы простого алгоритма перевода изображения в черно-белый вид

```
grayscale_arr = np.mean(image_arr, axis=2).astype(np.uint8)
```

Здесь мы рассчитываем средние значения по последней оси массива, в которой и хранится информация о цветах пикселей. Если

итеративный способ решения будет однозначно худшим, это еще не значит, что создание собственной векторизованной функции – это единственно верный вариант. В данном случае, как вы видите, лучшее решение рождается из понимания работы *встроенных* векторизованных функций.

Вывод

С точки зрения оптимизации производительности встроенная в чистый Python векторизация не дает никаких бонусов. В то же время в других областях, таких как Cython или обработка с помощью графического процессора, векторизация может обеспечить значительный прирост быстродействия кода – вплоть до нескольких порядков. Таким образом, если вы прониклись общей идеей векторизации функций и поняли, как и для чего она применяется, главы, посвященные Cython и вычислениям при помощи графического процессора, не покажутся вам такими уж сложными.

Теперь, когда мы рассмотрели основы применения библиотеки NumPy с целью повышения эффективности кода, пришло время взглянуть на внутреннее устройство NumPy и узнать, как можно оптимально сконфигурировать библиотеку. Оказывается, опции, которые можно настроить при установке пакета, могут серьезно влиять на производительность вычислений, включая многопроцессность.

4.3. Оптимизация внутренней архитектуры NumPy

В этом разделе мы заглянем внутрь NumPy и попробуем настроить внутреннее убранство библиотеки под максимальную производительность. А начнем с обзора внутренней архитектуры NumPy.

Многие особенности библиотеки NumPy, которые делают ее столь эффективной, не реализованы в чистом языке Python. И это неудивительно. Именно расширения языка, в числе которых и сторонние библиотеки, за счет конфигурации своих настроек позволяют значительно повысить производительность кода в целом.

Темы, которые мы будем обсуждать в этом и следующих разделах, могут показаться довольно сухими и сложными тем, кто сосредоточен исключительно на написании программ на Python. Если вы уверены, что ваша библиотека NumPy работает оптимально, или у вас нет доступа к ее настройке, можете смело переходить к разделу «Потоки в NumPy», который имеет прямое отношение к Python. Если же у вас есть полный доступ к вашему стеку Python, вы любите копаться в системных настройках и хотите выжать максимум производительности из имеющихся инструментов, продолжайте чтение.

4.3.1. Обзор зависимостей в NumPy

Многие научные библиотеки, будь они на основе Python или нет, зависят от двух широко используемых библиотек API, связанных с вычислениями в области линейной алгебры: *BLAS* (Basic Library Algebra Subprograms) и *LAPACK* (Linear Algebra PACKage).

В библиотеке BLAS реализован набор базовых функций для работы с массивами и матрицами. К примеру, в ней реализованы такие операции, как сложение векторов и перемножение матриц.

Библиотека LAPACK включает в себя несколько алгоритмов линейной алгебры, в числе которых *сингулярное разложение матрицы* (singular value decomposition – SVD) – базовый алгоритм анализа главных компонент. На рис. 4.11 схематично показана архитектура библиотеки NumPy.

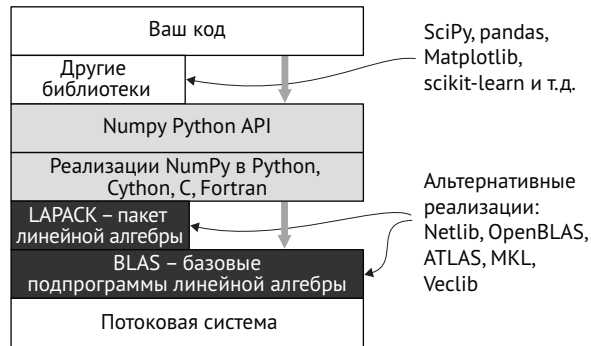


Рис. 4.11. Стек библиотеки NumPy, включая зависимости

Существует масса альтернативных реализаций библиотек, и ваш выбор может непосредственно сказываться на производительности решений. К примеру, стандартная реализация библиотеки LAPACK, которую можно скачать с сайта <https://netlib.org>, не поддерживает многопоточность и не отличается высокой эффективностью на современных архитектурах. Распространенными альтернативами библиотекам BLAS / LAPACK являются *OpenBLAS* и *Intel MKL*, и обе они поддерживают работу с потоками. Способ использования ресурсов компьютера может сильно варьироваться в зависимости от того, поддерживает ли ваша реализация NumPy многопоточность. К примеру, используя версию без многопоточности, вы максимально сможете запустить столько процессов, сколько ядер находится в вашем процессоре. В то же время с многопоточной реализацией BLAS и LAPACK вам придется следить за тем, чтобы не переусердствовать с использованием ресурсов центрального процессора.

Надеюсь, я убедил вас в важности понимания того, от каких именно библиотек зависит ваша реализация NumPy. В теории вы можете определить эти зависимости, запустив следующий код:

```
import numpy as np
np.show_config()
```

На практике же вам может понадобиться обратиться к файловой системе и системе управления пакетами, чтобы понять, что на самом деле происходит. К примеру, когда я подключил MKL с использованием Anaconda Python, вывод частично стал таким:

```
lapack_opt_info:
  libraries = [
    'lapack', 'blas', 'lapack', 'blas', 'cblas', 'blas', 'cblas', 'blas']
  library_dirs = ['/home/tra/anaconda3/envs/book-mkl/lib']
  language = c
  define_macros = [('NO_ATLAS_INFO', 1), ('HAVE_CBLAS', None)]
  include_dirs = ['/home/tra/anaconda3/envs/book-mkl/include']
```

Если вы запустите эту инструкцию, вам, возможно, повезет больше, но в моем случае вывод оказался крайне неинформативным (book-mkl – это название моего виртуального окружения, так что никаких выводов по этому делать не стоит). Чтобы определить, какие именно реализации у меня используются, мне пришлось выполнить в терминале команду `ls -l /home/tra/anaconda3/envs/book-mkl/lib/libcblas.so*`, и я заметил следующие вхождения: `libcblas.so.3 • libmkl_rt.so`. Судя по всему, библиотека MKL у меня подключена. Вы можете определить, какие библиотеки подключены в вашей системе.

СОВЕТ. NumPy не только использует в своей работе библиотеки BLAS и LAPACK, но также предоставляет к ним интерфейс Python, чтобы вы при желании могли обращаться к ним напрямую. Вернее, SciPy предоставляет такой интерфейс.

SciPy – это родственная NumPy библиотека, и у них общая история. В библиотеке SciPy реализуются функции более высокого уровня по сравнению с NumPy.

По причине такой тесной связи между NumPy и SciPy вас могут сбить с толку их API. В официальной документации SciPy эта ситуация очень четко поясняется. Ниже приведен фрагмент из документации к модулю линейной алгебры SciPy, называемому `scipy.linalg`:

Смотрите также: за другими функциями в области линейной алгебры вы можете обратиться к модулю `numpy.linalg`. Обратите внимание, что, хотя `scipy.linalg` импортирует большинство из них, идентично названные функции из модуля `scipy.linalg` могут предлагать больше возможностей или другой функционал.

Таким образом, иногда библиотека SciPy импортирует и реэкспортирует функции NumPy, иногда их API могут немного различаться, а бывает, что реализации функций могут оказаться совершенно разными.

Если вам нужно, вы можете обратиться к библиотекам BLAS и LAPACK напрямую. Соответствующие API для Python вы можете найти в модулях `scipy.linalg.blas` и `scipy.linalg.lapack` соответственно. Если вам кажется, что было бы логичнее разместить эти интерфейсы в модулях NumPy, а не SciPy, знайте, что вы не одиноки.

Получается, что вы можете использовать эти библиотеки напрямую из Python, но с точки зрения производительности лучше будет воспользоваться для этого языками низкого уровня. Здесь я об этом распространяться не буду, но вернусь к этой теме в главе, посвященной Cython.

Перед возвращением к практическому обсуждению внутреннего устройства NumPy нам необходимо уделить внимание влиянию процесса установки библиотеки на производительность. Для этого обратимся к дистрибутивам Python.

4.3.2. Настройка NumPy в дистрибутиве Python

В этом разделе мы поговорим о том, как можно убедиться в оптимальности установки библиотеки NumPy в вашем дистрибутиве. Конечно, мы не сможем охватить все существующие установки Python для всех операционных систем, так что сосредоточимся на стандартной реализации языка с `python.org` и `Anaconda Python`. Мы будем использовать Linux, поскольку выбор операционной системы здесь также имеет значение. Комментарии, сделанные по ходу, пригодятся и при использовании других дистрибутивов.

Если вы устанавливаете библиотеку NumPy в стандартном дистрибутиве, вам, скорее всего, придется использовать для этого команду `pip install numpy`. Это сработает только в том случае, если библиотеки BLAS и LAPACK в вашей операционной системе уже установлены. Другой вопрос заключается в том, какая именно версия установлена. Если речь идет о наиболее распространенной версии с NetLib, то она медленная и не поддерживает потоки. В плане производительности это будет ужас. Таким образом, вам нужно будет убедиться, что:

- вы установили что-то более эффективное вроде OpenBLAS или MKL от Intel;
- воспользовались приведенной ранее функцией `np.show_config` и привязали наиболее быструю версию BLAS/LAPACK к своей системе.

При использовании других дистрибутивов есть вероятность, что система управления пакетами этих дистрибутивов позаботится о библиотеках BLAS и LAPACK за вас. Зависимости также могут быть установлены автоматически. Например, когда вы в `Anaconda Python` выполняете команду `conda install numpy`, вы, скорее всего,

получите установленную библиотеку *OpenBLAS*, которая годится в большинстве случаев.

Хотя в основном в научных дистрибутивах Python с библиотеками будет все в порядке, вы можете захотеть что-то изменить. И почти все дистрибутивы позволяют это делать. Допустим, в *Anaconda* вы можете установить NumPy с поддержкой MKL следующим образом:

```
conda create -n book-mkl blas=*mk
conda activate book-mkl
conda install numpy
```

Мы создали новое виртуальное окружение *book-mkl*, чтобы не затронуть и не испортить текущее окружение. Также мы прописали строчку *blas=*mk*, тем самым установив сборку MKL библиотеки BLAS. Теперь можно устанавливать NumPy.

СОВЕТ. В большинстве случаев вам достаточно будет убедиться, что вы не используете медленную реализацию BLAS с NetLib. Чаще всего *OpenBLAS* или MKL вам будет вполне достаточно. Если в вашем случае используются другие реализации библиотек, вам следует их изучить.

Если вы хотите выбрать действительно оптимальную реализацию под себя и попытаться извлечь абсолютный максимум возможного из своей системы, вам придется самостоятельно сравнить быстродействие разных решений. В интернете есть множество различных шаблонных тестов, но вам лучше разработать свой собственный, исходя из своего кода, поскольку у разных реализаций разные сильные стороны, и единого критерия отбора здесь не существует.

Теперь, когда вы убедились, что библиотека NumPy у вас установлена и сконфигурирована, давайте воспользуемся полученными преимуществами.

4.3.3. Потоки в NumPy

Работа библиотеки NumPy с потоками определяется используемыми реализациями BLAS/LAPACK. Большинство реализаций этих библиотек являются многопоточными (NumPy умеет обходить правила GIL, так что мы здесь говорим о настоящем параллелизме), и вы можете воспользоваться ими, если необходимо. Но здесь есть два нюанса. Во-первых, все-таки далеко не все реализации BLAS/LAPACK поддерживают многопоточность, а во-вторых, вам может понадобиться, чтобы эти библиотеки работали исключительно с одним потоком.

Представьте следующий сценарий в нашем приложении для обработки изображений. Вам нужно обработать тысячи файлов, и для этого вы задействуете восемь параллельных процессов на своей 8-ядерной машине. Если ваш NumPy поддерживает многопо-

точность, в рамках каждого из восьми процессов может быть запущено восемь потоков, и в результате получится 64 конкурентных потока исполнения. А вам необходимо, чтобы параллельно обрабатывалось максимум восемь изображений.

Зачастую гораздо более эффективно обработка данных выполняется при наличии одного потока в процессе, а не восьми. Помните, что код, не использующий BLAS, в каждом процессе Python будет выполняться в однопоточном режиме, так что восемь потоков будут запускаться только при задействовании модулей NumPy/BLAS.

Иногда нам необходимо уменьшить количество потоков, используемых модулями BLAS и LAPACK, вплоть до одного. Это должно быть несложно, правда? К сожалению, вы не можете напрямую управлять количеством потоков, используемых в работе модулем NumPy, и вам придется конфигурировать непосредственно реализации BLAS/LAPACK, что сделает код непереносимым.

Для версии с NetLib все просто, поскольку она не поддерживает многопоточность. Но вам все равно необходимо избегать работы с ней, если вы нацелены на повышение производительности решения. Библиотеки OpenBLAS и MKL от Intel обладают разными интерфейсами, и у них может быть другая низкоуровневая зависимость для выполнения многопоточных вычислений: они *могут быть* скомпилированы и зависимы от OpenMP. Так что вам необходимо сконфигурировать ваши реализации BLAS/LAPACK и, *возможно*, используемую библиотеку многопроцессной обработки.

Для библиотеки OpenBLAS перед вызовом кода на Python нужно выполнить следующие инструкции:

```
export OPENBLAS_NUM_THREADS=1
export GOTO_NUM_THREADS=1
export OMP_NUM_THREADS=1
```

Стандартный способ конфигурирования OpenBLAS

Устаревшая переменная, основанная на оригинальном пакете GotoBLAS2, из которого возникла библиотека OpenBLAS

На случай, если OpenBLAS использует OpenMP

Требуемые действия для библиотеки MKL включают следующие:

```
export MKL_NUM_THREADS=1
export OMP_NUM_THREADS=1
```

Стандартный способ конфигурирования MKL

На случай использования OpenMP

Для других библиотек необходимые проверки вам нужно выполнять самостоятельно. Обращайте внимание на возможные зависимости от низкоуровневых библиотек, поддерживающих многопоточность, вроде OpenMP и учитывайте требования к их конфигурации.

С практической точки зрения здесь может возникнуть еще одна проблема, состоящая в том, что при перемещении кода с одной машины на другую – скажем, из среды разработки в рабочее окруже-

ние – связанные библиотеки могут меняться. Вы можете учесть это в своем установочном коде или устанавливать все необходимые переменные для всех возможных библиотек, что можно назвать более прагматичным решением.

Вывод

Возможно, копаться в низкоуровневых библиотеках, от которых зависит NumPy, – не самое большое удовольствие. Но если вы хотите использовать NumPy и весь связанный с этим модулем стек на полную мощность, без определения того, от каких реализаций библиотек он зависит и используют ли эти библиотеки многопоточность, вам не обойтись.

Заключение

- Представления массивов могут значительно превосходить копии массивов в эффективности использования как в плане расходования памяти, так и в отношении вычислительных ресурсов процессора. Вы должны использовать представления массивов всегда, когда это возможно и оправдано.
- В библиотеке NumPy заложены очень гибкие механизмы использования представлений, позволяющие по-разному интерпретировать имеющиеся данные с минимальными затратами вычислительных ресурсов и памяти.
- Хорошее понимание свойств `shape` (количество элементов в измерениях массива) и `strides` (шаги, которые необходимо сделать для достижения следующего элемента в соответствующем измерении массива) лежит в основе эффективного использования представлений массивов. Оба эти свойства могут меняться от представления к представлению, что способствует получению разных интерпретаций лежащих в их основе данных.
- Приемы программирования на основе массивов, заключающиеся в выполнении декларативных операций применительно ко всему массиву целиком вместо императивного стиля с обходом всех входящих в него значений, способны на несколько порядков повысить производительность исполняемого кода. Эти принципы необходимо использовать всегда, когда это возможно.
- Правила транслирования массивов, принятые в библиотеке NumPy и позволяющие применять операторы к массивам разной размерности, способствуют повышению быстродействия кода и его элегантности и лаконичности.
- Внутренняя архитектура библиотеки NumPy может быть оптимизирована с точки зрения времени выполнения инструк-

ций. Она, в свою очередь, зависит от низкоуровневых библиотек BLAS и LAPACK, которые предполагают разные варианты конфигурации.

- Убедитесь в том, что в вашей установке NumPy используются наиболее эффективные реализации библиотек для вашей архитектуры.
- Параллельное программирование в NumPy может быть осложнено по причине наличия зависимостей от сторонних библиотек линейной алгебры со своей семантикой многопоточности.
- Перед использованием многопроцессной обработки в Python и NumPy убедитесь в том, что лежащие в основе вашей реализации NumPy библиотеки поддерживают нужные вам режимы для работы с потоками. Если они не реализуют многопоточность, можете использовать альтернативные библиотеки. Иначе постарайтесь не использовать многопроцессную обработку поверх вызовов NumPy, работающих в многопоточном режиме.
- О библиотеке NumPy можно говорить очень долго, поскольку именно она лежит в основе всего анализа данных в Python. В связи с этим мы еще не раз будем возвращаться к ней в следующих главах книги.

Часть II

Аппаратное обеспечение

Вторая часть этой книги будет посвящена разработке на Python с точки зрения извлечения максимальной производительности при использовании распространенных средств аппаратного обеспечения. Сначала мы поговорим об использовании низкоуровневых языков программирования, с помощью которых можно повысить эффективность использования вычислительных мощностей системы, включая центральный процессор. Главным образом мы сосредоточим свое внимание на расширении языка Python, получившем название Cython, с помощью которого генерируется эффективный код на языке C. После этого мы поговорим о современных архитектурах аппаратного обеспечения, которые зачастую требуют применения на первый взгляд противоречащих здравому смыслу приемов и техник, позволяющих, как это ни странно, извлечь максимум производительности. Мы погрузимся в изучение того, как устроены современные библиотеки Python вроде NumExpr и что именно позволяет им использовать архитектуру аппаратного обеспечения для достижения высокой эффективности.

5

Реализация критически важного кода с помощью Cython

В этой главе мы обсудим следующие темы:

- как эффективно переписать код на Python;
- расширение Cython с точки зрения обработки данных;
- профилирование кода на Cython;
- использование Cython для реализации высокопроизводительных функций NumPy;
- обход GIL для реализации настоящего параллелизма на потоках.

Python медленный. Вернее, стандартная реализация Python медленная, и разработчикам с пользователями приходится дорого платить за динамическую природу языка. В то же время многие сторонние библиотеки Python, предназначенные для обработки данных, отличаются высокой производительностью как раз за счет своей реализации с помощью языков низкого уровня. Иногда и нам самим не помешает умение реализовывать высокоэффектив-

ные алгоритмы с использованием более производительных инструментов по сравнению с Python. В этой главе мы познакомимся с расширением Cython, способным преобразовывать код, написанный на Python, в код на языке C с сопутствующим этому повышением быстродействия.

Существует большое множество альтернатив расширению Cython, также способных интегрироваться с Python в деле увеличения производительности, так что мы начнем с обзора возможных решений, после чего погрузимся в мир Cython.

Если вы никогда ранее не использовали расширение Cython, это введение позволит вам получить начальные навыки в нем применительно к анализу данных совместно с библиотекой NumPy, являющейся ключевой в деле обработки информации в Python. После знакомства с расширением мы обсудим вопросы, связанные с профилированием и оптимизацией кода на Cython. Мы также напишем код, который позволит NumPy обойти ограничения GPL и тем самым выполняться в условиях полного многопоточного параллелизма. Завершим главу полноценным параллельным примером, в котором код на Cython будет действовать в обход правил GPL.

Но сначала давайте взглянем на альтернативы расширению Cython. Возможно, какие-то из этих вариантов подойдут вам лучше, особенно если вы достаточно хорошо знакомы с низкоуровневыми языками вроде C или Rust.

5.1. Обзор техник для эффективной реализации кода

Расширение *Cython* является одной из множества альтернатив реализации высокоэффективного кода. Мы в этой книге остановили свой выбор на нем по причине того, что для написания кода на Cython вам не придется изучать другие языки программирования, помимо базового Python, поскольку Cython является его надмножеством. В то же время вы должны знать и о других вариантах реализации эффективного кода, поскольку в вашем окружении могут присутствовать ограничения, которые потребуют использования одной из альтернатив.

Существующие варианты можно условно разбить на четыре группы, более подробно представленные в табл. 5.1:

- существующие библиотеки;
- Numba;
- высокоскоростные языки, такие как Cython, C и Rust;
- альтернативные реализации языка Python: PyPy, Jython, Iron Python и Stackless Python.

Таблица 5.1. Различные подходы к реализации эффективного кода

Библиотеки	Низкоуровневые языки	Альтернативные реализации Python	JIT-компиляторы
NumPy, SciPy, scikit-learn, PyTorch	C, Rust, Fortran, C++, Go, Cython	PyPy, IronPython, Jython, Stackless Python	Numba

NumPy является примером библиотеки, предлагающей эффективные приемы работы с данными в Python. Существуют и другие высокопроизводительные библиотеки, такие как pandas, scikit-learn и др. Перед тем как реализовывать свой собственный код, убедитесь, что похожий функционал уже не воплощен в одной из популярных библиотек.

Второй альтернативой, которую стоит упомянуть, является Numba. Это своего рода JIT-компилятор, преобразующий подмножество Python в быстрый нативный код. Использовать Numba бывает легче, чем прибегать к помощи сторонних языков, включая Cython. Кроме того, существуют оптимизации Numba для большого количества популярных библиотек, в числе которых NumPy и pandas, а также для эффективного использования с некоторыми архитектурами, включая архитектуру графического процессора. Причина того, почему мы в этой книге сделали упор на Cython, заключается в желании как можно более развернуто рассказывать о том, как работает то или иное решение. В то же время в компиляторе Numba есть некая скрытая «магия» в попытках создать максимально эффективный код. Мы же в этой книге стремимся рассказывать о приемах создания эффективного кода. Именно поэтому нас больше интересуют открытые решения, а не просто магические средства оптимизации. На практике вы не должны сбрасывать компилятор Numba со счетов. Более того, зачастую он позволяет добиваться схожих с Cython решений с точки зрения эффективности, требующих меньших затрат энергии. Мы более подробно поговорим о компиляторе Numba в этой книге, когда придет время. В частности, ему будет посвящено приложение Б.

В этой главе мы сосредоточимся на реализации существующего решения с использованием более низкоуровневых подходов. Так получилось, что ближе всех остальных решений низкого уровня к Python оказалось именно расширение Cython. Но это не значит, что у него нет альтернатив. Самым популярным языком в этой области является C, но и им дело не ограничивается, поскольку вы можете реализовывать свои решения с использованием языков C++, Rust, Julia и др. Cython во многом облегчает вашу жизнь как разработчика именно по причине тесной интеграции с Python. Если вы решите использовать другой язык для реализации низкоуровневых решений, придется исследовать возможности встраива-

ния кода на этом языке в Python. Например, для C и C++ вы можете воспользоваться встроенным модулем *ctypes* или *SWIG*.

Наконец, можете сделать выбор в пользу альтернативного интерпретатора Python, а не использовать стандартную реализацию CPython. Если вы плотно работаете с языком Java, вам может оказаться по душе интерпретатор Jython. Для разработчиков .Net больше подойдет IronPython. Наиболее близкой альтернативой интерпретатору CPython является реализация PyPy, более быстрая по причине своей природы JIT-компилятора. Несмотря на это, у PyPy есть и свои ограничения в виде ограниченного количества библиотек, в которых реализована поддержка этого интерпретатора. На данный момент для большинства решений наиболее подходящей реализацией языка остается CPython.

Многие из перечисленных выше вариантов могут быть использованы совместно. Ключевым аспектом этой главы будет попытка связать воедино внешнюю библиотеку, коей является NumPy, с низкоуровневой природой расширения Cython.

Теперь, когда мы вместе проговорили существующие варианты оптимизации кода, давайте перейдем к конкретному примеру с использованием Cython, который продемонстрирует возможности этого расширения.

5.2. Беглый обзор расширения Cython

Хотя это не ознакомительная книга по Cython, можно с большой долей уверенности предположить, что очень многие из моих читателей никогда не использовали это расширение на практике. В этом разделе мы напишем небольшой демонстрационный проект с уклоном в производительность. При этом мы не будем вдаваться в подробности, связанные с компиляцией кода на Cython. Да, это важная часть общего процесса, но она далеко не так тесно связана с вопросами эффективности итогового решения. В интернете полно литературы, посвященной вопросам компиляции кода на Cython и управления памятью. К тому же вы всегда можете начать с официальной документации Cython по адресу <https://cython.readthedocs.io/en/latest>.

В этом разделе мы обратимся к примеру из предыдущей главы, связанному с обработкой изображения, и создадим фильтр, принимающий на вход изображение, генерирующий его черно-белую версию и затемняющий ее в соответствии с другим изображением такого же размера. Наша первая реализация не будет отличаться высокой скоростью (над эффективностью мы поработаем позже), но она позволит вам познакомиться с основами расширения Cython, без которых двигаться дальше будет просто невозможно. На рис. 5.1 приведены изображения, которые помогут понять суть данного примера.

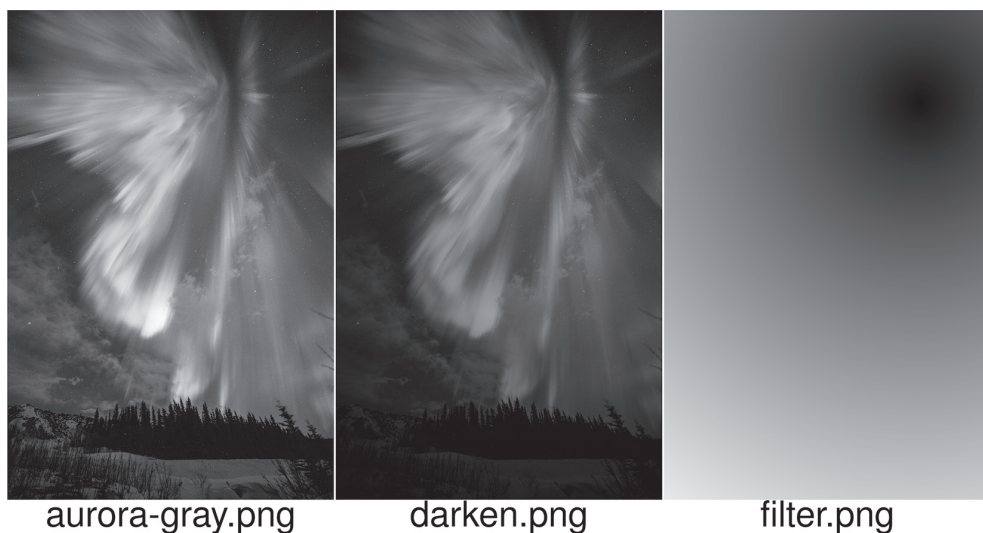


Рис. 5.1. Оригинальное изображение (в черно-белом виде), обработанное изображение и примененный фильтр

Давайте уже приступим и реализуем процесс применения фильтра с помощью расширения Cython. Для сравнения производительности вы можете загрузить из репозитория реализацию на чистом Python, которая на моем компьютере отработала примерно за 35 с.

5.2.1. Наивная реализация в Cython

Наш механизм фильтрации, как и в предыдущем примере, будет использовать для обработки изображений библиотеки NumPy и Pillow. Исходные изображения будут в цвете, так что мы будем иметь дело с тремя цветовыми компонентами. Значения фильтра будут варьироваться в диапазоне от 0 до 255, где 0 означает отсутствие затемнения, а 255 – полное затемнение. Сначала мы преобразовываем изображение в черно-белый вид, а затем осуществляем процесс затемнения.

Итак, давайте выполним прикидочную тестовую реализацию решения с использованием расширения Cython. Мы разделим код на две части: Python-код, вызывающий Cython-код в обычном исходном файле с расширением .py, и чистый код на Cython в файле с расширением .pyx. Это расширение берет свои истоки в проекте Pyrex, из которого в свое время появился Cython. Код на языке Python можно найти в файле с именем 05-cython/sec1-intro/apply_filter.py:

```

import numpy as np
from PIL import Image

import pyximport;
pyximport.install(
    language_level=3,
    setup_args={'include_dirs': np.get_include()})
import cyfilter

image = Image.open("../04-numpy/aurora.jpg")
gray_filter = Image.open("../filter.png").convert("L")
darken_arr = cyfilter.darken_naive(image_arr, gray_arr)
Image.fromarray(darken_arr).save("darken.png")

```

Модуль `pyximport` отвечает за компиляцию и загрузку кода на Cython

Нам нужна версия Python 3

Нам необходимо использовать заголовки NumPy для компиляции

Мы реализуем код на Cython в модуле с именем `cyfilter`

Единственная концептуально новая часть в приведенном выше коде связана с модулем `pyximport`. Этот модуль отвечает за компиляцию и связку с кодом на Cython.

Связка с кодом на Cython

Помните о том, что расширение Cython является надмножеством Python, и с его помощью код компилируется в C, а затем становится доступным в виде внешнего модуля. Это не такой простой процесс, как импортирование обычного родного модуля в Python.

Существует несколько способов осуществления этого процесса, как указано в документации к расширению Cython. Мы не будем перечислять их все, а остановимся на трех основных:

- подход, который мы использовали (с `pyximport`), предполагает компиляцию кода и его легкую и прозрачную привязку. Каждый раз при импортировании модуля Cython код будет преобразовываться в C, компилироваться и привязываться. Таким образом, мы будем терять какое-то время только в момент запуска. При выполнении профилирования кода не забудьте каким-то образом учесть это время;
- если вы используете Jupyter/IPython Notebook, можете воспользоваться инструкцией `%cython`. За подробностями можно обратиться к документации Cython или IPython;
- вызов `cython` с файлом `.рух` напрямую. Этот подход подразумевает, что мы будем выполнять привязку вручную. Позже в этой главе мы воспользуемся этим подходом, но только для того, чтобы взглянуть на сгенерированный код.

Если вы работаете с Jupyter или в целом с интерпретатором IPython, вам будет доступна инструкция `%cython`, с которой все будет просто. В этом случае я настоятельно рекомендую пользоваться именно ей. Но если вы в будущем планируете распространять свой код среди пользователей стандартного интерпретатора Python, вы не сможете воспользоваться этой удобной опцией.

Обращение к cython напрямую бывает полезно при необходимости проинспектировать полученный код на языке C. Других практических причин пользоваться этой утилитой я не вижу.

При отправке кода в рабочую среду или распространении среди пользователей вам придется полагаться на альтернативные способы. Вполне вероятно, что вы должны будете выполнить предварительную компиляцию вашего кода на Cython для целевой архитектуры, поскольку требовать от пользователей наличия полного стека, необходимого для компиляции кода на C, — это уже слишком. Подготовка кода для распространения — это отдельная большая тема, выходящая за рамки этой книги.

На данный момент наш код на Cython ничем не отличается от кода на Python, за исключением расширения `ruх`. Этот код можно найти в файле `05-cython/sec2-intro/cyfilter.pyх`:

```
#cython: language_level=3
import numpy as np

def darken_naive(image, darken_filter):
    nrows, ncols, _rgb_3 = image.shape
    dark_image = np.empty(shape=(nrows, ncols), dtype=np.uint8)
    for row in range(nrows):
        for col in range(ncols):
            pixel = image[row, col]
            mean = np.mean(pixel)
            dark_pixel = darken_filter[row, col]
            dark_image[row, col] = int(mean * (255 - dark_pixel) / 255)
    return dark_image
```

Первая строка кода — это на самом деле не комментарий, а инструкция для Cython, чтобы компиляция выполнялась под Python версии 3

За исключением первой строки кода и имени функции, код ничем не отличается от версии, реализованной на Python. Если сейчас запустить файл на Python, обращающийся к этому модулю, вы будете сильно разочарованы. На моем компьютере этот код выполнялся за 33 с, что лишь на 2 с быстрее по сравнению с родной версией на Python. В следующем разделе мы поработаем над оптимизацией кода и узнаем, почему первая версия нам не удалась.

Cython как компилируемый язык программирования

Cython, в отличие от Python, является компилируемым языком, а не интерпретируемым. Это серьезное отличие, влекущее за собой массу последствий, и в частности они касаются этапа обнаружения ошибок. Взгляните на приведенный ниже код:

```
def so_wrong():
    return a + 1
```

В Python этот код выдаст ошибку только на этапе выполнения, так что ничто не мешает вам выложить его в рабочую среду в таком виде. Что касается языка Cython, то здесь этот код завершится ошибкой еще на этапе компиляции. В этом отношении Cython является более безопасным, но будьте готовы к тому, что он будет постоянно надоедать вам сообщениями об ошибках, которые Python просто не заметит.

5.2.2. Использование аннотаций типов в Cython для повышения производительности

Перед тем как погрузиться в поиск причин низкой производительности предыдущего кода, давайте соорудим более быструю версию, чтобы было с чем сравнивать. Эта версия будет целиком полагаться на систему *аннотаций* (annotations) типов в Cython:

```
#cython: language_level=3
import numpy as np
cimport numpy as cnp
```

Импортируем определение уровня C для NumPy

```
def darken_annotated(
    cnp.ndarray[cnp.uint8_t, ndim=3] image,
    cnp.ndarray[cnp.uint8_t, ndim=2] darken_filter):
```

Первый параметр мы определяем в виде массива NumPy уровня C с тремя измерениями. Помните, что для цветных изображений мы учитываем по три компоненты (RGB) для каждого пикселя. Тип данных элементов – беззнаковый целочисленный 8-битный

```
    cdef int nrows = image.shape[0]
    cdef int ncols = image.shape[1]
    cdef cnp.uint8_t dark_pixel, mean
    cdef cnp.ndarray[np.uint8_t] pixel
```

Также мы типизируем все локальные переменные. Присвоить переменным кортежи в Cython не всегда бывает возможно, так что мы разбили кортеж на две переменные

Второй параметр представлен двумерным массивом с элементами такого же типа

Типизирование переменных должно выполняться в начале функции, так что мы определяем переменные внутреннего цикла перед входом в него

```
    cdef cnp.ndarray[cnp.uint8_t, ndim=2] dark_image =
np.empty(shape=(nrows, ncols), dtype=np.uint8)
    for row in range(nrows):
        for col in range(ncols):
            pixel = image[row, col]
            mean = (pixel[0] + pixel[1] + pixel[2]) // 3
            dark_pixel = darken_filter[row, col]
            dark_image[row, col] = mean * (255 - dark_pixel) // 255
    return dark_image
```

Это чуть более эффективный способ вычисления среднего значения

Самым важным отличием этого фрагмента кода от предыдущего является наличие аннотаций типов. К примеру, простое определение переменной `nrows` превратилось в `cdef int nrows`: тем самым мы говорим Cython, что объявляем переменную целочисленного (`int`) типа. Эта аннотация предназначена для кода на C, поскольку

расширение Cython конвертирует весь свой код в этот низкоуровневый язык. Также допустимо использовать определения уровня C для внешних библиотек, таких как NumPy. Именно такие определения мы импортировали в строке `import numpy as np`. После этого мы можем типизировать массивы данных.

ПРЕДУПРЕЖДЕНИЕ. Аннотации типов в Cython в корне отличаются от современных аннотаций в Python, и они никакого отношения друг к другу не имеют.

Обратите внимание, что, за исключением вычисления среднего значения и разделения кортежа на две переменные, в коде практически ничего не поменялось. По сути, те же два вложенных цикла обладают той же вычислительной сложностью.

При этом быстроедействие этого кода с 30 с возросло до 1,5 с – в 20 раз. Это уже что-то¹.

СОВЕТ. Сопровождайте аннотациями *все* ваши переменные в Cython.

Но код, который мы написали, может работать в несколько раз быстрее. Позже в этой главе мы обсудим техники, которые позволяют ускорить его, но сейчас поговорим о том, почему же аннотации так важны.

5.2.3. Как аннотации типов влияют на производительность

Почему же аннотации типов так важны с точки зрения эффективности программного кода? Чтобы ответить на этот вопрос, нам необходимо взглянуть на код на языке C, сгенерированный для наших функций. Не пугайтесь, если вы не знаете C, читать код всегда гораздо легче, чем писать его, так что ничего сложного тут не будет.

Мы рассмотрим простой пример с прибавлением к аргументу функции числа 4. Ниже показан код на Cython с аннотациями и без них. Найти код можно в файле `05-cython/sec2-intro/add4.pyx`:

```
#cython: language_level=3

def add4(my_number):
    i = my_number + 4
    return i

def add4_annotated(int my_number):
    cdef int i
    i = my_number + 4
    return i
```

¹ Тему замера времени выполнения кода мы обсуждали в главе 2. При использовании IPython/Jupyter вы можете воспользоваться удобной инструкцией `%timeit`. В стандартном Python можно прибегнуть к помощи модуля `timeit`. В данном случае я просто засекаю время. Приблизительных измерений зачастую бывает вполне достаточно.

Все просто, не так ли? Чтобы рассмотреть сгенерированный код на языке C, запустим Cython напрямую, как показано ниже:

```
cython add4.pyx
```

В результате будет создан файл на языке C с именем `add4.c`. В моей версии Cython получилось около 3000 строк кода. Но не стоит пугаться! Большинство из этих строк представляют стандартные шаблоны, а отыскать и изучить нужные нам строки кода будет очень просто. Их можно найти рядом с комментариями, в которых написаны строки вашего исходного кода Cython. Например, показанный ниже код был сгенерирован автоматически в виде комментариев:

```
/* "add4.pyx":9
 *
 *
 * def add4_annotated(int my_number):
 *
 cdef int i
 *
 i = my_number + 4
 */
```

Для каждой нашей функции Cython генерирует две функции на C: *обертку* (wrapper), реализующую интерфейс между Python и C, и саму функцию.

Таким образом, для нашей функции `add4` мы увидим следующую функцию-обертку:

```
static PyObject *__pyx_pw_4add4_1add4(
    PyObject *__pyx_self, PyObject *__pyx_v_my_number)
```

Если мыслить категориями Python, а не C или Cython, здесь все логично: функция возвращает и принимает объекты Python. А Python вообще знает только объекты.

Ниже показана сигнатура функции-обертки `add4_annotated`:

```
static PyObject *__pyx_pw_4add4_3add4_annotated(
    PyObject *__pyx_self, PyObject *__pyx_arg_a) {
```

Типы данных здесь точно такие же, и это понятно: помните, что Python понимает исключительно объекты. Таким образом, на уровне интерфейса Python эти функции ничем не отличаются.

Обе функции-обертки производят операции упаковки и распаковки, после чего вызывают соответствующие реализации. Ниже показана реализация функции `add4`:

```
static PyObject *__pyx_pf_4add4_add4(
    CYTHON_UNUSED PyObject *__pyx_self, PyObject *__pyx_v_my_number)
```

Типы данных по большей части такие же, поскольку мы не делаем аннотаций, и Cython создал обобщенную функцию.

Ниже показана сигнатура для функции `add4_annotated`:

```
static PyObject *__pyx_pf_4add4_2add4_annotated(
    CYTHON_UNUSED PyObject *__pyx_self, int __pyx_v_my_number) {
    ----
```

Обратите внимание, что для аргумента `my_number` установлен нативный тип C, это больше не объект Python: `int __pyx_v_my_number`.

Если взглянуть на обертки для обеих функций, вы увидите, что обертка для аннотированной функции намного более сложная, с многочисленными упаковками и распаковками типов. Неаннотированная обертка способна просто передавать параметры своей реализации, поскольку она взаимодействует только с объектами Python. В то же время обертка аннотированной функции должна уметь управлять преобразованиями между объектами Python и целочисленными значениями.

Итак, после этого затянувшегося вступления мы возвращаемся к нашей насущной задаче в виде реализации с виду простого математического действия `i = my_number + 4`. Вот что мы увидим в версии без аннотаций:

```
__pyx_t_1 = __Pyx_PyInt_AddObjC(__pyx_v_my_number, __pyx_int_4, 4, 0,
0);
if (unlikely(!__pyx_t_1)) __PYX_ERR(0, 5, __pyx_L1_error)
__Pyx_GOTREF(__pyx_t_1);
__pyx_v_i = __pyx_t_1;
__pyx_t_1 = 0;
```

Здесь мы видим вызов функции `__Pyx_PyInt_AddObjC` для прибавления целого числа к объекту. Эта функция реализована в файле `add4.c`. Если не боитесь, можете посмотреть на этого монстра. Внутри вы увидите множество вызовов функций CPython, много условий `if` на языке C и даже вызов `goto`! И все это в функции, которая прибавляет четверку к переменной. Как-то тяжело, нет?

Есть и еще одна серьезная проблема с этим кодом, состоящая в том, что, поскольку мы имеем дело с объектами Python, Cython не может обойти правила GIL. Код Python всегда испытывает ограничения в виде пресловутого GIL, тогда как код, написанный на низкоуровневых языках, в определенных обстоятельствах может игнорировать этот механизм. В текущем виде от ограничений GIL нам никак не избавиться, а значит, мы не можем рассчитывать на параллельное выполнение потоков.

На самом деле первая из озвученных выше проблем, связанная со сложной реализацией вычисления суммы, в данном случае имеет гораздо более разрушительный эффект по сравнению с ограничениями `GPU`. Даже если бы мы могли – а мы не можем – создать версию с параллельными вычислениями, потери, связанные с неуклюжим расчетом суммы, перевесили бы все приобретения от одновременного использования нескольких ядер процессора.

А теперь без лишних промедлений давайте взглянем на реализацию функции `add4_annotated`:

```
__pyx_v_i = (__pyx_v_my_number + 4)
```

Да, все так просто – обычное сложение на уровне языка `C`, которое выполнится на несколько порядков быстрее версии без аннотаций типов.

Вывод

Аннотации типов помогают Cython избавиться от кода на языке `C` от большей части инфраструктуры, связанной с интерпретатором `CPython`. В результате аннотированная версия кода на Cython будет выполняться значительно быстрее по сравнению с неаннотированной, в связи с чем я советую всегда, когда это возможно, пользоваться аннотациями типов.

5.2.4. Типизация возвращаемых из функции значений

Также вы можете задать тип возвращаемого функцией значения, как показано ниже:

```
cdef int add4_annotated_cret(int my_number):
    return my_number + 4
```

Обратите внимание, что теперь не только возвращаемое значение получило свой тип данных, но и сама функция стала объявляться с помощью ключевого слова `cdef`, а не `def`, как раньше. Эта функция может быть вызвана только из программы на языке `C`. Если вы попытаетесь вызвать ее в `Python`, она не будет работать ввиду отсутствия обертки. Какие существуют преимущества у такого подхода? Для функций Cython, которые вызываются только из других функций Cython, можно создать объявление, которое будет доступно как из `Python`, так и из Cython (в этом случае будут использованы разные интерфейсы при вызове из `Python` и Cython):

```
---
cpdef int add4_annotated_cpret(int my_number):
    return add4_annotated_cret(my_number)
----
```

Здесь вы будете одновременно получать и функцию-обертку, и низкоуровневую реализацию.

Таким образом, у вас есть три способа объявления функций: исключительно для Cython – в виде `cdef`, для Python и Cython (`cpdef`) и только для Python (`def`). Всегда, когда вам необходимо будет проходить через родной интерфейс Python, вы будете платить производительностью. Проходя через интерфейс Cython, вы не будете ощущать такой дополнительной нагрузки. Как мы видели в предыдущем разделе, функция `def` генерирует оба уровня, но это лишь особенности конкретной реализации, не связанные напрямую с использованием ключевого слова `def`.

Почему бы всегда не использовать объявление `cpdef` вместо `def` и `cdef`? Иногда вам может понадобиться использовать `def` для установки дополнительных ограничений на реализацию функции, а иногда может возникнуть явная необходимость в добавлении аннотаций. `cdef` нужно использовать в случаях использования типов данных, которые Python не поймет, например *указателей* (pointers), присутствующих в коде Cython.

Вывод

Всегда используйте аннотацию типов в Cython. Преимущества здесь огромны, тогда как неудобства сводятся лишь к необходимости писать аннотации. Если это возможно, используйте ключевое слово `cdef`. В противном случае рассмотрите возможность рефакторинга кода, чтобы фрагменты, связанные с Python, были вынесены в `def/cpdef`, а все операции с высокой вычислительной нагрузкой выполнялись в `cdef`.

Теперь, когда вы понимаете, почему аннотации так важны с точки зрения производительности, давайте посмотрим, как можно улучшить код на Cython с помощью его профилирования.

5.3. Профилирование кода на Cython

Вернемся к нашему коду для обработки изображения на Cython. И хотя он отработал гораздо быстрее, чем код, написанный на Python, особенно шустрым назвать его не получается. В конце концов мы имеем дело с простым применением фильтра к изображению, а операция выполняется больше секунды. Да, интуиция подсказывает нам, что наш код несовершенен, но, как мы узнали в главе 2, применительно к оценке производительности интуиция не лучший помощник, а значит, наш путь снова лежит в область профилирования кода, на этот раз на Cython.

Профилирование кода, написанного на Cython, укладывается в общую парадигму оценки производительности кода, написанного на Python. Таким образом, все те техники, которые мы озвучили

в главе 2, прекрасно подойдут и здесь, а это значит, что мы можем применить к нашей функции построчное профилирование для поиска узких мест.

5.3.1. Использование встроенной инфраструктуры профилирования Python

Начнем с использования встроенных средств профилирования кода. Первое, что нам нужно сделать, – это аннотировать наш код на Cython, чтобы его можно было профилировать. Сделать это довольно просто. Ниже приведена дополнительная аннотация нашей функции затемнения изображения. Код можно найти в файле 05-cython/sec3-profiling/cython_prof.py:

```
# cython: profile=True
import numpy as np

cimport cython
cimport numpy as cnp

def darken_annotated(
    cnp.ndarray[cnp.uint8_t, ndim=3] image,
    cnp.ndarray[cnp.uint8_t, ndim=2] darken_filter):
    cdef int nrows = image.shape[0]
    ...
```

← Говорим Cython о том, что мы собираемся выполнять профилирование кода

Это не сложнее, чем добавить в код глобальную директиву. Если вы по какой-то причине не хотите профилировать ту или иную функцию в коде, просто добавьте перед ней инструкцию `@cython.profile(False)`.

В качестве расширения примера из главы 2 давайте воспользуемся встроенным модулем `pstats` для сбора профилировочной информации. Ниже показан код вызывающей функции, который можно найти в файле `code/05-cython/sec3-profiling/apply_filter_prof.py`:

```
import cProfile
import pstats
import pyximport

import numpy as np
from PIL import Image

pyximport.install(
    setup_args={
        'include_dirs': np.get_include()})

import cyfilter_prof as cyfilter

image = Image.open("../04-numpy/aurora.jpg")
gray_filter = Image.open("../filter.png").convert("L")
image_arr, gray_arr = np.array(image), np.array(gray_filter)
```

← Мы позаботимся о запуске профилирования внутри кода

← Модуль `pstats` обрабатывает вывод профайлера

```
# Профилирование кода
cProfile.run("cyfilter.darken_annotated(image_arr, gray_arr)",
             "apply_filter.prof")
s = pstats.Stats("apply_filter.prof")
s.strip_dirs().sort_stats("time").print_stats()
```

← Вызываем профайлер для нашей функции

← Воспользуемся модулем pstats для вывода собранной статистики

В плане профилирования в этом коде нет ничего такого, что относилось бы конкретно к Cython. Вывод приведенного выше кода показан ниже:

```
Tue May 10 14:43:03 2022    apply_filter.prof
      5 function calls in 0.707 seconds
      Ordered by: internal time

ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
      1   0.707    0.707    0.707    0.707  cyfilter_prof.pyx:9
                                   (darken_annotated)
      1   0.000    0.000    0.707    0.707  {built-in method
                                   builtins.exec}
      1   0.000    0.000    0.707    0.707  <string>:1(<module>)
      1   0.000    0.000    0.707    0.707
                                   {cyfilter_prof.darken_annotated}
      1   0.000    0.000    0.000    0.000  {method 'disable' of
                                   '_lsprof.Profiler' objects}
```

Подробности вывода мы обсуждали в главе 2. Как мы видели там же, встроенный профайлер зачастую выводит меньше информации, чем нам бы хотелось. Давайте пойдем дальше и рассмотрим процесс построчного профилирования кода, на этот раз в специфике Cython.

5.3.2. Использование *line_profiler*

Здесь, как и в главе 2, мы воспользуемся модулем *line_profiler*. Для этого сперва подготовим наш код на Cython к построчному профилированию. Код можно найти в файле 05-cython/sec3-profiling/cython_lprof.py:

```
# cython: linetrace=True
# cython: binding=True
# cython: language_level=3
import numpy as np
cimport cython
cimport numpy as cnp

cpdef darken_annotated(
    cnp.ndarray[cnp.uint8_t, ndim=3] image,
    cnp.ndarray[cnp.uint8_t, ndim=2] darken_filter):
    cdef int nrows = image.shape[0] # Explain
```

← Указываем Cython подготовить код для построчного профилирования

← Нам нужно выполнить связывание наподобие Python

```

cdef int ncols = image.shape[1]
cdef cnp.uint8_t dark_pixel
cdef cnp.uint8_t mean # define here
cdef cnp.ndarray[cnp.uint8_t] pixel
cdef cnp.ndarray[cnp.uint8_t, ndim=2]
    dark_image = np.empty(shape=(nrows, ncols), dtype=np.uint8)
for row in range(nrows):
    for col in range(ncols):
        pixel = image[row, col]
        mean = (pixel[0] + pixel[1] + pixel[2]) // 3
        dark_pixel = darken_filter[row, col]
        dark_image[row, col] = mean * (255 - dark_pixel) // 255
return dark_image

```

Единственное изменение, которое нам пришлось внести, состоит в указании Cython подготовить код для построчного профилирования. Для этого мы воспользовались директивой `# cython: linetrace=True`. Вы также можете активировать возможность построчного профилирования на уровне функций. В этом случае вместо общих директив можете вставлять отдельные аннотации для функций, которые собираетесь профилировать:

```

@cython.binding(True)
@cython.linetrace(True)

```

Как вы помните из главы 2, процесс профилирования по строкам выполняется достаточно долго. В связи с этим Cython требует, чтобы вы не только аннотировали свой код при помощи директивы `linetrace`, но также явным образом запрашивали трассировку во время использования кода. Чтобы увидеть, как это происходит, давайте взглянем на код, вызывающий нашу функцию (на стороне Python):

```

import pyximport
import line_profiler
import numpy as np
from PIL import Image

pyximport.install(
    language_level=3,
    setup_args={
        'options': {'build_ext':
            {"define": 'CYTHON_TRACE'}},
        'include_dirs': np.get_include())

import cyfilter_lprof as cyfilter

image = Image.open("../04-numpy/aurora.jpg")
gray_filter = Image.open("../filter.png").convert("L")
image_arr, gray_arr = np.array(image), np.array(gray_filter)

```

← Импортируем модуль `line_profiler`

← Нам нужно скомпилировать код на языке C с макросом `CYTHON_TRACE`

```

profile = line_profiler.LineProfiler(
    cyfilter.darken_annotated)
profile.runcall(cyfilter.darken_annotated, image_arr, gray_arr)
profile.print_stats()

```

Здесь мы явным образом вызываем модуль `line_profiler`

Нужно не забыть активировать код на языке C, который выполняет всю работу. Этот код обернут в макрос и компилируется только в том случае, если компилятору передается директива `CYTHON_TRACE`. Мы делаем это с помощью инструктирования модуля `ruхimport` посредством системы `distutils`. В целом обсуждение системы макросов C и сборочной инфраструктуры Python выходит за рамки данной книги. Но вы должны убедиться, что макрос `CYTHON_TRACE` определен в той системе, которую вы используете для компиляции кода C. И помните, что `ruхimport` – это лишь один из вариантов.

Здесь мы воспользовались механизмом из модуля `line_profiler` напрямую, тогда как в главе 2 применили иной подход, вызвав наш код из скрипта `keгnprof`. Мы создаем объект `LineProfiler`, вызываем функцию `darken_annotated` внутри него и выводим на экран статистику. До запуска кода в качестве разминки для мозгов попытайтесь предположить, где именно окажется узкое место. После этого взгляните на рис. 5.2.

Total time: 3.58894 s
File: cyfilter_lprof.pyx
Function: darken_annotated at line 11

Line #	Hits	Time	Per Hit	% Time	Line Contents
11					cpdef darken_annotated(12 cnp.ndarray[cnp.uint8_t, ndim=3] image, 13 cnp.ndarray[cnp.uint8_t, ndim=2] darken_filter): 14 cdef int nrows = image.shape[0] # Explain 15 cdef int ncols = image.shape[1] 16 cdef cnp.uint8_t dark_pixel 17 cdef cnp.uint8_t mean # define here 18 cdef cnp.ndarray[cnp.uint8_t] pixel 19 cdef cnp.ndarray[cnp.uint8_t, ndim=2] dark_image = np.empty(shape=(nrows, ncols), dtype=cnp.uint8) 20 for row in range(nrows): 21 for col in range(ncols): 22 pixel = image[row, col] 23 mean = (pixel[0] + pixel[1] + pixel[2]) // 3 24 dark_pixel = darken_filter[row, col] 25 dark_image[row, col] = mean * (255 - dark_pixel) // 255 26 return dark_image
11	1	2.0	2.0	0.0	
15	1	0.0	0.0	0.0	
18	1	7.0	7.0	0.0	
20	1	0.0	0.0	0.0	
21	2048	337.0	0.2	0.0	
22	2799616	2151338.0	0.8	59.9	
23	2799616	481388.0	0.2	13.4	
24	2799616	477328.0	0.2	13.3	
25	2799616	478551.0	0.2	13.3	
26	1	1.0	1.0	0.0	

Рис. 5.2. Вывод статистической информации о профилировании кода

Время выполнения нашей функции составило 3,5 с, что больше по сравнению с тем временем, которое мы видели ранее. Не забывайте, что процесс профилирования несет с собой большие накладные расходы. Мы не будем сравнивать времена при обычном и построчном профилировании, поскольку они будут сильно отличаться. Нам также следует проявлять терпение при выполнении профилирования кода.

Как видно на рис. 5.2, вполне безобидная строка кода `pixel = image[row, col]` отнимает порядка 60 % всего времени. Признать-тесь, вы ожидали этого?

Чтобы понять, что именно приводит к такому замедлению, проще всего снова запустить инструкцию `cython cyfilter_lprof.py` и посмотреть на сгенерированный код. Здесь нам проще будет выпол-

нить анализ кода на С, чем в первом примере. Мы можем создать соответствующий веб-отчет прямо с помощью Cython, выполнив следующую инструкцию с опцией `-a`: `cython -a cyfilter_lprof.py`. В результате будет создан файл `cyfilter_lprof.htm`, который можно просмотреть в любом браузере. На рис. 5.3 показано основное представление нашей функции. Вы можете щелкать по строкам кода и смотреть, какой код на языке С был для них сгенерирован. Выделенные строки в скрипте (в браузере они должны быть подсвечены желтым) говорят о взаимодействии с движком Python, а это является индикатором возможного замедления работы кода.

```
Generated by Cython 0.29.21

Yellow lines hint at Python interaction.
Click on a line that starts with a "+" to see the C code that Cython generated for it.

Raw output: cyfilter\_lprof.c

+01: # cython: language_level=3
02: # cython: linetrace=True
03: # cython: binding=True
+04: import numpy as np
05: cimport cython
06: cimport numpy as cnp
07:
08:
09: @cython.binding(True)
10: @cython.linetrace(True)
+11: cpdef darken_annotated(
12:     cnp.ndarray[cnp.uint8_t, ndim=3] image,
13:     cnp.ndarray[cnp.uint8_t, ndim=2] darken_filter):
+14:     cdef int nrows = image.shape[0] # Explain
+15:     cdef int ncols = image.shape[1]
16:     cdef cnp.uint8_t dark_pixel
17:     cdef cnp.uint8_t mean # define here
18:     cdef cnp.ndarray[cnp.uint8_t] pixel
+19:     cdef cnp.ndarray[cnp.uint8_t, ndim=2] dark_image = np.empty(shape=(nrows, ncols), dtype=np.uint8)
+20:     for row in range(nrows):
+21:         for col in range(ncols):
+22:             pixel = image[row, col]
+23:             mean = (pixel[0] + pixel[1] + pixel[2]) // 3
+24:             dark_pixel = darken_filter[row, col]
+25:             dark_image[row, col] = mean * (255 - dark_pixel) // 255
+26:     return dark_image
```

Рис. 5.3. Веб-представление файла `cyfilter_lprof.html`. Подсвеченные строки говорят о взаимодействии с Python

Если вы раскроете строку под номером 22, в которой находится наш «безобидный» код `pixel = image[row, col]`, то с удивлением обнаружите, что тут выполняется не только присваивание. Вместо этого здесь производится множество вызовов на языке С с участием таких подозрительных с точки зрения производительности объектов, как `__Pyx_PyInt_From_int`, `PyTuple_New`, `__Pyx_PyObject_GetItem` и `__Pyx_SafeReleaseBuffer`. Как-то многовато всего для простой операции присваивания.

Вывод

Несмотря на все наши усилия по добавлению аннотаций типов в код на Cython, процесс профилирования выявил, что внутри продолжают использоваться массивы NumPy. А если низкоуровневый код взаимодействует с объектами Python, такими как NumPy, то и весь движок Python продолжает участвовать в процессе запу-

ска и выполнения кода, что неминуемо приводит к падению производительности нашего решения.

В результате возникает вопрос: а можем ли мы как-то более эффективно обращаться с этими массивами, что позволит сделать «безобидный» код присваивания действительно безобидным? Ответ вас обрадует – можем!

5.4. Оптимизация доступа к массивам в Cython с помощью *memoryview*

Для ускорения кода нам необходимо уменьшить количество взаимодействий с объектами Python до минимума, а в идеале до нуля. Таким образом, нам нужно избавиться от встроенных объектов Python и представлений массивов NumPy. В нашем примере массивы представлены в виде объектов Python, и необходимо изменить это положение вещей.

5.4.1. Использование представлений памяти

Расширение Cython располагает концепцией *представлений памяти* (*memoryview*) для массивов NumPy, схожей с одноименной концепцией, которую мы обсуждали в предыдущей главе. Таким образом, у Cython есть возможность обращаться напрямую к сырому представлению массивов без использования вычислительных средств Python. Мы разделим наш код на Cython на две функции: одна будет взаимодействовать с объектами Python, которые просто не способны показывать высокую скорость, а вторая будет работать на скоростях языка C. В результате одна будет принимать массивы NumPy и подготавливать объекты *memoryview*, а вторая – применять фильтр. Код можно найти в папке 05-cython/sec4-memoryview. Давайте начнем с функции, принимающей на вход массивы NumPy и подготавливающей представления памяти:

```
cpdef darken_annotated(
    cnp.ndarray[cnp.uint8_t, ndim=3] image,
    cnp.ndarray[cnp.uint8_t, ndim=2] darken_filter):
    cdef int nrows = image.shape[0]
    cdef int ncols = image.shape[1]
    cdef cnp.ndarray[cnp.uint8_t, ndim=2] dark_image =
        np.empty(shape=(nrows, ncols), dtype=np.uint8)
    cdef cnp.uint8_t[:, :] dark_image_mv
    cdef cnp.uint8_t[:, :] image_mv
    cdef cnp.uint8_t[:, :] darken_filter_mv
    dark_image_mv = dark_image
    darken_filter_mv = darken_filter
    image_mv = image
    darken_annotated_mv(image_mv, darken_filter_mv, dark_image_mv)
    return dark_image
```

Наконец,
вызываем
новую функ-
цию `darken_annotated_mv`, которая
будет рабо-
тать только
с представ-
лениями

Объявление
представления
памяти, которое
будет направле-
но на исходные
данные массива `dark_image`

Разрешаем Cython обращаться к исходным данным массива NumPy

Обратите внимание на синтаксис объявления *представлений памяти* (memoryview). Они должны иметь тип данных языка C, и их размеры должны быть известны (например, [:,:,]) для трех измерений в изображении). Cython позаботится о том, чтобы переменные, объявленные как представления памяти, обращались к сырым данным массивов с использованием правильных значений свойств `strides` и `shape`.

Теперь давайте взглянем на новую внутреннюю функцию:

```
cpdef darken_annotated_mv(
    cnp.uint8_t[:,:,:] image_mv,
    cnp.uint8_t[:, :] darken_filter_mv,
    cnp.uint8_t[:, :] dark_image_mv):
    cdef int nrows = image_mv.shape[0]
    cdef int ncols = image_mv.shape[1]
    cdef cnp.uint8_t dark_pixel
    cdef cnp.uint8_t mean # define here
    cdef cnp.uint8_t[:] pixel
    for row in range(nrows):
        for col in range(ncols):
            pixel = image_mv[row, col]
            mean = (pixel[0] + pixel[1] + pixel[2]) // 3
            dark_pixel = darken_filter_mv[row, col]
            dark_image_mv[row, col] = mean * (255 - dark_pixel) // 255
```

Меняем типы входных параметров с массивов NumPy на представления

Вывод функции теперь тоже представлен параметром

В результате полученный код оказался весьма похож на исходную версию. Типы входных данных изменились с массивов на представления памяти, а для порядка мы также передали выходное представление в качестве параметра.

Как показано на рис. 5.4, быстродействие немного выросло.

Total time: 1.82865 s
File: cyfilter_mv.pyx
Function: darken_annotated_mv at line 10

Line #	Hits	Time	Per Hit	% Time	Line Contents
10					cpdef darken_annotated_mv(
11					cnp.uint8_t[:,:,:] image_mv,
12					cnp.uint8_t[:, :] darken_filter_mv,
13					cnp.uint8_t[:, :] dark_image_mv):
14	1	2.0	2.0	0.0	cdef int nrows = image_mv.shape[0]
15	1	2.0	2.0	0.0	cdef int ncols = image_mv.shape[1]
16					cdef cnp.uint8_t dark_pixel
17					cdef cnp.uint8_t mean # define here
18					cdef cnp.uint8_t[:] pixel
19	1	0.0	0.0	0.0	for row in range(nrows):
20	2048	358.0	0.2	0.0	for col in range(ncols):
21	2799616	469718.0	0.2	25.7	pixel = image_mv[row, col] <1>
22	2799616	452761.0	0.2	24.8	mean = (pixel[0] + pixel[1] + pixel[2]) // 3
23	2799616	452165.0	0.2	24.7	dark_pixel = darken_filter_mv[row, col]
24	2799616	453647.0	0.2	24.8	dark_image_mv[row, col] = mean * (255 - dark_pixel) // 255

Рис. 5.4. Построчное профилирование обновленных функций

Мы получили примерно 50-процентное улучшение по скорости, а наша строка преткновения `pixel = image_mv[row, col]` стала действительно более безобидной, поскольку теперь для ее работы

не нужны объекты Python. И все же интуиция нам подсказывает, что функция расходует слишком много времени для простой манипуляции с изображением.

Оказывается, в нашем исправленном коде осталось достаточно много взаимодействий с механизмами Python. Если мы воспользуемся инструкцией `cython -a` для генерации веб-страницы с представлением нашего кода на языке C, то увидим, как показано на рис. 5.5, что у нас присутствует довольно много выделенных строк.

```
+09: cpdef darken_annotated_mv(
10:     cnp.uint8_t[:, :, :] image_mv,
11:     cnp.uint8_t[:, :] darken_filter_mv,
12:     cnp.uint8_t[:, :] dark_image_mv):
+13:     cdef int nrows = image_mv.shape[0]
+14:     cdef int ncols = image_mv.shape[1]
15:     cdef cnp.uint8_t dark_pixel
16:     cdef cnp.uint8_t mean # define here
17:     cdef cnp.uint8_t[:] pixel
+18:     for row in range(nrows):
+19:         for col in range(ncols):
+20:             pixel = image_mv[row, col]
+21:             mean = (pixel[0] + pixel[1] + pixel[2]) // 3
+22:             dark_pixel = darken_filter_mv[row, col]
+23:             dark_image_mv[row, col] = mean * (255 - dark_pixel) // 255
24:
```

Рис. 5.5. Вывод для функции на основе *memoryview*. Подсвеченные строки символизируют связь с движком Python

Вывод

Обычно время, потраченное на создание представлений памяти на основе массивов NumPy, себя оправдывает, поскольку это позволяет Cython работать напрямую с представлением массивов в памяти и избегать взаимодействий с Python. В результате быстроедействие программы может серьезно возрасти. В нашем случае мы провели дополнительное профилирование и выяснили, что взаимодействия с движком Python у нас остались, и в следующем разделе мы с этим поборемся.

5.4.2. Избавление от всех взаимодействий с Python

В нашей программе осталось три вида взаимодействия с движком Python, которые показаны на рис. 5.5:

- определение функции `cpdef` генерирует функцию на языке C с заглушкой для Python. Мы можем заменить определение на `cdef`;
- функция неявным образом возвращает объект `None`, что ожидается от всех функций Python. Одного этого уже достаточно, чтобы был запущен процесс управления объектами Python, даже если речь идет об объекте `None`;
- механизм *memoryview* в NumPy по-прежнему пытается помогать нам с определением границ массивов. Например, при указа-

нии индекса, находящегося за пределами массива, возникнет исключение Python.

Давайте решим все эти проблемы одним махом. Для этого нам необходимо внести изменения в определение функции, как показано ниже:

<div style="border-left: 1px solid black; padding-left: 5px;"> Деактивируем контроль за границами массивов </div>	<pre> @cython.boundscheck(False) cdef void darken_annotated_mv(cnp.uint8_t[:, :, :] image_mv, cnp.uint8_t[:, :] darken_filter_mv, cnp.uint8_t[:, :] dark_image_mv) nogil: </pre>	<div style="border-left: 1px solid black; padding-left: 5px;"> Меняем определение функции на <code>cdef</code> (без заглушки для Python) и определяем тип возвращаемого значения <code>void</code>, принятый в C </div> <div style="border-left: 1px solid black; padding-left: 5px; margin-top: 10px;"> Теперь можно сказать Cython, что в этой функции могут не соблюдаться правила GIL </div>
---	---	--

Позже мы еще затронем тему контроля за границами массивов. В данный момент ваш код может завершиться аварийно, если не будут соблюдены допустимые рамки существующих массивов. В нашем случае это не станет проблемой, но далее в этой главе посмотрим, как не допустить подобного.

Аннотация *nogil* является здесь опциональной. Мы ничего от этого не выиграем. Использование этой опции позволяет использовать настоящий параллелизм при выполнении программы, и позже мы поговорим об этом подробнее. Cython выдаст ошибку во время компиляции, если при использовании опции *nogil* вы не избавитесь от всех связей с Python. Таким образом, мы смогли применить ее лишь потому, что исправили все недочеты и избавились от подсвеченных линий в скомпилированном коде.

На моем ноутбуке с процессором Intel i5 CPU с рабочей частотой 1,6 ГГц код выполнялся за 0,04 с. Если вы помните, начали мы с 35 секунд с нативной реализацией на Python и 18 секунд – с первой реализацией на Cython.

Вывод

Чтобы избавиться от последних связей между Cython и Python, вы можете изменить определения функций, что позволит избежать совершения или возврата вызовов функций Python. В сочетании с полным аннотированием кода на Cython, включая типизацию возвращаемых значений и использование представлений памяти вместо обычных массивов NumPy, у вас в руках будет полная стратегия по избавлению вашего быстрого кода от привязок к медленному Python.

Позже мы еще вернемся к обсуждению контроля за границами массивов и оптимизации NumPy. Мы также уделим время параллелизму. Но сейчас пришло время узнать, как можно реализовать универсальные функции NumPy при помощи Cython. Это бывает очень полезно, поскольку универсальные функции поддерживают правила транслирования NumPy.

5.5. Написание обобщенных универсальных функций NumPy на Cython

В этом разделе мы применим альтернативный подход к решению задачи фильтрации изображения – с помощью *универсальной функции* (universal function), написанной на Cython. В предыдущей главе мы уже говорили о том, что механизмы универсальных функций значительно облегчают нам жизнь за счет использования таких полезных возможностей, как транслирование (broadcasting). Также универсальные функции дают ряд дополнительных привилегий и в целом имеют много общего с парадигмой программирования для графического процессора. Но стоит помнить, что они не представляют собой универсальное вычислительное решение для всех задач, и в следующем разделе мы рассмотрим один из примеров на эту тему.

В главе 4 мы также усвоили, что универсальные функции работают поэлементно. В нашем случае это означает, что обработка будет производиться по пикселям. Наш код будет состоять из двух частей: универсальной функции и кода для ее регистрации. Начнем с самой функции. Код можно найти в папке 05-cython/sec5-ufunc:

```
# cython: language_level=3
import numpy as np
cimport cython
cimport numpy as cnp

cdef void darken_pixel(
    cnp.uint8_t* image_pixel,
    cnp.uint8_t* darken_filter_pixel,
    cnp.uint8_t* dark_image_pixel) nogil:
    cdef cnp.uint8_t mean
    mean = (image_pixel[0] + image_pixel[1] + image_pixel[2]) // 3
    dark_image_pixel[0] = mean * (255 - darken_filter_pixel[0]) // 255
```

Обратите внимание на использование нотации указателей (*)

Поскольку мы теперь работаем с конкретными пикселями, наш код существенно упростился из-за отсутствия необходимости использования циклов для прохода по массивам/изображениям.

Фундаментальное отличие этой реализации состоит в том, что вместо передачи функции числовых значений в виде `cnp.uint8_t` мы передаем указатели на эти значения с помощью конструкции `cnp.uint8_t *`. Если вы не использовали в своей работе язык C, эта концепция может оказаться для вас новой. Для нашего конкретного примера использование указателей не имеет серьезных последствий, но при работе с более сложными сценариями вам придется почитать соответствующие разделы в документации по Cython. Единственным следствием применения указателей здесь является то, что результат вычислений нашей функции мы записываем обратно во входной параметр. Наконец, наша функция помечена

ключевым словом `nogil`, позволяющим ей запускаться параллельно. Мы в ней никаких объектов Python не используем, так что сможем без проблем обойти все ограничения, накладываемые GIL.

Наша универсальная функция, как и в предыдущей главе, является обобщенной, поскольку первым параметром (`image_pixel`) мы передаем не примитивный тип, а массив, – в цветном изображении пиксель обладает тремя компонентами RGB.

Теперь нам необходимо написать функцию-обертку для нашей универсальной функции. К сожалению, ее шаблон будет более длинным и запутанным:

```
cdef cnp.PyUFuncGenericFunction loop_func[1]
cdef char all_types[3]
cdef void *funcs[1]

loop_func[0] = cnp.PyUFunc_FF_F
all_types[0] = cnp.NPY_UINT8
all_types[1] = cnp.NPY_UINT8
all_types[2] = cnp.NPY_UINT8

funcs[0] = <void*>darken_pixel

darken = cnp.PyUFunc_FromFuncAndDataAndSignature(
    loop_func, funcs, all_types,
    1,
    2,
    1,
    0,
    "darken",
    "Darken a pixel", 0
    "(n),( )->()"
)
```

Количество входных типов

Количество входных параметров

Количество выходных параметров

Сигнатура Numpy

Нам понадобится переменная для хранения всех типов входных и выходных параметров

Все функции, реализующие универсальную функцию

Указываем типы данных для двух входных и одного выходного параметра

Список функций, реализующих универсальную функцию

Создание оболочки универсальной функции

Нам необходимо указать типы данных для всех параметров, для которых у нас выделена переменная `all_types`. По сигнатуре обобщенной универсальной функции `(n),()->()` мы видим, что на вход поступает массив `(n)` с тремя компонентами нашего исходного пикселя и примитивное значение `()`, представляющее затемняющий черно-белый пиксель, а на выход пойдет также значение примитивного типа `()`, характеризующее насыщенность серого в итоговом пикселе.

Вас может сбить с толку возможность наличия нескольких функций в одной реализации, ведь мы храним целый список функций `funcs`, а не одну функцию. В данном примере нам нужна была только одна функция `darken_pixel`, но мы могли бы прописать разные функции для разных входных и выходных параметров – скажем, одну для `NPY_UINT8`, а другую для `NPY_UINT16`.

Теперь можно использовать нашу реализацию как любую другую универсальную функцию. Например, так:

```
import pyximport

import numpy as np
from PIL import Image

pyximport.install(
    language_level=3,
    setup_args={
        'options': {"build_ext": {"define": 'CYTHON_TRACE'}},
        'include_dirs': np.get_include())

import cyfilter_uf as cyfilter

image = Image.open("../04-numpy/aurora.jpg")
gray_filter = Image.open("../filter.png").convert("L")
image_arr, gray_arr = np.array(image), np.array(gray_filter)

darken_arr = cyfilter.darken(image_arr, gray_arr)
```

Вывод

Реализация универсальных функций NumPy с помощью Cython зачастую возможна и предпочтительна, особенно с учетом того, что в этом случае вы сможете сделать их более быстрыми. Но бывает, что одних универсальных функций NumPy недостаточно для реализации задуманного алгоритма, например когда вам необходимо иметь доступ к другим элементам массива, а не только к текущему. Чтобы пояснить на практике, как быть с этими и другими сложностями при работе с массивами в Cython, мы рассмотрим и сами реализуем знаменитую игру «Жизнь» (англ. Game of Life).

5.6. Продвинутая работа с массивами в Cython

В этом разделе мы закрепим свои знания в области взаимодействия Cython и NumPy, погрузившись в оптимизацию работы с массивами. В результате мы хотим прийти к многопоточному параллельному низкоуровневому коду без всяких ограничений в виде GIL.

Здесь мы начнем работать с новым проектом, в рамках которого создадим цветную версию знаменитой игры «Жизнь» (*Game of Life*), придуманной Джоном Конвеем (John Conway). Подробности о ней вы можете найти по адресу <https://conwaylife.com>. Это игра, в которой не участвует ни один игрок. Она сама развивается в зависимости от заданного начального состояния, и именно фантазия при определении начального состояния игры составляет львиную

долю интереса. Под начальным состоянием имеется в виду сетка произвольного размера, каждая из ячеек которой может быть либо живой (включенной), либо мертвой (выключенной). После запуска игры каждая ячейка игрового поля будет циклически менять свое состояние согласно следующим простым правилам:

- если у живой ячейки есть два или три живых соседа, она остается жить;
- мертвая ячейка с тремя живыми соседями оживает;
- все остальные ячейки умирают или остаются мертвыми.

Наше игровое поле ограничено со всех сторон, но при определении соседей слева ячейка из крайнего левого столбца будет обращаться к ячейкам из крайнего правого столбца, и наоборот. Это же касается верхней и нижней строк игрового пространства.

На рис. 5.6 показаны три примера развития фигур во времени. Первый пример – это бесконечное вращение вертикальной линии из трех живых ячеек. Второй – устойчивое состояние квадрата размером 2×2 (его внешний вид с течением времени не меняется). А третий – исчезновение фигуры.

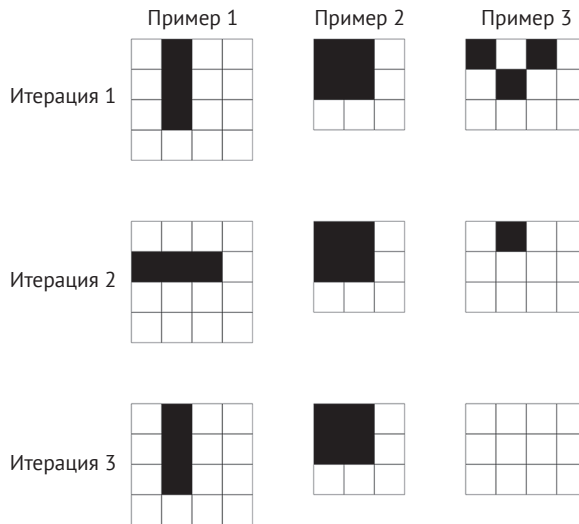


Рис. 5.6. Три примера развития фигур согласно правилам игры

Мы реализуем расширенную версию игры под названием QuadLife¹, в которой каждая ячейка может принимать четыре возможных состояния: красная, зеленая, синяя и желтая. Мне нравится это расширение просто потому, что оно симпатично выглядит. В этой версии добавляются два новых правила:

¹ Вы можете найти информацию о различных разновидностях игры на сайте <https://conwaylife.com>

- если какой-то цвет присутствует у большинства соседей ячейки, этот цвет присваивается новой ячейке;
- если все три соседние ячейки окрашены в разные цвета, новая ячейка приобретает четвертый цвет, отсутствующий в этом списке.

Как и в наших предыдущих примерах на Cython, эта реализация будет включать в себя две компоненты: вызывающий код на Python и вычислительный блок с использованием Cython.

Первая часть на Python пока будет простой и понятной. Этот код можно найти в папке 05-cython/sec6-quadlife. Он также приведен ниже с комментариями:

```
import sys

import numpy as np
import pyximport
pyximport.install(
    language_level=3,
    setup_args={
        'include_dirs': np.get_include()})

import cquadlife as quadlife

SIZE_X = int(sys.argv[1])
SIZE_Y = int(sys.argv[2])
GENERATIONS = int(sys.argv[3])

world = quadlife.create_random_world(SIZE_Y, SIZE_X)
for i in range(GENERATIONS):
    world = quadlife.live(world)
```

Настраиваем pyximport для включения NumPy

Читаем параметры из командной строки

Используем функцию (которая будет определена позже) для создания случайного мира

Запускаем алгоритм Quadlife столько раз, сколько укажет пользователь

При вызове этого скрипта мы будем передавать на вход три параметра, отвечающие за размер сетки и количество регенераций или циклов. На данный момент скрипт ничего не возвращает, он просто запускает игру. Позже мы немного изменим обработку результата.

Для начала будет достаточно, чтобы скрипт генерировал случайный мир путем вызова функции `create_random_world`, – этого хватит, чтобы протестировать его работу. Позже мы кое-что изменим в этом плане. Мы будем использовать массивы NumPy `SIZE_Y`, `SIZE_X` заданных пользователем размеров, которые поначалу будут заполнены случайными числами в диапазоне от 0 до 4. При этом ноль будет соответствовать мертвой ячейке. После этого мы запустим функцию с именем `live` в количестве раз, равном переданному на вход значению `GENERATIONS`, причем при первом вызове на вход функция будет принимать случайно сгенерированный мир, а при последующих – миры, образованные после очередной итерации.

В целом в представленном коде нет для вас ничего нового, и у вас не должно возникнуть проблем с ним.

Теперь приступим к нашему коду на Cython. Процедура создания случайного мира не требует особой оптимизации, поскольку она выполняется лишь раз в самом начале запуска:

```
#cython: language_level=3
import numpy as np

cimport cython
cimport numpy as cnp

def create_random_world(y, x):
    cdef cnp.ndarray [cnp.uint8_t, ndim=2] world =
        np.random.randint(0, 5, (y, x), np.uint8)
    return world
```

Теперь начинается самое интересное. Наша реализация будет включать в себя концептуально новые приемы, но на самом деле мы будем пользоваться знаниями, полученными в предыдущих разделах.

5.6.1. Обход ограничений GIL по запуску нескольких потоков одновременно

Первое, что нам необходимо сделать, – это убедиться в том, что наш внутренний цикл будет освобожден от ограничений, связанных с GIL. С этой целью мы создадим в Cython верхнеуровневую функцию `live`, которая будет заниматься преобразованием массивов NumPy в *представления памяти* (`memoryview`):

```
def live(cnp.ndarray[cnp.uint8_t, ndim=2] old_world):
    cdef int size_y = old_world.shape[0]
    cdef int size_x = old_world.shape[1]
    cdef cnp.ndarray[cnp.uint8_t, ndim=2] extended_world =
        np.empty((size_y + 2, size_x + 2), dtype=np.uint8) # пустой
    cdef cnp.ndarray[cnp.uint8_t, ndim=2] new_world =
        np.empty((size_y, size_x), np.uint8)
    cdef cnp.ndarray[cnp.uint8_t, ndim=1] states = np.empty((5,),
        np.uint8)

    live_core(old_world, extended_world, new_world, states)
    return new_world
```

Преобразование в представления памяти обусловлено сигнатурой вызываемой функции `live_core` (об этом мы поговорим далее), но нам все равно нужен слой, в котором объекты Python будут преобразовываться в свободные от ограничений GIL структуры. В переменной `old_world` хранится мир, переданный на вход; `new_world` – новый мир, а `extended_world` и `states` – это внутренние переменные функции `live_core`, которые объявляются здесь. Перед тем как по-

грузиться в алгоритм, заложенный в функцию `live_core`, давайте обсудим некий прием алгоритмической оптимизации, который мы будем применять.

Правила игры предполагают, что границы сетки как бы соединены между собой последовательно. Это означает, что при определении состояний соседних ячеек в крайнем левом столбце мы будем обращаться к ячейкам в крайнем правом столбце, и наоборот. Во избежание сложностей расчетов для пограничных элементов с образованием бесчисленного количества условий `if` и увеличением времени обработки мы решили создать временную расширенную сетку мира в переменной с именем `extended_world` в виде массива с размерностями $(y+2, x+2)$. При этом в дополненных по бокам и сверху и снизу ячейках будут присутствовать значения с противоположных сторон сетки, как показано на рис. 5.7.

0	1	2
3	0	4

4	3	0	4	3
2	0	1	2	0
4	3	0	4	3
2	0	1	2	0

Рис. 5.7. Расширенная сетка, используемая для вычисления нового мира

Цель применения этого алгоритма состоит в том, чтобы упростить расчеты при создании нового мира и не городить бесчисленные условия `if`. За свою идею мы заплатим памятью, поскольку теперь нам придется хранить в ней дополнительную сетку даже большего размера по сравнению с рабочей. С подобными компромиссами вам придется встречаться на своем пути оптимизатора сложных вычислительных задач постоянно. Здесь очень трудно или даже невозможно создать универсальную инструкцию, которая будет годиться на все случаи жизни оптимизатора. Все и всегда зависит от вычислительной сложности и требовательности к памяти тех или иных алгоритмов, а также от доступных вам ресурсов.

Ниже приведен код для реализации нашего расширенного мира. Обратите внимание, что в коде не используются проверки на границы сетки, чтобы не добавлять лишние условия:

```
@cython.boundscheck(False)  ← Отключаем проверку на границы, None
@cython.nonecheck(False)    ← и циклические переходы индексов в массивах
@cython.wraparound(False)
cdef void get_extended_world(  ← Используем определение cdef, чтобы
    cnp.uint8_t[:, :] world,    избежать ограничений GIL
    cnp.uint8_t[:, :] extended_world):  ← Типизируем все элементы
    cdef int y = world.shape[0]    в сигнатуре функции
```

```

cdef int x = world.shape[1]
extended_world[1:y+1, 1:x+1] = world  ←

extended_world[0, 1:x+1] = world[y-1, :] # верх
extended_world[y+1, 1:x+1] = world[0, :] # низ
extended_world[1:y+1, 0] = world[:, x-1] # лево
extended_world[1:y+1, x+1] = world[:, 0] # право

extended_world[0, 0] = world[y-1, x-1]
# верх слева
extended_world[0, x+1] = world[y-1, 0]
# верх справа
extended_world[y+1, 0] = world[0, x-1]
# низ слева
extended_world[y+1, x+1] = world[0, 0]
# низ справа

```

Копирование существующего мира в середину расширенного мира может оказаться дорогим с точки зрения ресурсов

Копирование содержимого переменной `world` в середину переменной `extended_world` в перспективе может привести к серьезным расходам в отношении использования памяти и вычислительных ресурсов. В то же время расходы на вычисления могут быть компенсированы за счет более простого основного алгоритма¹. Чисто в педагогических целях мы оставим этот прием, поскольку он позволит существенно снизить сложность алгоритма обработки.

Мы могли заметить, что некоторые строки в приведенном выше коде могли быть записаны в более лаконичной форме. Например, строка

```
extended_world[1:y+1, 1:x+1] = world
```

могла быть записана как:

```
extended_world[1:-1, 1:-1] = world
```

Но здесь мы не можем позволить себе такую лаконичность, поскольку ранее отключили проверку на *циклические переходы* (`@cython.wgaraound(False)`) индексов в массивах, чтобы повысить эффективность итогового кода на C. В результате мы потеряли возможность использовать отрицательные индексы в массивах. Но это приемлемый компромисс, поскольку разрешение циклических переходов подразумевает использование механизмов CPython, что замедлит наше решение в целом. Таким образом, чтобы избавиться от следов Python и позволить нашему коду обходить блокировки, обусловленные присутствием GIL, мы идем на этот шаг.

¹ Для определения того, получите ли вы такую компенсацию, нужно будет выполнить профилирование кода, и сейчас вы уже знаете, как это делать.

ПРЕДУПРЕЖДЕНИЕ. Отказ от проверок на границы или циклические переходы индексов в массивах может приводить к возникновению *ошибок сегментации* (segmentation faults) и аварийному завершению программы. Если вы наблюдаете подобные ошибки, уберите этот декоратор во время разработки. Ваш код должен надежно работать как с декораторами проверки, так и без них.

Также мы применили приемы оптимизации, обсуждавшиеся ранее: использовали определение `cdef`, выполнили полную типизацию параметров и переменных и воспользовались представлениями памяти вместо массивов NumPy. Теперь если вы выполните команду `cython -a squadlife.pyx`, то не увидите никаких подсвеченных строк в итоговом коде на C. А это значит, что наш код никак не привязан к Python.

В реализации основной функции, изменяющей состояния ячеек, мы воспользуемся созданным ранее расширенным миром. Приведенный ниже код реализует правила игры QuadLife, описанные ранее. Поскольку код получился довольно длинным, мы подробно прокомментируем его, включая проблемные места, которые уже постарались устранить:

```
@cython.boundscheck(False)
@cython.nonecheck(False)
@cython.wraparound(False)
cdef void live_core(
    cnp.uint8_t[:,:] old_world,
    cnp.uint8_t[:,:] extended_world,
    cnp.uint8_t[:,:] new_world,
    cnp.uint8_t[:] states):
    cdef cnp.uint16_t x, y, i
    cdef cnp.uint8_t num_alive, max_represented
    cdef int size_y = old_world.shape[0]
    cdef int size_x = old_world.shape[1]
    get_extended_world(old_world, extended_world)

    for x in range(size_x):
        for y in range(size_y):
            for i in range(5):
                states[i] = 0
            for i in range(3):
                states[extended_world[y, x + i]] += 1
                states[extended_world[y + 2, x + i]] += 1
                states[extended_world[y + 1, x]] += 1
                states[extended_world[y + 1, x + 2]] += 1

    num_alive = states[1] + states[2] + states[3] + states[4]
    if num_alive < 2 or num_alive > 3:
        # Слишком мало или слишком много соседей
        new_world[y, x] = 0
```

Отключаем проверку на границы, None и циклические переходы индексов в массивах

Используем определение `cdef` во избежание передачи объектов Python. Также описываем возвращаемое значение как `void`, что в C означает пустоту

Некоторые внутренние переменные (`states` и `extended_world`) определены вне функции, а мы используем доступную память

При вызове функции `get_extended_world` выделяется вся нужная память

Типизируем все параметры

Типизируем все локальные переменные

Реализация выражения `sum(states[:1])`

```

elif old_world[y, x] != 0:
    # Остаемся жить
    new_world[y, x] = old_world[y, x]
elif num_alive == 3: # Оживаем
    max_represented = max(states[1], max(states[2],
    max(states[3], states[4])))
    if max_represented > 1:
        # правило большинства для выбора цвета
        for i in range(1, 5):
            if states[i] == max_represented:
                new_world[y, x] = i
                break
    else:
        # используем цвет, которого нет
        for i in range(1, 5):
            if states[i] == 0:
                new_world[y, x] = i
                break
else:
    new_world[y, x] = 0 # Остаемся мертвыми

```

Реализация выражения `max(states[:1])`

Реализация выражения `states[1:].index(max_represented)`

Функция оказалась достаточно сложной, но большинство присутствующих в ней техник мы уже подробно обсудили ранее, а здесь просто собрали воедино в реалистичном примере. Внимательно прочитайте код и все комментарии к нему, и вы без труда все поймете.

При ближайшем рассмотрении кода у вас могут появиться вопросы относительно замены функций `sum` и `index` более декларативными выражениями. Причина в том, что эти функции используют движок CPython, а мы делаем все, чтобы избежать связи с ним. Похожая аргументация уместна и применительно к функции `max`, но в этом случае мы не сможем обойтись единственным вызовом. При использовании обобщенных функций вам может понадобиться выполнить их профилирование и заменить оптимизированными не-обобщенными функциями.

ПРИМЕЧАНИЕ. Поскольку смысл этой игры состоит в создании красивых итерационных визуализаций, мы создадим для нее простой графический интерфейс. Для этого воспользуемся встроенным модулем *tkinter* совместно с графическим пакетом *Pillow*. Мы не будем здесь обсуждать полученный код, поскольку это выходит за рамки книги. Но вы можете найти его в файле `05-cython/sec6-quadlife/gui.py`.

Итак, мы создали наше приложение, и теперь необходимо проверить его быстроедействие. Этим мы сейчас и займемся.

5.6.2. Базовый анализ производительности

В репозитории вы можете найти версию приложения на чистом Python. Мы будем использовать ее для сравнения быстрогодействия

с нашей реализацией на Cython. Запуск версии Python с размерами игрового поля 1000×1000 для 200 итераций на моем компьютере занял чуть меньше 1000 с, что составляет порядка 17 мин. Код Cython выполнен за 2,5 с.

ПРЕДУПРЕЖДЕНИЕ. Наша реализация завязана на использование довольно большого объема памяти. Будьте осторожны при ее запуске с очень большими размерами сетки. В целом в этой книге мы уделяем большое внимание требовательности описываемых алгоритмов к памяти. Если алгоритм способен обходиться оперативной памятью, он будет работать гораздо быстрее по сравнению с алгоритмом, требующим в процессе своей работы сохранения данных на диск. Всегда, когда это возможно, мы будем стараться использовать алгоритмы, уместящиеся в памяти. Если это невозможно, постараемся оптимизировать хранение информации для повышения эффективности ее обработки.

Теперь давайте проверим производительность на игровом поле размером $400 \times 900\,000$ всего на четырех итерациях. На моем компьютере этот запуск занял порядка 44 с. Как вы думаете, если поменять размерности и проверить работу программы на поле $900\,000 \times 400$, какие будут результаты? Количество ячеек то же, и такое же число итераций. Вы, наверное, удивитесь, но результат окажется совершенно иным, и даже не близко к нашему первому случаю. Запуск займет всего 20 с. Как итог – исходные данные такие же, а результат совершенно иной. В чем дело? Ответ на этот вопрос ждет вас в главе 6. Чтобы еще больше вас заинтриговать, скажу, что относительные результаты на моем и вашем компьютере могут сильно отличаться.

Перед тем как перейти к заключительному разделу главы, посвященному параллельной многопоточности в Cython, давайте закольцуем тему с нашей увлекательной игрой «Жизнь» и сгенерируем по ней целое видео. Этот процесс потребует от нас рассмотрения вопросов, связанных с вычислительной сложностью алгоритмов, и использования теории при написании более эффективных программ.

5.6.3. Космические войны в Quadlife

В репозитории вы также найдете код, с помощью которого можно сгенерировать видео нашей игры со стартовой позицией с космическими кораблями и оборонительными редутами. Сам код не относится к теме оптимизации, так что мы не будем рассматривать его подробно. Если захотите воспроизвести его самостоятельно, вам понадобятся библиотеки Pillow для работы с изображениями и ffmpeg – для обработки видео. Скрипт для запуска из командной оболочки располагается в файле `05-cython/generate_video.sh`. На рис. 5.8 показана начальная расстановка сил с инвертированными цветами.

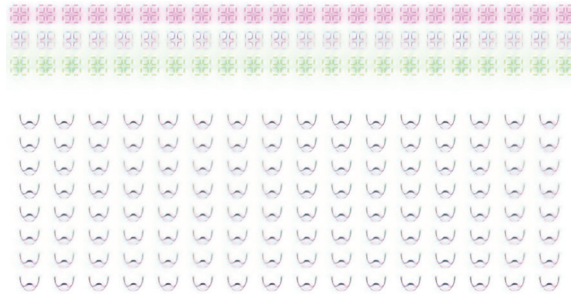


Рис. 5.8. Начальная позиция для видеоигры QuadLife

Сохраненное видео вы можете посмотреть по адресу https://www.youtube.com/watch?v=E0B1fDKU_MI. А на сайте <https://conwaylife.com> можете также загрузить различные шаблоны, подобные тому, который мы использовали при записи видео.

Код из репозитория, находящийся в файле `05-cython/patterns.py`, можно использовать для создания таких видео. По умолчанию будет запущена игра из 400 итераций на карте размером 400×250 , и у меня такой запуск занял порядка 1 с. В режиме HD с разрешением 1920×1080 с теми же 400 итерациями потеря времени составила 11 с. На 800 итераций ушло 22 с, а эмуляция режима 4K с разрешением 3840×2160 и 400 кадрами заняла 48 с. При частоте смены кадров 40 к/с 90-минутная война в режиме 4K будет сгенерирована примерно за 196 мин. Для сравнения, создание аналогичного видео с использованием кода на Python заняло бы порядка 54 дней!

Значение вычислительной сложности алгоритмов

Хотя эта книга посвящена исключительно практическим методам оптимизации кода, без теоретических вопросов, связанных с вычислительной сложностью алгоритмов, нам здесь просто не обойтись. Под вычислительной сложностью подразумеваются ресурсы, необходимые для работы алгоритма, — главным образом это время выполнения и память.

К примеру, мы ожидаем, что увеличение количества итераций алгоритма должно приводить к линейному росту времени выполнения кода. Но когда мы имеем дело с двумерным игровым полем со стороной n , рост будет квадратичным: обработка поля со стороной 20 ячеек будет обрабатываться не в два, а в четыре раза медленнее по сравнению с игровым пространством со стороной 10 ячеек. Если мы говорим о стороне 200, коэффициент замедления составит умопомрачительные 400, а не 20, как можно было ожидать изначально. Квадратично будут расти и требования алгоритма к затрачиваемой памяти.

В современном мире с постоянным ростом объемов данных это означает, что способности многих алгоритмов к масштабированию может оказаться недостаточно, и им на смену должны будут прийти более продвинутые решения. В этой книге мы не касаемся вопросов вычислительной сложности алгоритмов напрямую, но иногда будем напоминать вам об их важности.

Если вы думаете, что мы покончили с нашей игрой, то нет. Мы избавили ее код от взаимодействий с Python, а значит, полностью подготовились к тому, чтобы в обход GIL реализовать настоящий параллелизм с многопоточностью. Давайте сделаем это в нашем заключительном разделе главы.

5.7. Параллелизм с Cython

Мы сделали очень много, чтобы избавить код от оков GIL, так что сделать его параллельным теперь не составит труда. Воспользуемся внутренним механизмом параллельных вычислений, заложенным в Cython, и код, который вы можете найти в папке 05-cython/sec7-parallel, покажется вам достаточно простым.

Cython располагает декларативными параллельными функциями, базирующимися на OpenMP. В свою очередь *OpenMP* представляет собой мультиплатформенную библиотеку для работы с параллельными примитивами. Одна из функций, обеспечивающих параллельные вычисления, — *prange*, — разбивает итерации цикла *for* на отдельные потоки, и использовать ее крайне просто, что видно по приведенному ниже коду:

```
from cython.parallel import prange  ← Импортируем функцию prange
@cython.boundscheck(False)
@cython.nonecheck(False)
@cython.wraparound(False)
cdef void live_core(
    cnp.uint8_t[:,:] old_world,
    cnp.uint8_t[:,:] extended_world,
    cnp.uint8_t[:,:] new_world,
    cnp.uint8_t[:] states) nogil:  ← Теперь ключевое слово nogil является
                                обязательным
    cdef cnp.uint32_t x, y, i
    cdef cnp.uint8_t num_alive, max_represented
    cdef int size_y = old_world.shape[0]
    cdef int size_x = old_world.shape[1]
    get_extended_world(old_world, extended_world)

    for x in prange(size_x):  ← Просто меняем range на prange
        for y in range(size_y):
            ...
```

Да, вот так все просто. Но не забывайте, какое количество работы по избавлению от ограничений GIL мы выполнили до этого, только чтобы добраться до этого этапа. Также нам нужно не забыть добавить аннотации к функции `get_extended_world`:

```
@cython.boundscheck(False)
@cython.nonecheck(False)
@cython.wraparound(False)
```

```
cdef void get_extended_world(  
    cnp.uint8_t[:,:] world,  
    cnp.uint8_t[:,:] extended_world) nogil:  
...
```

Расширение Cython посредством библиотеки OpenMP предлагает воспользоваться несколькими функциями для простой и понятной реализации полноценного параллельного кода. И это может быть крайне полезно в самых разных ситуациях. Нашей главной задачей было избавить наш код от всех ограничений, наложенных `GiL`, что мы успешно сделали.


Большую часть времени мы посвятили взаимоотношениям между Python, для которого характерен `GiL`, и многопоточным параллелизмом. Также мы узнали о примитивах параллельных вычислений Cython, основанных на библиотеке OpenMP, которые необходимо использовать при написании параллельного кода. Но главным действующим лицом остается сама параллельная обработка данных. И здесь мы приоткрыли завесу того, в какую сторону необходимо двигаться, если вы хотите реализовать в Python полноценный многопоточный параллелизм. В целом же парадигма параллельного программирования – это целый дивный мир, для углубления в который вам придется искать специализированные ресурсы.

Не забывайте о том, что, используя расширение Cython, вы, по сути, переходите в парадигму языка C. И хотя у вас нет никаких причин отказываться от использования функционала библиотеки OpenMP в Cython, помните, что вы к ней жестко не привязаны. Это означает, что вы вольны использовать любые другие библиотеки для работы с параллельной обработкой на базе языка C. Другое дело, что для этого вам придется опуститься в парадигме программирования еще на уровень ниже.

Заключение

- Используя один лишь функционал родного интерпретатора Python CPython, вы не сможете реализовать быстрый код для выполнения сложных операций.
- Существует немало способов ускорить работу вашего кода на Python, включая использование оптимизированных библиотек, низкоуровневых языков программирования, компиляторов наподобие Numba или даже альтернативных интерпретаторов Python вроде PyPy.
- Расширение Cython представляет собой надмножество языка Python, позволяет компилировать его код в C и тем самым приблизиться по скорости выполнения к этому низкоуровневому языку программирования без необходимости изучения всех его тонкостей.

- Код на Cython можно профилировать подобно коду на Python.
- Написание эффективного кода с использованием расширения Cython предполагает использование аннотаций типов данных для их приведения к стандартам языка C, которые отличаются от выводов анализатора типов `pyrex`.
- С помощью инспектора кода Cython вы можете легко выявить строки в сгенерированном коде на C, взаимодействующие с интерпретатором Python, которые желательно переписать или оптимизировать.
- Вы должны постараться избавиться от максимально возможного количества взаимодействий с CPython в своем сгенерированном коде, вплоть до изменения всей архитектуры кода, чтобы попытаться отвязать неэффективные внутренние циклы в коде от интерпретатора Python. Это может позволить на несколько порядков увеличить скорость выполнения вашего кода.
- Расширение Cython очень тесно интегрировано с библиотекой NumPy, что позволяет очень эффективно работать с массивами. Для непосредственного взаимодействия между Cython и NumPy можно использовать так называемые *представления памяти* (`memoryviews`), что избавит вас от присутствия промежуточного слоя в виде интерпретатора Python.
- Независимость от CPython – это первый шаг к независимости от ограничений пресловутого GPL. Если вам удастся обойти правила GPL, перед вами откроется путь к разработке полноценных многопоточных параллельных приложений на Cython.
- Помните о наличии альтернативы Cython в виде JIT-компилятора Numba: использовать его бывает зачастую проще, хотя в богатстве настроек он уступает Cython.



Иерархия памяти, хранение данных и работа с сетью

В этой главе мы обсудим следующие темы:

- эффективное использование кеша центрального процессора и оперативной памяти;
- использование библиотеки Blosc для сжатия массивов данных;
- использование библиотеки NumExpr для ускорения выражений NumPy;
- разработка архитектуры клиент–сервер для быстрых сетей.

Ни у кого не возникает сомнений в том, что аппаратное обеспечение оказывает влияние на производительность. Но как именно – ответ на этот вопрос может быть не таким очевидным. Цель этой главы – создать у вас представление о том, как эффективность выполнения программ зависит от аппаратного обеспечения и что необходимо сделать в этой области, чтобы максимизировать производительность. В этой связи мы также поговорим об архитек-

туре современного «железа» и сетей применительно к обработке данных в Python.

Очень часто бывает, что последствия, связанные с хранением данных или аппаратным обеспечением, контринтуитивно влияют на быстродействие программ. К примеру, в определенных обстоятельствах обработка сжатых данных может выполняться быстрее, чем несжатых.

Принято считать, и это звучит вполне логично, что распаковка данных с их последующей обработкой занимает больше времени по сравнению с одной только обработкой. В конце концов, распаковка – это ведь лишние действия. Как это может быть более эффективным? Но современное аппаратное обеспечение способно сделать неочевидное вероятным.

Чтобы выжимать максимум возможного из современных архитектур, для начала нам нужно понять, что делает некоторые наши базовые представления контринтуитивными. Таким образом, нам лучше будет начать с введения в современные архитектуры с точки зрения производительности. Эта тема достойна отдельной книги, так что мы остановимся только на самых неочевидных вещах и познакомимся с иерархией памяти: от кеша центрального процессора через оперативную память, жесткие диски и локальные сети до широкомасштабных сетей.

При этом нас будет интересовать повышение эффективности хранения и обработки информации как в плане занимаемого места в памяти, так и с точки зрения времени, необходимого для ее обработки. После знакомства с базовыми принципами организации иерархии памяти мы обратимся к специализированным библиотекам Python, направленным на эффективную работу с аппаратным обеспечением. Сначала рассмотрим работу библиотеки `Blosc`, предназначенной для сжатия и обработки двоичных данных, которая может быть использована для создания сжатых представлений массивов `NumPy`, работать с которыми можно так же эффективно, как и с несжатыми данными. В процессе вы узнаете, как при помощи использования кеша центрального процессора можно практически нивелировать время упаковки и распаковки данных. После этого вы познакомитесь с библиотекой `NumExpr`, позволяющей существенно ускорить вычисление выражений `NumPy` при работе с большими массивами данных – опять же с использованием парадигмы обработки данных при помощи кеша.

В конце главы мы поговорим об облачных и кластерных вычислениях на основе очень быстрых локальных сетей. Большинство кода для проведения анализа данных запускается в кластерах или в облаке, что можно реализовать в таких сетях, так что вам будет очень полезно узнать, как именно это делается.

Интерпретация производительности

Поскольку эта глава посвящена вопросам, связанным с производительностью, результаты, полученные на вашей машине, могут серьезно отличаться от моих. Данные, которые уместятся в кеш центрального процессора на моем компьютере, могут не уместиться на вашем. Более того, если вы будете запускать код из этой главы на машине с графическим интерфейсом пользователя, то не сможете адекватно оценить доступную вместимость кеша процессора, поскольку все запущенные на компьютере процессы будут конкурировать за эти ресурсы.

Все примеры из этой главы мы прогоняли на выделенном сервере без графического интерфейса со следующими характеристиками: Intel Xeon 8375C CPU @ 2,90 ГГц, 32 ядра, кеш L1 2 Мб, кеш L2 40 Мб, кеш L3 54 Мб, DRAM 16 Гб. В разделе, посвященном библиотеке NumExpr, мы приведем конкретные примеры того, как сильно могут меняться результаты замеров в зависимости от аппаратного обеспечения.

Давайте начнем с обзора характеристик современных архитектур аппаратного обеспечения с точки зрения непредвиденных последствий для эффективности кода на Python. Для проверки примеров из этой главы у вас должна быть установлена библиотека `blosc` (`conda install blosc`). Если вы используете Docker, в вашем образе уже установлено все необходимое.

6.1. Как современная архитектура аппаратных средств влияет на эффективность кода Python

В данном разделе мы рассмотрим современные тенденции развития архитектуры аппаратного обеспечения в разрезе потенциального влияния на эффективность выполняемого программного кода Python. Архитектура аппаратных средств включает в себя как непосредственное «железо», находящееся внутри компьютера (центральный процессор, оперативная память и локальные диски), так и сетевые интерфейсы. При рассмотрении вопросов, касающихся локального хранилища и особенно сетевой инфраструктуры, мы не сможем обойти вниманием проблемы, связанные с архитектурой системного программного обеспечения, в частности файловых систем и сетевых протоколов. Опять же, на эту тему можно написать не одну книгу, а мы сосредоточимся на задачах, имеющих непосредственное отношение к эффективности кода на Python и для решения которых существуют специализированные библиотеки.

Начнем с очень простого, на первый взгляд, примера, который должен послужить для вас предметом мотивации и вдохновить на изучение особенностей архитектуры аппаратного обеспечения, влияющих на производительность выполнения кода. Если вам интересно,

как можно на пару порядков увеличить быстродействие операций, которые, на ваш взгляд, ускорить невозможно, читайте далее.

6.1.1. Неожданное влияние современной архитектуры на производительность

Для своего первого примера мы возьмем квадратный массив NumPy и удвоим значения в первой строке и в первом столбце. Можно предположить, что эти операции займут приблизительно одинаковое время, поскольку мы имеем дело с массивом с равными сторонами. Все вполне очевидно. Или нет?

Зачем гадать? Можно ведь замерить время удвоения значений в строке и колонке массива:

```
import numpy as np
```

```
SIZE = 1000
```

```
mat = np.random.randint(10, size=(SIZE, SIZE))
```

```
double_column = 2 * mat[:, 0]
```

```
double_row = 2 * mat[0, :]
```

Если вы используете интерпретатор IPython в Jupyter, то можете измерить производительность этих инструкций, просто добавив перед ними ключевое слово `%timeit`

Мы создали матрицу размером 1000×1000 и заполнили ее случайными значениями в интервале от 0 до 9. Позже мы проведем этот эксперимент с матрицами со стороной 1000 и 10 000 элементов.

Еще раз отметим, что мы имеем дело с квадратными матрицами, а это значит, что при вычислении переменных `double_column` и `double_row` нам потребуется выполнить одинаковое количество математических операций. Подсознательно кажется, что это какая-то странная задача, не стоящая нашего внимания. Очевидно, что время расчета для строки и колонки должно быть практически одинаковым. Но на этот раз очевидность не означает правильность.

Давайте снова вернемся к нашему коду и прочитаем, что написано между строк. Мы создали матрицу размером 1000×1000, вмещающую 10 000 целочисленных значений. С учетом того, что по умолчанию для целых чисел в памяти выделяется 8 байт, наша матрица будет занимать в памяти 80 Кб. На моем компьютере время удвоения значений в столбце составило 750 нс, а в строке – 715 нс. Не очень большая разница. А с учетом гранулярности операции она может быть вызвана механизмом профилирования. Пока ничего удивительного.

Давайте увеличим размер матрицы до 10000×10000 элементов. Таким образом, она будет вмещать в себя 1 млн элементов общим объемом 8 Мб. Теперь время удвоения значений в столбце составило 1,99 мкс, а в строке – 1,50 мкс. Опять же, ничего особенно выдающегося.

Продолжим масштабирование задачи и увеличим размер матрицы до 10 000×10 000. При этом содержимое матрицы должно занимать в памяти 800 Мб, так что сначала убедитесь, что вы распо-

лагаете такими ресурсами. Итак, в этом случае удвоение значений в столбце заняло 74,9 мкс, а в строке – 4,51 мкс!

Как видите, в матрице такого размера математические операции над элементами из одной строки выполняются в 16 раз быстрее, чем из одного столбца. Давайте разберемся! Здесь явно что-то связано с архитектурой и внутренним представлением данных в NumPy – что-то ведь привело к тому, что две одинаковые операции выполняются за такое разное время.

Причины может быть две. Первая связана с временем доступа к кешу центрального процессора и к оперативной памяти, а вторая – с внутренним представлением массива. В комбинации два этих фактора и привели к такой разнице в быстродействии. Мы коснемся каждого из них в отдельности.

6.1.2. Влияние кеша процессора на эффективность алгоритма

Сначала давайте рассмотрим *кратковременную память* (transient memory). Обычно мы оперируем терминами *динамической оперативной памяти* (dynamic random access memory – DRAM), но на самом деле вычисления производятся в *регистрах центрального процессора* (CPU registers), т. е. на самом нижнем уровне в иерархии памяти, и передаются через несколько слоев кеша процессора. В табл. 6.1 приведен пример характеристик современного компьютера.

Таблица 6.1. Иерархия памяти с объемами и временами доступа для гипотетической современной рабочей станции

Тип	Объем	Время доступа
Центральный процессор		
Кеш первого уровня (L1 cache)	256 Кб	2 нс
Кеш второго уровня (L2 cache)	1 Мб	5 нс
Кеш третьего уровня (L3 cache)	6 Мб	30 нс
Оперативная память		
DIMM	8 Гб	100 нс

Время доступа к *кешу первого уровня* (L1 cache) примерно соответствует скорости работы современных процессоров. Если вы помните, рабочая частота процессора 2 ГГц означает его способность выполнять $2 \cdot 10^9$ рабочих циклов в секунду, а наносекунда как раз и составляет 10^{-9} секунды.

Если данные, необходимые процессору для расчетов, находятся в кеше первого уровня, то скорость их обработки будет соответствовать рабочей частоте процессора. В то же время необходимость обращаться за данными к оперативной памяти приводит

к длительным простоям центрального процессора. В таких условиях не будет ничего удивительного, если процессор на протяжении 90 % времени будет просто ждать информацию.

Теперь давайте вернемся к нашему примеру. Почему же удвоение значений в столбце квадратной матрицы может выполняться намного дольше, чем удвоение значений в строке? Все довольно просто. Представьте, что мы работаем со следующей матрицей:

I11	I12	I13	I14
I21	I22	I23	I24
I31	I32	I33	I34
I41	I42	I43	I44

В памяти элементы матрицы будут представлены последовательно в следующем виде:

I11 I12 I13 I14 I21 I22 I23 I24 I31 I32 I33 I34 I41 I42 I43 I44

Когда вы обращаетесь к элементу I11, центральный процессор извлечет в память еще несколько элементов, а не один запрашиваемый. Таким образом, выполнение последовательности математических операций $2 \cdot I11$, $2 \cdot I12$, $2 \cdot I13$, $2 \cdot I14$ потребует единственной переброски данных из памяти в кеш. В то же время операции $2 \cdot I11$, $2 \cdot I21$, $2 \cdot I31$, $2 \cdot I41$ будут сопряжены с дорогостоящей передачей данных для каждой из них, поскольку физически эти элементы располагаются в памяти не рядом. Получается, что в первом случае нам потребуется выполнить четыре операции умножения с одной операцией переноса данных, а во втором – по четыре операции каждого типа.

Конечно, мы здесь работаем с упрощенным примером. В зависимости от размера матрицы и объема кеша вполне возможно, что все ваши данные в обоих случаях поместятся в кеш. Именно поэтому при работе с небольшими массивами мы практически не видели различий во времени выполнения операций. При увеличении размеров массивов разница становится ощутимой и вполне может превышать целый порядок, что мы и видели в нашем примере.

СОВЕТ. Применительно к представлениям массивов существует еще одна проблема, состоящая в том, что последовательно в памяти могут располагаться или строки массивов, или их столбцы. Первый вариант характерен для кода на базе C, а второй – для Fortran. Для нас это очень важно, поскольку внутренне вычисления в библиотеке NumPy реализованы с применением двух этих языков программирования. Таким образом, вы должны понимать, с какой реализацией вы работаете, чтобы оптимально адресовать массивы.

Далее в этой главе мы рассмотрим примеры использования библиотек Blosc и NumExpr, позволяющих повысить эффективность работы с кешем центрального процессора.

6.1.3. Современные устройства постоянного хранения

Еще одной проблемой на пути производительности является область постоянного хранения информации. Наиболее распространенными устройствами хранения данных являются *жесткий диск* (hard disk drive – HDD) и *твердотельный накопитель* (solid-state drive – SSD). Устройства постоянного хранения проигрывают оперативной памяти во времени доступа на несколько порядков: если у SSD это время исчисляется микросекундами, то у HDD и во все миллисекундами. Мы не будем глубоко погружаться в эту тему (хотя в главе 8 коснемся ее под несколько иным углом), но отметим, что техники, применимые к оперативной памяти (о них мы поговорим в следующем разделе), вполне применимы и к устройствам постоянного хранения. К примеру, бывают ситуации, когда со сжатыми данными можно работать быстрее, чем с несжатыми: стоимость распаковки может быть значительно ниже по сравнению с чтением большого объема сырых данных.

В дополнение к устройствам постоянного хранения и оперативной памяти мы также располагаем технологиями удаленного хранилища данных и удаленных вычислений. В теории обе эти концепции должны серьезно уступать локальному хранилищу информации. К примеру, доступ к информации через интернет обычно бывает небыстрым и непредсказуемым. В то же время в современных кластерах локального вычисления сетевые интерфейсы могут быть достаточно быстрыми. Насколько быстрыми? Доступ к удаленному серверу в таких сетях может выполняться даже быстрее, чем к локальному диску! В последнем разделе главы мы рассмотрим такую ситуацию более подробно. Как вы увидите, стандартные сетевые протоколы, которые мы используем для доступа к удаленным веб-службам, могут оказаться довольно медленными при работе в быстрой локальной сети.

Вывод

В этом разделе я пытался донести до вас мысль о том, что некоторые устоявшиеся парадигмы относительно вычислений и хранения данных в современном мире могут не работать. Как мы видели, внешне одинаковые операции могут занимать совершенно разное время в зависимости от способа хранения данных. Таким образом, если мы хотим, чтобы центральный процессор максимально быстро справлялся с нашими задачами, мы должны максимально близко к нему располагать данные, используемые нашим алгоритмом. К сожалению, даже их размещения в оперативной памяти мо-

жет быть недостаточно, поскольку в этом случае процессор может простаивать долгое время в ожидании данных. Так что наша главная цель – пробовать всегда, когда это возможно, помещать данные для работы в кеш первого уровня.

Но кроличья нора на самом деле еще глубже, поскольку зачастую бывает быстрее распаковать данные на лету с использованием продвинутых алгоритмов, чем использовать сырые данные. Именно это позволяет сделать библиотека Blosc, о которой мы поговорим в следующем разделе.

6.2. Эффективное хранение данных при помощи Blosc

Blosc представляет собой высокопроизводительный фреймворк, призванный сделать обработку сжатых данных более быстрой по сравнению с сырыми несжатыми данными. Как это вообще возможно? В предыдущем разделе мы уже говорили, что центральный процессор может простаивать большую часть времени при работе с данными, располагающимися в оперативной памяти. А если количество циклов процессора, необходимых для упаковки/распаковки данных, достаточно мало, чтобы уместиться во время простоя процессора, то все операции, связанные с компрессией, окажутся бесплатными.

6.2.1. Сжимаем данные, экономим время

Чтобы продемонстрировать, как можно сэкономить время при помощи компрессии данных, рассмотрим три разных способа создания массивов NumPy с последующим их сохранением на диск и извлечением посредством библиотек NumPy и Blosc. Мы сравним использованный объем памяти и время обработки для каждого подхода, хотя это не такая очевидная задача, как может показаться.

Начнем с создания массивов и написания вспомогательных функций:

```
import os
import blosc
import numpy as np

random_arr = np.random.randint(256, size=(1024, 1024, 1024)).astype(np.
uint8)

zero_arr = np.zeros(shape=(1024, 1024, 1024)).astype(np.uint8)

rep_tile_arr = np.tile(
    np.arange(256).astype(np.uint8), 4*1024*1024).
reshape(1024,1024,1024)
```

```

def write_numpy(arr, prefix):
    np.save(f"{prefix}.npy", arr)
    os.system("sync")

def write_blosc(arr, prefix, cname="lz4"):
    b_arr = blosc.pack_array(arr, cname=cname)
    w = open(f"{prefix}.bl", "wb")
    w.write(b_arr)
    w.close()
    os.system("sync")

def read_numpy(prefix):
    return np.load(f"{prefix}.npz")

def read_blosc(prefix):
    r = open(f"{prefix}.bl", "rb")
    b_arr = r.read()
    r.close()
    return blosc.unpack_array(b_arr)

```

Команда `sync` отвечает за сброс данных на диск

Библиотека NumPy сама следит за целостностью данных на диске

Перед сохранением массивов NumPy с помощью Blosc их необходимо упаковать

Перед чтением массивов NumPy с помощью Blosc их необходимо распаковать

Начали мы с создания трех трехмерных массивов: первый из них заполнили случайными значениями в диапазоне от 0 до 255, второй – нулями, а третий – повторами последовательных чисел от 0 до 255. Мы использовали разное заполнение массивов, чтобы продемонстрировать различия в отношении компрессии данных в зависимости от характера хранящейся информации. К примеру, массив, состоящий из нулей, можно сжать очень легко и быстро, массив с повторами одного шаблона данных потребует чуть больше ресурсов, а случайные значения практически не поддаются компрессии.

После этого мы создали три вспомогательные функции для чтения массивов и записи их на диск, при этом операция записи получилась не самой тривиальной. Поскольку нам необходимо точно замерять выполнение операции записи, мы вынуждены сбрасывать буферы потоков, для чего воспользовались функцией `sync`. Функция `sync` недоступна в Windows.

Теперь давайте оценим быстрдействие записи наших массивов:

```

os.system("sync")
%time write_numpy(zero_arr, "zero")
%time write_blosc(zero_arr, "zero")
%time write_numpy(rep_tile_arr, "rep_tile")
%time write_blosc(rep_tile_arr, "rep_tile")
%time write_numpy(random_arr, "random")
%time write_blosc(random_arr, "random")

```

Начали мы с вызова функции `sync`, которая очищает входные буферы ввода/вывода операционной системы, насколько это возможно. Затем воспользовались инструкцией `%time` для измерения времени выполнения функций записи. И хотя мы могли безопасно

воспользоваться инструкцией `%timeit`, по крайней мере для операций записи, мы хотим избежать любой возможности оптимизации наших вызовов со стороны операционной системы, что может затруднить интерпретацию полученных результатов. В табл 6.2 собраны итоговые показатели.

Таблица 6.2. Длительность операций записи массивов при помощи NumPy и Blosc в секундах

Массив	NumPy	Blosc
zero_arr	7,49	0,53
rep_tile_arr	7,49	0,53
random_arr	7,5	8,13

Запись массивов с нулевыми и повторяющимися элементами с помощью библиотеки Blosc была выполнена в 15 раз быстрее по сравнению с NumPy. Что касается массива случайных элементов, то здесь NumPy оказался даже чуть быстрее. Какой из этих случаев встречается на практике чаще? Со случайными данными в массивах приходится работать достаточно редко: обычно содержимое таблиц отвечает определенным шаблонам. В то же время не стоит рассчитывать на то, что вам часто придется работать с такими идеальными случаями, как матрица из нулей или полностью повторяющихся шаблонов.

В целом можно сделать вывод, что использование библиотеки Blosc приводит к существенному сокращению времени выполнения операций. А как насчет места на диске? Эта важная метрика при работе с большими данными посредством Blosc также не подкачала: для массивов `zero_arr` и `rep_tile_arr` выигрыш составил 250 и 200 раз, а размер массива случайных чисел не изменился.

6.2.2. Операции чтения (буферы памяти)

Теперь посмотрим, как обстоят дела с чтением массивов данных. В теории нам нужно просто прочесть данные из файлов, да? Но с учетом того, что мы записывали их на диск, операционная система могла поместить их в промежуточные буферы памяти, что помешает нам при сравнении результатов. Иными словами, кэширование может негативно сказаться на результатах профилирования кода. Таким образом, для чистоты эксперимента необходимо убедиться в том, что мы *действительно* читаем данные с диска, а не из временных буферов, которые могут работать намного быстрее. А значит, нам нужно сбросить буферы.

Самым радикальным вариантом решения этой задачи является перезагрузка компьютера. Но мы также можем указать операционной системе считать кэши недействительными. К сожалению, эта операция является зависимой от операционной системы. Здесь

я покажу вариант для Debian/Ubuntu и производных от них операционных систем, который не будет работать на Windows, Mac или на некоторых других дистрибутивах Linux. Вы должны найти способ, подходящий для вашей операционной системы.

Итак, выполните следующую команду от имени root:

```
sync; echo 3 > /proc/sys/vm/drop_caches
```

Теперь мы можем проводить тестирование, будучи уверенными, что данные не будут читаться из буферов в памяти:

```
%time _ = read_numpy("zero")
%time _ = read_blosc("zero")
%time _ = read_numpy("rep_tile")
%time _ = read_blosc("rep_tile")
%time _ = read_numpy("random")
%time _ = read_blosc("random")
```

Результаты приведены в табл. 6.3.

Таблица 6.3. Длительность операций чтения массивов при помощи NumPy и Blosc в секундах

Массив	NumPy	Blosc
zero_arr	7,02	0,63
rep_tile_arr	7,04	0,61
random_arr	7,37	8,58

Результаты получились приблизительно такими же, как в случае с записью массивов. При работе с неслучайными данными Blosc значительно превосходит NumPy, а это означает, что вам необходимо пользоваться этой библиотекой, когда вы сталкиваетесь с большими данными.

До этого момента мы не обращали внимания на используемый алгоритм сжатия (compression algorithm) данных, мы просто полагались на метод, принятый по умолчанию. В то же время библиотека Blosc предлагает вам богатый выбор алгоритмов сжатия.

6.2.3. Влияние алгоритма сжатия на эффективность хранения

В данном разделе мы не будем сравнивать все доступные в библиотеке Blosc алгоритмы сжатия данных. Мы лишь посмотрим, как можно пользоваться существующими методами компрессии и напомним, что в будущем могут появиться и другие. Алгоритмы сжатия отличаются по скорости и эффективности в зависимости от области их использования. Зная эти особенности, вы можете в любой ситуации выбрать наиболее подходящий алгоритм, исходя из

ваших потребностей и условий задачи. Мы ограничимся сравнением методов сжатия *LZ4* и *Zstandard*.

При этом здесь мы не будем писать данные на диск, поскольку уже поняли, как трудно проводить сравнение этих операций. Вместо этого мы будем выполнять операции в памяти и сжимать данные с помощью все той же библиотеки Blosc, которую мы уже видели в деле. Сначала воспользуемся алгоритмом LZ4, а затем применим метод Zstandard:

```
%timeit rep_lz4 = blosc.pack_array(rep_tile_arr, cname='lz4')
rep_lz4 = blosc.pack_array(rep_tile_arr, cname='lz4')
%timeit rep_std = blosc.pack_array(rep_tile_arr, cname='zstd')
rep_std = blosc.pack_array(rep_tile_arr, cname='zstd')
print(len(rep_lz4) // 1024)
print(len(rep_std) // 1024)
```

Создаем представление массива в памяти с использованием метода сжатия LZ4

Создаем представление массива в памяти с использованием метода сжатия Zstandard

Результаты показаны в табл. 6.4.

Таблица 6.4. Время выполнения операции и размер сжатых данных при использовании алгоритмов LZ4 и Zstandard

	LZ4	Zstandard
Время (мс)	527	919
Размер (Кб)	5204	366

Как вы помните из предыдущего раздела, алгоритм сжатия данных LZ4 позволил сократить объем данных по сравнению с чистым NumPy в 200 раз. Из этого, а также из приведенных в табл. 6.4 данных несложно заключить, что алгоритм Zstandard позволяет улучшить результаты NumPy в 2800 раз ($200 * 14$, где 14 – это разница между методами LZ4 и Zstandard).

Но это не все, на что способна библиотека Blosc. Помимо богатого выбора алгоритмов сжатия, она позволяет вам менять представление входных данных на лету, что также может позволить снизить объем итоговых данных. Давайте посмотрим, как это работает.

6.2.4. Использование сведений о представлении данных для повышения эффективности сжатия

Представьте, что вы точно знаете, что в ваших данных есть определенный регулярный шаблон, например цифры часто следуют по порядку. Скажем, в документе, с которым вы работаете, встречаются такие последовательности 8-битных чисел:

3,4,5,6

Обычно такая последовательность будет кодироваться в двоичном формате следующим образом:

00000011/00000100/00000101/00000110

Теперь представьте, что вы берете самый старший бит каждого числа и кодируете его, затем следующий, и так до восьмого. В итоге вы получите следующий результат:

```
0000000000000000000000000000000011110011010
```

Второй шаблон выглядит гораздо более регулярным. Именно это позволяет системам сжатия данных добиваться впечатляющих показателей. И библиотека Blosc предоставляет вам возможность воспользоваться этим:

```
for shuffle in [blosc.BITSHUFFLE, blosc.NOSHUFFLE]:
    a = blosc.pack_array(rep_tile_arr, shuffle=shuffle)
    print(len(a))
```

Версия со сжатием занимает 4 600 034 байт, а без сжатия – 5 345 500 байт. При этом операция сжатия обошлась нам по времени не очень дорого – 596 мс против 524 мс.

Вывод

Грамотное использование иерархии памяти и процессорной обработки может существенно повысить эффективность операций по работе с массивами. Библиотека Blosc позволяет в большинстве случаев работать со сжатыми данными быстрее, чем с исходными, что является плюсом к тому, что сжатые массивы занимают меньше места в памяти.

Давайте сделаем еще шаг вперед и воспользуемся похожими техниками при анализе данных. Для этого нам необходимо познакомиться с библиотекой NumExpr.

6.3. Ускорение NumPy с помощью NumExpr

Библиотека Blosc – лишь один из примеров того, как можно за счет эффективного использования иерархии памяти повысить производительность процесса обработки данных. Мы можем пойти дальше и развить эту концепцию путем обработки выражений NumPy с помощью библиотеки NumExpr.

NumExpr представляет собой средство вычисления числовых выражений для NumPy, которое может позволить ощутимо ускорить обработку этих выражений по сравнению с традиционными средствами NumPy. Мы просто передаем ему выражение – например, `a + b` – и получаем результат. Но постойте, какой в этом смысл?

Разве не этим самым занимается сам NumPy? Дело в том, что библиотека NumExpr способна подменять некоторый функционал NumPy посредством эффективной реорганизации вычислений при работе с большими данными. Один из принципов NumExpr состоит в том, чтобы *не* генерировать полные промежуточные представления для частей выражения: все вычисления выполняются порциями, помещающимися в кеш первого уровня.

6.3.1. Быстрая обработка выражений

Теперь давайте рассмотрим несколько примеров использования библиотеки NumExpr при вычислениях в NumPy и сравним эффективность:

```
import numpy as np
import numexpr as ne
```

```
a = np.random.rand(100000000).reshape(10000,10000)
b = np.random.rand(100000000).reshape(10000,10000)
f = np.random.rand(100000000).reshape(10000,10000).copy('F')
```

Этот массив представлен с использованием формата языка Fortran

```
%timeit a + a
```

```
%timeit ne.evaluate('a + a')
```

```
%timeit f + f
```

```
%timeit ne.evaluate('f + f')
```

```
%timeit a + f
```

```
%timeit ne.evaluate('a + f')
```

```
%timeit a**5 + b
```

```
%timeit ne.evaluate('a**5 + b')
```

```
%timeit a**5 + b + np.sin(a) + np.cos(a)
```

```
%timeit ne.evaluate('a**5 + b + sin(a) + cos(a)')
```

В библиотеке NumExpr применяется функция `evaluate` для обработки выражений

Итак, мы создали четыре квадратные матрицы для последующих вычислений. При этом последняя из них организована по принципам языка Fortran. Вывод инструкций `%timeit` собран в табл. 6.5.

Таблица 6.5. Сравнение времени выполнения в библиотеках NumPy и NumExpr в миллисекундах

Выражение	Среднее время NumPy	Среднее время NumExpr	Ускорение (раз)
<code>a + a</code>	224	58	3,8
<code>f + f</code>	224	58	3,8
<code>a + f</code>	577	153	3,7
<code>a**5 + f</code>	1690	87	19,4
<code>a**5 + f + sin(a) + cos(a)</code>	3840	153	25,1

На моем аппаратном обеспечении библиотека NumExpr показала гораздо более лучшие результаты. Обратите внимание на выражения, сочетающие в себе матрицы в разных форматах: C и Fortran. Такие выражения выполняются дольше по сравнению с однородными. С повышением сложности выражений преимущество использования библиотеки NumExpr становится еще более ощутимым из-за появления дополнительных областей применения оптимизации.

Но не стоит боготворить библиотеку NumExpr. Бывают случаи, когда ее применение приводит к падению быстродействия кода. Далее в этой главе мы продемонстрируем несколько таких примеров. Но начнем с качественных различий в производительности, вызванных аппаратным обеспечением.

6.3.2. Влияние архитектуры аппаратных средств на результаты

Как мы уже говорили в начале этой главы, ваши результаты при выполнении представленных в книге фрагментов кода могут существенно отличаться от приведенных. Давайте для интереса сравним результаты, полученные на сервере и на ноутбуке, на котором я пишу этот текст (Linux с графическим интерфейсом и текстовым редактором). Я не буду приводить информацию об объемах кеша, поскольку она может сбивать с толку: в процессе работы компьютера столько разных процессов конкурируют за кеш, что его эффективность и влияние оценить будет затруднительно. В табл. 6.6 приведены результаты вычисления выражений с использованием библиотеки NumExpr на сервере и ноутбуке.

Таблица 6.6. Влияние архитектуры на производительность

Выражение	Ускорение на сервере	Ускорение на ноутбуке
$a + a$	3,8	0,7
$f + f$	3,8	0,8
$a + f$	3,7	1,3
$a^{**}5 + f$	19,4	11,5
$a^{**}5 + f + \sin(a) + \cos(a)$	25,1	6,7

Как видите, эффект от применения библиотеки NumExpr в случае использования рабочего ноутбука значительно снизился. Более того, для некоторых простых операций время выполнения с использованием NumExpr даже ухудшилось по сравнению с NumPy. Одна из причин состоит в непредсказуемости доступа к кешу центрального процессора на рабочей станции из-за большого количества конкурентных процессов.

СОВЕТ. Не ждите ощутимого прироста производительности при выполнении операций на локальной машине с огромным количеством параллельно запущенных процессов, включая все графические приложения вроде браузера и текстового редактора. В таких условиях возникает слишком отчаянная конкурентная борьба за ресурсы кеша процессора, и результаты выполнения операций могут сильно варьироваться от запуска к запуску. Техники, основывающиеся на активном использовании кеша центрального процессора, могут эффективно применяться на выделенных серверах, а не на типичных рабочих станциях. Так что, если вы тестируете подходы, связанные с использованием кеша, тестируйте их на серверах.

Предыдущий пример продемонстрировал, что далеко не все сценарии годятся для использования библиотеки NumExpr. Давайте точнее сформулируем, когда именно обращение к этой библиотеке будет нецелесообразным.

6.3.3. Когда не стоит использовать библиотеку NumExpr

Существует несколько сценариев, в которых библиотека NumExpr не поможет вам увеличить быстродействие кода. Давайте их обсудим.

Наиболее важным фактором здесь является размер массивов, с которыми вы работаете: NumExpr показывает наилучшие результаты при взаимодействии с объемными массивами. Давайте повторим наш предыдущий пример, но с массивами меньшего размера:

```
small_a = np.random.rand(100).reshape(10, 10)
small_b = np.random.rand(100).reshape(10, 10)

%timeit small_a + small_a
%timeit ne.evaluate('small_a + small_a')
%timeit small_a**5 + small_b + np.sin(small_a) + np.cos(small_a)
%timeit ne.evaluate('small_a**5 + small_b + sin(small_a) + cos(small_a)')
```

Простое сложение здесь выполняется с помощью библиотеки NumExpr в 15 раз медленнее, да и вычисление более сложного выражения занимает на 30 % времени больше. Но это не такая большая проблема. В конце концов, вы вряд ли решите оптимизировать свои вычисления при работе с такими маленькими массивами данных. Наша головная боль – это большие данные.

Также библиотека NumExpr не сможет показать себя во всей красе при работе на локальной машине, на которой одновременно запущено множество процессов. Вотчина NumExpr – выделенные серверы с возможностью контролировать количество запущенных приложений и процессов. Это означает, что в среде распределенных кластеров, которые зачастую используются в научно-образовательной сфере, эффективность NumExpr будет непостоянной.

Наконец, библиотека NumExрг поддерживает только множество операторов, входящих в состав NumPy, и скорость выполнения других вычислений может остаться неизменной.

Теперь, когда мы поговорили о разных способах оптимизации использования динамической памяти, давайте переключимся на обсуждение локальных сетей. Современные локальные сети могут работать быстрее, чем постоянные запоминающие устройства, и это переворачивает с ног на голову некоторые устоявшиеся догмы.

6.4. Производительность при использовании локальных сетей

При написании приложений, работающих в сети, мы неизбежно полагаемся на инфраструктуру, которая может иметь самые разные характеристики. Обычно принято считать, что сеть – это что-то совсем ненадежное в отношении скорости, задержек и отказоустойчивости. Однако при использовании действительно высокоскоростных сетевых решений в локальной среде эти опасения бывают напрасны. Современные сетевые коммутаторы способны поддерживать скорость до 2 Тб/с на магистральной и до 56 Гб/с на каждый сетевой порт. Для сравнения, большинство жестких дисков работают на скорости 6 Гб/с. Только задумайтесь над этим: в высокоскоростной сети взаимодействие с соседним компьютером выполняется быстрее, чем чтение данных с диска. Если вы работаете именно с такой сетью, читайте дальше.

Традиционные программные сетевые фреймворки совершенно не подходят для скоростей, поддерживаемых в современных локальных сетях. Вы ведь не считаете нормальной идею доступа к жесткому диску с использованием REST по протоколу HTTPS? Высокоскоростные сети демонстрируют большую производительность по сравнению с жесткими дисками, так что нам необходимо найти для них наиболее эффективный способ обмена информацией.

Перед тем как выработать подходящее решение, давайте подумаем о том, почему нам не подходят обычные варианты. В данном разделе мы реализуем серверную часть службы *pastebin* не на основе REST. Служба *pastebin* позволяет пользователям сохранять фрагменты текста и делиться ими через интернет. Мы напишем клиентскую часть, с помощью которой можно будет посылать текст на сервер и запрашивать текст на чтение. Также напишем серверную часть, которая будет отвечать за запись фрагментов текста и отдавать их по запросу. Чтобы понять, о чем речь, вы можете посмотреть на готовую службу по адресу <https://pastebin.com>. Мы будем исходить из того, что клиент и сервер у нас находятся в высокоскоростной сети.

6.4.1. Причины неэффективности вызовов *REST*

Для начала давайте поищем узкие места в решении, основанном на архитектурном стиле *REST*. Взаимодействие клиента и сервера в архитектуре *REST* происходит в формате *JSON* по протоколу *HTTPS*. *JSON* представляет собой определенный текстовый формат, а значит, требует времени для анализа (парсинга) и занимает много места. Протокол *HTTPS* в свою очередь является надстройкой над *HTTP* с добавлением правил авторизации и шифрования с использованием открытого ключа. Таким образом, этот протокол несет и дополнительную нагрузку по сравнению с *HTTP*.

Что касается *HTTP*, то этот протокол работает поверх интернет-протокола под названием *TCP* (*Transmission Control Protocol*), который устанавливает соединение между двумя точками – в нашем случае между клиентом и сервером. Созданное подключение гарантирует доставку данных без потерь. Но сам протокол является довольно громоздким, по крайней мере, для нашей высокоскоростной сети. Судите сами – одна лишь установка соединения требует отправки и получения по крайней мере трех пакетов с информацией между клиентом и сервером.

После установки соединения *TCP* вступает в игру часть протокола *HTTP*, отвечающая за безопасность, и эти операции выполняются посредством протокола *TLS* (*Transport Layer Security*). Этот протокол квитирует установление связи, что требует передачи и получения еще нескольких пакетов между клиентом и сервером. А с учетом того, что в этот процесс вовлечена криптография, вычислительная нагрузка требуется немалая. При этом стоит отметить, что время вычислений будет относительно: в высокоскоростных сетях оно будет составлять большую часть общего процесса, а если ваш сервер располагается на другом конце света, это время будет незаметно на фоне ожидаемо больших затрат.

После всего этого мы будем готовы к отправке полезных пакетов в формате *JSON*, которым потребуется парсинг. По окончании работы мы закроем соединения на обоих уровнях: *HTTPS* и *TCP*.

За это время нам придется отправить и получить минимум 20 сетевых пакетов. С учетом скорости локальной сети можно предположить, что подавляющая часть времени будет потрачена на протоколы. Давайте создадим реализацию с использованием всего двух пакетов, что является абсолютным минимумом, – одного для запроса, второго – для ответа.

6.4.2. Наивный клиент на основе *UDP* и *msgpack*


Наша реализация будет предельно простой. Здесь важно будет не понять код, а осознать заложенные в нем компромиссы.

Начнем с клиента. Наш клиент будет посылать текст на сервер *pastebin* и извлекать его. Первые строки кода такие:

```
import socket

host = '127.0.0.1'
port = 54321

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

**Создание нового сокета UDP,
SOCK_DGRAM**


Вместо применения полноценного протокола HTTP(S) и стека TCP мы решили вовсе обойтись без использования *протоколов прикладного уровня* (application protocol) и заменить протокол TCP на UDP (User Datagram Protocol). Протокол UDP не устанавливает соединений, а просто посылает пакеты. Вы можете размышлять о протоколе UDP как о почтовых услугах, а о TCP – как о телефоне. На почте письма могут теряться, доставляться в неправильном порядке и неправильно маршрутизироваться. В то же время телефонное соединение подразумевает выделенный канал связи, чтобы информация не терялась. С точки зрения накладных расходов протокол UDP (почта) гораздо менее требовательный в сравнении с TCP (телефон).

В коде, показанном выше, используется низкоуровневый модуль *socket* для создания конечной точки назначения UDP. Мы указали адрес сервера 127.0.0.1, который в данном случае представляет локальный компьютер, и номер порта 54321.

Эта простая реализация использует пару допущений, о которых вы должны знать и которые должны быть для вас приемлемыми:

- *мы не используем зашифрованный канал для передачи информации, а значит, передаваемые данные могут перехватить и изменить.* При работе в высокоскоростных локальных сетях это не такая большая проблема, как при взаимодействии через интернет. Кроме того, стоит понимать, что, если злоумышленники смогут получить доступ к вашей магистральной сети, это будет гораздо хуже, чем если они перехватят ваши данные, поскольку будет означать, что ваша инфраструктура сама по себе ненадежна;
- *передача данных по протоколу UDP не предполагает гарантий доставки пакетов.* Это значит, что при передаче данных от клиента серверу и обратно они могут быть потеряны. Опять же, в высокоскоростных локальных сетях эта проблема представляет меньше опасности по сравнению с работой в сети Интернет. Но забывать о возможных потерях не стоит, и в последнем разделе этой главы мы коснемся этой темы более подробно.

Теперь давайте завершим написание клиента, снабдив его функциями отправки и получения сообщений от сервера:


```

import msgpack

def send_text(sock, text):
    pack = msgpack.packb({'command': 0, 'text': text})
    sock.sendto(pack, (host, port))
    text_id_enc = sock.recv(10240)
    return int.from_bytes(text_id_enc, byteorder='little')

def request_text(sock, text_id):
    pack = msgpack.packb({'command': 1, 'text_id': text_id})
    sock.sendto(pack, (host, port))
    text = sock.recv(10240)
    return text

text_id = send_text(sock, 'trial text')
returned_text = request_text(sock, text_id)

```

Воспользуемся внешней библиотекой `msgpack` для кодирования сложных структур данных

Упаковываем словарь в массив байтов с помощью библиотеки `msgpack`

Получаем ответ от сервера

Посылаем сообщение UDP на сервер

Функция `send_text` служит для отправки текста на сервер. Запрос содержит команду (`command`) со значением 0, что означает необходимость сохранить присланный текст. Мы могли бы использовать более многословное значение вроде текста "store text", но это заняло бы больше места и привело к снижению эффективности приложения. Текст мы посылаем как есть, но, как мы видели в предыдущих разделах, можно было бы применить к нему алгоритмы сжатия для экономии времени и памяти, особенно если мы работаем с объемными текстами.

Ответ от сервера не кодируется с помощью библиотеки `msgpack`. С учетом того, что мы получим числовое значение идентификатора, с которым текст был сохранен, поступим гораздо проще – восстановим его из потока байтов. Это должно быть даже быстрее, чем использование `msgpack`.

В функции `request_text` команда (`command`) содержит значение 1, а также передается идентификатор текста для получения, и все это упаковывается с помощью `msgpack`. После отправки запроса мы получаем ответ в виде текста.

В конце кода посылаем текст на сервер и извлекаем его обратно при помощи полученного идентификатора. Теперь мы готовы к реализации серверной части кода. После этого вернемся к клиентской части, чтобы повысить ее надежность в отношении потери пакетов.

6.4.3. Сервер на основе UDP

Серверный код будет базироваться на встроенном модуле `socketserver`, предоставляющем вспомогательные классы для написания полноценных серверов на основе сокетов:


```

import os
import socketserver

import msgpack

class UDPPProcessor(socketserver.BaseRequestHandler):
    def handle(self):
        request = msgpack.unpackb(self.request[0])
        socket = self.request[1]
        if request['command'] == 0:
            text = request['text']
            w = open(f'texts/{self.server.snippet_number}.txt', 'w')
            w.write(text)
            w.close()
            socket.sendto(self.server.snippet_number.to_bytes(
                4, byteorder='little'), self.client_address)
            self.server.snippet_number += 1
        elif request['command'] == 1:
            text_id = request['text_id']
            f = open(f'texts/{text_id}.txt')
            text = f.read()
            f.close()
            socket.sendto(text.encode(), self.client_address)

host = '127.0.0.1'
port = 54321

try:
    os.mkdir('texts')
except FileExistsError:
    pass

with socketserver.UDPServer((host, port), UDPPProcessor) as server:
    server.snippet_number = 0
    server.serve_forever()

```

Мы реализуем код сервера внутри класса-обработчика

Класс-обработчик требует реализации метода handle для воплощения нужного нам функционала

Создаем сервер UDP

Инициализируем внутреннюю переменную для идентификаторов текста

Переопределенный метод `handle` начинается с приема команды, чтобы понять, какую именно операцию хочет выполнить клиент. Если он хочет сохранить текст, мы считываем его и записываем на диск, а если извлечь, то получаем текст по идентификатору и возвращаем клиенту.

Высокую производительность реализованной схемы мы обсуждали в предыдущем разделе – связана она с использованием библиотеки `msgpack` и протокола UDP. Теперь, когда мы написали код клиента и сервера, давайте вернемся к первому из них и сделаем его более надежным.

6.4.4. Безопасность на клиенте с помощью тайм-аутов

В нашей первоначальной реализации клиента сообщение отправляется на сервер, и начинается ожидание ответа. Но протокол UDP, как мы уже говорили, не гарантирует доставки пакетов, так

что нам необходимо предусмотреть механизм тайм-аутов. Несмотря на это, в высокоскоростных локальных сетях потеря пакетов UDP – это большая редкость.

Ниже приведена новая реализация клиента с использованием декоратора:

```
import functools

def timeout_op(func, max_attempts=3):
    @functools.wraps(func)
    def wrapper(*args, **kws):
        attempts = 0
        while attempts < max_attempts:
            try:
                return func(*args, **kws)
            except socket.timeout:
                print('Timeout: retrying')
                attempts += 1
        return None
    return wrapper

@timeout_op
def send_text(sock, text):
    ...

@timeout_op
def request_text(sock, text_id):
    ...

sock = socket.socket(
    socket.AF_INET,
    socket.SOCK_DGRAM)
sock.settimeout(1.0)  ← Устанавливаем тайм-аут для сокета
```

Здесь мы просто применили наш декоратор к функциям `send_text` и `request_text`. По умолчанию сокет работает в блокирующем режиме, т. е. ждет, когда будет получен ответ. Поэтому мы после создания сокета воспользовались методом `settimeout`, чтобы сделать его неблокирующим с возвратом через секунду в случае отсутствия ответа. Такого простого механизма на тайм-аутах должно быть достаточно для нашего клиента как раз потому, что мы исходим из предположения о том, что наша высокоскоростная локальная сеть достаточно надежна, чтобы не терять пакеты UDP.

Есть смысл сделать нечто похожее и на стороне сервера. Здесь может возникнуть проблема с семантикой повторения операций: если вы сохраните текст дважды, вы впустую потратите дисковое пространство. Будьте внимательны при создании операций на сервере. Может оказаться, что их повторения будут приводить к излишнему расходованию ресурсов.

6.4.5. Прочие предпосылки для оптимизации сетевых вычислений

В нашей реализации используется протокол UDP с целью существенного снижения объема накладных расходов при отправке сообщений, но иногда вам может понадобиться прибегнуть к использованию протокола TCP или даже HTTPS или других протоколов, работающих поверх TCP. Ниже приведены несколько советов, которые могут пригодиться вам, если это ваш случай:

- *если ваш клиент посылает несколько запросов на сервер, постарайтесь использовать для них одно соединение.* Так вы будете вынуждены лишь раз открывать и закрывать соединение при отправке сообщений;
- иногда бывает возможно заранее открыть соединение TCP, до наступления пиковой нагрузки. Это позволит не включать время для установки соединения в критически важные лимиты пикового общения между клиентом и сервером. Эта техника так же, как и предыдущая, обычно применяется совместно с подключением к базе данных и называется *пулом соединений* (connection pooling);
- *если протокол UDP для вас слишком прост, а TCP слишком требователен, можете воспользоваться относительно новым протоколом QUIC.* Аббревиатура QUIC изначально расшифровывалась как Quick UDP Internet Connections (быстрые интернет-соединения UDP). Как понятно из этого названия, разработчики этого протокола старались объединить в нем все лучшее, что есть в протоколе UDP.

Заключение

- Хорошее понимание иерархии памяти лежит в основе написания эффективных приложений. Большинство разработчиков знают о различиях в отношении производительности при использовании оперативной памяти, дискового хранилища и сетевых ресурсов, но далеко не все понимают, как можно наладить эффективное взаимодействие между кешем процессора и динамической оперативной памятью.
- Относительно медленный доступ к динамической оперативной памяти (DRAM) вынуждает центральный процессор простаивать в течение многих рабочих циклов. Перенос как можно большего объема данных в кеш процессора позволит существенно ускорить их обработку.
- Алгоритмы, помогающие избегать процессорного голодания, способны работать крайне эффективно, но их семантика


не всегда интуитивно понятна. Например, в некоторых ситуациях бывает быстрее работать напрямую со сжатыми данными, чем с сырыми. Это объясняется тем, что на распаковку данных иногда требуется времени меньше, чем на извлечение большего их объема в несжатом виде из оперативной памяти.

- Библиотека Blosc позволяет эффективно работать со сжатыми представлениями массивов, а бонусом является экономия драгоценного места в памяти.
- Библиотека NumExpr предлагает свои услуги по вычислению выражений NumPy за меньшее время и с меньшими затратами памяти по сравнению с традиционными механизмами обработки NumPy. Эта библиотека активно задействует использование кеша первого уровня центрального процессора и другие техники, позволяющие уменьшить время обработки выражений – иногда более чем на порядок.
- Современные сетевые архитектуры предлагают гораздо более низкое время доступа к сетевым ресурсам, что позволяет обращаться к соседним машинам в сети быстрее, чем к локальным дискам.
- Стандартные методики REST API слишком медлительны и неэффективны для использования в высокоскоростных локальных сетях.
- Взаимодействие по сети можно ускорить сразу несколькими способами: от выбора протокола (TCP против UDP) до отказа от HTTPS и использования более быстрых методов сериализации данных по сравнению с JSON.

Часть III

Приложения и библиотеки для современной обработки данных

Третья часть книги будет в основном посвящена проблемам при работе с данными, и по большей части мы будем обсуждать в ней особенности различных библиотек Python для анализа данных. Сначала поговорим о вездесущей библиотеке `pandas`, незаменимой при работе с датафреймами. Также мы не обойдем вниманием современную библиотеку `Apache Arrow`, которая среди прочего может ускорять работу пакета `pandas`. После этого обратим взор на библиотеки, позволяющие извлечь максимум возможного в области хранения данных. В частности, рассмотрим библиотеку `Zarr`, предназначенную для хранения N-мерных массивов, и библиотеку `Parquet`, проявляющую себя при работе с датафреймами. Также мы поговорим об эффективной работе с наборами данных, объем которых превышает доступную память.



Высокопроизводительный *pandas* и *Apache Arrow*

В этой главе мы обсудим следующие темы:

- оптимизация использования памяти в процессе создания датафреймов в *pandas*;
- снижение вычислительных затрат при выполнении операций в *pandas*;
- использование Cython, NumExpr и Numpy для ускорения работы *pandas*;
- оптимизация *pandas* при помощи *Apache Arrow*.

Анализ данных в Python уже практически стал синонимом названия библиотеки *pandas*. *Pandas* представляет собой библиотеку для работы с датафреймами или эффективной обработки табличных данных. В языке Python библиотека *pandas* фактически стала стандартом для работы с табличными данными, размещающимися в памяти. В этой главе мы поговорим о способах оптимизации этой библиотеки. При этом рассмотрим два основных подхода, один из которых связан с возможностями оптимизации, заложенными в самом *pandas*, а второй предусматривает использование сторонней библиотеки *Apache Arrow*.

Библиотека Apache Arrow представляет собой языково-независимый функционал для эффективного доступа к колоночным данным, позволяющий делиться такими данными между разными реализациями языков и передавать их различным процессам и даже компьютерам. Эта библиотека может использоваться в качестве дополнения к pandas для повышения производительности, поскольку предоставляет быстрые алгоритмы выполнения базовых операций, таких чтение файлов CSV, преобразование датафреймов pandas в формат низкоуровневых языков для более быстрой обработки и сериализация датафреймов для их передачи между машинами.

Начнем мы с обзора техник, позволяющих оптимизировать использование библиотеки pandas. В этом разделе мы будем говорить отдельно об экономии времени обработки и занимаемой памяти. С учетом того, что библиотека pandas работает с данными в памяти, разработчику необходимо бросить все силы на то, чтобы датафреймы занимали как можно меньше места в памяти, что позволит ему не только выполнять сложную аналитику, но и загружать большие объемы информации без необходимости хранить данные на диске (к вопросам хранения данных на диске вернемся в главе 10).

Далее мы снова поговорим о библиотеках NumPy, Cython и NumExpr, но на этот раз используем их с целью оптимизации работы с датафреймами в pandas. Поскольку в основе библиотеки pandas лежит NumPy, для оптимизации работы с ней можно без труда обращаться к библиотекам Cython и NumExpr.

После этого речь пойдет о библиотеке Apache Arrow, причем сразу в двух аспектах. Во-первых, мы увидим альтернативные реализации стандартных алгоритмов pandas в Apache Arrow. Например, рассмотрим вариант чтения файла CSV не напрямую, а с промежуточной остановкой в Arrow и оценим эффективность этого способа. Во-вторых, мы воспользуемся Arrow для эффективной передачи датафреймов в форматы низкоуровневых языков программирования с целью более быстрой обработки данных посредством имеющихся в этих языках алгоритмов.

Но начнем мы с оптимизации загрузки данных в стандартном pandas. Если вы используете менеджер пакетов conda, вам необходимо установить PyArrow: на момент написания книги наиболее простым способом установить эту библиотеку был запуск команды `pip install pyarrow`. Если вы работаете в Docker, воспользуйтесь образом `tiagoantao/python-performance-dask`.

7.1. Оптимизация памяти и времени при загрузке данных

Нашей первой задачей будет оптимизация памяти и времени при загрузке данных в *датафрейм* (data frame) pandas. В следующем разделе

мы перейдем к процессу оптимизации операций по анализу данных. Для нашего примера будем использовать набор данных с поездками знаменитых желтых такси в Нью-Йорке. Компания Taxi and Limousine Commission (TLC) выложила в общий доступ набор данных, который можно скачать по адресу <http://mng.bz/516D> (<https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page>). Мы воспользуемся данными Yellow Car за январь 2020 года. В нем присутствует полная информация обо всех поездках такси, включая время посадки и высадки, количество пассажиров, стоимость поездки, сумму чаевых и т. д.

Начнем с локальной загрузки данных. Несмотря на то что pandas может загружать данные непосредственно с удаленного источника, мы не хотим каждый раз ожидать загрузки, чтобы не тратить время и лишний раз не нагружать сервер. В процессе получения данных нас будет интересовать два вопроса: сколько места понадобится pandas для загрузки всего набора данных или отдельных колонок и как можно снизить объем используемой памяти. Загрузить архив весом 566 Мб вы можете по адресу https://tiago.org/yellow_tripdata_2020-01.csv.gz.

7.1.1. Сжатые и несжатые данные

Давайте начнем с загрузки данных в датафрейм (код можно найти в файле 07-pandas/sec1-intro/read_csv.py):

```
import pandas as pd

df = pd.read_csv("yellow_tripdata_2020-01.csv")
```

На моем компьютере операция загрузки заняла около 10 с. Как вы уже видели в предыдущих главах, предварительное сжатие данных может оказать положительный эффект в отношении времени, требующегося для загрузки. Давайте попробуем сжать данные при помощи архиватора xz и снова загрузить их. Вам понадобится установить архиватор xz и с его помощью сжать файл `yellow_tripdata_2020-01.csv`, а затем загрузить его в датафрейм:

```
df = pd.read_csv("yellow_tripdata_2020-01.csv.xz")
```

Библиотека pandas достаточно продвинута, чтобы автоматически определять тип компрессии исходя из расширения файла, но можно указать этот параметр и явно. На моем компьютере эта операция заняла 15 с. Раньше было быстрее, зато теперь файл занимает всего 74 Мб, что в 7 раз меньше оригинала. Как видите, время загрузки в данном случае нам сократить не удалось, так что придется поискать компромисс между размерами файла и временем загрузки. Этот баланс зависит от вашей конкретной задачи. Подробнее мы поговорим об этом, когда будем обсуждать библиотеку Apache

Arrow, а сейчас просто сравним три типа архивации по объему файлов и времени для их загрузки (результаты приведены в табл. 7.1). Как и всегда, время будет напрямую зависеть от ваших мощностей, но относительная разница между методами будет сохраняться. В зависимости от ваших требований вам может понадобиться максимально быстрая загрузка данных или предельная экономия места на диске.

Таблица 7.1. Сравнение методов архивации файла CSV по времени загрузки и размеру файла

Приложение	Время чтения (с)	Размер (Мб)
Нет	10	566
gip	12	105
bzip2	26	103
xz	15	74

Не воспринимайте информацию о размерах файлов для разных методов компрессии как истину в последней инстанции – всегда проверяйте эффективность алгоритмов на своих данных.

Вывод

В данном примере и в целом в этой главе главной идеей является не то, какие относительные результаты мы наблюдаем, а сам факт того, что разные реализации и алгоритмы могут приводить к совершенно разным цифрам. В плане оптимизации памяти и времени (на самом деле чаще памяти *или* времени) вам необходимо помнить о двух важных вещах при загрузке данных. Следует понимать алгоритмы, которые используете, а не воспринимать их просто как черный ящик. В этом случае вы сможете прогнозировать определенные результаты в плане производительности. Кроме того, нужно четко представлять требования своей задачи и понимать, на что необходимо в первую очередь направить стрелы оптимизации: на время или ресурсы памяти.

7.1.2. Определение типов данных колонок

При загрузке данных вы получите следующее предупреждение (по крайней в pandas версии 1.0.5):

```
DtypeWarning: Columns (6) have mixed types. Specify the dtype option on
➡ import or set low_memory=False
```

Это сообщение говорит о том, что загрузчик данных не смог корректно вывести типы данных всех колонок в наборе данных.

ПРЕДУПРЕЖДЕНИЕ. Не поддавайтесь соблазну установить опцию `low_memory=False`, как советует `pandas`. При работе с большими данными ваша программа может превысить имеющиеся ресурсы и завершиться аварийно.

Обычно подобные сообщения говорят о том, что какие-то из столбцов в вашем наборе данных были загружены с использованием обобщенных типов данных. К примеру, колонка с целочисленными значениями могла быть загружена в виде объектов с соответствующими накладными расходами на их хранение. Позже в этой главе мы рассмотрим конкретные примеры.

Перед тем как опускаться до уровня детализации колонок, давайте узнаем, сколько памяти в целом занимает весь датафрейм. В `pandas` для этого предусмотрен специальный метод, который подходит лучше, чем более общие методы, упомянутые в предыдущих главах:

```
df.info(memory_usage="deep")
```

Сокращенный вывод этого метода приведен ниже:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6405008 entries, 0 to 6405007
Data columns (total 18 columns):
#   Column                Dtype
---  -
0   VendorID              float64
1   tpep_pickup_datetime  object
2   tpep_dropoff_datetime object
3   passenger_count       float64
4   trip_distance         float64
5   RatecodeID            float64
6   store_and_fwd_flag    object
7   PULocationID          int64
8   DOLocationID          int64
9   payment_type          float64
...
17  congestion_surcharge   float64
dtypes: float64(13), int64(2), object(3)
memory usage: 2.0 GB
```

Здесь мы видим информацию о типе данных каждой колонки, количестве записей и занимаемом размере. Как видите, в нашем случае файл размером 566 Мб разросся до 2 Гб! С учетом того, что речь идет о текстовых данных, это довольно странно.

Давайте посмотрим, сколько места в памяти занимает каждая колонка, а заодно запросим количество уникальных значений в них:

```
def summarize_columns(df):
    for c in df.columns:
        print(c, len(df[c].unique()),
              df[c].memory_usage(deep=True) // (1024**2), sep="\t")

summarize_columns()
```

Сокращенный вывод показан ниже:

tpep_pickup_datetime	2134342	object	464
passenger_count	11	float64	48
trip_distance	5606	float64	48
RatecodeID	8	float64	48
store_and_fwd_flag	3	object	401
PULocationID	261	int64	48
payment_type	6	float64	48
fare_amount	5283	float64	48
improvement_surcharge	3	float64	48
total_amount	12488	float64	48
congestion_surcharge	8	float64	48

Каждая из колонок с типом `object` занимает больше 400 Мб (причины этого мы подробно обсуждали в главе 2). Тип `float64` требует выделения 64 бит, или 8 байт, для каждого значения с плавающей запятой. Получаем общий объем 48 Мб. То же касается и типа данных `int64`. Можем ли мы сократить место в памяти, занимаемое датафреймом, если изменим типы данных? Конечно, и существенно.

Давайте начнем с колонок `tpep_pickup_datetime` и `tpep_dropoff_datetime`. Как ясно из названий, в этих колонках хранятся даты и времена. Вы можете выяснить это с помощью инструкции `df["tpep_pickup_datetime"].head()`. Давайте преобразуем эти колонки в формат `datetime` следующим образом:

```
df["tpep_pickup_datetime"] = pd.to_datetime(df["tpep_pickup_datetime"])
df["tpep_dropoff_datetime"] = pd.to_datetime(df["tpep_dropoff_datetime"])
```

Одно лишь это действие позволило уменьшить размер каждого столбца с 464 до 48 Мб, а датафрейма в целом – с 2 до 1,2 Гб. Этого уже достаточно, чтобы осознать всю важность использования подходящих типов данных.

Также стоит отметить, что в нашем наборе данных присутствуют дискретные поля, содержащие одно из нескольких возможных значений. К примеру, поле `payment_type` может содержать одно из шести числовых значений, но при этом обладает 8-байтовым типом `float64`. Давайте приведем его к 1-байтовому целочисленному типу:

```
import numpy as np
df["payment_type"] = df["payment_type"].astype(np.int8)
```

Заметьте, что 8-битный целочисленный тип данных располагается в модуле NumPy, что лишний раз подтверждает родственность между библиотеками pandas и NumPy.

К сожалению, это преобразование не может быть выполнено. Если вы просмотрите значения в колонке, то обнаружите в ней присутствие незаполненных значений (*NA*) вперемешку с числовыми. Можно заменить пропуски нулями, поскольку в других случаях нули в этой колонке не используются. В противном случае мы могли бы выбрать другое значение:

```
df["payment_type"] = df["payment_type"].fillna(0).astype(np.int8)
```

Это изменение позволило, как и ожидалось, уменьшить размер колонки с 48 до 6 Мб (вместо 8 байт на одно значение стал выделяться 1 байт). У нас есть шесть колонок, размер значений в которых можно уменьшить с 64 до 8 бит, а две колонки требуют 16 бит на значение. Итого еще 450 Мб экономии. В результате мы уже снизили объем датафрейма с 2 Гб до 750 Мб.

Не обманывайтесь на предмет того, как легко и просто мы все решили. Кодирование значений и избавление от пропущенных значений – процесс весьма громоздкий и долгий, который зачастую так просто не решается.

Колонка `store_and_fwd_flag` – один из примеров таких сложных для разбора полей. Здесь хранятся булевы значения, показывающие, была ли оплата сохранена в памяти автомобиля по причине недоступности сервера, хранящего оплаты. Для многих записей значение в этой колонке неизвестно (т. е. мы не знаем, оно `true` или `false`). Если бы у нас не было пропущенных значений, мы могли бы дать колонке логический тип, предполагающий выделение 1 бита на значение. Но в присутствии третьего значения нам необходимо использовать следующий по вместимости 8-битный тип данных. Таким образом, добавление одного значения приводит к 8-кратному увеличению размера столбца в памяти. Мы применим следующее преобразование:

```
df["store_and_fwd_flag"] = df["store_and_fwd_flag"].fillna(" ").apply(ord).apply(lambda x: [32, 78, 89].index(x) - 1).astype(np.int8)
```

Мы преобразовали пропущенные значения в пробелы и извлекли коды ASCII для всех допустимых значений: 32 для пробела, 78 для `N` и 89 для `Y`. С помощью функции `index` мы можем закодировать пропущенные значения (32) как `-1`, `N` (78) как `0` и `Y` (89) как `1`.

Вывод

В этой книге мы еще не раз коснемся темы представления пустых значений в столбцах, особенно в следующей главе, где будем обсуждать вопросы хранения данных и конкретно формат Parquet. На данный момент вам достаточно будет знать, что наиболее распространенной причиной перерасхода памяти (а следовательно, и снижения скорости вычислений) является представление колоночных данных с более обобщенными типами, чем требуется. Чем более объемный тип данных выбран, тем больше места в памяти будет занимать колонка и тем медленнее будут выполняться вычисления. Изменение типов данных колонок – не всегда тривиальная задача, но с ее помощью можно существенно сократить объем памяти, занимаемой датафреймом.

7.1.3. Эффект изменения точности типа данных

Еще одной техникой, способствующей уменьшению занимаемого колонкой места в памяти, является снижение *точности* (precision) выбранного типа данных без его изменения. К примеру, мы можем преобразовать тип данных колонки с денежными суммами из float64 в float32, т. е. изменить точность с двойной на одинарную, что позволит значительно снизить объем используемой памяти:

```
df["fare_amount_32"] = df["fare_amount"].astype(np.float32)
```

После этого необходимо проверить, сколько составили наши потери от изменения точности. Для этого можно воспользоваться следующей простой инструкцией:

```
(df["fare_amount_32"] - df["fare_amount"]).abs().sum()
```

Таким образом, мы можем вычислить разницу между одинарной и двойной точностью. При этом необходимо оперировать абсолютными величинами, чтобы избежать эффекта взаимного погашения отклонений. В нашем случае суммарная ошибка при изменении точности составила огромные 0,063 долл.:

```
df = pd.read_csv(
    "yellow_tripdata_2020-01.csv.gz",
    dtype={
        "PULocationID": np.uint8,
        "DOLocationID": np.uint8
    },
    parse_dates=[
        "tpep_pickup_datetime",
        "tpep_dropoff_datetime"
    ],
    converters={
        "VendorID":

```

Создаем конвертеры – в основном для преобразования пропущенных значений

Указываем разные типы для колонок

Ограничиваем хранение некоторых колонок восемью битами на одно значение

Есть и другой способ указать типы данных колонок

```

        lambda x: np.int8(["", "1", "2"].index(x)), <—
"store_and_fwd_flag":
        lambda x: ["", "N", "Y"].index(x) - 1,
"payment_type":
        lambda x: -1 if x == "" else int(x),
"RatecodeID":
        lambda x: -1 if x == "" else int(x),
"passenger_count":
        lambda x: -1 if x == "" else int(x)
    }
)
    
```

Явным образом приводим колонку VendorID к типу данных np.int8

Если сейчас выполнить команду `df.info(memory_usage="deep")`, мы увидим, что датафрейм занимает в памяти 757,4 Мб. Обратите внимание при этом, что большинство числовых типов отображаются с 64-битной длиной, включая поле `VendorID`, несмотря на то что мы преобразовали его в `np.int8`, и, судя по всему, безуспешно.

Но у нас есть поля, которые явно могут занимать меньше места. Поскольку мы решили пожертвовать точностью, то преобразуем 64-битные числа с плавающей запятой в 16-битные. Заодно приведем 64-битные целочисленные значения к 8-битным, поскольку в нашем конкретном случае 8-битного диапазона от -128 до 127 будет вполне достаточно для хранения этих значений:

```

for c in df.columns:
    if df[c].dtype == np.float64:
        df[c] = df[c].astype(np.float16)
    if df[c].dtype == np.int64:
        df[c] = df[c].astype(np.int8)
    
```

Теперь наш датафрейм занимает 250,4 Мб в памяти, тогда как начинали мы с 2 Гб. Неплохо.

7.1.4. Кодирование и снижение объема данных

Если вам это нужно, вы можете попытаться еще больше сократить объем памяти, требующейся для хранения данных. К примеру, в некоторых числовых колонках на самом деле хранится небольшое количество дискретных значений. Давайте попробуем найти их:

```

for c in df.columns:
    cnts = df[c].value_counts(dropna=False) <—
    if len(cnts) < 10: <—
        print(cnts)
    
```

Функция `value_counts` возвращает значения из колонки с количеством их появлений

Выводим на экран колонки с количеством значений, меньшим десяти

Здесь мы выводим имена колонок, в которых содержится менее десяти уникальных значений. Десять – это произвольное число, вы можете ввести любое другое. Некоторые из этих колонок мы уже

оптимизировали, однако две из них хранятся в 16-битном формате без особых на то причин. В колонке `improvement_surcharge` хранятся лишь три уникальных значения: 0, 0,3 и -0,3. Эти значения можно легко сконвертировать в 0, -1 и 1, что позволит задать для колонки более экономичный тип данных. В поле `congestion_surcharge` хранятся следующие значения: -2,5, -0,75, 0,5, 0,0, 0,5, 0,75, 2,0, 2,5 и 2,5. Как вы уже догадались, если умножить эти значения на 4, мы получим целые числа. Таким образом, мы сможем использовать для их хранения 8-битный целочисленный тип данных, кодируя числа с помощью умножения на 4 и декодируя с помощью деления.

Наконец, есть и радикальный способ сэкономить память, который состоит в загрузке только тех данных, которые нужны вам для работы. Для нашей следующей задачи будет достаточно информации о времени посадки и высадки в такси, а также о сумме чаевых. Ограничить данные можно непосредственно в pandas следующим образом:

```
df = pd.read_csv(
    "yellow_tripdata_2020-01.csv.gz",
    dtype={
        "congestion_surcharge": np.float16,
    },
    parse_dates=[
        "tpep_pickup_datetime",
        "tpep_dropoff_datetime"],
    usecols=[
        "congestion_surcharge",
        "tpep_pickup_datetime",
        "tpep_dropoff_datetime"],
)
```

В результате датафрейм стал занимать 109,9 Мб, что составляет порядка 5 % от исходного объема в 2 Гб. Мы добились этого, ограничив количество извлекаемых колонок и изменив некоторые типы данных.

Кажущаяся безопасность инструкции `inplace=True`

В большинстве методов библиотеки pandas предусмотрена возможность изменения существующей структуры данных на месте, без возвращения измененного датафрейма или ряда данных. Таким образом, мы можем сэкономить половину памяти, потеряв при этом исходные данные. К примеру, вы можете удалить все строки, в которых присутствуют пропущенные значения, следующим образом:

```
new_df = df.dropna()
```

В результате вы получите три датафрейма, которые будут занимать вдвое больше памяти. В качестве альтернативы можете использовать следующий синтаксис:

```
df.dropna(inplace=True)
```

Это позволит изменить исходный датафрейм. Сработает такой метод не всегда, но во многих ситуациях он является простейшим способом снижения объема используемой памяти.

Но будьте осторожны: во время выполнения этой операции pandas выделит место для обоих массивов. Таким образом, в процессе исполнения этого кода памяти потребуется вдвое больше. Получается, что, по сути, эта опция просто добавляет коду читаемости, но тот же функционал может быть реализован при помощи использования оператора *del* после вызова метода без опции *inplace*.

Библиотека Arrow предлагает более интересный подход к экономии памяти с помощью параметра *self_destruct*, о котором мы поговорим далее в этой главе.

В этом разделе мы продемонстрировали, как выбор правильных представлений для колонок влияет на объем итогового датафрейма. На практике вы можете задать все требуемые преобразования прямо при загрузке данных в pandas:

```
df = pd.read_csv(
    "yellow_tripdata_2020-01.csv.gz",
    dtype={
        "VendorID": np.int8,
        "trip_distance": np.float16,
        "PULocationID": np.uint8,
        "DOLocationID": np.uint8,
    },
    parse_dates=[
        "tpep_pickup_datetime",
        "tpep_dropoff_datetime"],
    converters={
        "VendorID":
            lambda x: np.int8(["", "1", "2"].index(x)),
        "store_and_fwd_flag":
            lambda x: ["", "N", "Y"].index(x) - 1,
        "payment_type":
            lambda x: -1 if x == "" else int(x),
        "RatecodeID":
            lambda x: -1 if x == "" else int(x),
        "passenger_count":
            lambda x: -1 if x == "" else int(x)
    }
)
```

Можно задать желаемый тип данных для определенных колонок

Колонки с датами обрабатываются отдельно от остальных

Преобразования во время загрузки

Обратите внимание, что для целочисленных значений и чисел с плавающей запятой всегда будут предлагаться более объемные типы данных, так что вам наверняка придется приводить их к более экономичным вариантам. Теперь, когда мы наиболее эффек-

тивно загрузили данные в память, давайте посмотрим, как можно повлиять на скорость анализа данных в pandas.

Вывод

Мы привели несколько примеров, с помощью которых продемонстрировали важность выбора правильных типов данных для колонок с точки зрения занимаемого ими места в памяти. Изменение типа данных и точности – это действия, не предполагающие серьезных компромиссов.

Эти основные подходы к преобразованию данных могут быть расширены, и мы поговорим об этом далее в этой книге. В частности, большая часть заключительной главы будет посвящена снижению объема данных в памяти и их представлению.

Удобно, что обычно не приходится конвертировать данные из одного типа данных в другой после их загрузки, поскольку все это можно сделать уже на этапе получения данных. В следующем примере мы воспользуемся функцией `read_csv` для выполнения большинства преобразований на лету.

7.2. Техники для повышения скорости анализа данных

Давайте обратимся к данным о нью-йоркских такси и выполним небольшой статистический анализ. К примеру, мы определим процент оплаты, приходящийся на чаевые. Мы не будем специально концентрироваться на анализе данных с использованием статистики, поскольку книга не об этом. Здесь нас больше интересует, как можно эффективно извлечь данные и подготовить их к анализу любой степени сложности. Начнем с применения техник индексирования датафреймов и способов прохождения по строкам.

Давайте загрузим данные. Нам понадобится лишь три поля. Исходный код можно найти в файле `07-pandas/sec2-intro/index.py`:

```
df = pd.read_csv(
    "yellow_tripdata_2020-01.csv.gz",
    dtype={
        "congestion_surcharge": np.float16,
    },
    parse_dates=[
        "tpep_pickup_datetime",
        "tpep_dropoff_datetime"],
    usecols=[
        "congestion_surcharge",
        "tpep_pickup_datetime",
        "tpep_dropoff_datetime"],
)
```

7.2.1. Использование индексирования для ускорения доступа к данным

Давайте выберем все записи с определенной датой и временем посадки в такси:

```
df[df["tpep_pickup_datetime"] == "2020-01-06 08:13:00"]
```

На моем компьютере время выполнения этой инструкции, измеренное с помощью ключевого слова `timeit`, в среднем оказалось равно 17,1 мс. Можно попробовать предварительно отсортировать датафрейм по выбранной колонке:

```
df_sorted = df.sort_values("tpep_pickup_datetime")
df_sorted[df_sorted["tpep_pickup_datetime"] == "2020-01-06 08:13:00"]
```

К сожалению, время выполнения сильно не изменилось: `pandas` игнорирует сортировку при выборе строк. При этом можно ожидать большого выигрыша в скорости при использовании индекса:

```
df_pickup = df.set_index("tpep_pickup_datetime")
df_pickup_sorted = df_pickup.sort_index()
df_pickup.loc["2020-01-06 08:13:00"]
df_pickup_sorted.loc["2020-01-06 08:13:00"]
```

Здесь мы проиндексировали датафрейм по полю `tpep_pickup_datetime`. Поиск элемента в неотсортированном, но проиндексированном датафрейме (`df_pickup`) заметных улучшений не дал. В то же время после сортировки (датафрейм `df_pickup_sorted`) мы нашли нужный нам элемент всего за 395 мкс, что в 40 раз быстрее первоначального поиска.

Этот метод поиска содержит массу неудобств, самым очевидным из которых является то, что он может быть применен только к проиндексированному полю. Таким образом, если вам необходимо будет осуществить поиск по другой колонке, придется устанавливать индекс на нее или воспользоваться индексом на основе нескольких колонок. Исходя из этого, прием с индексированием полей нельзя назвать основным решением при поиске, но рассмотренный пример дает понять, как важно правильно установить индексы в датафрейме с точки зрения производительности. Полагаясь на индексы (а по большей части в `pandas` индексы игнорируются), вы жертвуете единообразием в написании запросов в обмен на возможный выигрыш в скорости. К примеру, если мы хотим задать двойное условие в запросе, можем написать следующий код:

```
df[
    (df["tpep_pickup_datetime"] == "2020-01-06 08:13:00") &
    (df["congestion_surcharge"] > 0)]
```

Обратите внимание, что к обеим колонкам мы обращаемся одинаково. В присутствии индекса на колонке `trip_pickup_datetime` мы могли бы воспользоваться альтернативным синтаксисом, приведенным ниже:

```
my_time = df_pickup_sort.loc["2020-01-06 08:13:00"]
my_time[my_time["congestion_surcharge"] > 0]
```

СОВЕТ. Все аргументы, приведенные здесь в контексте индексирования датафреймов, могут быть применены к объединению датафреймов с помощью метода `df.join`, и даже с большим эффектом в плане производительности.

Теперь, когда мы узнали, как использование индексов в датафреймах может влиять на быстродействие выполняемых операций, давайте воспользуемся всем датафреймом для расчета средней доли сумм чаевых.

7.2.2. Техники перемещения по строкам

В этом разделе мы рассмотрим разные способы прохода по датафреймам. Мы рассчитаем долю чаевых в общей сумме, что потребует прохождения по всему набору данных и извлечения сумм чаевых и общих сумм оплаты.

Начнем с чтения данных и удаления записей с нулевыми суммами. Код можно найти в файле `07-pandas/sec2-speed/traversing.py`:

```
df = pd.read_csv("../sec1-intro/yellow_tripdata_2020-01.csv.gz")
# ^^ replace

df = df[(df.total_amount != 0)]
df_10 = df.sample(frac=0.1)
df_100 = df.sample(frac=0.01)
```

Заметьте, что мы сделали две выборки из нашего датафрейма объемом 10 и 1 % от общего объема, что поможет нам в дальнейшем при проверке производительности.

Давайте начнем с традиционной для Python техники, не основанной на pandas или NumPy-подобных приемах вроде векторизации. Это будет обычный цикл `for` по всем строкам:

```
def get_tip_mean_explicit(df):
    all_tips = 0
    all_totals = 0
    for i in range(len(df)):
        row = df.iloc[i]
        all_tips += row["tip_amount"]
        all_totals += row["total_amount"]
    return all_tips / all_totals
```

Обычный цикл в Python с использованием количества строк в датафрейме

Доступ к строке по позиции

Именно так писали бы разработчики на Python, не вооруженные pandas или NumPy. Как и ожидалось, производительность этого кода оставляет желать много лучшего – на моем компьютере длительность его выполнения исчислялась минутами¹.

Существует два альтернативных подхода с применением циклов `for`, которые дадут результаты получше. Первый из них связан с методом датафрейма с именем `iterrows` и показан ниже:

```
def get_tip_mean_iterrows(df):
    all_tips = 0
    all_totals = 0
    for i, row in df.iterrows():
        all_tips += row["tip_amount"]
        all_totals += row["total_amount"]
    return all_tips / all_totals
```

Это все еще цикл `for`, но здесь мы воспользовались итератором pandas, возвращающим текущую позицию и строку. Время исполнения кода чуть улучшилось, но не сильно.

СОВЕТ. Если вы только начинаете работать с pandas и NumPy, ваша привычка использовать циклы `for` в своих алгоритмах будет совершенно понятной и оправданной. В перспективе необходимо будет осваивать другие приемы множественных вычислений, такие как векторизация (позже мы рассмотрим пару таких примеров), но и на первых порах мы советуем вам стараться отказываться от использования явных итераций и конструкций на основе `iterrows`. Среди всех подходов, использующих ключевое слово `for`, вариант с методом `itertuples` сэкономит вам больше всего времени, и при этом не придется отказываться от привычной парадигмы.

Наш заключительный пример с использованием циклов `for` как раз будет основан на методе `itertuples`. Здесь мы будем на каждой итерации цикла получать *кортеж* (tuple) значений:

```
def get_tip_mean_itertuples(df):
    all_tips = 0
    all_totals = 0
    for my_tuple in df.itertuples():
        all_tips += my_tuple.tip_amount
        all_totals += my_tuple.total_amount
    return all_tips / all_totals
```

¹ Если вы планируете проверить этот код на эффективность, лучше воспользуйтесь сокращенными версиями датафрейма из переменных `df_10` или `df_100`: вы успеете почувствовать всю неловкость ситуации с ожиданием, но при этом вам не придется ждать целую вечность.

Хотя внешне этот подход мало чем отличается от предыдущих, время выполнения цикла составило уже 18 с.

Теперь перейдем к рассмотрению приемов, основанных на диалекте библиотеки pandas. Начнем с применения метода *apply*, который концептуально схож с функцией *map*. Каждая строка в этом случае будет обрабатываться отдельно:

```
def get_tip_mean_apply(df):
```

```
    frac_tip = df.apply(
        lambda row: row["tip_amount"] / row["total_amount"], axis=1
```

Применяем метод *mean* для вычисления итогового результата

Метод *apply* может быть применен как к колонкам (поведение по умолчанию), так и к строкам, как в нашем случае. Значение параметра *axis* по умолчанию равно нулю, что приводит к обработке столбцов. Для прохода по строкам мы изменили значение на единицу

```
    )
    return frac_tip.mean()
```

Использование метода *apply* позволило снизить время выполнения кода до 9,5 с, что вдвое быстрее по сравнению с предыдущим примером, но далеко не идеально.

Перед обсуждением решений, основанных на векторизации, давайте попробуем применить другую нотацию внутри метода *apply*:

```
def get_tip_mean_apply2(df): # df_10: 14,9 с
```

```
    frac_tip = df.apply(
        lambda row: row.tip_amount / row.total_amount, axis=1
    )
```

```
    return frac_tip.mean()
```

Разница заключается в обращении к колонкам датафрейма: мы использовали нотацию с точкой (*row.tip_amount* и *row.total_amount*) вместо квадратных скобок (*row["tip_amount"]* и *row["total_amount"]*). Результат оказался похуже – в районе 14 с на моем компьютере.

СОВЕТ. На разных версиях pandas относительные результаты применения разных методов могут варьироваться, поскольку нет никакой гарантии, что алгоритмы не подверглись доработкам. Это применимо как к обращению к значениям в строках, так и к любым другим алгоритмам. Так что всякий раз, когда производительность кода вас не удовлетворяет, попробуйте использовать другие алгоритмы (*for*, *apply*, векторизацию и т. д.) и разные методы доступа к объектам (как к атрибуту или как к значению в словаре). Не принимайте на веру все относительные результаты алгоритмов, приведенные в этой книге.

Теперь давайте рассмотрим оптимальный способ вычисления с использованием pandas и применим векторизованный подход:

```
def get_tip_mean_vector(df):  
    frac_tip = df["tip_amount"] / df["total_amount"]  
    return frac_tip.mean()
```

Здесь мы извлекаем объекты *Series* для колонок `tip_amount` и `total_amount` и делим их друг на друга, после чего вызываем метод `mean`. Время выполнения кода обрушилось на несколько порядков и достигло 32 мс.

Помните, что здесь мы приводим самый простой пример для иллюстрации важности выбора наиболее подходящего метода при итерациях по строкам. Для более сложных вычислений постарайтесь прийти к векторизованному решению. Если это невозможно, вы можете разбить вычисление на части, чтобы выделить векторизованные (быстрые) составляющие. После этого можно заняться оптимизацией кода, не поддающегося векторизации.

Вывод

Процесс загрузки данных зачастую не рассматривается разработчиками как важная часть задачи по обработке данных, которая может внести существенный вклад с точки зрения экономии памяти и ускорения операций. Правильный выбор типов данных для колонок – это основной способ оптимизации использования памяти во время загрузки, но и выбор точности в рамках типа данных играет большую роль. В библиотеке *pandas* вы можете типизировать колонки непосредственно в момент загрузки данных, так что впоследствии вам этим заниматься не придется.

Обработка данных после их загрузки – это отдельный пласт работы, подлежащий оптимизации. Существует два основных подхода к оптимизации выполняемых операций над данными. Первый из них связан с индексированием данных, способным ускорить некоторые операции, но несущим с собой несколько серьезных недостатков. Также вы можете воспользоваться итерациями по строкам. Способов для этого немало, и все они отличаются по скорости и могут быть применены в зависимости от задачи. Здесь главное понимать, что построчный анализ лучше реализовывать в декларативном виде, по возможности применяя векторизацию и избегая явных итераций. Теперь, когда мы познакомились с некоторыми подходами к оптимизации, характерными для *pandas*, давайте рассмотрим низкоуровневые методы по повышению производительности кода.

7.3. Взаимодействие pandas с NumPy, Cython и NumExpr

В следующих нескольких разделах мы разовьем дискуссию, начатую в предыдущих главах, и вновь поговорим о NumPy (глава 4),

Cython (глава 5) и NumExpr (глава 6), но на этот раз применительно к обработке данных в pandas. Если вы пропустили основы по этим технологиям, можете вернуться к указанным главам книги и восполнить пробелы.

Цель этой части главы состоит в исследовании возможностей NumPy, Cython и NumExpr с точки зрения библиотеки pandas и повышении производительности операций, связанных с анализом данных. Чтобы не усложнять повествование, мы продолжим использовать наш пример из предыдущего раздела, касающийся расчета доли чаевых в общей сумме.

7.3.1. Явное использование NumPy

Во всех подходах, которые мы применяли в предыдущем разделе, неявным образом использовалась библиотека NumPy, поскольку на ней основан pandas. Но можно использовать NumPy и явным образом.

Начнем с явного преобразования данных из колонок датафрейма в представления NumPy с последующим выполнением операций, относящихся непосредственно к NumPy. Код можно найти в файле 07-pandas/sec3-numpy-numpyexpr-cython/traversing.py:

```
df_total = df["total_amount"].to_numpy()
df_tip = df["tip_amount"].to_numpy()

print(type(df_tip))

def get_tip_mean_numpy(df_total, df_tip):
    frac_tip = df_total / df_tip
    return frac_tip.mean()
```

Метод `to_numpy` ссылается непосредственно на лежащий в основе колонки массив NumPy

Теперь мы имеем дело с типом данных `numpy.ndarray`, а не `pandas.Series`

Метод `mean` относится к NumPy, а не к pandas

Выполняем деление с использованием векторизации, а не объектов Series

Векторизованный код на основе NumPy на моем компьютере выполнялся за 11 мс в сравнении с векторизованной версией pandas, которая, напомним, заняла 35 мс. Здесь мы взаимодействовали напрямую с массивами NumPy, а не с объектами pandas¹.

СОВЕТ. Метод `to_numpy` возвращает ссылку на соответствующий массив pandas. Если вам нужно создать копию этого объекта, чтобы вычисления не задели исходные данные, используйте этот метод с соответствующим аргументом `to_numpy(copy=True)`. Но помните о том, что в этом случае интерпретатору потребуется создать дубликат объекта, на что может понадобиться время.

¹ Конечно, с точки зрения Python эта разница, скорее, концептуальная: если бы мы передали в функцию объекты Series, операция была бы выполнена над ними. Мы здесь упоминаем об этом явно, тогда как сам язык в силу своей гибкости этого даже не заметит.

7.3.2. Pandas поверх NumExpr

Также вместо задействования движка pandas мы можем для выполнения операций воспользоваться уже знакомой вам библиотекой NumExpr. Эта библиотека позволяет вычислять выражения NumPy с потенциальным выигрышем в скорости – не только за счет своей эффективной многопоточной архитектуры, но и благодаря очень разумному обращению с памятью, что позволяет выполнять вычисления непосредственно с использованием кеша центрального процессора. В предыдущей главе мы уже подробно обсуждали принципы работы библиотеки NumExpr.

Ниже приведена простая реализация вычисления доли чаевых в общей сумме посредством библиотеки NumExpr:

```
def get_tip_mean_numexpr(df):  
    return df.eval("(tip_amount / total_amount).mean()"),  
    engine="numexpr")
```

Как и ожидалось, pandas расширяет язык NumExpr, добавляя в него поддержку датафреймов и объектов Series. Здесь мы в выражении обращаемся к столбцам датафрейма по именам `tip_amount` и `total_amount`, которые автоматически разрешаются в ссылки на колонки в pandas. Также вы можете воспользоваться функцией pandas `eval` в локальном пространстве имен. К примеру, предыдущий код можно переписать так, как показано ниже:

```
def get_tip_mean_numexpr(df):  
    return pd.eval("(df.tip_amount / df.total_amount).mean()"),  
    engine="numexpr")
```

Это позволит вам обращаться больше чем к одному датафрейму в выражении. Вы также сможете использовать переменные, не относящиеся к pandas. За подробностями можно обратиться к официальной документации по адресу <http://mng.bz/aMAX> (https://pandas.pydata.org/pandas-docs/stable/user_guide/enhancingperf.html#expression-evaluation-via-eval).

А как насчет быстродействия? Здесь мы остались в лиге векторизованного решения на pandas – около 35 мс – и также уступаем решению с явным использованием библиотеки NumPy, где выполнение заняло всего 11 мс. В чем причина?

Во-первых, мы платим определенную цену за преобразование выражения в исполняемый код. Таким образом, мы лишь подтверждаем свои слова из предыдущей главы о том, что NumExpr стоит применять только при работе с действительно большими объемами данных, достаточными для оправдания сопутствующих накладных расходов. Оценка того, сколько именно данных будет достаточно, должна выполняться отдельно для каждой задачи по-

средством профилирования кода. В рассматриваемом примере мы используем довольно приличный объем данных. Почему же мы не получили прибавки в производительности?

Возможности NumExpr генерировать эффективный программный код напрямую зависят от сложности выполняемых вычислений. Чем сложнее формулы, тем больше вероятность того, что мы сможем воспользоваться данными, сохраненными в кеше. Давайте рассмотрим один надуманный пример с четырехкратным сложением долей чаевых из нашего сценария: `tip_amount / total_amount + tip_amount / total_amount + tip_amount / total_amount + tip_amount / total_amount`.

Ниже приведены реализации с использованием NumPy и NumExpr:

```
def get_tip_mean_numpy4(df_total, df_tip):
    frac_tip = (
        df_total / df_tip +
        df_total / df_tip +
        df_total / df_tip +
        df_total / df_tip )
    return frac_tip.mean()

def get_tip_mean_numexpr4(df):
    return df.eval(
        "tip_amount / total_amount +"
        "tip_amount / total_amount +"
        "tip_amount / total_amount +"
        "tip_amount / total_amount", engine="numexpr").mean()
```

Единственное, что здесь изменилось, – это сложность формулы. При этом скорость решения с использованием библиотеки NumPy снизилась даже сильнее, чем линейно, – до 55 мс, тогда как вариант с применением NumExpr удержался на своей отметке в 35 мс! Это отнюдь не означает, что скорость выполнения выражений в NumExpr не зависит от сложности формулы. Это лишний раз доказывает то, о чем мы говорили в предыдущей главе: чем больше условий мы создадим для задействования кеша центрального процессора и избежания обращений к оперативной памяти, тем больший прирост в скорости сможем получить даже в случаях, когда это кажется необоснованным.

Вывод

Библиотека NumExpr способна раскрыть весь свой потенциал в присутствии больших объемов данных и сложных формул, а это означает, что ее бывает целесообразно применять в действительно сложных случаях. Как вы видели, в первом простом примере NumExpr со всеми своими возможностями не смогла дотянуться

до NumPy и pandas. Это еще раз подтверждает мысль о том, что оптимальное решение должно выбираться только исходя из конкретной задачи и ее требований. Универсального решения здесь просто не существует, и не стоит везде пытаться использовать представленные в этой книге лаконичные выражения NumExpr. Каждой двери – свой ключик. Теперь давайте обратимся к расширению Cython и посмотрим, сможет ли оно помочь с производительностью решения, основанного на pandas.

7.3.3. Cython и pandas

Теперь мы еще раз перепишем наш код для вычисления доли чаевых в общей сумме, но на этот раз с использованием расширения Cython, в надежде получить решающий перевес. Этот раздел будет довольно коротким, поскольку между Cython и pandas нет непосредственной связи. Таким образом, в нашем примере мы будем использовать Cython исключительно на базе NumPy, как показано на рис. 7.1.

Если вы прочитали главу, посвященную расширению Cython, вы должны уверенно чувствовать себя с приведенным ниже кодом, в котором мы воспользуемся всеми эффективными приемами, рассмотренными ранее.

Вызывающий код на Python можно найти в файле 07-pandas/sec3-numpy-numexpr-cython/traversing_cython_top.py:

```
import pandas as pd
import numpy as np
import pyximport ← Для удобства воспользуемся библиотекой pyximport

pyximport.install( ← Настраиваем pyximport для включения NumPy
    setup_args={
        'include_dirs': np.get_include()})

import traversing_cython_impl as cy_impl

df = pd.read_csv("../sec1-intro/yellow_tripdata_2020-01.csv.gz")
df = df[(df.total_amount != 0)]
df_total = df["total_amount"].to_numpy() ← Получаем доступ к представлениям NumPy объектов Series
df_tip = df["tip_amount"].to_numpy()
get_tip_mean_cython = cy_impl.get_tip_mean_cython ← Вызываем реализацию функции на Cython
```

Код на Cython можно найти в файле 07-pandas/sec3-numpy-numexpr-cython/traversing_cython_impl.pyx:

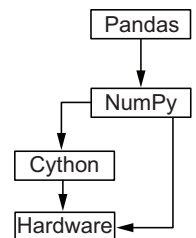


Рис. 7.1. pandas использует Cython посредством NumPy

```

import numpy as np
import cython
import numpy as cnp

@cython.boundscheck(False)
@cython.nonecheck(False)
@cython.wraparound(False)
@cython.cdivision(True)

cdef cnp.float64_t get_tip_mean_cython_impl(
    cnp.float64_t[:] df_total,
    cnp.float64_t[:] df_tip)
nogil:
    cdef cnp.float64_t frac_tip
    cdef int array_size = df_total.shape[0]
    cdef cnp.float64_t result = 0
    for i in range(array_size):
        result += df_tip[i] / df_total[i]
    return result / array_size

def get_tip_mean_cython(df_total, df_tip):
    return get_tip_mean_cython_impl(df_total, df_tip)

```

Доступ к функциям поддержки Cython

Доступ к функциям NumPy на уровне языка C

Отключаем проверку на границы, None и циклические переходы индексов в массивах

Используем операцию деления из языка C без дополнительных проверок. Это должна быть единственная новая для вас концепция

Полностью избавившись от всех зависимостей от чистого Python, мы можем указать функции не использовать ограничения GIL

Используем определение функции языка C (cdef) и соответствующий тип возвращаемого значения (cnp.float64_t). Это позволит коду на Cython избавиться от накладных расходов языка Python

Для всех переменных мы используем типы данных Cython

Входные параметры прописываем как представления памяти (memoryviews) из 64-битных чисел с плавающей запятой, которые работают значительно быстрее по сравнению с объектами Python и даже массивами NumPy

Деление выполняем в самом конце, что значительно более эффективно

У нас есть функция-мост, которую можно вызвать из Python. В ней массивы NumPy будут неявным образом преобразовываться в представления памяти

Весь приведенный код сопровождается многословными комментариями, хотя все это мы уже проходили в главе, посвященной Cython. Единственной новинкой для вас здесь является аннотация `cdivision`, которая приведет к тому, что при делении на ноль не будет возникать ошибка на уровне Python, что добавит коду быстродействия. Таким образом, при появлении нуля в знаменателе программа будет завершаться аварийно. Однако, если вы помните, мы тщательно проверили наш исходный набор данных на нулевые суммы еще в Python, так что для нашего случая это вполне приемлемое ограничение.

При изменении кода не забывайте выполнять команду `cython -a` для создания отчета HTML о потенциальных взаимодействиях с интерпретатором Python, которые могут стать камнем преткновения в плане быстродействия. Кстати, а как у нас с этим самым быстродействием? 8,51 мс – абсолютный чемпион!

Вывод

Поскольку библиотека pandas основана на NumPy, при ее использовании мы неявным образом работаем с NumPy, сами того не зная.

Но если вы хотите повысить быстродействие вычислений, вы всегда можете напрямую воспользоваться структурами данных NumPy. Также для увеличения производительности можно прибегнуть к услугам расширения Cython, способного оптимизировать работу структур NumPy при работе с pandas. Если вам приходится сталкиваться с обработкой больших массивов данных с помощью сложных алгоритмов, можете взглянуть в сторону библиотеки NumExpr. Из-за большого количества факторов, включающих программное и аппаратное обеспечение, характер данных, а также долго- и краткосрочные цели, невозможно провести универсальное сравнение всех приведенных здесь методов. Что идеально годится для моей задачи и моих вводных данных, может совершенно не подойти в вашем случае. Только умение выбирать поможет вам остановиться на оптимальном для вашей ситуации решении.

В следующем разделе мы рассмотрим еще один способ оптимизации библиотеки pandas. Воспользуемся программной платформой Apache Arrow для ускорения некоторых распространенных операций, таких как чтение данных с диска.

7.4. Чтение данных в pandas с помощью Arrow

В этом разделе мы попробуем ускорить процесс загрузки данных в датафрейм pandas с использованием программной платформы Apache Arrow. Но перед этим позвольте сделать шаг назад и взглянуть на проблематику взаимодействия pandas и Arrow в целом.

ПРИМЕЧАНИЕ. В этой главе мы лишь приведем пример совместной работы pandas и Arrow, а не будем вдаваться в подробности работы этой платформы (хотя в конце раздела все же проведем небольшую аналитику). За подробностями о технологии Apache Arrow вы можете обратиться на официальный сайт <https://arrow.apache.org>.

7.4.1. Взаимодействие между pandas и Apache Arrow

Apache Arrow представляет собой языково-независимый формат хранения колоночных данных в памяти. Независимость от языка означает, что Apache Arrow никоим образом не привязан ни к Python/pandas, ни к какому другому стеку технологий. По своей сути эта платформа состоит из набора библиотек, предназначенных для выполнения базовых операций в таких низкоуровневых языках программирования, как C, Rust или Go, хотя встречаются и некоторые реализации с использованием JavaScript. Для медленных языков предусмотрены обертки, включающие быстрые реализации. Например, реализация Arrow на Python – это всего лишь обертка для реализации на языке C++.

В данном разделе мы будем рассматривать Arrow в качестве дополнения к pandas: в частности, от Arrow нам необходимо будет

получить ускорение некоторых базовых функций pandas, а не его полноценную замену. Возможно, в будущем, когда аналитическая база Arrow значительно вырастет, эта платформа сможет конкурировать с библиотекой pandas и рассматриваться в качестве ее замены, но пока это время не настало.

В этом разделе мы заменим с помощью Arrow только механизм загрузки данных в pandas, а конкретно чтение из файла CSV, а также вскользь коснемся аналитики, доступной в Arrow. В следующем разделе затронем тему межпроцессного взаимодействия (Interprocess Communication – IPC) с использованием сервера IPC Plasma. В обоих случаях нашей целью будет выяснить, сможем ли мы добиться прироста производительности.

Платформа Arrow предлагает обширный функционал. В следующей главе мы подробнее коснемся вопросов хранения данных и рассмотрим разные форматы файлов и их влияние на производительность. Arrow также умеет взаимодействовать с разными серверными хранилищами. Кроме того, в этой платформе предусмотрен функционал удаленных вызовов процедур (remote procedure calls – RPC) для передачи данных между компьютерами. Обзор архитектуры Arrow в ее текущем виде представлен на рис. 7.2.

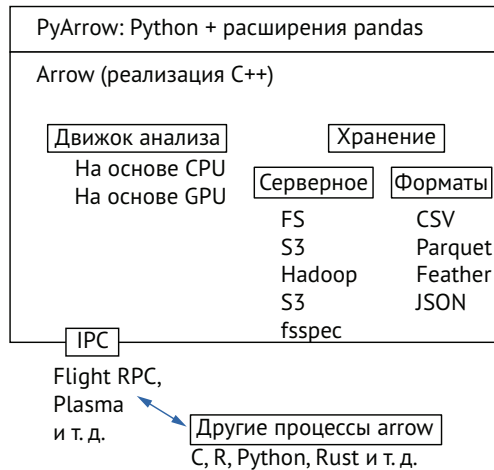


Рис. 7.2. Внутренняя архитектура PyArrow

В Python присутствует обертка для реализации Arrow на C. Часть на языке C состоит из нескольких компонентов, главным из которых является аналитический движок, способный в своих вычислениях использовать ресурсы графического процессора и поддерживающий такие серверные хранилища, как Amazon S3 и Hadoop, и различные форматы файлов, включая CSV, Parquet и JSON. Наконец, эта реализация предлагает функционал для межпроцессного и межмашинного взаимодействия для обеспечения эффективной

коммуникации с процессами, которые могут быть запущены на других серверах. Наиболее важно то, что формат хранения данных, реализованный в Arrow, не зависит от конкретного языка программирования или архитектуры аппаратных средств.

Теперь давайте посмотрим, как Arrow проявит себя при чтении данных в сравнении с pandas.

7.4.2. Чтение из файла CSV

В первом разделе этой главы мы использовали pandas для чтения данных о поездках в нью-йоркских такси из файла CSV. Чтение из файлов – одна из областей, которая была модернизирована в Apache Arrow по сравнению с pandas. Здесь используется многопоточный движок и более продвинутая система вывода типов данных. Как мы уже говорили ранее, технологии сегодня так быстро развиваются, что на момент выхода книги pandas, и Arrow могут сильно измениться, и взаимодействие между ними – тоже. Пока что платформа Apache Arrow предлагает более современную и гибкую архитектуру, а с учетом того, сколько приложений уже написано с использованием традиционных методов pandas, трудно себе представить, что эта библиотека вдрут сделает какой-то качественный скачок.

Итак, прочитаем тот же файл CSV с помощью Arrow и посмотрим, сколько места в памяти займут загруженные данные. Представленный ниже код можно найти в файле 07-pandas/sec4-arrow-intro/read_csv.py:

```
from pyarrow import csv  <— Импортируем обработчик файлов
                           CSV из модуля PyArrow

table = csv.read_csv("../sec1-intro/yellow_tripdata_2020-01.csv.gz")

tot_bytes = 0
for name in table.column_names:  <— Проходим по всем столбцам для опре-
    col_bytes = table[name].nbytes      деления их типа и объема в памяти
    col_type = table[name].type
    print(name, col_bytes // (1024 ** 2))
    tot_bytes += col_bytes
print("Total", tot_bytes // (1024 ** 2))
```

Время загрузки, по сравнению с pandas, на моем компьютере снизилось в шесть раз: с 12 до 2 с. Ниже приведен сокращенный вывод:

```
VendorID int64 48
tpep_pickup_datetime timestamp[s] 48
passenger_count int64 48
trip_distance double 48
store_and_fwd_flag string 34
total_amount double 48
Total 865
```


Без применения всякой оптимизации Arrow сумел уместить загруженные данные в 865 Мб в сравнении с 2 Гб в pandas. Если помочь pandas с определением типов данных, можно добиться сокращения объема данных до 250 Мб, но сравнивать этот результат с базовой загрузкой в Arrow будет несправедливо. Как же Arrow это удастся?

Без особого знания характера данных автоматическая система определения типов неплохо справляется со своей задачей. Хотя могла бы и лучше. В поле VendorID присутствует всего три значения: 1, 2 и null, так что для него подходящим типом данных мог быть int8, который уместит все нужные значения. Что касается использования типов double для полей с типом float, Arrow просто не берет на себя смелость предположить, что нам хватит и меньшей точности данных.

ПРИМЕЧАНИЕ. Типы данных в Arrow отличаются от типов в Python и NumPy/pandas, хотя преобразование между ними выполняется достаточно легко. И хотя с точки зрения программного кода тут все просто, существуют важные отличия в представлении разных типов.

Вы могли заметить, что поле VendorID, в котором присутствуют значения null, получило целочисленный тип данных. В pandas/NumPy это было бы невозможно без предварительного преобразования значений NA. Arrow воспринимает пропущенные значения совершенно иначе в сравнении с NumPy/pandas: в нем выделяется дополнительный массив бит с одним значением для каждой строки, указывающим на то, является ли значение пропущенным. Таким образом, в обмен на скромную для большинства типов прибавку к необходимой для хранения данных памяти мы получаем возможность сохранить пропущенные значения в полях. В результате целые числа оказываются представлены подходящим для них целочисленным типом, несмотря на наличие пустых значений.

Как видите, с помощью такого представления пропущенных значений в Arrow мы можем существенно снизить требования к выделению памяти для многих колонок, но в некоторых случаях нам нужно явно уведомить Arrow о своих намерениях. В PyArrow это можно сделать при помощи класса *ConvertOptions* следующим образом:

```
convert_options = csv.ConvertOptions(
    column_types = {
        "VendorID": pa.bool_()
    },
    true_values=["Y", "1"],
    false_values=["N", "2"])
```

Arrow может вывести тип данных для поля store_and_fwd_flag как булев на основе значений true_values и false_values, но для числового поля VendorID ему необходимо явно указать требуемый тип

Мы оповещаем Arrow о том, что значения Y и 1 должны быть сконвертированы в true. Это правило будет использоваться только для колонок с булевым типом. Для значений false мы также предоставляем соответствия

```
table = csv.read_csv(  
    "../sec1-intro/yellow_tripdata_2020-01.csv.gz",  
    convert_options=convert_options  
)  
  
print(  
    table["store_and_fwd_flag"].unique(),  
    table["store_and_fwd_flag"].nbytes // (1024 ** 2),  
    table["store_and_fwd_flag"].nbytes // 1024  
)
```

← Передаем обработчику файлов CSV
опции преобразования

Поле `VendorID` технически не является булевым, но, поскольку в нем содержится лишь два значения, мы можем сделать его логическим для экономии места.

Поскольку пропущенные значения в Arrow обрабатываются отдельно, расход памяти для колонок с небольшим количеством возможных значений с пропусками будет минимальным. Идеальный случай – два возможных значения с преобразованием в булев тип данных.

Для поля `store_and_fwd_flag` мы в коде вывели все уникальные значения (как и ожидалось, их всего три) и объем занимаемой памяти сначала в Мб, а затем в Кб. Получилось всего 790 Кб.

К сожалению, с целью проведения анализа данных нам необходимо конвертировать полученную таблицу из Arrow в pandas, поскольку их внутренние форматы существенно отличаются. При выполнении этой операции мы потеряем время и израсходуем некоторый объем памяти:

```
table_df = table.to_pandas()
```

Как мы и предполагали, датафрейм в pandas потребовал для хранения больше памяти. То же поле `store_and_fwd_flag` получило объектный тип данных, несмотря на все наши усилия по его преобразованию в Arrow.

На первый взгляд кажется, что все улучшения, которых мы добились за счет использования Arrow, были безвозвратно потеряны. Но это не так. Проблемы у нас две: время, требуемое для преобразования данных, и занимаемая память. И обе они решаемые.

Время, необходимое для конвертации таблицы из Arrow в pandas, составляет малую долю нашего выигрыша от перехода к загрузке данных при помощи Arrow. Как вы помните, в pandas это время составляло порядка 12 с, а в Arrow – всего 2 с. При этом время, необходимое для преобразования загруженных данных из одного формата в другой, составило на моем компьютере всего 23 мс, что несопоставимо с полученной выгодой.

Что касается проблем с памятью, они существуют, и в какой-то момент с текущим подходом будет выделено вдвое больше памяти, чем

необходимо. К счастью, в Arrow предусмотрено решение этой ситуации при помощи параметра *self_destruct*, позволяющего уничтожить структуру данных Arrow во время преобразования. Это не приведет к перерасходу памяти, а заплатить придется потерей данных в формате Arrow:

```
mission_impossible = table.to_pandas(self_destruct=True)
```

Вывод

Как показал этот пример, библиотека Arrow предлагает современный и эффективный функционал, позволяющий сэкономить как время, так и память в сравнении с pandas. На текущий момент Arrow значительно уступает pandas в аналитической мощи, так что эти библиотеки желательно использовать совместно. Несмотря на это, давайте потратим пару минут и посмотрим, как осуществляется анализ данных в Arrow. Возможно, в будущем такой подход будет востребован.

7.4.3. Анализ данных в Arrow

Поскольку эта книга посвящена тому, как вы прямо сегодня можете повысить эффективность своего кода в Python, нас в основном интересует применение библиотеки Arrow на службе у pandas. Но дело в том, что Arrow умеет и самостоятельно анализировать данные. И хотя разнообразием аналитических приемов эта библиотека пока похвастаться не может, особенно в сравнении с pandas, с течением времени ее потенциал в этой области существенно возрастет.

Давайте вернемся к расчету доли чаевых в общей сумме оплат за такси в Нью-Йорке и попробуем обойтись одним лишь Arrow:

```
import pyarrow.compute as pc

t0 = table.filter(
    pc.not_equal(table["total_amount"], 0.0))

pc.mean(pc.divide(t0["tip_amount"], t0["total_amount"]))
```

Чисто архитектурно этот код, как и любой код преобразования данных в Arrow, сильно отличается от pandas. Первое, что стоит отметить, – это то, что адресация стала более низкоуровневой: к примеру, если попытаться выполнить здесь операцию *not_equal* с целочисленным аргументом (0) вместо числа с плавающей запятой (0.0), вы получите ошибку несоответствия типов данных. Второе отличие состоит в том, что интерфейс является гораздо более функциональным по сравнению с объектно ориентированным. Обратите внимание, что здесь мы вызываем функции верх-

него уровня с массивами, передаваемыми в качестве параметров, а не методы самих массивов данных. Наконец, система обработки ошибок в Arrow основана на кодах ошибок, а не на исключениях. В том, чтобы сохранять минималистичный подход и сопоставлять лежащие в основе Arrow библиотеки на языке C с как можно меньшим количеством инструкций Python, есть своя выгода, и главным преимуществом здесь является скорость выполнения кода.

На моем компьютере время вычисления доли чаевых составило порядка 15 мс, что практически вдвое быстрее по сравнению с эквивалентной версией расчета в pandas (`get_tip_mean_vector`).

Мы увидели, как эффективно Arrow может загружать данные в pandas. Но есть и другой способ совместного использования этих библиотек, и мы поговорим о нем далее. Давайте посмотрим, как Arrow может помочь в плане взаимодействия разных языков. Помните при этом, что эта библиотека предлагает одинаковый формат для разных реализаций.

7.5. Использование механизма взаимодействий в Arrow для делегирования задач более эффективным языкам и системам

Одним из преимуществ Arrow является наличие стандартного механизма хранения данных в памяти, что позволяет совместно использовать представление структуры данных в разных реализациях и разных языках. При этом совместное использование осуществляется с нулевым копированием, что способствует очень эффективной передаче структур данных. В этом разделе мы узнаем, что делает архитектуру Arrow наиболее эффективной в сравнении с конкурентами. Мы также реализуем пример с использованием сервера Plasma, входящего в состав Arrow, демонстрирующий межпроцессное взаимодействие.

Главной целью этого раздела будет демонстрация эффективного взаимодействия процессов при помощи Arrow. С учетом того, что сервер Plasma находится в состоянии разработки, а вы можете самостоятельно реализовать совместное использование памяти между процессами, воспринимайте все сказанное здесь больше как иллюстрацию некоего шаблона для разработки. При этом код из этого раздела будет полностью рабочим и функциональным.

7.5.1. Предпосылки архитектуры межязыкового взаимодействия Arrow

Представьте себе сценарий, в котором большую часть обработки данных вы производите на Python, но для выполнения специфической операции вам необходимо воспользоваться функционалом языка R.

Существует множество способов осуществить такое межпроцессное и межязыковое взаимодействие, но мы для примера рассмотрим два подхода без участия библиотеки Arrow и два – с ее участием. Эти подходы проиллюстрированы на рис. 7.3 и описаны ниже:

- первый подход связан с записью наших данных в формате обычного файла (например, CSV) в Python и чтением его в R. Это будет довольно эффективно с точки зрения расходования памяти, но времени потребуется довольно много по причине использования диска для записи;
- второй подход предполагает использование пакета *rpy2*¹ с конвертацией датафреймов между pandas и R. Это потребует вдвое больших затрат как в отношении памяти, так и в плане затрачиваемого времени. На самом деле здесь все еще хуже, как мы увидим совсем скоро;
- третий подход состоит в том, что вы преобразовываете датафрейм pandas в формат Arrow, после чего передаете полученные данные в R, извлекаете таблицу из формата Arrow и обрабатываете ее. Это требует времени для выполнения двух преобразований (из pandas в Arrow и из Arrow в R). Расходование памяти в этом случае зависит от того, будете ли вы уничтожать исходные данные в процессе преобразования. Если да, то дополнительная память не потребуется. В противном случае вам придется выделить еще столько же памяти, сколько занимает исходный набор данных;

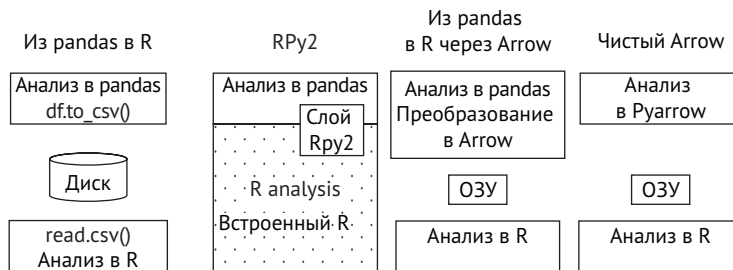


Рис. 7.3. Альтернативы для межязыкового взаимодействия Python и R

- наконец, лучший подход с использованием Arrow для тех случаев, когда вы выполняете обработку в Python и R, предполагает использование только Arrow, без pandas вовсе. В этом случае вы будете передавать лишь указатель на память, что не требует от вас никаких дополнительных расходов.

¹ Если вы также используете R – а в анализе данных они с Python часто применяются совместно, – вам стоит рассмотреть вариант интеграции этих языков при помощи пакета *rpy2*. Он позволит встроить процесс R в Python и обеспечит элегантный доступ к примитивам для коммуникации между Python и R.

На первый взгляд может показаться, что второй подход, связанный с конвертацией датафреймов между pandas и R, должен быть более эффективным по сравнению с третьим – с двумя преобразованиями (из pandas в Arrow и из Arrow в R). Но это не так сразу по трем причинам:

- формат Arrow в памяти позволяет делиться данными между разными реализациями Arrow вне зависимости от языка. Таким образом, передача структуры данных Arrow сводится к передаче указателя на память;
- при преобразовании данных из pandas в R сам механизм конвертации работает только на одной стороне (в Python или в R), а значит, на противоположной платформе, которая не является для него родной, он может функционировать очень неэффективно;
- конвертеры Arrow реализованы на языках низкого уровня C/C++ в многопоточном стиле и изначально нацелены на максимальную эффективность. Мы уже затрагивали эту тему в предыдущем разделе, когда говорили о чтении данных из CSV.

Есть и еще одна важная причина, состоящая в том, что в сложных системах вам понадобится использовать больше конвертеров. К примеру, если вы используете pandas, Java, R и Rust, вам понадобятся конвертеры pandas/Java, pandas/R, Java/R, Java/Rust, pandas/Rust и R/Rust. При использовании Arrow в качестве промежуточного формата количество конвертеров снизится до четырех: pandas/Arrow, Java/Arrow, R/Arrow и Rust/Arrow. Если платформ будет больше, разница станет еще более очевидной.

В этой книге мы не будем касаться других языков программирования, помимо Python, но вы можете сами поэкспериментировать с ними и произвести замер производительности каждого из способов.

Далее мы на примере Python покажем, как можно эффективно обмениваться данными между процессами. В реальных сценариях один или более компонентов в этой схеме будут реализованы с использованием другого языка программирования.

7.5.2. **Операции с нулевым копированием с использованием сервера Plasma от Arrow**

Давайте вернемся к нашему старому доброму набору данных о нью-йоркских желтых такси и выполним какую-нибудь сквозную аналитику. Мы разделим работу на три процесса: один будет считывать данные и отправлять их на обработку, второй, собственно, будет заниматься обработкой, а третий – извлекать и отображать результат. Есть две главные причины для использования такой архитектуры. Первая состоит в том, что нужная вам реализация алгоритма может находиться в отдельном процессе, который никак не может быть напрямую связан с Python. Вторая связана с тем, что сложный

код, в котором выполняются вычисления, желательно держать отдельно от аналитики.

Библиотека Arrow предоставляет сервер *Plasma*, в обязанности которого входит управление распределенной памятью. Он позволяет регистрировать, считывать и записывать объекты, а также обеспечивает механизм для отслеживания существующих в памяти объектов. Наличие такого сервера значительно облегчает взаимодействие между разными процессами. Сервер Plasma является локальным, и это означает, что он не может быть доступен по сети, а обращение к нему выполняется только через локальный сокет. Основные функции сервера Plasma сводятся к облегчению доступа к распределенной памяти, нахождению и предоставлению доступа к существующим объектам и обеспечению доступа к общей памяти для процессов, не пересекающихся во времени, – например, если процесс-потребитель был запущен уже после того, как процесс-поставщик был остановлен.

Первое, что нам необходимо сделать, – это запустить сервер Plasma:

```
plasma_store -s /tmp/fast_python -m 1000000000
```

Мы будем использовать сокет UNIX /tmp/fast_python в качестве основы для межпроцессного взаимодействия. При этом мы выделили 1 Гб памяти для общего использования.

Наш первый процесс будет отвечать за чтение из файла CSV и записи на сервер Plasma: для этого мы подключаемся к сокету Plasma socket, читаем данные при помощи Arrow и размещаем их в Plasma. Исходный код можно найти в файле 07-pandas/sec5-arrow-plasma/load_csv.py:

```
import os
import sys

import pyarrow as pa
from pyarrow import csv
import pyarrow.plasma as plasma

csv_name = sys.argv[1]
client = plasma.connect("/tmp/fast_python")

convert_options = csv.ConvertOptions(
    column_types={
        "VendorID": pa.bool_()
    },
    true_values=["Y", "1"],
    false_values=["N", "2"])

table = csv.read_csv(
    csv_name
    convert_options=convert_options
)
```

Подключаемся к серверу Plasma с помощью сокета

Предполагаем, что исходные данные находятся в формате поездок такси от компании NYC


```

pid = os.getpid()

plid = plasma.ObjectID(
    f"csv-{pid}".ljust(20, " ").encode("us-ascii"))
client.put(table, plid)

```

Создаем идентификатор для нашей таблицы

Помещаем объект в Plasma

При размещении объекта на сервере Plasma мы должны дать ему идентификатор, т. е. как-то озаглавить его. Мы будем использовать имя, которое будет начинаться с `csv-`, после чего будет следовать ID нашего процесса. Для наших целей такой принцип именования объектов вполне сойдет, вы же можете реализовать любой алгоритм именования, который вас удовлетворит и обеспечит гарантию того, что имена не будут пересекаться. Сервер Plasma требует, чтобы идентификатор был представлен в формате из 20 байт, так что мы просто добились пробелами лишние знаки и закодировали идентификатор с помощью кодека US-ASCII, который возвращает список байтов. Вы можете использовать любой кодек по вашему усмотрению при условии, что он каждый символ конвертирует в 1 байт, – иначе вы получите слишком длинный массив байтов.

Для поиска и извлечения наших таблиц мы воспользовались идентификатором объекта. Существуют и более продвинутые техники с использованием метаданных и прочего, но для нашего простого примера будет вполне достаточно идентификатора.

ПРЕДУПРЕЖДЕНИЕ. Если у вас закончится вся выделенная память, сервер Plasma начнет избавляться от старых объектов.

Перед реализацией двух других процессов давайте напишем вспомогательный скрипт, который будет перечислять все объекты CSV, находящиеся в Plasma. Это позволит нам следить за содержимым хранилища. Также мы будем мониторить файлы с результатами, которые создадим позже. Их имена будут начинаться с префикса `result-`. Представленный здесь код можно найти в файле `07-pandas/sec5-arrow-plasma/list_csvs.py`:

```

import pyarrow as pa
import pyarrow.plasma as plasma

client = plasma.connect("/tmp/fast_python")

all_objects = client.list()

for plid, keys in all_objects.items():
    try:
        plid_str = plid.binary().decode("us-ascii")
    except UnicodeDecodeError:
        continue

```

Извлекаем список всех объектов

В результате декодирования идентификаторов могут возвращаться невалидные строки, поэтому мы должны перехватывать соответствующие исключения


```

if plid_str.startswith("csv-"):
    print(plid_str, plid)
    print(keys)
elif plid_str.startswith("result-"):
    print(plid_str, plid)
    print(keys)

```

После получения списка всех сохраненных объектов в виде словаря мы осуществляем поиск объектов, идентификаторы которых начинаются с `csv-` или `result-`. Поскольку не все идентификаторы сохраненных объектов могут быть преобразованы в строку (с помощью сервера Plasma можно делиться и другими объектами), мы позаботились о перехвате соответствующего исключения, в результате которого просто переходим к следующему объекту, поскольку никакой ошибки здесь нет.

Для всех объектов, удовлетворяющих нашим критериям, мы распечатываем декодированный идентификатор, исходный и некоторые связанные метаданные. Ниже приведен пример вывода:

```

csv-579123      ObjectID(6373762d35373931323320202020202020202020)
{'data_size': 822037944, 'metadata_size': 0, 'ref_count': 0,
 'create_time': 1616361341, 'construct_duration': 0,
 'state': 'sealed'}

```

Теперь давайте реализуем наш вычислительный сервер. В его основе будет бесконечный цикл, занятый поиском объектов с идентификаторами, начинающимися с `csv-`. Если очередной объект найден, а файла с результатом для него еще нет, мы производим вычисление и результат помещаем обратно в Plasma в файле с тем же идентификатором, но с префиксом `result-`. Код можно найти в файле `07-pandas/sec5-arrow-plasma/compute_stats.py`:

```

import time

import pandas as pd
import pyarrow as pa
from pyarrow import csv
import pyarrow.compute as pc
import pyarrow.plasma as plasma

client = plasma.connect("/tmp/fast_python")
while True:
    client = plasma.connect("/tmp/fast_python")
    all_objects = client.list()

    for plid, keys in all_objects.items():
        plid_str = ""
        try:
            plid_str = plid.binary().decode("us-ascii")

```

```

except UnicodeDecodeError:
    continue
if plid_str.startswith("csv-"):
    original_pid = plid_str[4:]
    result_plid = plasma.ObjectID(
        f"result-{original_pid}".ljust(
            20, " ")[:20].encode("us-ascii"))
    if client.contains(result_plid):
        continue
    print(f"Working on: {plid_str}")
    table = client.get(plid)
    t0 = table.filter(pc.not_equal(table["total_amount"], 0.0))
    my_mean = pc.mean(pc.divide(t0["tip_amount"], t0["total_
amount"])).as_py()
    result_plid = plasma.ObjectID(
        f"result-{original_pid}".ljust(20, " "[:20]
        .encode("us-ascii"))
    client.put(my_mean, result_plid)
time.sleep(0.05)

```

Получаем таблицу из Plasma

Проверяем наличие результата

Помещаем результат в Plasma

Большинство конструкций, использованных в представленном коде, мы уже видели в этом и предыдущих разделах. Единственной концептуальной новинкой здесь можно назвать разве что применение метода *contains*, позволяющего перед использованием метода *get* для извлечения таблицы из Plasma проверить наличие результата по ней.

Наконец, давайте посмотрим на итоговые результаты. Код, приведенный ниже, можно найти в файле `07-pandas/sec5-arrow-plasma/show_results.py`:

```

import pyarrow as pa
import pyarrow.plasma as plasma

client = plasma.connect("/tmp/fast_python")

all_objects = client.list()

for plid, keys in all_objects.items():
    try:
        plid_str = plid.binary().decode("us-ascii")
    except UnicodeDecodeError:
        pass
    if plid_str.startswith("result-"):
        print(plid_str, client.get(plid, timeout_ms=0))

```

Ничего нового для вас в этом фрагменте кода нет. Отметим только, что в последней строке кода мы извлекаем объект без блокировки, передав ноль в качестве значения параметра `timeout_ms`. По умолчанию Plasma использует блокирующую семантику, но вы всегда можете задать нужный вам тайм-аут.

О сервере Plasma можно рассказывать долго – например, об использовании низкоуровневых API для извлечения и помещения объектов или об эффективной передаче объектов pandas. Но, поскольку архитектура Arrow/Plasma постоянно развивается, нам важнее было остановить свое внимание здесь именно на теме межпроцессных взаимодействий.

Мы еще вернемся к разговору об Arrow в следующей главе, когда будем говорить о концепции хранения данных. На этом фронте Arrow тоже есть что нам предложить.

Заключение

- pandas является наиболее распространенной библиотекой для анализа данных в мире Python, но, как бы хороша и удобна она ни была, эффективности в отношении времени вычислений и хранения данных в памяти ей не хватает.
- Простые техники загрузки данных могут оказаться весьма неэффективными в плане расхода памяти – например, если оставить в списке колонки те, которые вам не нужны при выполнении анализа или не задать типы данных для колонок непосредственно во время загрузки.
- Разумное использование индексирования способно снизить время обработки данных, хотя гибкость этой техники в pandas оставляет желать лучшего.
- Разные подходы к выполнению итераций по строкам в датафрейме могут отличаться по скорости более чем на два порядка. Избегайте простых циклов всегда, когда это возможно, отдавая предпочтение векторизованным операциям.
- Несмотря на то что библиотека pandas основана на NumPy, иногда для повышения эффективности операций можно и нужно обращаться к структурам данных NumPy, извлеченным из датафрейма pandas, напрямую.
- Расширение Cython может быть использовано совместно с pandas, хоть и опосредованно через структуры данных NumPy. В целом эта техника может существенно повлиять на эффективность кода.
- При работе с объемными наборами данных, над которыми требуется выполнить сложные вычисления, можно добиться серьезного прогресса в эффективности при помощи библиотеки NumExpr.
- Платформа Apache Arrow может использоваться для решения самых разных задач. В этой главе мы узнали о том, как ее можно применять совместно с pandas, в частности для эффектив-

ного чтения данных. Но и в других областях Apache Arrow бывает незаменима.

- Архитектура Arrow может быть использована для эффективной передачи данных между различными процессами в рамках одной машины посредством сервера Plasma. Эта возможность может оказаться очень удобной при использовании одновременно нескольких языков и фреймворков, поскольку данная технология абсолютно не зависит от окружения.

8

Хранение больших данных

В этой главе мы обсудим следующие темы:

- знакомство с `fsspec` – библиотекой абстракций над файловыми системами;
- эффективное хранение неоднородных колоночных данных с помощью `Parquet`;
- обработка файлов данных в памяти при помощи библиотек `pandas` или `Parquet`;
- обработка однородных многомерных массивов данных с помощью библиотеки `Zarr`.

При работе с большими данными во главу угла ставится вопрос их хранения. Нам необходимо максимально быстро, насколько это возможно, взаимодействовать с данными (читать и записывать), при этом предпочтительно иметь возможность делать это параллельно из разных процессов. Вместе с тем нам бы хотелось, чтобы данные занимали как можно меньше места, поскольку хранить действительно большие объемы может быть накладно.

В этой главе мы рассмотрим разные подходы к эффективному хранению больших данных. Начнем с короткой дискуссии о биб-

библиотеке *fsspec*, служащей в качестве абстракции для файловых систем, как локальных, так и удаленных. И хотя сама библиотека *fsspec* напрямую не влияет на производительность, многие приложения ее используют для доступа к хранилищам при построении эффективных реализаций хранения данных.

После этого мы поговорим о формате файлов Parquet, используемом для хранения неоднородных колоночных наборов данных. Формат Parquet поддерживается в Python посредством платформы Apache Arrow, о которой мы говорили в предыдущей главе.

Далее обсудим концепцию секционного чтения очень больших наборов данных, иногда называемую *подходом с использованием внешней памяти* (out-of-core approach). Зачастую мы вынуждены иметь дело с наборами данных, которые не могут быть обработаны в памяти целиком. Секционное чтение позволяет обрабатывать данные частями с помощью уже известных вам библиотек. Это довольно простое, но весьма эффективное решение. В нашем примере мы возьмем объемный датафрейм *pandas* и преобразуем его в файл Parquet. В заключение главы познакомимся с *Zarr* – современным форматом данных и библиотекой для хранения однородных многомерных массивов, таких как массивы *NumPy*, в постоянной памяти.

Для проверки примеров из этой главы вам будет необходимо установить библиотеки *fsspec*, *Zarr* и *Arrow*, предоставляющую доступ к интерфейсу Parquet. Если вы используете *conda*, то для установки достаточно будет запустить команду `conda install fsspec zarr pyarrow`. Docker-образ `tiagoantao/python-performance-dask` включает в себя все необходимые библиотеки. Давайте начнем с краткого обзора библиотеки *fsspec*, позволяющей взаимодействовать с разными типами файловых систем, как локальными, так и удаленными, с использованием одного и того же API.

8.1. Универсальный интерфейс для доступа к файлам: *fsspec*

Существует огромное множество систем для хранения файлов, начиная от традиционных локальных файловых систем и заканчивая облачными хранилищами вроде Amazon S3 и протоколами наподобие SFTP и SMB (файловый обменник в Windows). Этот список очень велик, особенно если включать в него объекты, подобные файловым системам. Например, *zip*-файл представляет собой, по сути, контейнер для файлов и директорий, сервер HTTP предполагает наличие дерева файлов и т. д.

Если работать со всеми файловыми системами по отдельности, придется изучать огромное количество разных API, что очень долго и муторно. Лучше изучить одну библиотеку *fsspec*, представляющую собой некую абстракцию над файловыми системами разных

типов при помощи одного универсального API. Таким образом, вам достаточно будет изучить работу с одним API, чтобы взаимодействовать с разными типами файловых систем. Но есть и нюансы вроде того, что вы не можете ожидать от локальной файловой системы в точности такого же поведения, как от удаленной. В то же время эта библиотека нивелирует эти различия и значительно облегчает доступ к различным файловым системам с минимальными накладными расходами.

8.1.1. Использование *fsspec* для поиска файлов в репозитории *GitHub*

Для демонстрации принципов работы библиотеки *fsspec* давайте используем ее для доступа к репозиторию *GitHub*, поиска *zip*-файлов и определения того, содержат ли эти архивы файлы *CSV*. В данном примере мы будем воспринимать репозиторий *GitHub* в качестве файловой системы. И это, в принципе, не так далеко от истины, ведь, если вдуматься, репозиторий *GitHub* является не чем иным, как деревом директорий с версионированным содержимым.

В качестве примера мы будем использовать репозиторий этой книги. В папке `08-persistence/sec1-fsspec` находится архив `dummy.zip`, содержащий два простых файла *CSV*. Мы в своем коде пройдем по репозиторию, найдем все *zip*-файлы – в нашем случае он будет один, – откроем их и с помощью функции `describe` из библиотеки *pandas* проанализируем содержимое хранящихся в архивах файлов *CSV*.

Давайте начнем с осуществления доступа к нашему репозиторию и получения списка файлов в корневой директории:

```
from fsspec.implementations.github import GithubFileSystem

git_user = "tiagoantao"
git_repo = "python-performance"

fs = GithubFileSystem(git_user, git_repo)
print(fs.ls(""))
```

Здесь мы импортировали класс *GithubFileSystem*, передали в него имя пользователя и название репозитория и вывели на экран содержимое верхнеуровневой директории. Обратите внимание, что к корневой папке мы здесь обращаемся с помощью пустой строки, а не традиционного для таких случаев слеша (`/`). Библиотека *fsspec* также содержит множество других классов для доступа к различным хранилищам, таким как локальная файловая система, сжатая файловая система, *Amazon S3*, *Argow*, *HTTP*, *SFTP* и т. д.

У объекта *fs* есть несколько методов, общих с интерфейсами файловой системы *Python*. Например, для прохождения по файло-

вой системе, что необходимо для поиска архивов, можно воспользоваться методом *walk*, очень напоминающим одноименный метод из модуля *os*:

```
def get_zip_list(fs, root_path=""):
    for root, dirs, fnames in fs.walk(root_path):
        for fname in fnames:
            if fname.endswith(".zip"):
                yield f"{root}/{fname}"
```

Таким образом, мы создали генератор *get_zip_list*, выдающий полные пути к архивным файлам. Обратите внимание, что этот код практически идентичен тому, в котором вы использовали бы метод *os.walk*, если бы в качестве атрибута *root_path* был передан слеш (/).

Ограничения интерфейса *fsspec*

Хотя библиотека *fsspec* предоставляет универсальный и простой доступ к разным типам файловых систем, она не может скрыть их семантические различия. В действительности иногда нам и не нужно, чтобы она их скрывала. На примере класса *GitHubFileSystem* можно выделить два аспекта, в которых проявляются подобные различия:

- дополнительный функционал. Внутри репозитория вы можете перемещаться по папкам и файлам с учетом временного аспекта, не ограничиваясь только текущим временем в основной ветке проекта (ветке *master*). Вы можете указать конкретную ветку (*branch*) или *тег* (*tag*) и просмотреть состояние репозитория на выбранный момент времени;
- ограничения. Вы будете не только испытывать характерные для удаленных файловых систем проблемы вроде неработающего кода из-за отсутствия интернет-соединения, но и сталкиваться с ограничениями скорости передачи данных при выполнении многократных запросов на сервер.

Теперь, когда мы подготовили механизм для извлечения архивов из репозитория, будем копировать их в локальную файловую систему. Идея состоит в том, что для поиска в этих архивах файлов *CSV* мы будем открывать их локально:

```
def get_zips(fs):
    for zip_name in get_zip_list(fs):
        fs.get_file(zip_name, "/tmp/dl.zip")
        yield zip_name
```

Далее нам необходимо просмотреть содержимое архивов. Для простоты можно воспользоваться встроенным в Python модулем *zipfile*, как показано ниже:


```

import zipfile
import pandas as pd

def describe_all_csvs_in_zips(fs):
    for zip_name in get_zips(fs):
        my_zip = zipfile.ZipFile("/tmp/dl.zip")
        for zip_info in my_zip.infolist():
            print(zip_name)
            if not zip_info.filename.endswith(".csv"):
                continue
            print(zip_info.filename)
            my_zip_open = zipfile.ZipFile("/tmp/dl.zip")
            df = pd.read_csv(zipfile.Path(my_zip_open, zip_info.
filename).open())
            print(df.describe())

```

Открываем файл с помощью модуля zipfile

Метод `infolist` специфичен для модуля `zipfile` и нуждается в дополнительном изучении

Обратите внимание, что для использования библиотеки `zipfile` нам придется разобраться с ее API. Мы начали с конструктора, после чего вызвали метод `infolist`, но пришлось повторно открывать файл из-за особенностей библиотеки `zipfile`.

8.1.2. Использование `fsspec` для поиска zip-файлов

Наш предыдущий код был простой демонстрацией того, от каких проблем вас может уберечь использование библиотеки `fsspec`. Эта библиотека предлагает интерфейс для доступа к zip-файлам, а значит, мы можем переписать наш предыдущий код следующим образом:

```

from fsspec.implementations.zip import ZipFileSystem

def describe_all_csvs_in_zips(fs):
    for zip_name in get_zips(fs):
        print(zip_name)
        my_zip = ZipFileSystem("/tmp/dl.zip")
        for fname in my_zip.find(""):
            if not fname.endswith(".csv"):
                continue
            print(fname)
            df = pd.read_csv(my_zip.open(fname))
            print(df.describe())

```

Метод `find`, как и все остальные, может быть использован применительно к любой файловой системе, а не только к zip

Как и метод `find`, метод `open` также доступен для всех типов файловых систем


Помимо создания объекта `ZipFileSystem`, этот код очень напоминает нашу реализацию механизма работы с репозиторием GitHub и очень близок к концепции работы с файлами в Python. Таким образом, нам больше нет необходимости отдельно изучать интерфейс библиотеки `zipfile`.

8.1.3. Доступ к файлам с использованием библиотеки `fsspec`

Также библиотека `fsspec` может быть использована для открытия файлов напрямую, хотя семантика ее применения значительно от-

личается от стандартной функции `open`. Например, для открытия zip-файла с помощью функции `open` из модуля *fsspec* вам необходимо выполнить следующий код:

```
d1f = fsspec.open("/tmp/dl.zip")
with d1f as f:
    zipf = zipfile.ZipFile(f)
    print(zipf.infolist())
d1f.close()
```



Для открытия файла мы должны использовать инструкцию `with`
Снова воспользуемся встроенным в Python модулем `zipfile` для анализа файла

Вывод будет следующим:

```
[
  <ZipInfo filename='dummy1.csv' filemode='-rw-rw-r--' file_size=22>,
  <ZipInfo filename='dummy2.csv' compress_type=deflate
    filemode='-rw-rw-r--' file_size=56 compress_size=54>
]
```

Обратите внимание на необходимость использования конструкции `with` после функции `open` для получения корректного файлового дескриптора, что отличается от методики применения стандартной функции `open`.

8.1.4. Использование цепочки URL для обращения к разным файловым системам

Давайте вернемся к нашему zip-файлу в репозитории GitHub. Поскольку сами архивы можно рассматривать как контейнеры для других файлов, по сути, к нашему zip-файлу можно относиться как к файловой системе внутри другой файловой системы. Библиотека *fsspec* предлагает нам декларативный и очень простой способ получения доступа к нашим данным с помощью так называемой *цепочки URL* (URL chaining). Иногда можно взять поток и интерпретировать его как файловую систему. Простой пример разъяснит ситуацию. Давайте распечатаем содержимое файла `dummy1.csv`:

```
d1f = fsspec.open("zip://dummy1.csv::/tmp/dl.zip", "rt")
with d1f as f:
    print(f.read())
```

Посмотрите на цепочку URL в действии: с помощью нее мы извлекли файл `dummy1.csv` из `/tmp/dl.zip`. При этом вам не понадобилось заранее распаковывать zip-файл, библиотека *fsspec* позаботилась об этом за вас.

Помните, что мы назвали нашу реализацию функции `get_zips` простой? Все дело в том, что нам нет необходимости явно загружать файл благодаря возможности использовать цепочки URL, как показано ниже:

```
d1f = fsspec.open("zip://dummy1.csv:github://tiagoantao:python-
performance@08-persistence/sec1-fsspec/dummy.zip")
```

```
with d1f as f:
    print(pd.read_csv(f))
```

Здесь мы вручную прописали всю цепочку URL целиком, чтобы вы лучше усвоили ее строение.

8.1.5. Замена реализации файловой системы

Поскольку библиотека `fsspec` служит для создания абстракции любого типа файловой системы, мы можем легко менять реализацию файловой системы на лету. К примеру, давайте изменим реализацию GitHub на локальную файловую систему следующим образом:

```
import os
from fsspec.implementations.local import LocalFileSystem

fs = LocalFileSystem()
os.chdir("../..")
```

Предполагается, что вы запускаете скрипт из папки `08-persistence/sec1-fsspec`, так что путь `../..` должен привести вас в корневую директорию книги в репозитории.

Мы просто использовали реализацию `LocalFileSystem` вместо `GitHubFileSystem`, вот и все. Поскольку наш исходный файл находится на два уровня ниже в иерархии, пришлось подниматься по дереву выше с помощью функции `chdir`. Теперь наш код работает в пространстве локальной файловой системы, а не GitHub. Вы можете проверить это, запустив инструкцию `describe_all_csvs_in_zips(fs)`.

8.1.6. Взаимодействие с PyArrow

Наконец, стоит упомянуть, что библиотека `PyArrow`, о которой мы говорили в предыдущей главе, может напрямую взаимодействовать с `fsspec`, как показано ниже:

```
from pyarrow import csv
from pyarrow.fs import PyFileSystem, FSSpecHandler

zfs = ZipFileSystem("/tmp/dl.zip")
arrow_fs = PyFileSystem(FSSpecHandler(zfs))
my_csv = csv.read_csv(arrow_fs.open_input_stream("dummy1.csv"))
```

Здесь важно отметить, что в библиотеке `Arrow` предусмотрена концепция файловой системы, что позволяет ей бесшовно интегрироваться с `fsspec`. Файловая система `Arrow` может взаимодействовать с `fsspec` посредством класса `pyarrow.fs.FSSpecHandler`. После

установления такого соответствия примитивы файловой системы Arrow могут напрямую использоваться поверх fsspec.

СОВЕТ. Библиотека fsspec поддерживает возможность частичной загрузки данных с удаленных серверов, что может быть очень полезно при работе с большими данными, где нам может понадобиться только часть объемного файла. Это возможно только в случае, если сервер, к которому мы обращаемся, поддерживает порционную загрузку данных. Например, в GitHub не реализована такая поддержка, а в S3 реализована. Вы можете воспользоваться этой возможностью путем активации кеша, передав функции `open` значение `readahead` в качестве параметра `cache_type`.

Это было небольшое отступление от общей канвы книги, поскольку библиотека fsspec напрямую не относится к теме производительности. Хотя она активно используется в библиотеках, тесно связанных с эффективной обработкой данных, таких как Dask, Zarr и Arrow. Но теперь мы вернемся к нашей привычной повестке и познакомимся с подходами к эффективному хранению неоднородных колоночных данных, также именуемых датафреймами.

8.2. Parquet: эффективный формат хранения колоночных данных

Хранение данных в формате CSV сопряжено с большими проблемами. Во-первых, из-за невозможности строго задать типы данных для колонок зачастую можно получить неожиданные значения. Кроме того, сам этот формат данных далек от идеала в плане эффективности. К примеру, числовые данные можно хранить в двоичном виде гораздо более эффективно, чем в текстовом. Помимо этого, при работе с файлами CSV у нас нет возможности перейти к конкретной строке или колонке за фиксированное время, а позицию в файле невозможно рассчитать по причине того, что в файле CSV строки могут варьироваться по размеру.

Apache Parquet закрепился в аналитической среде как эталонный формат хранения неоднородных колоночных данных. Суть этого формата предполагает возможность обращения только к тем колонкам, которые вам нужны, а также использования техники сжатия и кодирования данных для повышения эффективности.

В этом разделе мы узнаем, как можно использовать формат Parquet для хранения датафреймов, продолжив работать с нашим привычным набором данных по нью-йоркским такси. В процессе работы мы узнаем о некоторых полезных возможностях формата Parquet.

ПРЕДУПРЕЖДЕНИЕ. Формат хранения данных Parquet зародился в мире Java, а точнее в экосистеме Hadoop. И хотя реализации этого формата в Python отлично подходят для рабочих проектов, они не полностью отвечают спецификации технологии. К примеру, мы не можем указать во всех подробностях, как именно мы хотим кодировать колонки, а также нам будут недоступны детали хранения колонок, о чем поговорим чуть позже. Но для подавляющего большинства задач функционала Parquet в Python будет вполне достаточно, и он постоянно расширяется.

Напомним, что мы на протяжении нескольких глав работали с набором данных по поездкам на нью-йоркских такси. Среди прочего эти данные содержат информацию о времени посадки и высадки пассажиров, начальной и конечной точках, стоимости, налоговых сборах и чаевых. Здесь мы продолжим использовать файл, с которым начали работать в предыдущей главе, хранящий данные о поездках за январь 2020 года. И первое, что мы сделаем, это преобразуем файл CSV в формат Parquet. Для этого воспользуемся библиотекой Apache Arrow, с которой познакомились ранее. Представленный ниже код можно найти в файле 08-persistence/sec2-parquet/start.py:

```
import pyarrow as pa
from pyarrow import csv
import pyarrow.parquet as pq

table = csv.read_csv(
    "../../07-pandas/sec1-intro/yellow_tripdata_2020-01.csv.gz")
pq.write_table(table, "202001.parquet")
```

Мы просто вызвали функцию *write_table* из модуля *parquet* библиотеки PyArrow. В результате мы получили двоичный файл размером 111 Мб. Сжатая версия файла CSV весила 105 Мб, а полная – 567 Мб. Поскольку Parquet является структурированным двоичным форматом, мы вполне могли ожидать разницу в объеме при хранении одного и того же содержимого. Здесь важно не фиксироваться на деталях, а задуматься о соотношении размеров.

8.2.1. Исследование метаданных Parquet

Давайте рассмотрим некоторые особенности формата Parquet на примере полученного файла:

```
parquet_file = pq.ParquetFile("202001.parquet")

metadata = parquet_file.metadata
print(metadata)
print(parquet_file.schema)
```

```
group = metadata.row_group(0)
print(group)
```

Сокращенный вывод показан ниже:

```
<pyarrow._parquet.FileMetaData object at 0x7f90858879f0>
  created_by: parquet-cpp-arrow version 4.0.0
  num_columns: 18
  num_rows: 6405008
  num_row_groups: 1
  format_version: 1.0
  serialized_size: 4099
<pyarrow._parquet.ParquetSchema object at 0x7f9193aeed00>
required group field_id=0 schema {
  optional int32 field_id=1 VendorID (Int(bitWidth=8,
isSigned=false));
  optional int64 field_id=2 tpep_pickup_datetime (
    Timestamp(isAdjustedToUTC=false, timeUnit=milliseconds,
      is_from_converted_type=false, force_set_converted_type=false));
  ....
<pyarrow._parquet.RowGroupMetaData object at 0x7f90858ad0e0>
  num_columns: 18
  num_rows: 6405008
  total_byte_size: 170358087
```

Начали мы с вывода метаданных о файле. Здесь представлена общая информация, такая как количество колонок (18) и строк (6 405 008). Также мы видим, что в файле содержится одна *группа строк* (row group). Группа строк представляет собой раздел, состоящий из определенного количества записей: в объемных файлах может быть несколько групп строк. В группе находятся все данные о колонках для строк, входящих в ее состав. Помните, что данные в файлах Parquet организованы по колонкам. Совсем скоро вы это увидите.

После вывода информации о файле в целом мы попросили показать нам данные о схеме файла. Сокращенный вывод приведен ниже:

```
required group field_id=0 schema {
  optional int32 field_id=1 VendorID (Int(bitWidth=8, isSigned=false));
  optional int64 field_id=2 tpep_pickup_datetime (
    Timestamp(isAdjustedToUTC=false, timeUnit=milliseconds,
      is_from_converted_type=false, force_set_converted_type=false));
  optional double field_id=5 trip_distance;
  optional binary field_id=7 store_and_fwd_flag (String);
}
```

Здесь мы видим определение поля VendorID, занимающего 8 бит без знака

Здесь выведена информация по всем колонкам в наших данных. Например, поле `VendorID` представлено типом `int32`, но при этом оно занимает 8 бит с отсутствующим знаком. В этой колонке могут присутствовать лишь два значения и `null`, так что восьмью бит без знака для этого вполне достаточно. В теории мы могли бы использовать для хранения этого поля и меньше бит – формат `Parquet` это позволяет.

В колонке `trip_pickup_datetime` у нас находятся даты, и с точки зрения хранения такие поля наиболее важны, поскольку точность типа данных в них напрямую влияет на занимаемое место в памяти. В библиотеке `pandas` по умолчанию точность полей с датой и временем установлена до наносекунд. Также обратите внимание на информацию о поле `store_and_fwd_flag`: здесь стоит отметить, что текст хранится в двоичном виде.

8.2.2. Кодирование колонок в *Parquet*

Теперь давайте внимательно рассмотрим метаданные для отдельных колонок:

```
tip_col = group.column(13) # tip_amount
print(tip_col)
```

Сокращенный вывод:

```
physical_type: DOUBLE
num_values: 6405008
path_in_schema: tip_amount
statistics:
  has_min_max: True
  min: -91.0
  max: 1100.0
  null_count: 0
  distinct_count: 0
  num_values: 6405008
  physical_type: DOUBLE
  logical_type: None
  converted_type (legacy): NONE
compression: SNAPPY
encodings: ('PLAIN_DICTIONARY', 'PLAIN', 'RLE')
has_dictionary_page: True
```

Здесь начинается статистическая информация о колонке

Используемый для колонки алгоритм сжатия

Метаданные начинаются с информации о физическом типе поля, количества значений в нем и его имени. Из раздела статистической информации о поле мы узнаем, что минимальное значение в нем составляет `-91` (похоже на отрицательную сумму чашевых, возможно, введенную по ошибке), а максимальное – `1000`. Дальше интереснее.

В целях экономии места на диске `Parquet` при хранении выполняет сжатие информации в колонках. Помимо экономии дисково-

го пространства, сжатые данные могут обрабатываться в памяти быстрее по причине эффективного задействования кеша центрального процессора, о чем мы говорили в предыдущей главе. При этом к разным колонкам могут применяться разные алгоритмы сжатия, а к каким-то компрессия не применяется вовсе.

Для колонки в нашем примере был использован метод сжатия *Snappy*. Этот метод ставит во главу угла скорость доступа к информации, жертвуя эффективностью сжатия, в отличие от того же метода *gzip*, который также можно использовать. Вы можете сами узнать, какие алгоритмы применяются в *Arrow* на момент использования. На странице <https://facebook.github.io/zstd/#benchmarks> доступны сравнительные данные по разным алгоритмам сжатия, на основании которых вы можете сделать свой выбор.

Допустим, можете указать алгоритм сжатия *ZSTD* следующим образом:

```
pq.write_table(table, "202001_std.parquet", compression="ZSTD")
```

В этом примере мы применили метод сжатия *ZSTD* ко всем колонкам в таблице. Это позволило уменьшить размер итогового файла со 110 Мб с алгоритмом *Snappy* до 82 Мб.

Parquet также предусматривает возможность кодирования колонок не только с использованием значений напрямую, но и с помощью словарей. В этом случае длинные значения конвертируются в косвенные ссылки, что позволяет сэкономить немало места. Чтобы понять, как это работает, представим, что суммы чаевых представлены типом данных *double*, требующим 64 бит, при этом у нас в таблице всего 3626 уникальных значений чаевых:

```
print(len(table["tip_amount"].unique()))
```

Использование словаря поможет снизить количество бит на одно значение с 64 до 12, которых хватит для кодирования 4096 уникальных значений. Нам также понадобится хранить сам словарь из 3626 пар ключ/значение. Однако с учетом того, что уникальных значений у нас здесь довольно много, кодирование с помощью словаря может оказаться не слишком эффективным. Вы можете указать в методе *write_table*, будет ли столбец закодирован со словарем.

Последний вариант кодирования по порядку, но не по значению – это *кодирование длин серий* (*Run Length Encoding – RLE*). Давайте рассмотрим преимущества этого метода на простом и довольно странном примере. Создадим датафрейм с оригинальной колонкой *VendorID* и еще одной такой же колонкой, но с отсортированными значениями:


```
import pyarrow.compute as pc

silly_table = pa.Table.from_arrays([
    table["VendorID"],
    table["VendorID"].take(
        pc.sort_indices(table["VendorID"])]),
    ["unordered", "ordered"]
])
```

Таким образом, мы получили две колонки с одинаковым наполнением, но одна из них отсортирована, а вторая нет. Давайте посмотрим, сколько места каждая из них займет в файле с форматом Parquet:

```
pq.write_table(silly_table, "silly.parquet")
silly = pq.ParquetFile("silly.parquet")
silly_group = silly.metadata.row_group(0)
print(silly_group.column(0))
print(silly_group.column(1))
```

Неотсортированные данные заняли в файле 953 295 байт, а отсортированные – всего 141 байт! Дело в том, что механизм RLE обеспечивает хранение данных посредством запоминания самого значения и количества его последовательных повторений. Получается, что отсортированный столбец идеально будет подходить для такого метода кодирования. В нашей колонке есть всего три уникальных значения (1, 2 и null), и они упорядочены. Таким образом, при использовании метода RLE мы можем хранить данные в таком сжатом формате: 1.0 2094439 / 2.0 4245128 / null 65441.

Кодирование длин серий позволяет очень сильно сократить объем данных при хранении. Наш случай является идеальным для применения этого метода, но в целом он хорошо подходит для упорядоченных полей и столбцов с небольшим количеством повторяющихся значений. При отходе от этих требований убедитесь, что выбранный вами метод кодирования позволяет получить достаточную степень сжатия данных.

Сокращение объема файлов позволяет ускорить обработку данных. В главе 6 мы уже упоминали о том, что при размещении данных в более быстрых типах памяти вы можете ожидать повышения скорости работы с ними на несколько порядков.

Формат Parquet постоянно совершенствуется, так что со временем вы можете ожидать появления новых, еще более эффективных методов хранения информации. Также этот формат позволяет секционировать данные, что положительно влияет на эффективность. Мы поговорим об этом в следующем разделе.

8.2.3. Секционирование наборов данных

Чтобы разобраться в том, что из себя представляет *секционирование* (partitioning) и какие последствия влечет, давайте разобьем наш набор данных на секции с использованием колонок `VendorID` и `passenger_count`. Поскольку секции данных не могут основываться на отсутствующих значениях, мы исключим их из набора данных. Сделаем мы это только с целью демонстрации. В обычных условиях вы не должны удалять строки с пустыми значениями просто потому, что вам так удобно:

```
from pyarrow import csv
import pyarrow.compute as pc
import pyarrow.parquet as pq

table = csv.read_csv(
    "../07-pandas/sec1-intro/yellow_tripdata_2020-01.csv.gz")

table = table.filter(
    pc.invert(table["VendorID"].is_null()))
table = table.filter(pc.invert(table["passenger_count"].is_null()))

pq.write_to_dataset(
    table, root_path="all.parquet",
    partition_cols=["VendorID", "passenger_count"])
```

Снова обращаем ваше внимание на разницу в подходах Arrow и pandas к обработке данных

При использовании библиотеки `pandas` строка с фильтрацией по столбцу `VendorID` выглядела бы так: `table = table[table["VendorID"].isna()]`.

Если вы теперь попытаетесь найти файл `all.parquet` на диске, то будете очень удивлены, обнаружив не файл с таким именем, а директорию. Сокращенное содержимое иерархии папок и файлов внутри этой директории показано ниже:

```
.
├── VendorID=1
│   ├── passenger_count=0
│   │   └── e59ac47b5193411e9772bfee9d423d61.parquet
│   ├── passenger_count=1
│   │   └── ee90fe5b818d4a37a32b5a415915610b.parquet
│   └── passenger_count=9
│       └── 002ff0bba1d340abb6174c5c64f779d7.parquet
└── VendorID=2
    ├── passenger_count=0
    │   └── 5809e29649524202a9b3cef5371c46d9.parquet
    └── passenger_count=9
        └── feaff7a23bbf4ae2b687b34dcaa10afb.parquet
```

Как видите, структура папок четко соответствует нашей стратегии секционирования. Папки первого уровня относятся к зна-

чениям поля `VendorID`, а папки второго уровня – к значениям поля `passenger_count`.

Теперь у вас есть два варианта. Первый, менее интересный, заключается в загрузке всех данных в качестве таблицы:

```
all_data = pq.read_table("all.parquet/")
```

В этом случае все имеющиеся данные будут загружены в обычную таблицу. Того же эффекта можно добиться при помощи следующей последовательности инструкций:

```
dataset = pq.ParquetDataset("all.parquet/")
ds_all_data = dataset.read()
```

В качестве альтернативы вы можете загрузить каждый файл Parquet по отдельности. Для примера давайте загрузим секцию данных, соответствующую трем пассажирам и значению поля `VendorID`, равному единице:

```
import os
data_dir = "all.parquet/VendorID=1/passenger_count=3"
parquet_fname = os.listdir(data_dir)[0]
v1p3 = pq.read_table(f"{data_dir}/{parquet_fname}")
print(v1p3)
```

Имя файла в папке мы не знаем, поэтому обращаемся к нему по индексу 0 (берем первый по счету)

Если вы взглянете на вывод, то обнаружите, что колонки `VendorID` и `passenger_count` отсутствуют, поскольку их значения могут быть выведены из названия директорий.

ПРЕДУПРЕЖДЕНИЕ. Содержимое каждой директории может меняться. В нашем случае, с использованием библиотеки `PyArrow`, это один файл Parquet. Допустим, вы можете сказать Parquet, чтобы он разбил каждую секцию на файлы по группам строк. Таким образом, вам нужно следить за тем, как данные на самом деле записываются на диск, и соответствующим образом адаптировать свой код.

В чем смысл секционирования данных с точки зрения производительности? Это дает нам возможность загружать и обрабатывать каждый файл Parquet по отдельности. К примеру, мы можем значительно повысить быстродействие приложения, задействовав несколько процессов в рамках одной машины, каждый из которых будет заниматься своим файлом. Кроме того, мы можем обрабатывать отдельные файлы на разных компьютерах. Это не вполне очевидно, но файловая система может функционировать гораздо эффективнее, если конкурирующие загрузки будут производиться физически в разных частях диска. Также выигрыш будет наблю-

даться в отношении памяти, поскольку мы не будем загружать колонки, по которым производилось секционирование. Наконец, секционирование открывает дорогу к конкурентным операциям записи, что позволяет воспользоваться всеми преимуществами параллельных вычислений. Более подробно о конкурентной записи мы будем говорить в разделе 8.4.

Сам способ секционирования тоже играет важную роль в отношении производительности. К примеру, если первому продавцу (Vendor) соответствует вдвое меньше записей, чем второму, то на обработку данных по второму продавцу потребуется вдвое больше времени. Это может привести к нежелательному простоев в ожидании завершения обработки самой большой секции, поскольку нам важно поддерживать синхронность и единообразие. При этом поле `VendorID` может лучше подходить для секционирования, нежели поле `passenger_count`. У формата Parquet есть целый ряд других возможностей, но мы уже увидели, что в плане производительности ему действительно есть чем похвастаться.

8.3. Работа с наборами данных, не помещающимися в памяти, по-старому

В этом разделе мы на примере файлов Parquet и CSV продемонстрируем две простые техники работы с данными, не помещающимися в памяти: отображение в памяти (*memory mapping*) и порционирование (*chunking*). При этом существуют более прогрессивные способы выполнения этих задач, о которых мы поговорим в разделе 8.4 и в следующей главе. Но методы, рассмотренные здесь, очень важны для понимания более продвинутых библиотек, о которых мы будем говорить далее.

8.3.1. Отображение в памяти с помощью NumPy

Процедура *отображения в памяти* (*memory mapping*) подразумевает прямую ассоциацию между фрагментом памяти и содержимым файловой системы. Применительно к NumPy это означает, что к массиву, находящемуся в хранилище, можно обращаться с помощью обычного API NumPy, и сам NumPy обеспечивает перенос в оперативную память необходимых нам фрагментов массива. В большинстве случаев этим занимается ядро операционной системы. При обратной записи данных в хранилище производятся нужные изменения. Поскольку в этом случае вы фактически работаете с оперативной памятью, время выполнения операций может вырасти на несколько порядков. На рис. 8.1 схематично показана процедура отображения в памяти.

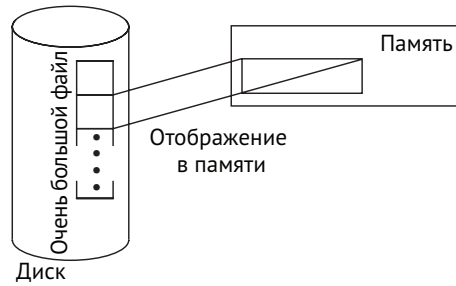


Рис. 8.1. Отображение в памяти позволяет эффективно отзеркалить фрагмент файла в оперативной памяти

Мы продемонстрируем это на простом абстрактном примере с созданием большого массива и чтением из него. Размер массива вы сможете задать по своему усмотрению. Для этого упражнения я рекомендую ставить размер массива, превышающий объем оперативной памяти вашего компьютера, но достаточный, чтобы разместить данные на жестком диске. Выполните следующий код:

```
import numpy as np
SIZE_IN_GB = 10
array = np.memmap("data.npy", mode="w+",
                  dtype=np.int8, shape=(SIZE_IN_GB * 1024, 1024, 1024))
print(array[-1, -1, :10])
```

Измените размер файла в Гб под ваши
требования

Вызов функции *np.memmap* довольно прост: вам достаточно передать ей на вход имя файла, режим доступа, а также тип элементов в массиве и его размер. Если после запуска этого кода вы откроете папку с исходным кодом, то увидите созданный в ней файл *data.npy* размером 10 Гб.

Массив данных будет заполнен нулями, так что на экран выведется десять нулей. Теперь давайте добавим двойку к каждому элементу в массиве:

```
array += 2
```

Как видите, обращаться с этим объектом можно точно так же, как с обычным массивом NumPy. При этом вы можете заметить, что эта операция займет несколько секунд. Это объясняется тем, что в процессе ее выполнения наш большой файл на диске целиком обновляется, а это не такая простая задача.

Теперь давайте откроем наш файл и выведем последние значения:

```
array = np.memmap("data.npy", mode="r", dtype=np.int8)
print(array.shape)
print(array[-10:])
```

Вывод будет таким:

```
(10737418240,)  
[222 ... 222]
```

Здесь важно отметить, что размерность массива не сохраняется вместе с ним, так что при отображении массива без указания его размерности вы получите одномерный массив. Вы должны самостоятельно следить за тем, в каком виде извлекать данные. В заключительной строке кода мы выводим на экран последние десять элементов массива – все двойки.

Копирование при записи в NumPy

Механизм отображения в памяти позволяет воспользоваться технологией *копирования при записи* (copy-on-write), дающей возможность иметь несколько копий массива на диске с отображением в памяти и при этом использовать существенно меньше памяти. Зачастую использование этой технологии ведет к ошибкам, в основном по причине того, что Python не лучшим образом подходит для работы с распределенными структурами данных, а также из-за неопределенностей семантики отображения данных в памяти при изменении лежащих в их основе файлов. Не думаю, что потенциальные преимущества способны покрыть связанные с использованием этой техники риски, если вы не уверены, что будете работать с данными исключительно на чтение. При желании ближе познакомиться с этой техникой вы можете прочитать великолепную статью Итамара Тернера-Трауринга (Itamar Turner-Trauring), размещенную по адресу <https://pythonspeed.com/articles/reduce-memory-array-copies>.

Лично я предпочел бы отказаться от явного применения техник отображения данных в памяти при наличии конкурентной записи и совместного использования данных, если только не был бы абсолютно уверен в том, что каждый процесс работает исключительно на чтение. Если же вы являетесь разработчиком низкоуровневой библиотеки, вам может понадобиться использовать технику отображения в памяти совместно с записью, но при этом вы вряд ли будете использовать Python для реализации ключевых фрагментов кода, так что решение этих проблем можно возложить на другие языки.

Помните о том, что, даже если вы явно не применяете технику отображения данных в памяти, ее могут задействовать многие из используемых вами фреймворков, так что знать о ней полезно. Теперь давайте обсудим еще одну технику для работы с большими файлами, а именно порционирование данных.

8.3.2. Порционирование данных при чтении и записи в датафрейм

Порционирование (chunking) данных, как ясно из названия, предполагает обработку файлов с разбиением содержимого на порции.

Таким образом, вы читаете или пишете информацию в файл по частям. Если вы уже работали с библиотекой *Zarr*, которую мы будем обсуждать в разделе 8.4, или *Dask*, которой будет посвящена глава 10, вы сталкивались с операцией порционирования.

Здесь мы продолжим работать с нашим набором данных, содержащим поездки на такси. Мы преобразуем файл из формата CSV в Parquet, но по частям. Несмотря на то что наш файл достаточно небольшой и мы могли бы обработать его в памяти практически на любом компьютере, представим, что на нашей машине недостаточно памяти для хранения всего набора целиком.

В процессе написания кода воспользуемся библиотекой *pandas* для чтения файла CSV, а затем с помощью *Aggow* запишем данные в файл с форматом Parquet. Мы могли бы все это проделать с одной только библиотекой *Aggow*, что было бы даже более эффективно, но мы хотим показать вам механизм порционирования данных, применяемый в *pandas*:

```
import pandas as pd

table_chunks = pd.read_csv(
    "../../07-pandas/sec1-intro/yellow_tripdata_2020-01.csv.gz",
    chunksize=1000000
)
print(type(table_chunks))  # Тип данных объекта table_chunks – pandas.
                           # io.parsers.TextFileReader
for chunk in table_chunks:  # Каждая порция (chunk) данных –
    print(chunk.shape)      # это датафрейм
```

Все, что нам пришлось сделать, это добавить параметр *chunksize* при вызове функции *read_csv*. В результате на выходе функции мы получили не датафрейм, а генератор порций, каждая из которых является датафреймом с максимальным количеством строк, равным одному миллиону.

Теперь займемся преобразованием данных в формат Parquet. Сперва нам нужно заново открыть файл. Мы уже прошили по разу по каждой порции данных, так что придется начать все сначала:

```
table_chunks = pd.read_csv(
    "../../07-pandas/sec1-intro/yellow_tripdata_2020-01.csv.gz",
    chunksize=1000000,
    dtype={
        "VendorID": float,
        "passenger_count": float,
        "RatecodeID": float,
        "PULocationID": float,
        "DOLocationID": float,
        "payment_type": float,
    }
)
```

Как видите, мы явным образом указали типы данных некоторых полей. Заметим, что тип одной и той же колонки может меняться от порции к порции. В большинстве своем это относится к целочисленным полям, которые могут содержать пропущенные значения. При обнаружении пропусков тип данных таких колонок будет устанавливаться в `float`, поскольку в `pandas` недопустимо наличие пропущенных значений в колонках с типом `integer`.

Теперь пройдем по нашим порциям данных и создадим файл `Parquet`:

```
first = True
writer = None
for chunk in table_chunks:
    chunk_table = pa.Table.from_pandas(chunk)
    schema = chunk_table.schema
    if first:
        first = False
        writer = pq.ParquetWriter(
            "output.parquet", schema=schema)
    writer.write_table(chunk_table)
writer.close()
```

Преобразовываем
датафрейм `pandas`
в таблицу `Arrow`

Создаем объект `writer`.
При инициализации
нам необходимо
указать схему

Интерфейс `ParquetWriter` позволяет записывать таблицу за таблицей в один и тот же файл. Каждая таблица при этом будет записана в отдельную группу строк. По сути, это и есть порция.

Читать данные из файла `Parquet` можно по-разному:

```
pf = pq.ParquetFile("output.parquet")
print(pf.metadata)
for groupi in range(pf.num_row_groups):
    group = pf.read_row_group(groupi)
    print(type(group), len(group))
    # break
table = pf.read()
table = pq.read_table("output.parquet")
```

Можно читать каждую группу
отдельно

Метаданные показывают, что в нашем файле `Parquet` находится семь групп строк. При этом у нас есть возможность читать их одну за другой. Если у вас достаточно памяти, есть два способа считать все группы строк и создать таблицу в памяти: это метод `read` из модуля `ParquetFile` и `read_table` из модуля `parquet`.

Теперь, когда вы знаете, как с помощью техники порционирования можно загружать и обрабатывать данные частями, мы перейдем к знакомству с библиотекой `Zarr`, позволяющей манипулировать очень большими однородными многомерными массивами вроде объектов `NumPy`.

8.4. Использование Zarr для хранения больших массивов

Зачастую большие наборы данных бывают представлены не в виде разнородных табличных данных, а в виде многомерных однородных массивов. И существует потребность хранить такие данные предельно эффективно.

Библиотека *Zarr* используется для хранения такого вида массивов на разных серверных архитектурах и в различных форматах кодировки и обладает функционалом конкурентной записи для эффективного генерирования данных.

Существует несколько проверенных временем стандартов представления подобных массивов (например, NetCDF и HDF5), но мы решили обратить внимание в этой книге на только зарождающийся формат *Zarr*, позволяющий значительно лучше оптимизировать данные для последующей обработки. Например, вы можете использовать конкурентную запись и разную организацию файловой структуры, и оба этих фактора оказывают существенное влияние на производительность. Конкурентная запись позволяет нескольким процессам работать одновременно с одной и той же структурой. А использование разных структур дает возможность извлечь все возможные выгоды из характеристик файловой системы.

Хотя *Zarr* является форматом файлов, зародился он именно в Python как часть одноименной библиотеки. По этой причине вы можете быть уверены, что в версии для Python *Zarr* реализует все свои основные возможности. Если вы планируете использовать файлы *Zarr* в других языках программирования, то сперва должны убедиться, поддерживает ли конкретная реализация формата нужные вам возможности. В некотором смысле формат *Zarr* является противоположностью *Parquet*: если *Parquet* пришел в Python из экосистемы Java, и именно поэтому в реализации *Parquet* для Python не прописаны все возможности, формат *Zarr* вырос в Python, и эта его реализация представляет золотой стандарт.

Формат *Zarr* появился в среде биоинформатики, и в данном разделе мы будем использовать пример именно из этой области, в частности, возьмем данные одного старого проекта по геномике, называющегося *HarMap* (<https://www.genome.gov/10001688/international-hapmap-project>). В рамках проекта были собраны разные генетические варианты (вариации в последовательностях ДНК) для множества представителей разных популяций. Не пугайтесь, вам не придется разбираться в научных подробностях этого проекта. Мы будем пользоваться упрощенной концепцией, которую обговорим в процессе работы над примером.

Начнем с заранее подготовленной базой данных *Zarr*, которую я сгенерировал на основании информации из проекта *HarMap*

в формате Plink (<https://www.cog-genomics.org/plink/2.0>). Вам не стоит беспокоиться из-за исходного формата данных, но, если действительно интересно, вы можете найти в сопроводительных материалах код для создания базы данных Zarr. Сам файл базы данных можно загрузить по адресу <https://tiago.org/db.zarr.tar.gz>. В нашу базу включена генетическая информация о 210 представителях разных популяций.

Одной из наших целей будет создание другой базы данных Zarr, пригодной для применения *метода главных компонент* (principal components analysis – PCA) – распространенной в мире генетики техники машинного обучения без учителя. Для этого нам придется переформатировать данные, полученные из источника. Мы не будем заходить далеко и применять метод главных компонент, а ограничимся лишь подготовкой данных для него.

8.4.1. Знакомство с внутренней структурой формата Zarr

Давайте начнем с анализа содержимого базы данных. Одновременно мы будем знакомиться с некоторыми терминами из области генетики:

```
import zarr
```

```
genomes = zarr.open("db.zarr")
genomes.tree()
```

Выводим древовидную
структуру содержимого файла

Zarr представляет собой иерархический контейнер для массивов, так что здесь мы имеем дело с древовидной структурой, в которой концевыми элементами являются массивы данных. Сокращенная версия вывода показана ниже:

```
├─ chromosome-1
│   ├── alleles(318558,<U2
│   ├── calls(318558,210)uint8
│   └── positions(318558,)int64
├─ chromosome-10
│   ├── alleles(216535,<U2
│   ├── calls(216535,210)uint8
│   └── positions(216535,)int64
```

Наши данные разбиты на хромосомы, каждая из которых содержит свою иерархию. Внутри хромосомы располагается список генотипированных позиций, по которым мы получаем буквы ДНК, в массиве `positions`. Возможные аллели (т. е. буквы ДНК) для каждой позиции находятся в массиве `alleles`. В основной матрице с именем `calls` располагаются аллели для каждого маркера среди 210 индивидуальных представителей. Таким образом, с учетом того, что в первой хромосоме насчитывается 318 558 маркеров, матрица `calls` будет обладать размером 318 558×210. Для каждо-

го представителя и маркера существует два показателя, которые в массиве `calls` закодированы одним числом.

Наша цель состоит в создании составной матрицы всех вызовов для передачи в нашу реализацию метода главных компонент. Не беспокойтесь о специфике – для нас важно только то, что у нас есть двумерный массив `calls` со значениями 0/1/2 в виде беззнаковых 8-битных целых чисел и два одномерных массива, один из которых представлен 64-битными целыми числами (`positions`), а второй – строками с максимальной длиной в два символа (`alleles`).

Перед тем как погрузиться в вопросы, связанные с производительностью, давайте вкратце обсудим, как можно осуществлять проход по данным в формате `Zarr`. Мы можем пройти по всем данным в структуре так, как показано ниже:

```
def traverse_hierarchy(group, location=""):
    for name, array in group.arrays():
        print(f"{location}/{name} {array.shape} {array.dtype}")
    for name, group in group.groups():
        my_root = f"{location}/{name}"
        print(my_root + "/")
        traverse_hierarchy(group, my_root)
```

Получаем все массивы
внутри группы

Получаем все группы
внутри группы

`traverse_hierarchy(genomes)`

При чтении файла `Zarr` возвращает объект `Group`. Метод `groups` позволяет получить генератор со всеми подгруппами, позволяющий свободно перемещаться по репозиторию `Zarr`.

Также вы можете получить произвольный доступ к содержимому базы данных, используя пути, подобно работе с папками. Например:

```
in_chr_2 = genomes["chromosome-2"]
pos_chr_2 = genomes["chromosome-2/positions"]
calls_chr_2 = genomes["chromosome-2/calls"]
alleles_chr_2 = genomes["chromosome-2/alleles"]
```

Переменная `in_chr_2` будет содержать объект `Group`, а в переменных `pos_chr_2`, `calls_chr_2` и `alleles_chr_2` будут находиться готовые к использованию массивы `chromosome-2/positions`, `chromosome-2/calls` и `chromosome-2/alleles` соответственно.

Давайте теперь извлечем из нашей структуры какую-нибудь информацию:

```
print(in_chr_2.info)
```

Вывод будет следующим:

```

Name      : /chromosome-2
Type      : zarr.hierarchy.Group
Read-only : False
Store type : zarr.storage.DirectoryStore
No. members : 3
No. arrays : 3
No. groups : 0
Arrays    : alleles, calls, positions

```

Мы видим, что наш объект Group содержит три элемента, все из которых являются массивами. Но здесь также могут быть и подгруппы.

Формат Zarr поддерживает большое разнообразие хранилищ. В нашем случае используется хранилище `zarr.storage.DirectoryStore`, но вы можете найти классы для работы с данными в памяти, с файлами `zip`, `DBM`, `SQL`, `fsspec`, `Mongo` и многими другими источниками.

Скоро мы увидим, что хранилище `DirectoryStore` поддерживает очень важную опцию, связанную с параллельными вычислениями, а сейчас давайте взглянем на структуру директорий. Если вы еще не заметили, `db.zarr` на самом деле является не файлом, а папкой. Ниже показана сокращенная версия ее иерархии:

```

.
├─ chromosome-1
│   ├── alleles
│   ├── calls
│   └── positions
├─ chromosome-10
│   ├── alleles
│   ├── calls
│   └── positions
...

```

Как видите, иерархия директорий полностью повторяет структуру группы Zarr, что значительно облегчает разработку.

8.4.2. Хранение массивов в Zarr

Теперь давайте поговорим о том, как в формате Zarr хранятся массивы, – это гораздо более сложная и интересная тема:

```
print(pos_chr_2.info)
```

Отсортируем вывод и разобьем его на несколько частей. Начнем с базовой информации, представленной ниже:

```

Type      : zarr.core.Array
Data type : int64
Shape     : (333056,)
Order     : C
Read-only : False
Store type : zarr.storage.DirectoryStore

```

Вы спокойно можете проинтерпретировать эту информацию, если читали предыдущие главы книги. Мы здесь имеем дело с объектом `zarr.coge.Agga`, хранящим 333 056 целочисленных значений с сортировкой `C`, каждое из которых занимает 64 бит и может быть перезаписано.

Теперь взглянем на опции, связанные с порционированием данных:

```
Chunk shape      : (41632,)
Chunks initialized : 8/8
```

Если вы помните, механизм порционирования, показанный на рис. 8.2, служит для разделения больших массивов данных на более мелкие части (порции), которыми гораздо легче манипулировать.

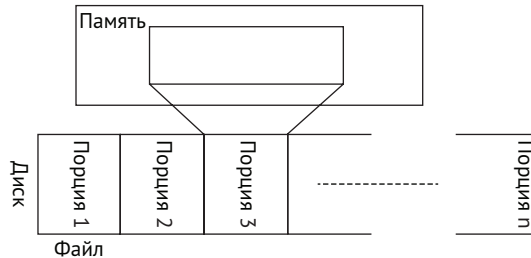


Рис. 8.2. Объемный файл с массивом может быть разбит на равные по размеру порции, которые могут быть обработаны по отдельности

`Zarr` говорит нам, что в каждой порции данных находится 41 632 элемента, а в сумме в восьми порциях как раз получается 333 056 элементов. Когда мы создавали массив во вспомогательном скрипте, то по своей наивности не указали желаемый объем порций, так что `Zarr` сделал это за нас. Но размеры порций могут и должны быть установлены при создании массива. Позже в этом разделе мы узнаем, почему.

Также обратите внимание на то, что все восемь порций данных у нас инициализированы, хотя при определенных условиях (например, когда мы имеем дело с пустыми массивами) в инициализации массивов нет нужды. Процесс инициализации массивов может позволить в перспективе сэкономить немало места на диске. Мы также увидим это позже при создании массивов.

Если вы откроете директорию `db.zarr/chromosome-2/position`, то увидите в ней восемь файлов с именами от 0 до 7: по одному файлу на порцию. Такое разделение файлов позволяет эффективно использовать формат `Zarr` для выполнения конкурентных операций записи, что не поддерживается во многих других системах хранения массивов.

Наконец, массивы в формате `Zarr` могут быть эффективно сжаты, что позволит экономить дисковое пространство, а с ним

и время выполнения, о чем мы уже говорили ранее в этой главе. Ниже приведен фрагмент метаданных, посвященный компрессии:

```
Compressor      : Blosc(cname='lz4', clevel=5,  
                        shuffle=SHUFFLE, blocksize=0)  
No. bytes       : 2664448 (2.5M)  
No. bytes stored : 687723 (671.6K)  
Storage ratio    : 3.9
```

В нашем примере массивы были сжаты при помощи библиотеки Blosc с использованием алгоритма LZ4. В исходном виде данные занимали 2 664 448 байт (333 056 элементов по 8 байт каждое для хранения 64-битных целых чисел). После сжатия объем данных снизился до 687 723 байт, а коэффициент сжатия составил 3,9. С учетом однородности наших массивов мы вполне могли ожидать, что компрессия сработает более эффективно по сравнению с неоднородными датафреймами. Разумеется, свои предположения мы можем строить применительно к усредненному случаю наполнения массивов. Например, массивы, заполненные случайными числами, очень трудно поддаются компрессии.

Для массива `calls` мы видим похожий вывод с поправкой на двумерность структуры. Ниже показан сокращенный вывод команды `print(calls_chr_2.info)`:

```
Shape           : (333056, 210)  
Chunk shape     : (41632, 27)  
Chunks initialized : 64/64
```

Здесь мы имеем дело с матрицей размером 333 056×210 и двумерными порциями данных.

СОВЕТ. Вы можете порционировать N-мерный массив с помощью массивов меньшей размерности, чем N. Например, наш двумерный массив мы могли бы разбить на одномерные массивы. Такой подход имеет смысл, если вам необходимо одновременно обрабатывать всю информацию из одного измерения. Как и в случае с любыми другими вопросами, касающимися порционирования данных, здесь все зависит от ваших нужд и конкретной ситуации.

В нашем случае каждое измерение было разбито на восемь интервалов, что в сумме позволило создать 64 порции данных. Если вы откроете директорию `db.zarr/chromosome-2/calls`, то увидите в ней 64 файла, названия которых следуют шаблону X.Y, где X и Y варьируются от 0 до 7, что соответствует номеру порции в рамках каждого измерения.

Наконец, у нас есть массив `alleles`, значения которого представлены в виде строки из двух символов (AT, CG, TC и т.д.). Сокращенный вывод инструкции `print(alleles_chr_2.info)` показан ниже:

Data type : <U2

Тип данных здесь представлен как строка в кодировке Unicode с фиксированным размером в 2 байта. Если вы помните, в главе 2 мы уже говорили, что в Python строки представлены очень необычно (или неприлично, в зависимости от точки зрения), и оценить реальный размер строки в байтах там – задача не из легких.

В то же время для эффективной работы со строками полезно точно представлять себе их размер. В формате Zarr предусмотрено два способа для представления строк. Если ваша строка состоит только из символов ASCII, вы можете представить ее как массив байтов. Для хранения расширенных строк можно воспользоваться юникод-представлением с фиксированным количеством символов, что отличается от строк с динамическим размером в Python. Если вам необходимо использовать строки переменной длины и разные кодировки, у Zarr найдутся для вас соответствующие кодировщики, но в плане производительности вы можете просесть. В связи с этим мы рекомендуем всегда, когда это возможно, использовать строки с фиксированной длиной.

Теперь, когда вы понимаете, как организованы данные в формате Zarr, давайте создадим массив, собрав в него все позиции по всем хромосомам. Нам нужно получить один массив по всем хромосомам – именно его мы отправим на вход методу главных компонент, для которого готовим данные.

8.4.3. Создание нового массива

Итак, мы собираемся создать новый массив, который может быть использован для алгоритмов машинного обучения без учителя вроде метода главных компонент. Для этого нам просто нужно собрать воедино все данные из массивов `calls`.

Но для начала нам необходимо узнать размер будущего массива. Это можно сделать, пройдя по существующему файлу Zarr и вычислив количество маркеров на одну хромосому. Это число будет варьироваться в зависимости от вашей задачи:

```
import zarr

genomes = zarr.open("db.zarr")

chrom_sizes = []
for chrom in range(1, 23):
    chrom_pos_array = genomes[f"chromosome-{chrom}/positions"]
    chrom_sizes.append(chrom_pos_array.shape[0])
total_size = sum(chrom_sizes)
```

Здесь мы просто собираем в список количество элементов в одномерных массивах с позициями. После этого рассчитываем их сумму, что дает нам необходимый размер нашего будущего массива.

Теперь, когда мы знаем, какое количество элементов нам понадобится, мы можем создать сам массив:

```
CHUNK_SIZE = 20000
all_calls = zarr.open(
    "all_calls.zarr", "w",
    shape=(total_size, 210),
    dtype=np.uint8, # изменение типа данных
    chunks=(CHUNK_SIZE,))
```

210 – это количество индивидуальных представителей в нашем наборе данных

Наиболее важный параметр с точки зрения производительности – это размер порции (`CHUNK_SIZE`). Мы выбрали такое значение, чтобы размер одной порции данных превышал 1 Мб, хотя вы для своей задачи можете выбрать любое другое значение. Если умножить 20 000 на 210, мы получим порцию размером около 4 Мб, но мы еще полагаемся на некоторое сжатие. Мы предполагаем, что информация обо всех представителях будет прочитана за раз, так что порционируем данные только по одному измерению. Вы можете поиграть с размером порции и посмотреть, как это влияет на производительность.

Основные правила по определению размера порции данных

Вывести какие-то общие постулаты, которым необходимо следовать при установке размера порции данных, очень сложно. Все обычно зависит от используемого алгоритма и прочих обстоятельств. И все же некоторые базовые идеи, которыми можно руководствоваться, сформулировать можно:

- не делайте порции данных слишком маленькими. Обычно их размер не должен быть меньше 1 Мб;
- порции данных должны комфортно помещаться в памяти;
- попробуйте использовать разные значения на разных измерениях. Это может серьезно повлиять на производительность и возможность выполнить все требуемые действия в памяти;
- выбор размера порции может зависеть от используемого типа хранилища. К примеру, `DirectoryStore` не справится с задачами масштабирования при наличии тысячи порций данных просто потому, что файловая система не предназначена для хранения такого количества файлов в одной папке. Для этой цели в библиотеке Zarr предусмотрен класс `NestedDirectoryStore`, позволяющий распределять порции данных по разным директориям. Как видите, очень важно понимать особенности разных хранилищ и классов, чтобы соответствующим образом корректировать размер порции.

Теперь давайте посмотрим информацию о созданном массиве `all_calls`. Сокращенная версия приведена ниже:


```

Type           : zarr.core.Array
Data type      : uint8
Shape          : (3976554, 210)
Chunk shape    : (20000, 210)
No. bytes      : 835076340 (796.4M)
No. bytes stored : 345
Storage ratio   : 2420511.1
Chunks initialized : 0/199

```

Здесь важно обратить внимание на количество хранящихся байт (No. bytes stored) и количество инициализированных порций данных (Chunks initialized). Тогда как ожидаемый суммарный размер массива равен 796,4 Мб, на данный момент в нем хранится всего 345 байт (!), поскольку никаких данных мы пока не сохраняли (и ни одна порция данных пока не инициализирована). По умолчанию Zarr считает значения в массиве, порции которого не были инициализированы, равными нулю. Если в этот момент вы откроете директорию `all_calls.zarr`, то увидите, что она пуста и не занимает места на диске.

На самом деле в ней есть один скрытый файл `.zarrгау`, где в формате JSON хранятся метаданные, которые мы передали Zarr при создании файла, и некоторые другие значения по умолчанию.

8.4.4. Параллельное чтение и запись массивов в Zarr

Теперь пришло время создать объединенный массив данных для передачи на вход метода главных компонент.

Мы рассмотрим два варианта сценария: последовательный и параллельный. Ниже приведен первый из них:

```

def do_serial():
    curr_pos = 0
    for chrom in range(1, 23):
        chrom_calls_array = genomes[f"chromosome-{chrom}/calls"]
        my_size = chrom_calls_array.shape[0]
        all_calls[curr_pos: curr_pos + my_size, :] = chrom_calls_array
        curr_pos += my_size

do_serial()
print(all_calls.info)

```

В этом коде мы последовательно собираем в один массив `all_calls` элементы из массивов `calls`. Обратите внимание, что все управление хранилищем при этом полностью абстрагировано поверх обычного интерфейса NumPy.

Если после запуска этого кода вывести информацию о массиве `all_calls`, вы увидите следующие изменения:

```

No. bytes      : 835076340 (796.4М)
No. bytes stored : 297035153 (283.3М)
Storage ratio   : 2.8
Chunks initialized : 199/199

```

Теперь, когда все наши порции данных инициализированы, объем наших реальных данных составил 283,3 Мб с коэффициентом сжатия 2,8. До компрессии данные занимали 796,4 Мб. Если вы теперь откроете директорию `all_calls.zarr`, то обнаружите в ней 199 файлов – по одному на порцию.

Предыдущему коду потребовалось на выполнение несколько секунд. Конечно, я не буду требовать от вас проверки этого фрагмента кода с данными, занимающими терабайты памяти, но вы и сами прекрасно понимаете, что при увеличении объемов исходной информации время обработки будет неумолимо расти.

Учитывая это, мы подготовили вторую версию нашей функции, которая будет читать данные из массивов хромосом и писать их в массив `all_calls`, при этом обе операции будут выполняться в параллельном режиме.

Немногие библиотеки поддерживают параллельные вычисления, но Zarr из их числа. Заранее разбив исходные данные на порции, мы тем самым значительно облегчили задачу Zarr по реализации параллельной записи. Наш код будет предельно простым, но при этом очень эффективным за счет использования особенностей файловой системы.

В теории мы можем писать данные любого объема, но более эффективно будет делать это порциями заданного ранее размера, поскольку в этом случае Zarr не придется выполнять конкурентную запись в один и тот же файл. Очень важно, чтобы размер порции идеально подходил вашей конкретной задаче, тогда вы сможете обрабатывать данные по частям наиболее эффективно.

В нашем случае нельзя идти просто по хромосомам, мы должны идти порциями. Ниже приведена функция для записи порции данных:

```

def process_chunk(genomes, all_calls, chrom_sizes, chunk_size, my_chunk):
    all_start = my_chunk * chunk_size
    remaining = all_start
    chrom = 0
    chrom_start = 0
    for chrom_size in chrom_sizes:
        chrom += 1
        remaining -= chrom_size
        if remaining <= 0:
            chrom_start = chrom_size + remaining
            remaining = -remaining
            break

```

Позиция для записи определяется как номер текущей порции, умноженный на размер порции

Проходим по размерам хромосом, пока не найдем, откуда начать

▶ while remaining > 0:

Порция может потребовать более одной хромосомы

```

    write_from_chrom = min(remaining, CHUNK_SIZE)
    remaining -= write_from_chrom
    chrom_calls = genomes[f"chromosome-{chrom}/calls"]
    all_calls[all_start:all_start + write_from_chrom, :] = chrom_calls[
        chrom_start: chrom_start + write_from_chrom, :]
    all_start = all_start + write_from_chrom

```

Не расстраивайтесь, если не все понимаете в этом фрагменте кода, поскольку в нем есть определенная специфика предметной области. Здесь важен сам подход. Мы стараемся обрабатывать данные порциями, что идеально подходит для работы с большими файлами, не помещающимися в памяти.

Теперь мы можем воспользоваться простым пулом из модуля `multiprocessing` и с помощью метода `map` обработать данные параллельно:

```

from functools import partial
from multiprocessing import Pool

partial_process_chunk = partial(
    process_chunk, genomes,
    all_calls, chrom_sizes, CHUNK_SIZE)

def do_parallel():
    with Pool() as p:
        p.map(partial_process_chunk, range(all_calls.nchunks))

do_parallel()

```

Мы воспользовались функцией `partial` из модуля `functools` для упрощенного доступа к функции `process_chunk`. Это позволило сделать вызовы метода `Pool.map` для параллельной обработки наших данных более читаемыми. За подробностями написания кода с многопроцессной обработкой можно обратиться к главе 3.

Заключение


- Библиотека `fsspec` выступает в роли универсального интерфейса для различных файловых систем, позволяя использовать один API для самых разных серверных архитектур.
- По причине использования единого API при работе с библиотекой `fsspec` вы можете очень легко менять серверную часть и не испытывать при этом больших сложностей.
- Хотя использование `fsspec` напрямую не сказывается на производительности, многие важные библиотеки, такие как `Arrow` и `Zarr`, активно поддерживают работу с этой библиотекой.

- Parquet представляет собой колоночный формат данных, позволяющий обеспечить более эффективное хранение. Данные в этом формате строго типизированы, организованы по колонкам и обычно хорошо сжимаются.
- В формате Parquet применяются продвинутое методы кодирования данных, включая кодирование со словарем и на основе длин серий. Это позволяет значительно повысить эффективность хранения данных, особенно при наличии повторяющихся значений и определенных шаблонов. Кроме того, этот формат активно развивается, и в будущем, возможно, нас ждут новые улучшения в плане организации и хранения данных.
- Формат Parquet поддерживает секционирование, что позволяет разработчикам обрабатывать данные в параллельном режиме.
- Наиболее распространенной техникой для работы с файлами, физически не помещающимися в памяти, является порционирование. Эта техника поддерживается во множестве библиотек, включая pandas, Parquet и Zarr.
- Zarr – это современная библиотека, позволяющая эффективно обрабатывать многомерные однородные массивы данных. Изначально она появилась в экосистеме Python и нативно предлагает интерфейсы, основанные на NumPy.
- Библиотека Zarr поддерживает параллельные вычисления. Также стоит отдельно отметить поддержку конкурентной записи, что реализовано далеко не во всех библиотеках.

Часть IV

Продвинутые возможности

Четвертая (и заключительная) часть книги, как ясно из ее названия, будет посвящена сложным темам, не вошедшим в другие части. Начнем мы с обсуждения преимуществ использования ресурсов графического процессора (GPU) в процессе обработки больших данных. Так получилось, что архитектура графических процессоров идеально подходит для обработки больших объемов данных, особенно если они хранятся в виде многомерных массивов. В заключение этой части мы поговорим о фреймворке Dask на базе Python, позволяющем выполнять параллельные вычисления с использованием множества компьютеров. Это решает проблему горизонтального масштабирования проектов, обрабатывающих действительно огромные данные с помощью сложных алгоритмов.



Анализ данных с использованием графического процессора

В этой главе мы обсудим следующие темы:

- использование архитектуры GPU для повышения эффективности алгоритмов анализа данных;
- применение JIT-компилятора Numba для преобразования кода на Python в эффективный низкоуровневый код для GPU;
- написание высокопараллельного кода GPU для работы с матрицами;
- использование родных для GPU библиотек для анализа данных в Python.

Графические процессоры (graphics processing unit – GPU) изначально задумывались для обработки графической информации в приложениях для рисования, создания анимации и компьютерного моделирования, а также – как без них? – в компьютерных играх.

В какой-то момент стало очевидно, что графические процессоры способны выполнять не только графическую обработку, но и любые другие вычисления. Таким образом появилось понятие *графи-*

ческих процессоров общего назначения (general-purpose computing on graphics processing units – *GPGPU*). В целом графические процессоры являются более перспективными в плане вычислений устройствами по сравнению с центральными процессорами по причине больших вычислительных мощностей. Современные GPU активно используются во многих приложениях в области научных вычислений и искусственного интеллекта. Также они получили широкое распространение в сфере науки о данных и в целом вносят весомый вклад в повышение быстродействия компьютеров.

В графических процессорах общего назначения используются архитектура и парадигма программирования, обусловленные аппаратными возможностями GPU и построенные на двух основных предположениях. Во-первых, графические процессоры должны быть способны выполнять огромное количество вычислений, поскольку в среде, связанной с графикой, очень много данных. Во-вторых, они должны уметь выполнять большое количество однотипных действий параллельно, поскольку все пиксели на экране необходимо обновлять одновременно. Эти требования серьезно повлияли на внутреннюю архитектуру графических процессоров. Например, такие процессоры обладают огромным количеством ядер, счет которых идет на тысячи и которые по большей части занимаются выполнением похожих задач в параллельном режиме. Для сравнения, типичные центральные процессоры насчитывают всего несколько ядер, работающих параллельно, каждое из которых занято своей задачей. Общая скорость обработки данных GPU обусловлена именно большим количеством ядер. При этом по мощности ядра графического процессора серьезно уступают ядрам центрального. Таким образом, главным активом графических процессоров является параллельность вычислений.

Все перечисленные выше отличия между двумя архитектурами обуславливают тот факт, что программирование под графический процессор сильно отличается от программирования под центральный. И одной перекомпиляцией кода здесь не ограничишься. Программирование под GPU, по крайней мере когда мы явно ставим себе такую цель, требует полного слома привычной парадигмы.

Привлечение графического процессора к интенсивным вычислениям в области анализа данных – вполне очевидная и выгодная мера, однако многие разработчики, столкнувшись с необходимостью менять свой подход к программированию, малодушно отказываются от идеи научиться чему-то новому и прогрессивному. В этой главе главные акценты будут сделаны на том, как изменить свое сознание и мышление, чтобы научиться писать эффективный код для графического процессора. В отличие от остальных глав этой книги, для чтения которых вам нужна была определенная подготовка, здесь мы в основном сосредоточимся на базовых вещах, необходимых для слома привычной парадигмы. Именно по этой причине все

примеры будут максимально упрощены, а сложные подробности – опущены. Мы не будем касаться таких продвинутых аспектов, как синхронизация потоков, а будем решать довольно простые задачи на параллельные вычисления, чтобы вы могли полностью сосредоточиться на новой для вас концепции. Многие задачи из области науки о данных довольно легко распараллеливаются, так что материал, изложенный в этой главе, хорошо подойдет для этой сферы.

СОВЕТ. Подробную информацию о вычислениях с использованием графического процессора вы можете почерпнуть из сторонних источников. Я бы порекомендовал руководство для программистов CUDA C++ от NVIDIA по адресу <http://mng.bz/61Bp> (<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>). Хотя это руководство посвящено языкам C и C++, первые четыре главы помогут вам понять в целом, как устроена архитектура и программная концепция графических процессоров. Кроме того, третья часть книги «Parallel and High Performance Computing»¹ авторов Боба Роби (Bob Robey) и Юлианы Замора (Yuliana Zamora), выпущенной издательством Manning в 2021 году, посвящена графическим процессорам. Главу 9 «Архитектуры и концепции GPU» вы можете прочитать бесплатно по адресу <http://mng.bz/oJ6y> (<https://livebook.manning.com/book/parallel-and-high-performance-computing/chapter-9/point-16671-1-351-1>).

Начнем мы со взгляда на архитектуру графических процессоров и их влияние на алгоритмы и процедуру разработки программного обеспечения в целом. При этом мы будем исходить из того, что у вас нет никаких специальных знаний в этой области, а просто покажем, как работает графический процессор.

Поскольку код, написанный на Python, не может быть напрямую запущен на GPU, мы будем использовать компилятор Numba, на лету транслирующий язык Python в низкоуровневую реализацию, совместимую как с центральным, так и с графическим процессором. С введением в компилятор Numba вы можете ознакомиться в приложении Б. В наших примерах мы будем явным образом разворачивать свои приложения на Python для GPU.

После того как мы разберемся с вопросами, касающимися аппаратной части GPU, которые являются фундаментом для понимания программной модели, мы приступим к использованию высокоуровневых библиотек для анализа данных. Хотя вы можете напрямую писать инструкции, которые будут обрабатываться графическим процессором, вам также будут доступны библиотеки для доступа к GPU, берущие на себя многие детали реализации. В данном случае мы будем обращаться к графическому процессору

¹ Роби Р., Замора Дж. Параллельные и высокопроизводительные вычисления / пер. с англ. А. Логунов. М.: ДМК Пресс, 2021, 800 с.

неявно, посредством внешних библиотек. Например, мы заменим библиотеку NumPy на CuPy. Однако, несмотря на то что использование библиотек позволяет снять большинство сложностей при реализации обращений к графическому процессору, просто поменять одни библиотеки на другие будет недостаточно. Давайте для начала познакомимся с архитектурой графического процессора с точки зрения необходимости изменения парадигмы программирования и производительности.

ПРИМЕЧАНИЕ. В этой главе мы будем работать с графическими процессорами NVIDIA с микроархитектурой Pascal и выше. Таким образом, мы устанавливаем привязку к конкретному производителю графического чипа. Конечно, мне бы хотелось сделать обзор без этого ограничения, но реальность такова, что в большинстве случаев вычисления при помощи GPU выполняются именно на процессорах от NVIDIA с поддержкой программно-аппаратной архитектуры CUDA. Это особенно важно в экосистеме Python с такими библиотеками, как CuPy или cuDF. Если вы хотите исследовать вопросы применения графических процессоров в вычислениях без привязки к производителю, обратитесь к платформам OpenCL (<https://www.khronos.org/opencl>) и Vulkan (<https://www.vulkan.org>).

Перед чтением главы вам необходимо убедиться, что у вас установлены все необходимые драйверы NVIDIA для производства вычислений при помощи графического процессора. Вам нужно установить набор инструментов *CUDA*, а также библиотеку *CuPy*. При использовании *conda* это можно сделать с помощью команды `conda install -c rapidsai -c nvidia -c numba -c conda-forge cupy cuda-toolkit`. Образ Docker для вычислений при помощи графического процессора находится в репозитории `tiagoantao/python-performance`.

9.1. Предпосылки для использования вычислительных мощностей GPU

Графический процессор на несколько порядков лучше справляется с задачами, относящимися к целому классу алгоритмов, чем центральный. В этом разделе мы познакомимся с архитектурой GPU и узнаем, когда именно необходимо задействовать графические ресурсы при выполнении вычислений.

9.1.1. Преимущества использования графического процессора

Для понимания того, почему графические процессоры при решении определенного спектра задач показывают столь высокие результаты, мы рассмотрим конкретный пример, на котором проиллюстрируем разницу в подходах между центральным и графическим процессором.

Представьте, что у вас есть массив из ста элементов, и вам нужно вернуть массив с удвоенными значениями из исходного:

```
import numpy as np
a = np.ones(100)
b = np.empty(100)
for i in range(100):
    b[i] = 2 * a[i]
```

Если вы будете решать эту задачу на машине с примитивным центральным процессором с одним потоком и одним вычислительным ядром, то псевдокод на ассемблере получится примерно следующий:

TMPVAR = A[0]	←		Получаем первый элемент из массива A и помещаем его в регистр TMPVAR
TMPVAR = 2 * TMPVAR	←		
B[0] = TMPVAR	←		Удваиваем значение в регистре
TMPVAR = A[1]			
TMPVAR = 2 * TMPVAR			Помещаем значение из регистра в первый элемент массива B
B[1] = TMPVAR			
...			
TMPVAR = A[99]			
TMPVAR = 2 * TMPVAR			
B[99] = TMPVAR			

В показанном выше псевдокоде мы берем значение первого элемента массива A, переносим его в регистр, удваиваем и записываем в первую позицию массива B. И эта операция повторяется сто раз для всех элементов в массиве.

Если вы помните, в главе 6 мы говорили, что извлечение значений из основной памяти – дело очень трудоемкое и дорогостоящее. Представим, что у нашего простенького центрального процессора отсутствует кеш. Также допустим, что операции чтения и записи (TMPVAR = A[0] и B[0] = TMPVAR) занимают по 90 временных единиц, а операция удвоения (TMPVAR = 2 * TMPVAR) – всего 2 временные единицы. Итого у нас есть по 100 операций чтения и записи плюс 100 операций удвоения, получаем $100 \cdot 90 + 100 \cdot 90 + 100 \cdot 2 = 18\,200$ единиц. Также помните, что наш простой процессор работает в последовательном режиме, так что очередная операция может начаться только тогда, когда завершится предыдущая.

Теперь представьте совершенно иную модель выполнения, в которой у вас 100 потоков запущены параллельно, и каждый из них может выполнить операцию чтения, удвоить значение и записать его. Предположим, что временные затраты на операции чтения и записи у нас прежние. В то же время на удвоение значения нам теперь требуется больше времени, поскольку у нас есть несколько вычислительных блоков, и каждый из них значительно уступает прежнему единому процессору. Скажем, удвоение теперь выполняется за 40 временных единиц.

Получается, что все потоки могут выполнить чтение одновременно. Через 90 временных единиц все потоки получают свои значения для удвоения и потратят 40 временных единиц на выполнение этой операции. Помните, что все потоки у нас работают параллельно и независимо друг от друга. Наконец, они одновременно осуществляют запись данных, на что требуется еще 90 временных единиц. Получается $90 + 40 + 90 = 220$ единиц.

Таким образом, если наш условный простой центральный процессор справился с этой задачей за 18 200 временных единиц, то графический – всего за 220, что более чем в 80 раз меньше. В то же время если бы вам потребовалось выполнить эту операцию применительно к одному элементу, у центрального процессора ушло бы на это $90 + 90 + 2 = 182$ временных единицы, тогда как графическому потребовались бы те же 220 единиц.

Этот пример наглядно демонстрирует пользу и вред от использования вычислительных мощностей графического процессора. По сути, применение GPU может быть оправдано при наличии задержек при обращении к памяти и необходимости выполнять одни и те же операции применительно к большим массивам данных. Вместе с тем пользоваться услугами графического процессора для произведения одной операции не имеет смысла. Если перейти на язык метафор, то центральный процессор можно сравнить с Ferrari, а графический – с автобусом. Если вам нужно перевезти из места в место 5 человек, Ferrari – прекрасный выбор, а если надо транспортировать 500 человек? Выбор очевиден.

Одна из сложностей, с которыми сталкиваются разработчики при изучении или реализации проектов с использованием графического процессора, состоит в преодолении собственного интуитивного представления о том, что большая часть кода выполняется последовательно. Да, много кода выполняется последовательно, но вместе с тем наиболее ресурсоемкие фрагменты всегда стараются сделать параллельными. Наиболее очевидным примером здесь может быть отображение пикселей на экране. На мониторе с разрешением 1920×1080 располагается порядка 2 млн пикселей. При этом каждый пиксель обрабатывается независимо, а значит, по крайней мере, в теории мы можем управлять ими всеми в параллельном режиме. Другой пример – это многомерные массивы, представляющие собой типы структур, с которыми активно работают в науке о данных. Каждый элемент массива может быть обработан отдельно, а значит, все элементы в идеале могут быть вычислены параллельно. Как видите, графические процессоры никогда не останутся без работы.

Чтобы лучше понять, почему для подобных задач лучше подходят именно графические процессоры, необходимо поближе познакомиться с их внутренней архитектурой. И мы сделаем это прямо сейчас.

9.1.2. Связь между центральным и графическим процессорами

Вычислительная модель графического процессора сильно отличается от модели центрального. Чтобы иметь возможность эффективно программировать под GPU – и вообще программировать под него – необходимо хорошо понимать его архитектуру со всеми отличительными особенностями.

В действительности графический процессор является *сопроцессором* (co-processor). Всем заправляет центральный процессор, что ясно из его названия, и код для него управляет верхнеуровневыми вычислениями. Существующая применительно к вычислениям с помощью GPU терминология хорошо объясняет связь между процессорами: в ней *хостом* (host) именуется центральный процессор, а графический получил термин *устройство* (device). И всем вычислительным процессом управляет код хоста.

Наиболее важным аспектом с точки зрения производительности здесь является то, что в подавляющем большинстве архитектур центральный и графический процессоры имеют раздельную память, никак не связанную между собой. Таким образом, у нас есть память [доступная для] хоста и память [доступная для] устройства.

Расходы на перемещение данных в память графического процессора могут быть очень существенными и не окупиться, если GPU предстоит выполнить не так много вычислительных операций. На рис. 9.1 схематически показана связь между процессорами.

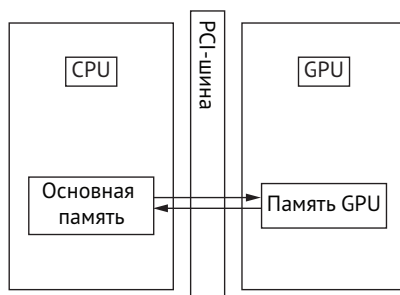


Рис. 9.1. CPU и GPU наделены своей обособленной памятью. Для произведения вычислений нам нужно перебросить данные в память GPU, а затем извлечь из нее результаты, что может занять немало времени

Производители GPU и совместимость программного обеспечения

Перед тем как углубиться в дебри внутренней архитектуры графического процессора, хотелось бы сказать пару слов о производителях чипов GPU и возможностях для адаптации программного обеспечения. На текущий момент существует два основных производителя графических процессоров: NVIDIA и AMD. В теории вы можете использовать для на-

писания кода интерфейсы, не зависящие от производителя. Для этого вам необходимо обратить внимание на такие программные решения, как OpenCL или Vulkan.

На практике же графические процессоры NVIDIA с огромным преимуществом доминируют в отношении использования для вычислений общего назначения. Это легко увидеть, если проследить за низкоуровневыми решениями, рассчитанными на GPU, большинство из которых привязано к архитектуре *Compute Unified Device Architecture (CUDA)* от NVIDIA. Помимо этого, многие библиотеки Python с поддержкой вычислений с помощью графических процессоров, такие как CuPy, CuDF, cuML и BlazingSQL, также основываются на технологии CUDA.

В этой главе мы будем рассматривать только решения на базе NVIDIA/CUDA. В то же время концептуально все сказанное может быть применено и к технологии AMD.

Еще одна проблема, проистекающая из наличия нескольких ключевых поставщиков графических процессоров, лежит в области терминологии. Дело в том, что термины из мира, абстрагирующегося от производителя, могут не совпадать с терминами от NVIDIA. Я буду по возможности использовать терминологию в отрыве от производителя, но также буду обращаться к терминам от NVIDIA, получившим большое распространение. Проблема с именованиями усложняется еще и тем, что, помимо терминов, введенных конкретными поставщиками, используются слова, берущие свое начало от истоков графических процессоров. Например, термин ядро CUDA также относится к потоковому процессору (не путайте с потоковым мультипроцессором) и шейдеру (shader).

9.1.3. Внутренняя архитектура графического процессора

В состав графического процессора входит несколько *потоковых мультипроцессоров* (streaming multiprocessor – SM). Их количество обычно варьируется от 1 до 30, а иногда и выше. Каждый потоковый мультипроцессор состоит из множества *потоковых процессоров* (streaming processor – SP), зачастую называемых *ядрами CUDA* (CUDA core). На рис. 9.2 в упрощенном виде показана схема основных компонентов графического процессора.

К примеру, если взять графическую карту NVIDIA RTX 2070, основанную на чипе NVIDIA Turing 106, то мы увидим, что на ней располагаются 36 потоковых мультипроцессоров, в каждом из которых по 64 ядра CUDA. В сумме получается 2304 ядра CUDA. Из этого можно сделать вывод о том, что этот графический процессор способен параллельно запустить 2304 отдельных потока. Как я уже говорил, мы будем допускать определенные упрощения, поскольку в действительности архитектура GPU на самом деле куда сложнее. В частности, для специалистов в области науки о данных важным фактором является наличие в графическом процессоре *тензорных ядер* (tensor core), которые могут быть использованы для ускорения функций ис-

кусственного интеллекта. Мы не будем затрагивать эти темы в данной книге, но вы при желании можете изучить их самостоятельно.

Организация памяти – также очень важный момент. Каждый потоковый мультипроцессор обладает собственным кешем *первого уровня* (L1 cache)¹, который может быть совместно использован всеми потоковыми процессорами в рамках данного мультипроцессора. Мы не будем обращаться к этому кешу напрямую, но он может быть использован для обмена состояниями между потоками, запущенными параллельно в одном мультипроцессоре. Также графический процессор располагает кешем *второго уровня* (L2 cache), доступ к которому есть у всех потоковых мультипроцессоров, и *основной памятью* (main memory). Тот же чип TU 106 располагает объемом кеша первого уровня, равным 64 Кб, и 4 Мб кеша второго уровня. Помимо этого, видеокарта RTX 2070 снабжена основной памятью объемом 8 Гб.

Как архитектура графических процессоров влияет на подход к программированию под них? Об этом мы узнаем в следующих разделах.

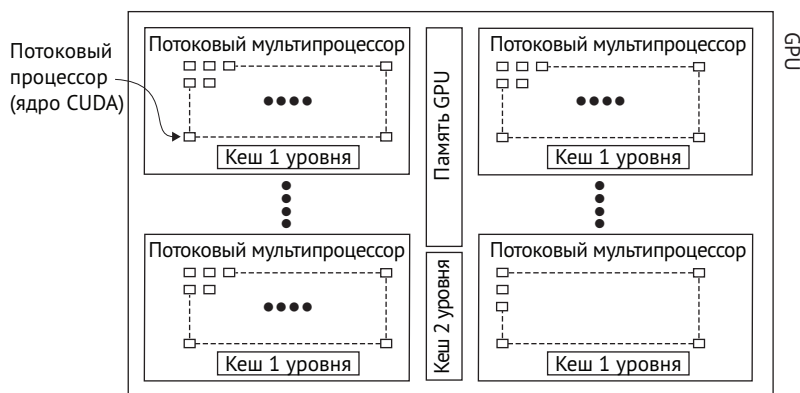


Рис. 9.2. Упрощенная схема архитектуры графического процессора: потоковые мультипроцессоры содержат потоковые процессоры (ядра CUDA) и локальный кеш. GPU включает потоковые мультипроцессоры, дополнительный кеш и основную память

9.1.4. Архитектура программного обеспечения

Теперь давайте посмотрим, как архитектура аппаратного обеспечения графического процессора может оказывать влияние непосредственно на программный код. Мы узнаем, какие шаги необходимо предпринять, чтобы нашу предыдущую задачу с удвоением всех значений в массиве реализовать в рамках графического процессора. Позже в этой главе мы напишем сам код, но сейчас проговорим основные шаги.

¹ Подробно о концепции кеша мы говорили в главе 6.

Итак, у нас есть матрица в памяти недалеко от центрального процессора, так что первое, что нам нужно сделать, это передать ее в память графического процессора. Эта операция может оказаться достаточно дорогой в плане исполнения и не оправдать себя, если нам нечем будет порадовать GPU в плане объема вычислений.

Представим, что у нас есть массив размером 1024×64 (т. е. содержащий 65 536 элементов). В графическом процессоре каждый элемент массива будет обрабатываться в отдельном *потоке* (thread). В результате мы поднимем 65 536 потоков. При этом в каждом потоке будет выполняться одинаковый код. Потоки необходимо будет разбить на *блоки потоков* (thread block) таким образом, чтобы все потоки из одного блока размещались в одном потоковом мультипроцессоре с общей памятью и примитивами синхронизации. В нашем случае потоки не будут использовать какие-то общие данные, поскольку алгоритм задачи самый простой, но без разбивки на блоки потоков нам все же не обойтись.

Если предположить, что в одном блоке будут располагаться 32 потока, нам понадобится 2048 блоков. Каждый блок будет выполняться в рамках одного потокового мультипроцессора.

Так как же мы будем вызывать наш код? Как вы помните, у нас всем управляет центральный процессор, и именно он будет инициировать *точку входа* (entry point) для GPU, который займется всеми вычислениями. Имя этой точки входа – *функция ядра* (kernel function). Итак, на данный момент мы знаем о программировании для GPU лишь то, что у нас должна быть некая точка входа – функция ядра, и в множестве потоков должен выполняться один и тот же код.

Теперь нам необходимо написать низкоуровневый код для запуска на графическом процессоре. Поскольку не существует аналога Cython для преобразования кода на Python в OpenCL C (или CUDA C), мы будем использовать компилятор Numba. Если вы никогда не использовали его, подробности найдете в приложении Б.

9.2. Использование компилятора Numba для генерации кода под GPU

После некоторых базовых приготовлений мы готовы приступить к написанию своей первой программы для графического процессора с помощью компилятора Numba. Для понимания основных проблем, связанных с программированием под GPU, мы начнем с нашего простого примера с удвоением значений в массиве. После этого мы реализуем генератор Мандельброта, который вы сможете сравнить с реализацией под центральный процессор в приложении Б. Опять же, если вы раньше не сталкивались с Numba, сначала вы можете ознакомиться с этим приложением.

9.2.1. Программное обеспечение для работы с GPU в Python

Перед написанием кода для графического процессора необходимо убедиться в том, что все нужные драйверы и программы для GPU установлены. А установка всего необходимого не всегда проходит гладко. Мы не сможем дать подробную инструкцию для всех операционных систем, но какие-то основные моменты проговорим.

Вероятно, первое, что вам нужно будет сделать, это установить драйверы ядра и перезагрузить компьютер. Также вам понадобится набор инструментов CUDA (CUDA toolkit), который поставляется в разных видах. Если вы используете Anaconda, самым простым способом установить его будет запуск команды `conda install cudatoolkit`.

У компилятора Numba есть возможность проинспектировать вашу инфраструктуру и дать отчет об имеющемся аппаратном обеспечении и библиотеках. Для этого можно в оболочке выполнить следующую команду:

```
numba -s
```

В результате вы получите детализированный отчет о вашей системе. Для определения того, доступен ли вам GPU, необходимо посмотреть, определилось ли аппаратное обеспечение и есть ли нужные библиотеки. По аппаратному обеспечению вам нужно найти показанный ниже раздел в отчете:

```
__CUDA Information__
CUDA Device Initialized           : True
CUDA Driver Version               : 11020
CUDA Detect Output:
Found 1 CUDA devices
id 0                             b'Tesla T4'           [SUPPORTED]
                                compute capability: 7.5
                                pci device id: 30
                                pci bus id: 0

Summary:
    1/1 devices are supported
```

Этот фрагмент позволяет понять, установлено ли устройство и поддерживается ли оно. Некоторые старые модели GPU могут не поддерживаться. Также вам нужно убедиться, что все библиотеки были обнаружены. Это прописано в следующем фрагменте отчета:

```
CUDA Libraries Test Output:
Finding cublas from Conda environment
    named libcublas.so.11.2.0.252
    trying to open library...           ok
Finding cusparse from Conda environment
    named libcusparse.so.11.1.1.245
    trying to open library...           ok
```

```

Finding cufft from Conda environment
  named libcufft.so.10.2.1.245
  trying to open library...           ok
Finding curand from Conda environment
  named libcurand.so.10.2.1.245
  trying to open library...           ok
Finding nvvm from Conda environment
  named libnvvm.so.3.3.0
  trying to open library...           ok
Finding libdevice from Conda environment
  searching for compute_20...         ok
  searching for compute_30...         ok
  searching for compute_35...         ok
  searching for compute_50...         ok

```

Этот вывод поможет вам выявить все возможные проблемы с библиотеками. Теперь давайте напишем наш первый код для графического процессора.

9.2.2. Основы программирования для GPU с помощью Numba

Перед тем как начать писать наш код, давайте посмотрим на то, как *не* надо это делать. Если вы помните, наша задача состоит в том, чтобы просто удвоить все значения в массиве. Ниже приведен стандартный код, который будет выполнен центральным процессором:

```

def double_not_this(my_array):
    for position in range(my_array):
        my_array[position] *= 2

```

for – это последовательная операция

Здесь мы просто последовательно проходим по массиву и удваиваем значения. Но если мы говорим о графическом процессоре, то в нем для каждого элемента будет создан отдельный поток, так что наш код должен производить действие только с одним элементом. Позже мы научим наш графический процессор применять этот код ко всем элементам в массиве. Ниже приведена первая версия кода:

```

from numba import cuda

@cuda.jit
def double(my_array):
    position = cuda.grid(1)
    my_array[position] *= 2

```

Компилируем функцию в CUDA

cuda.grid осуществляет доступ к текущей позиции в массиве, которая и должна быть обработана

Никакого вывода у этой функции нет, поскольку она будет выполняться графическим процессором.

Да, эта функция обрабатывает лишь один элемент! И такой подход сильно отличается от всего, что мы видели раньше, если не считать векторизации.

Также мы разместили перед нашей функцией декоратор `@cuda.jit`, тем самым попросив компилятор Numba сгенерировать версию нашего кода для CUDA. Кроме того, мы воспользовались магической функцией `cuda.grid` для извлечения единственной позиции в массиве, значение которой мы будем изменять. Позже мы увидим, как это работает. Наконец, изменили значение элемента массива, основываясь на позиции.

Наша функция не может ничего возвращать, поскольку она будет реализована в виде функции ядра GPU, так что нам необходимо передавать ей параметры для получения возвращаемых значений. Попробуем выполнить следующий код:

```
my_array = np.ones(1000)
double(my_array)
```

В результате мы получим ошибку такого вида:

```
Kernel launch configuration was not specified. Use the syntax:
kernel_function[blockspergrid, threadsperblock](arg0, arg1, ..., argn)
```

Ошибка возникла из-за того, что мы не сообщили компилятору Numba, как именно собираемся распределять наши вычисления. Как мы уже говорили в разделе, посвященном архитектуре графического процессора, необходимо разбить вычисление на блоки, а каждый блок – на потоки. Вот так работает нормально:

```
import numpy as np

blocks_per_grid = 50
threads_per_block = 20

my_array = np.ones(1000)
double[blocks_per_grid, threads_per_block](my_array)
assert (my_array == 2).all()
```

Синтаксис
вызова функ-
ции получился
не самым есте-
ственным

Проверка на то, что функция была при-
менена ко всем элементам в массиве

Обратите внимание, что синтаксис вызова функции вышел не очень привычным. В последней строке кода мы проверяем с помощью функции `assert`, что все элементы в массиве стали равны двум. При этом следует быть очень внимательным: если мы передадим неправильное количество блоков, не все значения могут быть удвоены.

Мы используем 20 потоков в каждом блоке, и, поскольку всего в нашем массиве 1000 элементов, нам понадобится 50 блоков. В основном принято использовать 32 потока на блок. С учетом организации памяти в графическом процессоре все потоки в рамках одного блока могут очень быстро обмениваться состояниями. Здесь мы

не будем рассматривать более продвинутые алгоритмы, в которых это может применяться. Таким образом, как вы видите, мы можем довольно гибко распределять вычисления между ядрами графического процессора.

Иногда, однако, невозможно получить точное количество элементов в массиве путем перемножения количества блоков и потоков – например, если количество значений в массиве соответствует простому числу. В таких случаях необходимо убедиться, что произведение количества блоков и потоков будет превышать количество элементов в обрабатываемом массиве. Ниже приведен пример:

```
threads_per_block = 16
blocks_per_grid = 63

my_array = np.ones(1000)
double[blocks_per_grid, threads_per_block](my_array)
assert (my_array == 2).all()
```

Здесь мы получим 1008 потоков (16×63). Если вам повезет, этот код сработает нормально. Но велика вероятность, что он завершится ошибкой.

В данном случае мы вызываем наш код для позиций в массиве с нулевой по 1007-ю, тогда как последние восемь позиций (с 1000-й по 1007-ю) находятся за пределами массива. В этот момент вам нужно забыть о том, что вы программируете на языке Python, и вспомнить, что написанный вами код был сконвертирован в язык низкого уровня. Это означает, что все стандартные проверки границ диапазонов, применяемые в Python, будут вам недоступны, и в качестве неожиданного сюрприза вы можете получить ошибку выделения памяти или, что еще хуже, беззвучную ошибку. Позже мы рассмотрим такой пример.

Исправить возникшую проблему очень просто:

```
@cuda.jit
def double_safe(my_array):
    position = cuda.grid(1)
    if position > my_array.shape[0]:
        return
    my_array[position] *= 2
```

Мы проверяем, превышает ли текущая позиция размер массива, и в этом случае осуществляем выход из функции. Теперь можем уверенно запускать наш код:

```
my_array = np.ones(1000)
double_safe[blocks_per_grid, threads_per_block](my_array)
assert (my_array == 2).all()
```

Свершилось! Мы выполнили наш код с использованием ресурсов графического процессора!

Теперь давайте вернемся к магической функции `cuda.grid`, позволяющей определить позицию для вычисления. Мы постараемся понять, как она работает, путем явного обращения к некоторым полезным свойствам. Иногда есть необходимость все держать под контролем и осуществлять доступ к нужным атрибутам напрямую, например при работе с массивами, содержащими более трех измерений:

```
@cuda.jit
def double_safe_explicit(my_array):
    position = cuda.blockIdx.x * cuda.blockDim.x + cuda.threadIdx.x
    if position >= my_array.shape[0]:
        return
    my_array[position] *= 2
```

Вызов в потоке позволяет получить доступ к номеру блока и потока, в которых выполняется код. Также можно узнать размер блока. Свойство `cuda.blockIdx` содержит индекс блока, в котором запущен текущий поток. Свойство `cuda.blockDim` дает доступ к размеру текущего блока, а `cuda.threadIdx` – к индексу потока внутри блока. Обладая всей этой информацией, вы можете с уверенностью сказать, что каждый поток адресует свою позицию в массиве.

Информация о расположении элемента для потока может быть одно-, дву- и трехмерной. Вероятно, вы заметили, что во всех трех вызовах в предыдущем коде мы обращались к параметру `.x`. Если вы имеете дело с двумерными или трехмерными массивами, можете воспользоваться параметрами `.y` и `.z`.

Теперь давайте взглянем на ту же функцию, способную работать с двумерными массивами:

```
@cuda.jit
def double_matrix_unsafe(my_matrix):
    x, y = cuda.grid(2)
    my_matrix[y, x] *= 2
```

Здесь мы вызываем функцию `cuda.grid(2)` для получения двух индексов. Обратите внимание, что мы вернулись к небезопасному коду, поскольку можем быть уверены, что у нас возникнет ошибка. Давайте запустим приведенный ниже код:

```
threads_per_block_2d = 16, 16
blocks_per_grid_2d = 63, 63

my_matrix = np.ones((1000, 1000))
double_matrix_unsafe[blocks_per_grid_2d, threads_per_block_2d](my_matrix)
print((my_matrix == 2).all())
```

Результатом выполнения этого кода будет вывод значения True, поскольку все элементы массива теперь содержат значение 2. Обратите внимание, что определения блоков и потоков стали двумерными, как и наш массив данных.

Если вы запустите этот код и вам повезет, то ошибка не появится. Но и результат вы наверняка увидите неправильный. Причина этого в том, что мы не выполняем проверку на границы массивов, и при выходе за границы одной строки переходим на следующую, что было невозможно в случае с одномерным массивом. Таким образом, в определенных позициях в массиве у нас могут оказаться не двойки, а четверки, поскольку код в них будет выполняться дважды.

Исправить это довольно просто. Ниже представлена окончательная версия с явным индексированием массивов:

```
@cuda.jit
def double_matrix(my_matrix):
    x = cuda.blockIdx.x * cuda.blockDim.x + cuda.threadIdx.x
    y = cuda.blockIdx.y * cuda.blockDim.y + cuda.threadIdx.y
    if x >= my_matrix.shape[0]:
        return
    if y >= my_matrix.shape[1]:
        return
    my_matrix[y, x] *= 2
```

Теперь, когда вы освоили основы вычислений с помощью графического процессора, давайте воссоздадим генератор Мандельброта с использованием GPU.

9.2.3. Создание генератора Мандельброта с помощью графического процессора

Мы уже познакомились почти со всеми концепциями компилятора Numba, так что можем попробовать воссоздать генератор Мандельброта с использованием ресурсов графического процессора. Чтобы собрать все концепции воедино и построить визуализацию, мы пойдем обходным путем – будем выполнять на первый взгляд логичные действия, но сами себя при этом заводить в тупик. Параллельно мы будем объяснять, почему те или иные шаги не сработали, чтобы в будущем вы смогли воспользоваться по назначению нашими граблями и не опирались только на свою интуицию, которая далеко не всегда является лучшим помощником.

Начнем с реализации *функции Мандельброта* (Mandelbrot function) для расчета значения в конкретной точке:

```
from numba import cuda

@cuda.jit(device=True)
```

```
def compute_point(c):  
    i = -1  
    z = complex(0, 0)  
    while abs(z) < 2:  
        i += 1  
        if i == 255:  
            break  
        z = z**2 + c  
    return 255 - (255 * i)
```

Обратите внимание, что к декоратору `@cuda.jit` мы добавили параметр `device=True`. Таким образом мы говорим компилятору Numba о том, что функция должна быть вызвана изнутри устройства. Функция устройства, в отличие от функции ядра, может возвращать значения.

Теперь давайте реализуем первую версию вызывающего кода – внешне логичную, но ошибочную:

```
@cuda.jit  
def compute_all_points_doesnt_work(start, end, size, img_array):  
    x, y = cuda.grid(2)  
    if x >= img_array.shape[0] or y >= img_array.shape[1]:  
        return  
    mandel_x = (end[0] - start[0])*(x/size) + start[0]  
    mandel_y = (end[1] - start[1])*(y/size) + start[1]  
    img_array[y, x] = compute_point(complex(mandel_x, mandel_y))
```

Несмотря на успешную компиляцию этого кода, при попытке его вызова вы получите следующую ошибку:

```
NotImplementedError: (UniTuple(float64 x 2), (-1.5, -1.3))
```

Проблема в том, что компилятор Numba не умеет обрабатывать в качестве входных параметров кортежи (по крайней мере пока). Это отсылает нас к более пространным размышлениям о том, что далеко не весь функционал, реализованный в Python, поддерживается в Numba. У вас всегда должна быть под рукой документация по этому компилятору, расположенная по адресу <http://numba.pydata.org>, чтобы вы могли быстро проверять, поддерживаются ли те или иные функции. Сейчас нет никакого смысла говорить о том, какие именно функции не реализованы в Numba, поскольку этот компилятор постоянно развивается, и функции, о которых мы могли здесь упомянуть, вполне вероятно могут быть воплощены в очередной версии Numba еще до выхода этой книги.

Итак, давайте избавимся в нашем решении от кортежей в качестве входных параметров:


```
@cuda.jit
def compute_all_points(startx, starty, endx, endy, size, img_array):
    x, y = cuda.grid(2)
    if x >= img_array.shape[0] or y >= img_array.shape[1]:
        return
    mandel_x = (end[0] - startx)*(x/size) + startx
    mandel_y = (end[1] - starty)*(y/size) + starty
    img_array[y, x] = compute_point(complex(mandel_x, mandel_y))
```

Сделаем лишь одно замечание по поводу последней строки кода: если помните, при работе с массивами NumPy первой идет координата *y*, поэтому мы написали `img_array[y, x]`.

Теперь давайте вызовем нашу функцию:

```
from math import ceil
import numpy as np
from PIL import Image

size = 2000
start = -1.5, -1.3
end = 0.5, 1.3

img_array = np.empty((size, size), dtype=np.uint8)
threads_per_block_2d = 16, 16
blocks_per_grid_2d = ceil(size / 16), ceil(size / 16)

compute_all_points(blocks_per_grid_2d,
    threads_per_block_2d)(start[0], start[1], end[0], end[1],
    size, img_array)

img = Image.fromarray(img_array, mode="P")
img.save("mandelbrot.png")
```

Здесь можно обратить внимание на то, как объявляется переменная для хранения количества блоков. С учетом того, что мы используем 16 потоков на каждом блоке для каждого измерения, размер мы должны вычислять как `size/16` с округлением результата в большую сторону.

Давайте выполним замер времени для этого вызова:

```
In [3]: %timeit compute_all_points(blocks_per_grid_2d, ...
72.6 ms ± 50.4 μs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Для сравнения, лучшая версия для центрального процессора из Приложения Б показала результат 539 мс. Но, признаться, это не самое честное соревнование, поскольку силы медленного CPU и быстрого GPU изначально не равны. Более того, здесь есть и другие факторы, такие как используемый алгоритм и необходимость передачи данных между центральным и графическим процессором, оказывающие существенное влияние на быстродействие. Несмотря на это, в целом должно быть понятно, что некоторые алгоритмы при запуске на графическом процессоре работают быстрее, чем на центральном.

Создав свой первый генератор Мандельброта с использованием GPU, мы получили неплохую прибавку в скорости. Теперь давайте применим иной подход. На этот раз мы создадим генератор с применением векторизации в NumPy, которая, как мы видели ранее, помогает ускорять алгоритмы при анализе данных.

9.2.4. Создание генератора Мандельброта с помощью NumPy

Наша заключительная версия будет основываться на универсальной функции NumPy, запускаемой на графическом процессоре. Мы уже видели все части по отдельности, так что нам будет несложно собрать готовое решение. Ниже показана функция вычисления точки вместе с векторизованной версией вызова:

```
from cuda import vectorize

size = 2000
start = -1.5, -1.3
end = 0.5, 1.3

def compute_point_255_fn(c):
    i = -1
    z = complex(0, 0)
    while abs(z) < 2:
        i += 1
        if i == 255:  ← Мы будем использовать упрощенную версию
                       расчета точек с жестко заданным пределом
            break
        z = z**2 + c
    return 255 - (255 * i) // 255

compute_point_vectorized = vectorize(
    ["uint8(complex128)"], target="cuda")(compute_point_255_fn)
```

Единственной новинкой здесь является использование параметра `target="cuda"` при вызове функции `vectorize` в последней строке кода.

Как вы помните из предыдущего раздела, нам необходимо подготовить массив с позициями, для которых необходимо выполнить вычисление:

```
def prepare_pos_array(start, end, pos_array):
    size = pos_array.shape[0]
    startx, starty = start
    endx, endy = end
    for xp in range(size):
        x = (endx - startx)*(xp/size) + startx
        for yp in range(size):
            y = (endy - starty)*(yp/size) + starty
            pos_array[yp, xp] = complex(x, y)

pos_array = np.empty((size, size), dtype=np.complex128)
img_array = np.empty((size, size), dtype=np.uint8)
```

Теперь можно осуществить замер времени для этой версии:

```
In [6]: %timeit compute_point_vectorized(pos_array)
222 ms ± 3.05 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Скорость оказалась ниже, чем в предыдущей версии для графического процессора, но все же лучше, чем в версии для центрального. При этом шаблон здесь отличается от версии для CPU, в самой быстрой из которых использовалась универсальная функция.

Функционал NumPy оказывается ограничен со стороны CUDA из-за вычислительной модели. Было бы куда лучше, если бы для GPU существовала родная реализация библиотеки NumPy. Знакомьтесь, CuPy!

9.3. Анализ производительности кода для GPU: приложение с использованием CuPy

В данном разделе мы реализуем решение с использованием родной для графического процессора версии библиотеки NumPy под названием CuPy.

ПРИМЕЧАНИЕ. Многие библиотеки для анализа данных, заточенные под центральный процессор, имеют свои аналоги для GPU. С помощью них вы можете взаимодействовать с графическим процессором, не имея глубоких практических знаний о его устройстве. Далее мы приведем список библиотек для анализа данных, предназначенных для работы с графическим процессором.

После создания решения на базе библиотеки CuPy мы обсудим вопросы, связанные с профилированием кода для графического процессора. Наш пример с использованием CuPy позволит нам познакомиться с некоторыми инструментами для оценки производительности кода для GPU и выполнения профилирования. Но прежде давайте узнаем, какие основные библиотеки в области науки о данных применяются для выполнения вычислений с помощью графической подсистемы.

9.3.1. Библиотеки для анализа данных на базе GPU

Если у вас есть доступ к графическому процессору, это еще не значит, что вам нужно программировать для него с чистого листа. В Python существует несколько библиотек, разработанных исключительно для GPU и являющихся близкими аналогами соответствующих библиотек для центрального процессора с очень похожими интерфейсами. Зачастую, пользуясь этими библиотеками, вам ничего не придется знать о специфике работы графического процессора. В табл. 9.1 перечислены все существующие библиотеки для работы с данными с использованием GPU с указанием их аналогов.

Таблица 9.1. GPU-библиотеки с аналогами для CPU

GPU	CPU	Применение
cuBLAS	BLAS	Базовая линейная алгебра
CuPy	NumPy	Обработка многомерных массивов
CuDF	pandas	Анализ колоночных данных
CuGraph		Алгоритмы на графах для датафреймов
CuML	scikit-learn	Машинное обучение
BlazingSQL		Интерфейсы SQL поверх колоночных данных

Существуют и другие библиотеки для ускорения кода, связанного с анализом данных. К примеру, с помощью библиотеки *cuDNN* вы можете повысить скорость работы библиотек для машинного обучения вроде PyTorch и TensorFlow.

При написании решений для анализа данных на основе графического процессора вы можете активно пользоваться всеми этими библиотеками. Мы в данном разделе воспользуемся одной из них, а именно CuPy.

9.3.2. Использование CuPy – версии библиотеки NumPy для GPU

Итак, в своем решении мы решили применить высокоуровневую библиотеку из области науки о данных, получившую название *CuPy*. Эта библиотека представляет собой версию NumPy для графического процессора. Многие высокоуровневые библиотеки для работы с GPU имеют очень схожие интерфейсы со своими аналогами, разработанными для центрального процессора, и поэтому вам придется изучить не так много новой информации. Кроме того, мы воспользуемся кодом, написанным в этом разделе, для представления инструментов и техник профилирования применительно к GPU. Как вы уже догадались, мы продолжим работать с генераторами Мандельброта и на этот раз напишем свою версию с использованием массивов CuPy.

9.3.3. Базовое взаимодействие с CuPy

Перед созданием очередной версии генератора Мандельброта давайте в общих чертах познакомимся с библиотекой CuPy. Для начала создадим массив размером 5000×5000 и удвоим элементы в нем:

```
import numpy as np
import cupy as cp

size = 5000

my_matrix = cp.ones((size, size), dtype=cp.uint8)
print(type(my_matrix))
```

```
np_matrix = my_matrix.get()
print(type(np_matrix))
```

```
2 * my_matrix
2 * np_matrix
```

Несмотря на сходство интерфейсов, CuPy и NumPy представляют собой разные библиотеки и работают с объектами разных типов. Зачастую бывает полезно импортировать обе библиотеки при проведении расширенного анализа данных.

Типом данных для массива в переменной `my_matrix` является `cupy.core.core.ndarray`, тогда как для массива в `np_matrix` тип данных – `numpy.ndarray`. При этом данные в переменной `my_matrix` физически размещаются в памяти графического процессора, так что при необходимости выполнения каких-либо действий над ними нам не придется переносить их из памяти центрального процессора в память GPU. К примеру, математическая операция `2 * my_matrix` целиком и полностью выполняется в графическом процессоре. Перенос данных из памяти GPU осуществляется при явном вызове метода `my_matrix.get()`, в результате чего создается независимое представление NumPy для исходного массива.

Базовое профилирование кода, предназначенного для графического процессора, не должно выполняться при помощи привычных для Python инструментов вроде модуля `timeit` или инструкции `%timeit` в IPython. Код для GPU выполняется независимо от кода для CPU, и данные о производительности, собранные способами, типичными для центрального процессора, не будут отражать реальное положение дел.

Библиотека CuPy предоставляет собственный простой механизм для профилирования кода. Давайте запустим операцию `2 * my_matrix` 200 раз и посмотрим на результат:

```
from cupyx.time import repeat
print(repeat(lambda : 2 * my_matrix, n_repeat=200))
```

Вывод на моем компьютере оказался таким:

```
<lambda>      : CPU: 60.910 us +/-14.344
                (min: 19.158 / max: 101.755) us
                GPU-0: 785.708 us +/-12.013
                (min: 749.760 / max: 822.656) us
```

Таким образом, в среднем каждая инструкция заняла порядка 60 мкс времени центрального процессора и 785 мкс времени графического процессора. Я запускал этот код на видеокарте Tesla T4 GPU на компьютере с центральным процессором Intel Xeon с рабочей частотой 2,50 ГГц.

Теперь давайте перейдем к практической реализации нашей новой версии генератора Мандельброта с использованием библиотеки CuPy. Мы не ставим себе цель продемонстрировать интерфейс этой библиотеки, поскольку он по определению приближен к стандартам NumPy. Мы также не будем заострять внимание на ограничениях CuPy по сравнению с NumPy в связи с тем, что баланс сил между ними стремительно меняется и к выходу книги может стать кардинально иным.

Таким образом, при реализации двух следующих версий генератора Мандельброта мы просто будем стараться извлечь максимум возможного из графического процессора. В процессе мы напишем функции обработки, работающие на GPU с использованием библиотеки CuPy. И наш первый пример продемонстрирует взаимодействие CuPy с компилятором Numba.

9.3.4. Создание генератора Мандельброта с помощью Numba

Библиотека CuPy очень органично взаимодействует с компилятором Numba: вы можете написать декорированную Numba функцию и использовать ее совместно с CuPy.

СОВЕТ. CuPy располагает собственным конвертером кода на Python в версию для GPU, который в отдельных случаях способен конкурировать с компилятором Numba. В настоящий момент этот движок поддерживает меньше возможностей Python в сравнении с Numba. Пока я бы советовал преимущественно использовать именно компилятор Numba, хотя со временем, возможно, конвертер CuPy выйдет на первые роли и будет поддерживать весь основной функционал Python.

Ниже приведена реализация генератора Мандельброта, написанная с использованием Numba и CuPy:

```
from math import ceil

import numpy as np
import cupy as cp
from numba import cuda
from PIL import Image

size = 2000
start = -1.5, -1.3
end = 0.5, 1.3

@cuda.jit
def compute_all_mandelbrot(startx, starty, endx, endy, size, img_array):
    x, y = cuda.grid(2)
    if x >= img_array.shape[0] or y >= img_array.shape[1]:
        return
    mandel_x = (end[0] - startx)*(x/size) + startx
    mandel_y = (end[1] - starty)*(y/size) + starty
```

```
c = complex(mandel_x, mandel_y)
i = -1
z = complex(0, 0)
while abs(z) < 2:
    i += 1
    if i == 255:
        break
    z = z**2 + c
img_array[y, x] = i
```

Похоже, что функция *repeat* предпочитает выводить время исключительно в микросекундах. Здесь мы получили результат 70 600 мкс, что соответствует 70 мс. Теперь, когда мы написали первую версию генератора с использованием CuPy, давайте создадим вторую версию, в которой встроим наш код на CUDA C в программу на Python.

9.3.5. Создание генератора Мандельброта с помощью CUDA C

На этот раз для создания множества Мандельброта (Mandelbrot set) мы воспользуемся векторизованной функцией. Она будет принимать массив со всеми позициями и для каждой из них вычислять значение. Мы реализуем нашу функцию на языке CUDA C.

Как и в случае с версией NumPy, начнем мы с подготовки массива позиций. Сделаем это в NumPy и затем перенесем в CuPy:

```
def prepare_pos_array(start, end, pos_array):
    size = pos_array.shape[0]
    startx, starty = start
    endx, endy = end
    for xp in range(size):
        x = (endx - startx)*(xp/size) + startx
        for yp in range(size):
            y = (endy - starty)*(yp/size) + starty
            pos_array[yp, xp] = complex(x, y)

pos_array = np.empty((size, size), dtype=np.complex64)
prepare_pos_array(pos_array)

cp_pos_array = cp.array(pos_array)
```

Код для подготовки массива остался практически неизменным. В последней строке мы преобразовываем массив NumPy в версию CuPy для GPU, что требует передачи данных.

Теперь нам необходимо подготовить наши переменные *threads_per_block* и *blocks_per_grid*. Чтобы не усложнять код на языке C, мы будем работать в одном измерении, а не в двух:

```
threads_per_block = 16 ** 2
blocks_per_grid = ceil(size / 16) ** 2
```

Мы соответствующим образом масштабируем одномерные блоки и потоки в блоках. Ниже приведена наша реализация:

```
c_compute_mandelbrot = cp.RawKernel(r'''
#include <cupy/complex.cuh>
extern "C" __global__
void raw_mandelbrot(const complex<float>* pos_array,
                    char* img_array) {
    int x = blockDim.x * blockIdx.x + threadIdx.x;
    int i = -1;
```



```

complex<float> z = complex<float>(0.0, 0.0);
complex<float> c = pos_array[x];
while (abs(z) < 2) {
    i++;
    if (i == 255) break;
    z = z*z + c;
}
img_array[x] = i;
}
''' , 'raw_mandelbrot')

```

В данной книге мы не ставили себе цель научить вас программировать на языке C, так что не будем вдаваться в подробности показанного выше кода. Но в принципе код здесь довольно простой, и вы должны без труда его прочитать и понять. Мы вычисляем позицию с помощью выражения `blockDim.x * blockIdx.x + threadIdx.x`. Код на C фактически воспринимает матрицу как одномерный массив, и это работает.

Наконец, давайте воспользуемся написанной функцией для вычисления множества Мандельброта на основании массива позиций:

```

c_compute_mandelbrot((blocks_per_grid,),
    (threads_per_block,), (cp_pos_array, cp_img_array))
img = Image.fromarray(cp.asnumpy(cp_img_array), mode="P")
img.save("cmandelbrot.png")

```

Обратите внимание на синтаксис вызова функции и указание количества блоков и потоков в каждом блоке. Это отличается от подхода с использованием компилятора Numba. В заключение мы преобразовываем массив CuPy в формат NumPy для вывода.

Давайте проведем базовый анализ производительности этого решения:

```

from cupyx.time import repeat

print(repeat(
    lambda: c_compute_mandelbrot((blocks_per_grid,),
        (threads_per_block,), (cp_pos_array, cp_img_array)),
    n_repeat=200))

```

На моем компьютере я получил такой результат:

```

<lambda>          :   CPU: 6.677 us +/- 2.769
                   (min: 4.377 / max: 25.978) us
                   GPU-0: 3149.825 us +/- 801.397
                   (min: 2635.584 / max: 5881.088) us

```

Итоговые 3,1 мс – это в 20 раз быстрее по сравнению с версией для Numba. Так что если код с использованием компилятора Numba получился недостаточно быстрым, вы можете предпринять попытку встраивания эффективного кода на CUDA C.

Теперь, когда мы вдоволь написались кода для графического процессора, давайте узнаем, какие существуют средства профилирования для GPU.

9.3.6. Средства профилирования кода для GPU

В данном разделе мы будем использовать базовый функционал средств профилирования NVIDIA для анализа производительности наших реализаций генератора Мандельброта. Профилирование кода выполняется с помощью общих механизмов, которые не зависят ни от CuPy, ни даже от Python, – вы можете применять их к любому коду, запущенному на GPU. Для демонстрации возможностей этих средств мы выполним профилирование нашей версии генератора Мандельброта с использованием NumPy и векторизации.

Воспользуемся программным пакетом *Nsight Systems* от NVIDIA. Выполним замер производительности в автономном режиме и отдельно проанализируем быстродействие кода при помощи графического интерфейса программы *Nsight Systems*. Это наиболее гибкий подход к анализу, поскольку он предполагает, что машина с графическим процессором располагается отдельно от аналитической машины. Например, когда компьютер с GPU размещен в облаке, а вы анализируете производительность на своем локальном компьютере.

После установки *Nsight Systems* вы можете сразу приступить к анализу быстродействия кода, воспользовавшись следующим синтаксисом:

```
nsys profile -o numba python mandelbrot_numba.py
nsys profile -o c python mandelbrot_c.py
```

Чтобы выполнить профилирование версии NumPy с векторизацией для GPU, можно запустить следующую команду:

```
nsys profile -o numpy python ../sec3-gpu/mandelbrot_numpy.py
```

В результате получим три файла с трассировкой: `numba.qdrep`, `c.qdrep` и `numpy.qdrep`.

Если помните, для версии NumPy среднее время, полученное при помощи инструкции `timeit`, составляло 222 мс, тогда как `cyrux.time.repeat` для версии с использованием компилятора Numba выдал 70 мс, а для CUDA C – 3 мс.

Для каждого варианта мы можем собрать некую базовую статистику. Начнем с версии NumPy:

```
nsys stats numpy.qdrep
```

Запуск этой команды приведет к объемному выводу. Мы сосредоточимся на части, касающейся графического процессора, приведенной ниже:

Time%	Total ns	Calls	Avg ns	Min ns	Max ns	StdDev	Name
96.1	368748545	1	368748545	368748545	368748545	0	cuMemcpyDtoH
3.6	13654495	1	13654495	13654495	13654495	0	cuMemcpyHtoD
0.1	540957	2	270478	234726	306231	50561	cuMemAlloc
0.1	371672	1	371672	371672	371672	0	cuModLdDataEx
0.0	133176	1	133176	133176	133176	0	cuLinkComplete
0.0	66248	1	66248	66248	66248	0	cuLinkCreate
0.0	49602	1	49602	49602	49602	0	cuMemGetInf
0.0	37495	1	37495	37495	37495	0	cuLaunchKernel
0.0	2071	1	2071	2071	2071	0	cuLinkDestroy

Обратите внимание, что львиная доля времени в нашей реализации ушла на передачу данных в память графического процессора и обратно: это первые две строчки с идентификаторами `cuMemcpyDtoH` и `cuMemcpyHtoD`, занимающие больше 99 %.

Также мы можем проанализировать время самих вычислений в ядре. Ниже приведена сокращенная версия по реализации для NumPy:

Time(%)	Total Time (ns)	Name
100.0	365860777	cupy::__main__:__vectorized_compute_point ...

Время получилось 365 860 777 нс, или 365 мс.

Теперь посмотрим на вывод для версии CuPy с использованием Numba:

Time(%)	Total Time (ns)	Name
100.0	189965876	cupy::__main__:compute_all_mandelbrot ...

Результат – 180 мс.

Наконец, посмотрим на быстроедействие реализации с использованием языка CUDA C:

Time(%)	Total Time (ns)	Name
100.0	5876134	raw_mandelbrot

Здесь результат получился 5,8 мс.

Как видите, версия с использованием NumPy оказалась вдвое более медленной по сравнению с Numba. В свою очередь версия с применением языка CUDA C показала скорость, в 32 раза превышающую реализацию с Numba.

Программный комплекс Nsight Systems обладает удобным графическим интерфейсом, который можно вызвать командой `nsys-ui`. В открывшемся окне вы сможете отслеживать выполнение процессов в реальном времени. Хотя с помощью картинки бывает трудно пере-

дать динамику происходящего, мы хотя бы попытались. На рис. 9.3 показан увеличенный фрагмент трассировки для генератора Мандельброта с использованием языка C. В приложении отображаются события CPU и GPU, но мы сконцентрируемся на последних. В основной области на рисунке вы видите два выделяющихся блока. Первый из них, располагающийся на шкале слева, отражает процесс преобразования массива NumPy с позициями в формат CuPy, т. е. его передачу с хоста на устройство. Эта операция воплощена в строке кода `sr_pos_aggay = sr.aggay(pos_aggay)`. Второй блок отвечает за вычисление функции Мандельброта.

Из этого раздела вы можете вынести две главные идеи:

- как и в случае с задачами, выполняющимися на центральном процессоре, при возникновении задержек, связанных с работой графического процессора, всегда лучше выполнить полное профилирование кода для выявления их причин, а не строить догадки;
- если для библиотек в Python, которыми вы обычно пользуетесь, существуют аналоги, предназначенные для работы с графическим процессором, почти всегда будет лучше воспользоваться ими вместо написания кода с нуля для воплощения той же реализации.

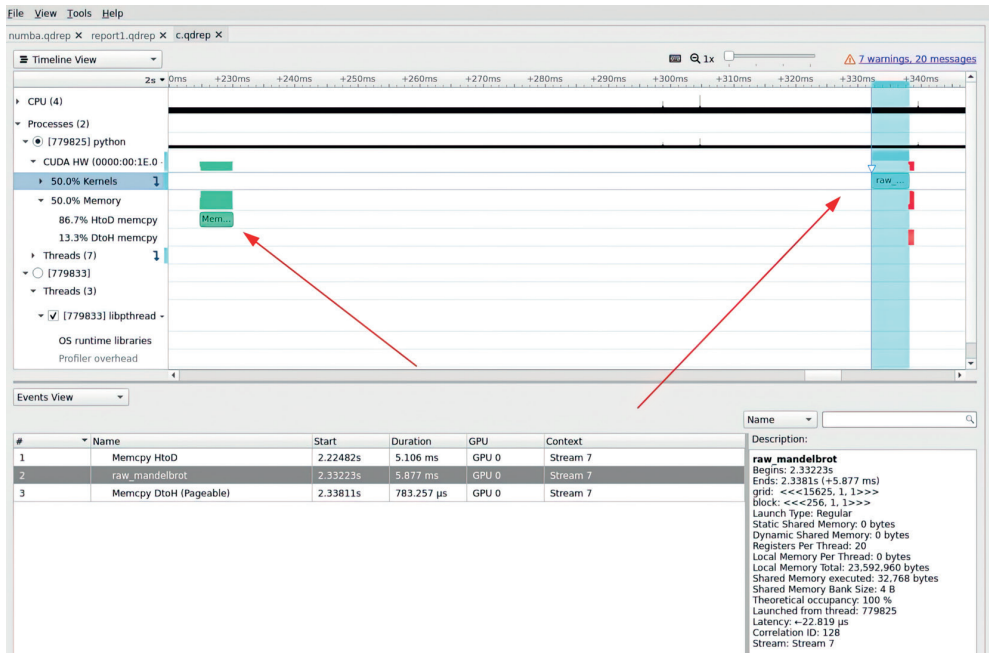


Рис. 9.3. Графический интерфейс Nsight Systems. Слева сверху: иерархическая структура всех процессов для GPU и CPU. Основное окно: временное представление запусков операций. Слева внизу: статистика по нескольким операциям GPU. Справа внизу: подробности для одного из блоков из основного окна.

Заключение

- Центральный процессор предоставляет разработчикам сразу несколько высокопроизводительных ядер для параллельного решения нескольких сложных задач. В противовес ему графический процессор обладает гораздо большим количеством ядер значительно более низкой производительности, способных эффективно выполнять несложные повторяющиеся действия.
- Ресурсы графического процессора идеально подходят для эффективной обработки данных, поскольку многие задачи из области науки о данных сводятся к работе с большими массивами, к элементам которых применяются одни и те же вычислительные алгоритмы.
- Существует множество производителей графических процессоров, но общепризнанным лидером в плане использования GPU в вычислительных целях является NVIDIA.
- При написании кода для графического процессора необходимо учитывать разницу в вычислительных моделях между GPU и CPU, которая обуславливает применение совершенно иного подхода к программированию.
- Стандартный код на Python не может быть напрямую запущен на графическом процессоре. Для этого необходимо использовать альтернативные способы, позволяющие задействовать в вычислениях ресурсы GPU.
- Существует множество высокоуровневых библиотек Python, с помощью которых можно перенаправлять вычисления на графический процессор, даже не разбираясь во всех тонкостях его работы.
- Многие библиотеки Python для работы с GPU являются практически прямой заменой аналогичным библиотекам, взаимодействующим с CPU. К примеру, библиотека CuPy предоставляет похожий интерфейс с NumPy, но при этом задействует ресурсы графического процессора, а библиотека cuDF обладает заметными сходствами с популярной библиотекой pandas.
- Компилятор Numba способен генерировать эффективный код, предназначенный для GPU, но для этого недостаточно просто снабдить инструкции на Python соответствующими аннотациями. Код необходимо полностью переписать для реализации параллелизма с применением одинаковых алгоритмов к большим массивам данных.
- Код Numba, даже написанный для GPU, может органично взаимодействовать с библиотекой NumPy, что позволяет вынести за скобки ресурсоемкие параллельные алгоритмы и при этом продолжать работать с привычным стеком для анализа данных в Python.

10

Анализ больших данных с использованием библиотеки Dask

В этой главе мы обсудим следующие темы:

- масштабирование вычислений с задействованием множества машин при работе с действительно большими данными;
- знакомство с моделью выполнения библиотеки Dask;
- исполнение кода с помощью планировщика `dask.distributed`.

Обработка больших объемов данных зачастую требует более одного компьютера по причине невозможности обеспечения процессинга на одной машине или сложности используемых алгоритмов, требующих существенных вычислительных ресурсов. На текущем этапе вы уже знаете, что нужно для осуществления более эффективного вычислительного процесса и организации хранения данных в виде, наиболее подходящем для обработки. В заключительной главе книги речь пойдет о *горизонтальном масштабировании* (scale out) проектов, т. е. задействовании большего числа компьютеров для выполнения необходимых вычислений.

С целью выполнения горизонтального масштабирования мы будем использовать библиотеку *Dask*, позволяющую внедрить параллельные вычисления в аналитике данных. *Dask* прекрасно интегрируется с прочими библиотеками в экосистеме Python, такими как NumPy и pandas, и отвечает всем нашим требованиям в отношении горизонтального масштабирования решений. В то же время *Dask* может использоваться для выполнения *вертикального масштабирования* (scale up), т. е. улучшения вычислительных результатов в рамках одного компьютера. В этом отношении использование данной библиотеки может рассматриваться в качестве альтернативы приемам, связанным с параллелизмом, о которых мы говорили в главе 3.

Среди конкурентов библиотеки *Dask* можно выделить один из самых популярных фреймворков – Spark. Эта технология изначально появилась в среде Java, в связи с чем она значительно меньше интегрирована в экосистему Python по сравнению с *Dask*. Лично я предпочитаю использовать в работе решения, прорастающие из Python для обеспечения максимально простого взаимодействия. В то же время многие техники, которые будут показаны в этой главе, могут быть с легкостью применены в других фреймворках.

Библиотека *Dask* содержит сразу несколько программных интерфейсов. Высокоуровневые API *Dask* очень напоминают традиционные библиотеки NumPy, pandas и другие аналитические пакеты. Вместе с тем те же простые в освоении интерфейсы *Dask* могут успешно использоваться применительно к объектам вроде датафреймов и массивов, которые целиком не помещаются в память, в чем библиотеки pandas и NumPy бессильны. Один из низкоуровневых интерфейсов *Dask* основывается на библиотеке `concurrent.futures`, упомянутой нами в главе 3, а другой позволяет распараллеливать обычный код на Python, в котором не используются массивы или датафреймы.

Главная цель этой главы состоит в том, чтобы вы поняли лежащую в основе библиотеки *Dask* *модель выполнения* (execution model), а также альтернативные способы планирования и принципы работы с данными, объем которых превышает доступную память. В процессе обсуждения мы никуда не денемся от вопросов, связанных с производительностью, но я уверен, что гораздо большую пользу на данном этапе вам принесет глубокое понимание модели вычислений, использующейся в *Dask*. Среда выполнения может сильно меняться – от одной машины до огромных кластеров – и значительно влиять на эффективность используемых приемов. Таким образом, в этой главе книги наш подход будет отличаться от всего, что вы видели раньше: вы просто получите в свое распоряжение необходимые строительные блоки, а как применить их в вашем актуальном окружении, решите сами.

В *Dask* используется совсем иная, более ленивая модель выполнения по сравнению с библиотеками вроде pandas и NumPy. Таким

образом, первый раздел этой главы будет посвящен различиям в семантике между этими библиотеками. Чтобы не отрываться от земли сразу, мы не будем на первых этапах затрагивать вопросы, связанные с параллелизмом. Также мы поначалу обойдем вниманием тему обработки данных, объем которых превышает доступную память. Мы просто поработаем с датафреймами в Dask и увидим, что этот процесс очень напоминает pandas.

Во втором разделе мы затронем тему секционирования больших данных и обсудим некоторые предпосылки библиотеки Dask, связанные с повышением производительности. Кроме того, я покажу несколько техник, способствующих повышению эффективности вычислений.

Третий раздел будет посвящен обсуждению планировщика Dask, позволяющего рационально распределять вычисления между несколькими компьютерами и архитектурами: от *высокопроизводительных вычислительных кластеров* (HPC cluster) до машин на основе графических процессоров. Поскольку было бы слишком оптимистично надеяться на то, что у всех читателей есть доступ к кластеру или облаку для проверки кода из книги, мы будем все реализовывать в рамках одной машины с возможностью горизонтального масштабирования.

Начнем мы повествование с модели выполнения, используемой в Dask. С учетом ленивой природы этой библиотеки вы обнаружите несколько важных отличий между ней и традиционными библиотеками, такими как pandas или NumPy. И эти отличия очень важно осмыслить перед тем, как приступить к реализации параллельных решений с использованием Dask.

Для запуска кода из этой главы вам необходимо установить библиотеку Dask, а также Graphviz, позволяющую выводить на экран графы задач. Если вы используете conda, запустите команду `conda install dask`. Что касается библиотеки Graphviz, то ее в настоящее время легче всего установить с помощью pip, даже если вы используете conda: `pip install graphviz`. Также вы должны убедиться, что у вас установлено основное приложение Graphviz. Образ Docker с установленными пакетами находится в репозитории `tiagoantao/python-performance-dask`.

10.1. Знакомство с моделью выполнения Dask

Параллельные решения – это всегда непросто, особенно когда они работают в рамках распределенной архитектуры. Перед тем как погрузиться в параллелизм с Dask, давайте поближе познакомимся с моделью выполнения этой библиотеки. Мы напишем pandas-подобное решение с использованием библиотеки Dask, не обращая внимания на особенности реализации: последовательное или параллельное вычисление. Пока мы будем касаться лишь модели выполнения, что поможет нам понять некоторые ключевые отличия между библиоте-

кой Dask и любой другой, например pandas. В следующем разделе мы уже сосредоточимся на способе осуществления вычислений и поговорим о принципах параллелизма применительно к Dask.

В данном примере мы воспользуемся данными Бюро переписи населения США (US Census) о налоговых сборах в 50 штатах. Для каждого штата мы получим полную информацию о налогах с разбивкой по источникам. Иными словами, в нашем распоряжении будет как полная сумма налоговых выплат по штатам, так и конкретные суммы по налогу на доходы, налогу с продаж, налогу на имущество и прочим видам налогов. Мы принимаем решение о том, в каком штате купить дом, и одним из факторов принятия решения для нас является информация о том, насколько велик в штате налог на имущество. Таким образом, мы хотим узнать, в каких штатах этот вид налога занимает относительно небольшую часть суммарных налоговых сборов. Проще говоря, нас интересует доля налога на имущество в общей сумме налогов по штатам.

Мы будем работать с небольшой таблицей, с которой нормально справилась бы библиотека pandas, но для нас сейчас размер не главное. Нас интересует исключительно модель выполнения. Данные находятся в репозитории в папке 10-dask, также их можно загрузить по адресу <http://mng.bz/41ND> (<https://www.census.gov/data/tables/2016/econ/stc/2016-annual.html>).

10.1.1. Шаблон pandas для сравнения

Давайте по привычке начнем обработку данных с помощью библиотеки pandas. Прочитаем данные из файла, очистим их и затем рассчитаем долю налога на имущество в общей сумме налогов по штатам:

```
import numpy as np
import pandas as pd

taxes = pd.read_csv("FY2016-STC-Category-Table.csv", sep="\t")
taxes["Amount"] = taxes["Amount"].str.replace(",", "")
taxes["Amount"] = taxes["Amount"].str.replace("X", np.nan).astype(float)

pivot = taxes.pivot_table(index="Geo_Name", columns="Tax_Type",
                           values="Amount")
has_property_info = pivot[pivot["Property Taxes"].notna()].index

pivot_clean = pivot.loc[has_property_info]
frac_property = pivot_clean["Property Taxes"] / pivot_clean["Total Taxes"]
frac_property.sort_values()
```

Очищаем данные
в колонке Amount,
чтобы сконвертиро-
вать их в тип float

Разво-
рачи-
ваем
таблицу
по полю
Tax_Type

Начали мы с чтения данных о налогах, содержащих информацию о штате (в поле Geo_Name), типе налога (Tax_Type) и сумме сборов по этому типу (Amount):

Geo_Name	Tax_Type	Amount
Alabama	Total Taxes	10,355,317
Alabama	Property Taxes	362,515
Alabama	Sales and Gross Receipts Taxes	5,214,390
Alabama	License Taxes	575,510
Alabama	Income Taxes	4,098,278
Alabama	Other Taxes	104,624
Connecticut	Total Taxes	15,659,420
Connecticut	Property Taxes	X
Connecticut	Sales and Gross Receipts Taxes	6,518,905
Connecticut	License Taxes	454,779
Connecticut	Income Taxes	8,322,645
Connecticut	Other Taxes	363,091
...		

Затем в колонке Amount мы меняем крестики (X) на пропущенные значения (NA). Для наших дальнейших вычислений нам необходимо, чтобы в колонке не присутствовали строковые значения.

После этого мы выполняем разворачивание или *сведение* (pivot) таблицы, заключающееся в создании отдельных столбцов для каждого типа налога и сокращении количества строк в таблице до одной строки на штат. Такое преобразование таблицы значительно облегчит подсчет, поскольку вся необходимая информация у нас будет содержаться в одной строке. Сокращенный результат этого преобразования показан ниже:

index	Income Taxes	Total Taxes	... Property Taxes
Alabama	4098278	10355317	362515
Colorado	711711	12887859	NaN

Затем мы избавляемся от штатов, по которым отсутствует информация о сумме налога на имущество, и рассчитываем долю этого вида налога в общей сумме налоговых сборов:

Nebraska	0.000024
New Jersey	0.000147
Iowa	0.000147
Massachusetts	0.000213
....	
Alaska	0.124577
New Hampshire	0.154625
Wyoming	0.177035
DC	0.326369
Vermont	0.338844

Теперь давайте взглянем на версию кода с использованием библиотеки Dask.

10.1.2. Решение на основе датафреймов Dask

Взгляните на код с использованием библиотеки Dask, который почти не отличается внешне от версии с применением pandas:

```
import numpy as np
import dask.dataframe as dd
```

← Импортируем интерфейс датафреймов Dask

```
taxes = dd.read_csv("FY2016-STC-Category-Table.csv", sep="\t")
taxes["Amount"] = taxes["Amount"].str.replace(",", "").replace("X",
np.nan).astype(float)
taxes["Tax_Type"] = taxes["Tax_Type"].astype(
    "category").cat.as_known()
```

← Необходимо указать, что в колонке Tax_Type хранятся категориальные данные

```
pivot = taxes.pivot_table(index="Geo_Name",
    columns="Tax_Type", values="Amount")
has_property_info = pivot[~pivot[
    "Property Taxes"].isna()].index
```

← Используем метод isna() вместо notna(), который в Dask не поддерживается

```
pivot_clean = pivot.loc[has_property_info]
frac_property = pivot_clean["Property Taxes"] / pivot_clean["Total Taxes"]
```

Как видите, код очень похож на предыдущий. Он кажется абсолютно таким же – такое ощущение, что можно просто скопировать и вставить код и заменить в строке импорта pandas на dask.dataframe.

ПРЕДУПРЕЖДЕНИЕ. Хотя внешне интерфейс работы с датафреймами в Dask очень напоминает pandas, некоторые возможности в нем не реализованы, а другие реализованы несколько иначе. В нашем небольшом примере нам пришлось отказаться от использования метода notna и явным образом задать категориальный тип для столбца. И таких мелочей великое множество. Я выбрал простой пример, чтобы пока не заострять ваше внимание на различиях в используемых техниках. Общие мысли пока должны быть такими: внешне код остался прежним за небольшими исключениями.

Однако показанный выше код делает совсем не то же, что аналогичный код на pandas. Что же он делает?

Вы удивитесь, но инструкция print(frac_property) в данном случае *не* приведет к выводу результата. Вы увидите не набор данных, а *план выполнения* (execution plan) или *граф задач* (task graph), в соответствии с которым будет вычислен результат. На графе задач узлы (nodes) представляют собой операции для выполнения, а ребра (edges) – отношения (dependencies) между операциями. Давайте посмотрим на наш граф.

Библиотека Dask может экспортировать визуальное представление графа задач следующим образом:

```
frac_property.visualize(filename="10-property.svg", rankdir="LR")
```

На рис. 10.1 показана часть графа задач, соответствующая двум первым строкам кода. Строка с функцией dd.read_csv представлена первым

(левым) узлом на графе. Часть строки кода с преобразованиями значений в колонке `taxes["Amount"].str.replace(",","").replace("X", np.nan).astype(float)` показана в нижней ветке узлов на графе, а саму операцию присваивания `taxes["Amount"] = ...` символизирует правый узел.

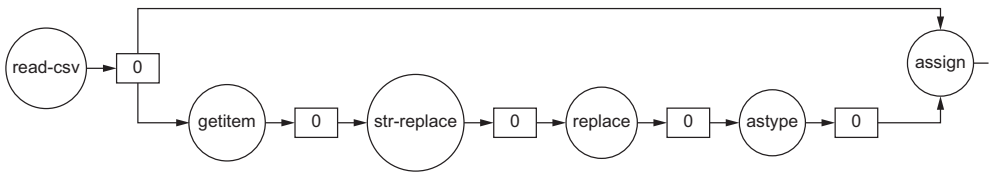


Рис. 10.1. Начало графа задач для нашего кода, состоящее из чтения файла CSV и преобразования значений в колонке Amount

Теперь наше вычисление готово к запуску. Результат можно получить, выполнив следующую инструкцию:

```
frac_property_result = frac_property.compute()
```

В итоге будут произведены необходимые вычисления и возвращен датафрейм `pandas` – такой же, как и в предыдущем примере.

На данном этапе нам не важно, как именно были произведены вычисления: последовательно, многопоточно, многопроцессно, в кластере, на GPU или в облаке. Главное, что вам сейчас необходимо понять, это то, что вычисления в Dask выполняются лениво (отложено), а код, который мы написали ранее, служит лишь для создания графа задач с целью последующего выполнения.

Теперь, когда вы понимаете разницу между ленивым или отложенным подходом к выполнению задач, принятым в Dask, и жадным или немедленным, характерным для `pandas` и `NumPy`, давайте погрузимся в вопросы вычислительной стоимости алгоритмов.

10.2. Вычислительная стоимость операций Dask

В этом разделе мы поговорим о вычислительной стоимости разных операций Dask. При этом наша дискуссия не будет зависеть от среды выполнения. Хотя на практике факторы, связанные с алгоритмами и платформами, на которых они выполняются, тесно переплетаются, значительно легче рассуждать о сложности алгоритмов как таковых. Последствия, связанные с тем, как Dask секционирует данные, не помещающиеся в памяти, никак не зависят от той инфраструктуры, которая будет заниматься непосредственно вычислениями. Проблематика, которой мы посвятим этот раздел, в равной степени касается любых распределенных систем вычисления, таких как `Spark` и многие другие.

Пояснять мы все будем на простых задачах. Первая из них будет связана с созданием колонки `year` с представлением года из

колонки `Survey_Year` в формате последних двух цифр. Например, число 2016 должно быть представлено как 16. Вторая задача предполагает создание колонки `k_amount`, в которой суммы из столбца `Amount` будут представлены в тысячах, т. е. исходные числа должны быть поделены на 1000. Третья задача будет состоять в получении штата с максимальной суммой, а четвертая – в сортировке штатов по суммарным налоговым сборам.

Мы продолжим использовать данные из предыдущего раздела. Несмотря на то что файл у нас довольно небольшой, это не мешает нам использовать секционирование с помощью Dask так же, как при работе с большими данными. Так или иначе, большие это данные или нет – зависит от доступного вам аппаратного обеспечения.

Перед тем как начать разбивать данные на секции, мы прочитаем их и создадим колонку с двузначным представлением года, как показано ниже:

```
import numpy as np
import dask.dataframe as dd

taxes = dd.read_csv("FY2016-STC-Category-Table.csv", sep="\t")
taxes["year"] = taxes["Survey_Year"] - 2000
taxes.visualize(filename="10-single.svg", rankdir="LR")
```

Если вы визуализируете граф задач, как показано на рис. 10.2, то увидите простую схему, похожую на ту, что видели ранее, – она состоит из узла чтения данных и последовательности узлов, связанных с их преобразованием.

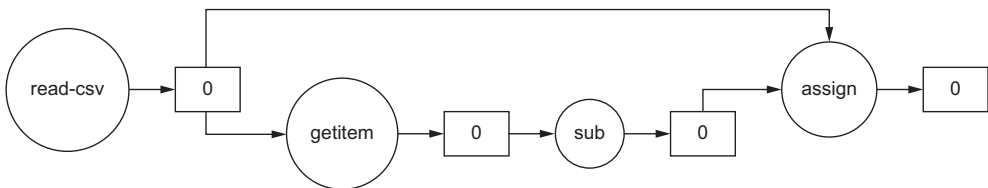


Рис. 10.2. Граф задач с чтением файла CSV и созданием колонки с вычитанием

Теперь давайте посмотрим, как будет выглядеть наш граф после разбиения исходного набора данных на секции.

10.2.1. Секционирование данных для обработки

Наш файл CSV достаточно мал, он занимает всего 15 Кб, но давайте представим, что мы не можем одновременно обрабатывать более 5 Кб информации. Чтобы реализовать секционирование на практике, достаточно передать функции `read_csv` дополнительный параметр `blocksize`, отвечающий за объем порции данных в байтах, как показано ниже:

```
taxes = dd.read_csv("FY2016-STC-Category-Table.csv",
    sep="\t", blocksize=5000)
taxes["year"] = taxes["Survey_Year"] - 2000
taxes.visualize(filename="10-block.svg", rankdir="LR")
```

На визуализации графа задач, представленной на рис. 10.3, видно, что он оказался разбит на три независимые части, при этом каждая из них будет обрабатывать не более 5000 Кб данных.

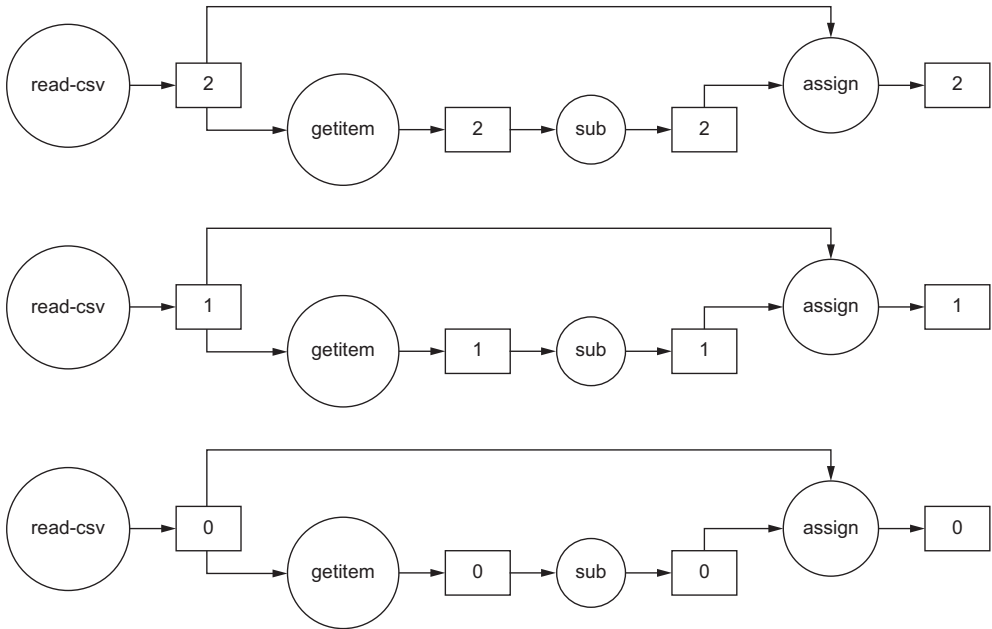


Рис. 10.3. Чтение файла CSV с секционированием и отдельная обработка каждой секции

Теперь, когда мы научились разбивать датафрейм на части, пришло время узнать, как эти самые датафреймы представлены в Dask. Мы разбили исходные данные на три секции, чтобы узнать, как это повлияет на отображение нашего графа задач. На рис. 10.4 показана высокоуровневая схема процесса секционирования. Как видите, все три составные части набора данных представляют собой датафреймы pandas.

Похожая реализация используется и для представления массивов Dask (аналог массивов NumPy в Dask). Каждая секция в этом случае содержит отдельный массив NumPy. Таким образом, библиотека Dask, будучи рожденной в экосистеме Python, активно использует существующие библиотеки в своей внутренней архитектуре. Давайте вернемся к нашему сценарию.

Теперь, когда мы узнали, как операция секционирования данных влияет на итоговые графы задач, давайте посмотрим, как в Dask реализован способ сокращения количества повторяющихся вычислений.

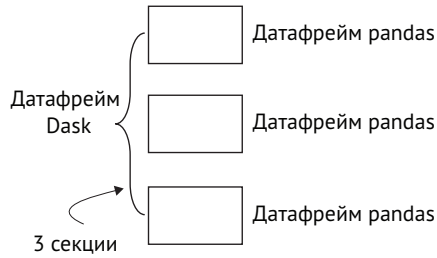


Рис. 10.4. Реализация датафреймов в библиотеке Dask

10.2.2. Сохранение промежуточных вычислений

Как мы уже упоминали в предыдущем разделе, данные в колонке с суммами нуждаются в предварительной очистке, чтобы они могли быть преобразованы в числовые значения. Но, поскольку эти значения могут понадобиться нам сразу в нескольких задачах, нам бы хотелось избежать необходимости преобразовывать строки в числа каждый раз, когда нам нужно обратиться к этому столбцу. С этой целью в Dask предусмотрена техника сохранения промежуточных состояний вычислений, реализованная так, как показано ниже:

```
taxes["Amount"] = taxes["Amount"].str.replace(" ",
    "").replace("X", np.nan).astype(float)
taxes = taxes.persist()
taxes.visualize(filename="10-persist.svg", rankdir="LR")
```

Хотя семантика работы метода *persist* будет зависеть от используемого планировщика, мы представим, что вычисление всех узлов уже началось, и граф задач по очистке данных в поле `Amount` будет выполнен. После запуска метода *persist* будет сохранено промежуточное состояние вычисления, и повторно очистка данных в столбце `Amount` выполняться уже не будет. В нашем примере мы использовали простую обработку данных, но на практике можем столкнуться с достаточно объемными графами задач с вычислениями, состояние которых необходимо сохранять для последующего использования.

Преимущество хранения отдельных данных состоит в том, что они остаются в нашей вычислительной среде, и мы можем использовать их в своих дальнейших параллельных вычислениях. В случае с использованием метода *compute* все данные вернулись бы в исходное состояние, и при необходимости вновь выполнить какие-либо вычисления вам пришлось бы собрать их вместе и сделать все расчеты заново, отправив данные всем задействованным процессам. Кроме того, если весь датафрейм не уместится в вашей памяти, выполнение метода *compute* приведет к аварийному завершению процесса.

СОВЕТ. Передача данных между блоками обработки может оказаться очень дорогой, особенно если речь идет о сетевом окружении, поскольку данные должны быть сериализованы. С другой

стороны, мы не можем хранить все, поскольку это может потребовать чрезмерно много памяти. Обычно кандидатами для использования метода `persist` являются узлы, производящие относительно немного данных и часто используемые повторно.

Теперь, когда мы научились оптимизировать составные части наших вычислений, давайте вернемся к задачам, связанным с извлечением штата с максимальной суммой налоговых сборов и упорядочиванием штатов по суммам налогов.

10.2.3. Реализации алгоритмов при работе с распределенными датафреймами

Для некоторых операций алгоритмы с использованием распределенных вычислений могут отличаться по стоимости в сравнении с последовательными алгоритмами, к которым мы привыкли, – в нашем случае это сродни сравнению датафреймов Dask и pandas.

Давайте выполним простую операцию. Если помните, нам нужно представить значения из колонки `Amount` в новой колонке `k_amount` в тысячах, как показано ниже:

```
taxes["k_amount"] = taxes["Amount"] / 1000  
taxes.visualize(filename="10-k.svg", rankdir="LR")
```

Граф задач для этой операции будет довольно простым, что видно на рис. 10.5. В данном случае все вычисления выполняются параллельно со всеми секциями, что весьма эффективно.

Теперь давайте усложним задачу и решим ее в распределенной среде. В данном случае нам необходимо вычислить максимальное значение по колонке с суммами налогов. Подумайте, как может для такой задачи выглядеть граф. Код для нее приведен ниже:

```
max_k = taxes["k_amount"].max()  
max_k.visualize(filename="10-max_k.svg", rankdir="LR")
```

Обратите внимание, что для каждой секции максимальное значение будет вычисляться отдельно. К сожалению, затем все вычисленные максимумы по секциям должны быть объединены в отдельный процесс для расчета наибольшего значения среди них. Это имеет свои последствия. После нахождения всех максимумов по секциям параллельное вычисление останавливается в последнем узле для расчета максимального значения среди выбранных. В этот момент данные должны быть переданы из узлов, вычислявших максимумы по секциям, в узел агрегации. На показанном на рис. 10.6 графе задач вы можете видеть эти операции передачи данных из узлов `series-max-chunk` в узел `series-max-agg`. Иными словами, нам пришлось от трех параллельных задач перейти к одной, и именно этот аспект может стать узким местом для параллелизма.

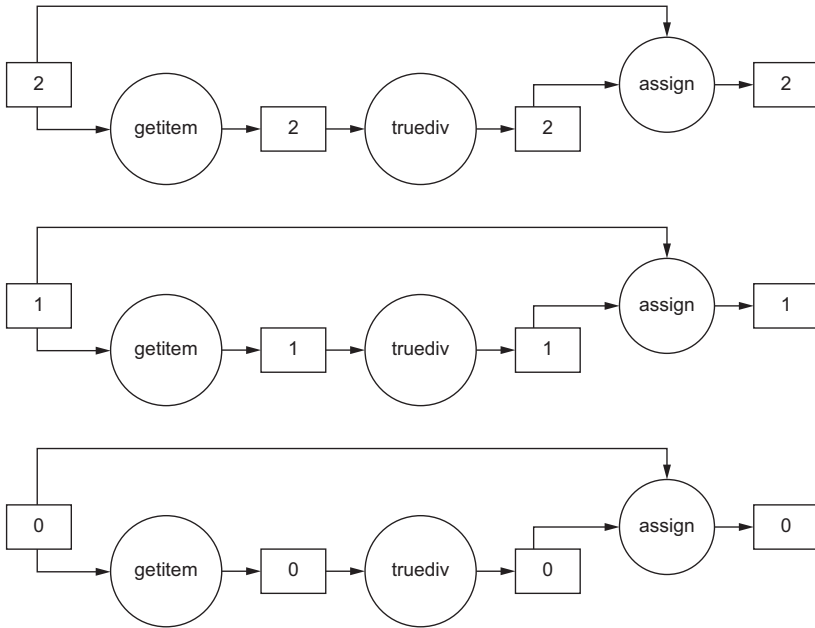


Рис. 10.5. Вычисления производятся отдельно, без необходимости взаимодействия между ними

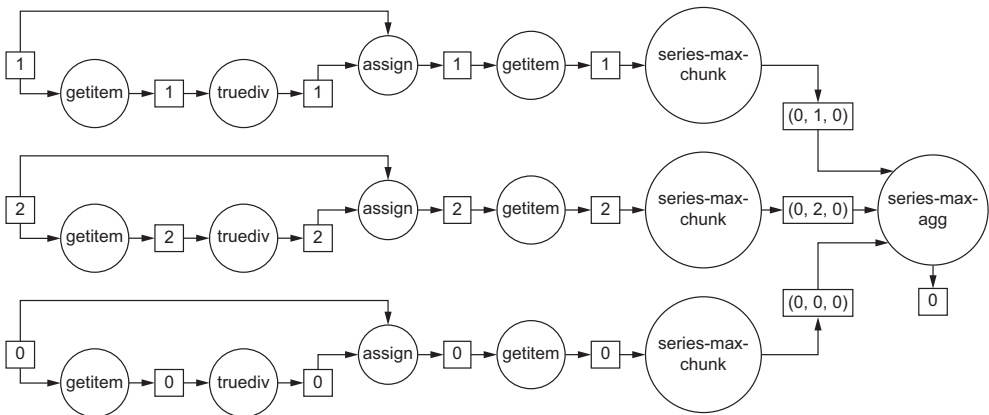


Рис. 10.6. Некоторые вычисления требуют отхода от принципов параллелизма, и один из примеров – расчет максимального значения

Основное, что можно заметить, это то, что стоимость операций при использовании библиотеки Dask может сильно варьироваться в зависимости от условий, в отличие от pandas или NumPy. Если операция требует взаимодействия между процессами или прекращения параллельного вычисления, можно ожидать увеличения ее стоимости. Точные расчеты зависят от используемой архитектуры и прочих факторов.

Если вы не знаете топологию графа для конкретной операции, вы всегда можете визуализировать полный граф задач и определить узкие места в его структуре. К примеру, на рис. 10.7 показан граф задач для довольно сложной операции сортировки. В данном случае узлы `barrier` и `shuffle-collect` привели к остановке параллельного вычисления для всего графа в целом:

```
sv = taxes.sort_values("k_amount")
sv.visualize(filename="10-sv.svg", rankdir="LR")
```

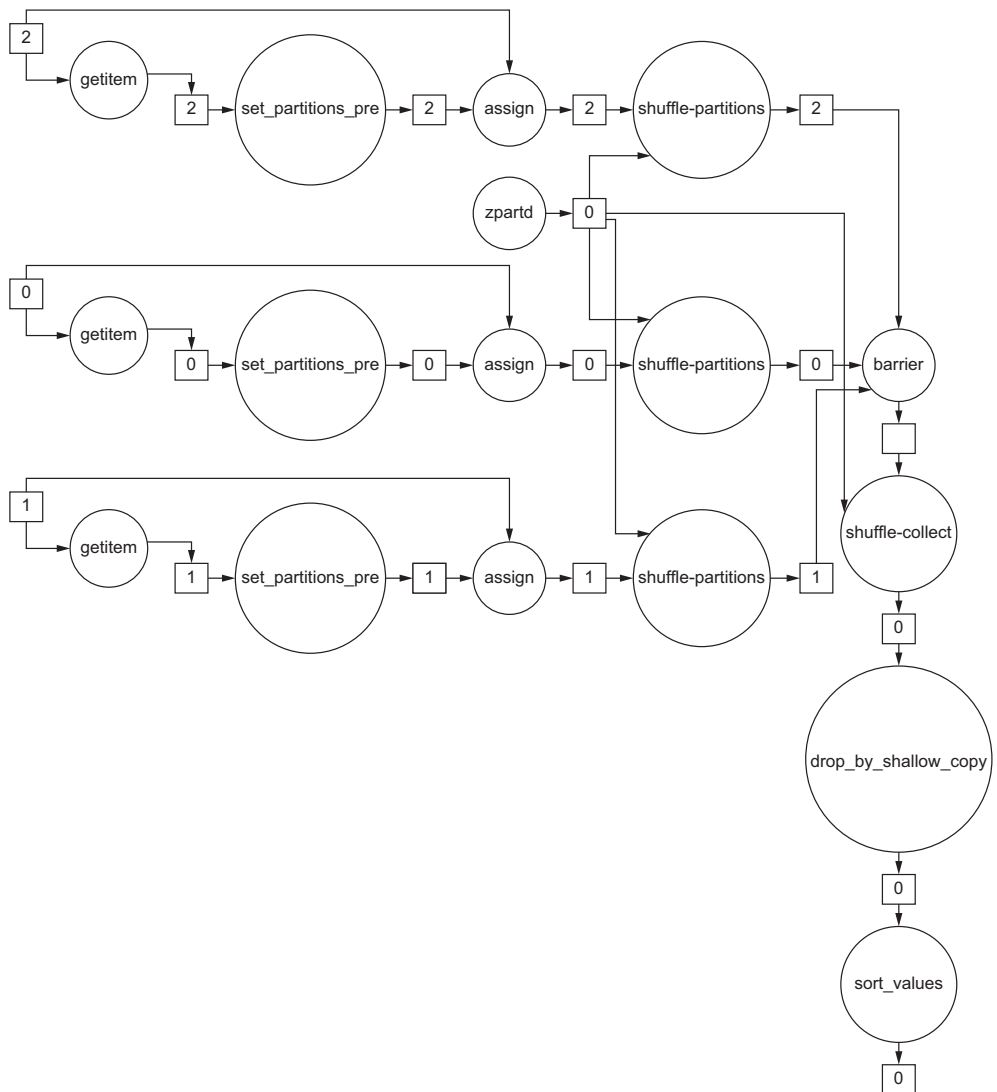


Рис. 10.7. Некоторые операции, такие как сортировка, могут оказаться достаточно сложными и дорогостоящими

Иногда вам может понадобиться выполнить проверку сразу двух операций на графе, поскольку Dask достаточно умен, чтобы произвести такую оптимизацию. К примеру, операции, следующие за `groupby`, могут быть оптимизированы совершенно по-разному. Более того, эффективность оптимизации может варьироваться от версии к версии Dask, так что нет иного способа проверить стоимость выполняемых операций, кроме как вывести граф задач и внимательно его проанализировать.

Мы не будем перечислять все дешевые и дорогие операции Dask в плане времени выполнения по двум причинам. Во-первых, здесь многое зависит от вашего вычислительного окружения. Например, стоимость одной и той же операции может сильно отличаться при выполнении в облаке или на мощном многоядерном компьютере. Во-вторых, технология Dask постоянно развивается, и реализации различных операций могут меняться от версии к версии. Гораздо важнее понимать принципы образования стоимости той или иной операции.

10.2.4. Рассекционирование данных

Иногда, в зависимости от графа задач и окружения выполнения, вы можете выиграть в плане гранулярности вычислений от объединения данных или их *рассекционирования* (`repartitioning`). Представьте, что вы завершили довольно дорогостоящую часть вычислений, потребовавшую в процессе создания дополнительных секций и потенциального вовлечения дополнительных узлов. После выполнения этого процесса, при переходе к менее интенсивным операциям, нам может понадобиться сократить количество секций. В примере ниже мы уменьшаем количество используемых секций с трех до двух:

```
taxes2 = taxes.repartition(npartitions=2)
taxes2.visualize(filename='10-repart.svg', rankdir='LR')
```

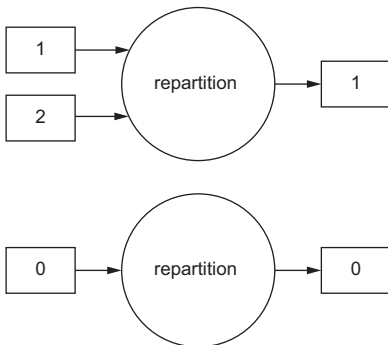


Рис. 10.8. Рассекционирование данных с разным количеством задач

Как видно на графе задач, показанном на рис. 10.8, проблема с таким рассекционированием состоит в том, что две из трех секций объединяются в новую секцию. Было бы эффективнее, если бы обе итоговые секции содержали примерно один объем данных.

Давайте посмотрим, что нужно сделать, чтобы сбалансировать полученные секции. Первая проблема, с которой мы столкнулись, обусловлена семантическими различиями между библиотеками Dask и pandas. И прежде чем двигаться дальше, надо с ними разобраться.

Метод *repartition* позволяет разбить данные, основываясь не только на количестве секций, но и на индексах внутри датафреймов. Для этого нужно знать индексы, и у нас все для этого есть:

```
print(taxes.index)
```

Вывод будет следующим:

```
Dask Index Structure:
```

```
npartitions=3
```

```
  int64
```

```
    ...
```

```
    ...
```

```
    ...
```

```
dtype: int64
```

```
Dask Name: assign, 6 tasks
```

Здесь мы получим лишь набор задач к исполнению, а не результат

Выполнение этого кода позволило нам получить список задач, а не непосредственно результат. Для получения индексов нам придется воспользоваться методом `compute` со всеми сопутствующими вычислительными сложностями, которые в большинстве случаев приведут к краху оптимизации на полпути.

В качестве альтернативы мы могли бы воспользоваться для расчетов границами каждой секции. Библиотека Dask дает такую возможность посредством следующей инструкции:

```
print(taxes.divisions)
```

К сожалению, вывод оказался удручающим:

```
(None, None, None, None)
```

Вместо значений или колонки с индексом мы получили `None`. Очевидно, что мы никак не сможем использовать эту информацию для расчета размеров секций.

Что ж, давайте погрустим: нам нужны данные из индекса, но достать их мы не можем. При выполнении функции `read_csv` в Dask мы не получим индекс со значениями, даже если предварительно сохраним данные (по крайней мере в текущей версии Dask).

Но мы можем установить индекс, чтобы в дальнейшем его использовать. Давайте воспользуемся колонками `Geo_Name` и `Tax_Type` для установки индекса:

```
taxes = taxes.set_index(["Geo_Name", "Tax_Type"])
```

К сожалению, в нынешней версии Dask эта операция не работает, поскольку Dask не поддерживает множественные индексы. Как видите, наши предположения о том, что Dask и pandas, будучи

очень близкими библиотеками в отношении реализации и воплощения, могут серьезно отличаться в плане работы с распределенными данными, оправдываются. Библиотека Dask делает все возможное, но пока в ней присутствуют подобные ограничения.

СОВЕТ. Убедитесь, что вы используете последнюю доступную версию библиотеки Dask. Может оказаться, что в ней некоторые ограничения уже сняты.

Хорошо, давайте попробуем задать индекс с использованием одной колонки:

```
taxes = taxes.set_index(["Geo_Name"])
print(taxes.npartitions)
print(taxes.divisions)
```

Вывод будет таким:

```
3
('Alabama', 'Iowa', 'North Carolina', 'Wyoming')
```

Что ж, отлично! Мы получили информацию о трех секциях, первая из которых начинается со штата Алабама (Alabama), вторая – со штата Айова (Iowa), а третья распространяется с Северной Каролины (North Carolina) до Вайоминга (Wyoming).

Как же Dask узнал эту информацию о секциях, если он такой ленивый? Мы не просили явным образом вычислять новый датафрейм. Тем не менее в некоторых случаях метод `set_index` выполняется в жадной манере, способствуя запуску всех вычислений, необходимых для создания датафрейма с индексом, что может потребовать немалых вычислительных ресурсов.

Как видите, Dask не всегда так ленив, и для определенных операций ему могут потребоваться существенные вычислительные мощности. Перед использованием незнакомой операции будет не лишним прочесть документацию к ней, особенно если у вас есть сомнения в том, что ее выполнение будет ленивым.

Теперь, когда мы наконец смогли создать индекс в датафрейме, можно сделать из трех секций две следующим образом:

```
taxes2 = taxes.repartition(divisions=[
    "Alabama", "New Hampshire", "Wyoming"])
print(taxes2.npartitions)
print(taxes2.divisions)
```

Вывод этой операции представлен ниже:

```
2
('Alabama', 'New Hampshire', 'Wyoming')
```

Всегда помните о том, что рассекционирование является довольно дорогостоящей в плане ресурсов операцией, и использовать ее можно либо на относительно небольших данных, либо заранее убедившись с помощью профилирования, что ее применение будет оправданным.

Стоит отметить, что все аргументы, которые мы использовали в отношении взаимодействия между датафреймами Dask и pandas, могут быть в полной мере применены к массивам Dask и NumPy. Массивы Dask в большинстве своем реализуются при помощи ленивых операций и являются подмножеством интерфейса массивов NumPy. В то же время иногда их реализация может отличаться от NumPy семантически.

Итак, мы готовы к сохранению нашего распределенного датафрейма на диск.

10.2.5. Хранение распределенных датафреймов

Чтобы сохранить датафрейм `taxes2` на диск, нужно выполнить следующую простую инструкцию:

```
taxes2.compute().to_csv("taxes2_pandas.csv")
```

При этом распределенный датафрейм Dask будет пересчитан в датафрейм pandas, и именно pandas мы поручим осуществлять запись на диск. Но собрать все данные из вычисляемых узлов едино может оказаться слишком дорого, кроме того, итоговый набор данных может просто не поместиться в память, так что такая опция не всегда может быть применима.

ПРЕДУПРЕЖДЕНИЕ. Будьте осторожны, употребляя слово *сохранить* (`persist`). В данном случае имеется в виду перенос данных в физическую память, например на диск. В то же время, как мы уже видели, в Dask есть метод `persist`, с помощью которого осуществляется вычисление и сохранение промежуточных объектов в каждой секции.

Мы можем попросить Dask сохранить данные непосредственно из узлов следующим образом:

```
taxes2.to_csv("partial-*.csv")
```

Вы помните, что в наших данных содержится две секции, так что в результате мы получим не один, а сразу два файла CSV с именами `partial-0.csv` и `partial-1.csv`, оба с заголовками. Если вам необходимо получить единый файл, вы можете сконкатенировать данные соответствующим образом.

С помощью формата Parquet, который мы обсуждали в главе 8, вы можете получить единую хранимую версию данных с независимой записью со стороны каждой секции:

```
taxes2.to_parquet("taxes2.parquet")
```

Если открыть соответствующую директорию, вы увидите следующую иерархию файлов:

```
taxes2.parquet/  
  _common_metadata  
  _metadata  
  part.0.parquet  
  part.1.parquet
```

Эту папку впоследствии можно прочитать как единый файл. В следующем примере мы воспользовались библиотекой Apache Arrow, о которой речь шла в главе 7:

```
from pyarrow import parquet  
taxes2_pq = parquet.read_table("taxes2.parquet")  
taxes_pd = taxes2_pq.to_pandas()
```

Таким образом, формат Parquet идеально подходит для осуществления распределенной записи и единообразного представления данных.

Итак, на данный момент мы узнали, как работает генератор задач в Dask, но о выполнении этих задач мы говорили довольно мало. Самое время перейти к заключительной части, а именно к использованию Dask для эффективных параллельных вычислений на разных архитектурах с помощью планировщика.

10.3. Использование распределенного планировщика Dask

Мы уже видели, что библиотека Dask по большей части работает в ленивом режиме и просто создает вычислительный граф задач, который впоследствии должен выполняться. С целью распределения процесса выполнения узлов по вычислительным ресурсам Dask использует *планировщик* (scheduler). При запуске графа задач на выполнение без предварительного конфигурирования планировщика Dask использует настройки планировщика по умолчанию в зависимости от коллекции, с которой вы работаете. Давайте в качестве примера возьмем наш датафрейм:

```
import dask  
from dask.base import get_scheduler  
import dask.dataframe as dd  
  
df = dd.read_csv("FY2016-STC-Category-Table.csv")  
print(get_scheduler(collections=[df]).__module__)
```

Функция `get_scheduler` возвращает функцию для запуска графа задач. В нашем случае она определена в модуле, название которого мы вывели на экран:

```
'dask.threaded'
```

Как видно по имени модуля, для датафреймов Dask по умолчанию использует многопоточный планировщик. Помимо него в распоряжении Dask есть еще два простых планировщика: многопроцессный и однопоточный. Последний хорошо подходит для отладки приложений и профилирования кода, поскольку не вносит лишних усложнений и все операции выполняет последовательно. Что касается рабочих окружений, здесь необходимо использовать более сложный *распределенный планировщик* (distributed scheduler), обладающий исключительной гибкостью и оставивший далеко позади все другие планировщики, которые есть в Dask.

Распределенный планировщик позволяет выполнять задачи на нескольких машинах, при этом, помимо прочих, он располагает реализациями для *высокопроизводительных вычислительных кластеров* (HPC cluster), подключений SSH и облачных провайдеров. Кроме того, этот планировщик поддерживает вычисления в рамках одной машины, причем как одно- и многопоточные, так и многопроцессные. Таким образом, он покрывает все вычислительные методы, используемые во встроенных планировщиках.

Далее мы будем использовать конфигурацию планировщика для одной машины, так что вам не придется получать доступ к кластеру или облаку, но при этом все перечисленные строительные блоки вы сможете использовать и при горизонтальном масштабировании архитектуры.

ПРИМЕЧАНИЕ. В этой главе вы заметите определенные пересечения материала с главой 3. Вы действительно можете использовать библиотеку Dask для создания параллельных решений в Python, которые мы реализовывали при помощи встроенных библиотек. При этом у решений с использованием Dask будет одно неоспоримое преимущество в виде возможности осуществления горизонтального масштабирования, т.е. использования нескольких машин.

Мы снова обратимся к нашему сценарию с созданием генератора Мандельброта из предыдущей главы, чтобы попрактиковаться с интерфейсом Dask для работы с массивами. Не забывайте, что существуют более подходящие альтернативы для реализации эффективного алгоритма генерирования множества Мандельброта (например, Cython или Numba). С учетом того, что мы будем работать в чистой реализации Python с массивами, разумеется, использование расширения Cython или компилятора Numba может дать весьма существенный прирост производительности. На самом же

деле наиболее эффективной реализацией этого алгоритма, особенно для очень больших изображений, является способ с привлечением библиотеки Dask *совместно* с Cython или Numba. Но начнем мы с изучения архитектуры `dask.distributed`.

10.3.1. Архитектура `dask.distributed`

Архитектура, показанная на рис. 10.9, содержит следующие компоненты:

- *единый централизованный планировщик* – этот планировщик ответственен за все задачи, запускаемые на обработчиках. Планировщик обладает собственным дашбордом, с помощью которого пользователи могут следить за ходом выполнения задач;
- *обработчики (workers)* – обработчики занимаются выполнением задач. На каждой машине может располагаться несколько обработчиков. При этом вы можете настроить каждый обработчик на работу с любым количеством потоков по вашему желанию. Таким образом, на практике мы получаем параллелизм либо посредством многопоточного исполнения – скажем, при наличии одного обработчика с количеством потоков, равным числу ядер процессора, – либо с помощью многопроцессности, когда на каждом ядре работает по одному обработчику. У каждого обработчика также есть свой дашборд и небольшой присоединенный процесс под названием *pannu*, осуществляющий постоянный контроль за обработчиком;
- *клиенты* – могут подключаться к Dask, использовать планировщик для развертывания на нем задач и отслеживать дашборды планировщика и обработчика.

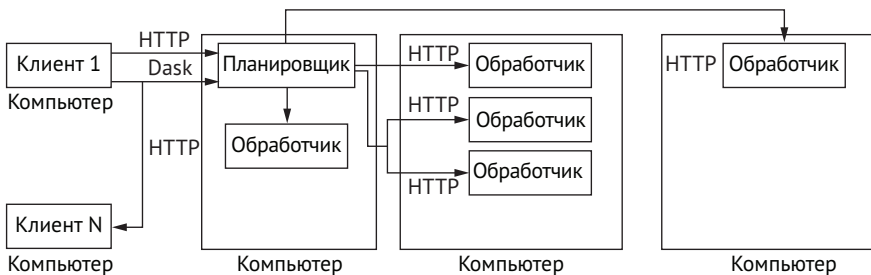


Рис. 10.9. Архитектура выполнения Dask

Обычно компоненты также включают в себя какое-то распределенное хранилище, но это будет зависеть от ваших конкретных условий.

Мы для нашего примера будем использовать одну машину, на которой и будут запущены все процессы. Существуют и более простые способы развертывания архитектуры, но в выбранном нами варианте все компоненты определяются явно.

Начнем с запуска планировщика:

```
dask-scheduler --port 8786 --dashboard-address 8787
```

Обработчики мы можем запустить на той же машине, где располагается планировщик, как показано ниже:

```
dask-worker --nprocs auto 127.0.0.1:8786
```

Опция `--nprocs auto` позволяет скрипту самому решить, сколько обработчиков и потоков запускать на нашей машине.

На моем компьютере с четырьмя ядрами и двумя потоками на каждое ядро для меня было создано четыре обработчика с двумя потоками для каждого. Эту информацию можно увидеть на дашборде планировщика. Для этого откройте адрес <http://127.0.0.1:8787> в браузере и перейдите на вкладку **Workers**. Результат, который я получил, показан на рис. 10.10.

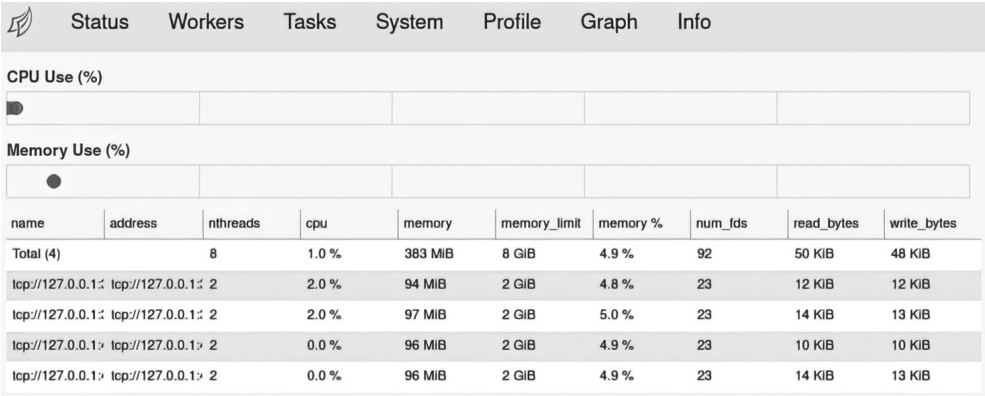


Рис. 10.10. Список всех обработчиков на дашборде

Решить, сколько именно обработчиков и потоков поднимать на одной машине, бывает непросто, поскольку это, в частности, зависит от рабочей нагрузки. Для большинства задач на базе библиотеки NumPy, которая при правильной настройке работает многопоточно, можно начать с одного обработчика на машине. NumPy будет использовать столько потоков, сколько ему понадобится, и в этом случае решение можно оставить за библиотекой. Аналогичную аргументацию относительно автоматического выбора количества обработчиков и потоков можно применить и к решениям на базе Numba или Cython, поскольку в обоих случаях может быть осуществлен обход ограничений GIL. Но условия и нагрузка могут быть разными. В нашем случае большая часть работы выполняется на чистом Python, в связи с чем мы будем использовать следующую конфигурацию для обработчиков:

```
dask-worker --nprocs 4 --nthreads 1 --memory-limit 1GB 127.0.0.1:8786
```

Мы подняли четыре процесса по одному потоку в каждом. Также мы обозначили ограничение на используемый объем памяти в 1 Гб, что составляет половину от всего объема памяти на моей машине. Я сделал так, поскольку запускаю множество сторонних процессов на своем компьютере, которым также требуется память. На выделенном сервере можно сделать этот лимит менее жестким, да и в целом конфигурация должна соответствовать вашим требованиям.

В целях обучения мы снизим количество обработчиков до двух, чтобы можно было наладить взаимодействие между обработчиками, и выделим 250 Мб памяти для каждого обработчика. На моем компьютере команда будет выглядеть так:

```
dask-worker --nprocs 2 --nthreads 1 --memory-limit 250MB 127.0.0.1:8786
```

Теперь нам необходимо подключиться к нашему планировщику с помощью кода на Python. Но, перед тем как приступить к решению конкретной задачи, давайте ознакомимся с нашей инфраструктурой:

```
from pprint import pprint
import dask.dataframe as dd
from dask.distributed import Client

client = Client('127.0.0.1:8786')
print(client)

for what, instances in client.get_versions().items():
    print(what)
    if what == 'workers':
        for name, instance in instances.items():
            print(name)
            pprint(instance)
    else:
        pprint(instances)
```

Мы подключаемся к планировщику по порту, указанному при его запуске

Метод `get_versions` возвращает информацию о различных компонентах системы Dask

Здесь мы подключаемся к планировщику путем создания объекта `Client`, указывающего на точку входа. Первый оператор `print` выведет на экран следующее:

```
<Client: 'tcp://192.168.2.20:8786' processes=2 threads=2, memory=500.00 MB>
```

Вывод отражает созданную нами инфраструктуру, состоящую из двух обработчиков с одним потоком и 250 Мб памяти на каждый обработчик.

После этого на экран выводятся версии всех используемых компонентов. Информация о планировщике выглядит так, как показано ниже:

```

scheduler
{'host': {'LANG': 'en_US.UTF-8',
          'LC_ALL': 'None',
          'OS': 'Linux',
          'OS-release': '5.13.0-19-generic',
          'byteorder': 'little',
          'machine': 'x86_64',
          'processor': 'x86_64',
          'python': '3.9.7.final.0',
          'python-bits': 64},
'packages': {'blosc': '1.9.2',
              'cloudpickle': '1.6.0',
              'dask': '2021.01.0+dfsg',
              'distributed': '2021.01.0+ds.1',
              'lz4': None,
              'msgpack': '1.0.0',
              'numpy': '1.19.5',
              'python': '3.9.7.final.0',
              'toolz': '0.9.0',
              'tornado': '6.1'}}

```

Здесь вы видите полную информацию о хосте, включая данные об операционной системе и типе процессора, а также о версии Python и установленных библиотек.

Ниже показан сокращенный вывод для двух наших обработчиков и клиента:

```

workers
tcp://127.0.0.1:32931
{'host': {'LANG': 'en_US.UTF-8',
...
          'python-bits': 64},
'packages': {'blosc': '1.9.2',
...
          'tornado': '6.1'}}
tcp://127.0.0.1:34719
{'host': {'LANG': 'en_US.UTF-8',
...
          'tornado': '6.1'}}

client
{'host': {'LANG': 'en_US.UTF-8',
...
          'tornado': '6.1'}}

```

Убедиться в том, что версии установленных библиотек являются совместимыми, бывает очень важно при работе в неоднородном кластере с участием множества машин. В нашем случае, когда у нас на одной машине запущен и планировщик, и клиент, и два обработчика, мы можем не беспокоиться о совместимости версий. Но когда в процесс вовлечено множество компьютеров, этот вопрос приобретает огромное значение. Теперь давайте разберем наше решение.

10.3.2. Запуск кода с помощью *dask.distributed*

Начали мы с подключения к планировщику:

```
from dask.distributed import Client
client = Client('127.0.0.1:8786')
```

Этот клиент будет неявно использоваться во всех наших дальнейших вызовах, если не указано иное. Вы помните по предыдущему разделу, что структуры данных Dask обладают планировщиками по умолчанию, но они будут автоматически заменены на распределенный планировщик.

СОВЕТ. Объект `client` предоставляет явный интерфейс, очень похожий на `API concurrent.futures`. Если вы хотите использовать этот интерфейс, обратитесь к главе 3. Мы здесь будем пользоваться распределенным фреймворком через интерфейсы, применяемые в науке о данных, в частности *dask.array*, имитирующий NumPy.

В нашем примере мы применим подход, связанный с использованием универсальных функций NumPy. Код функции для расчета одной точки в множестве Мандельброта останется таким же, как в предыдущей главе:

```
def compute_point(c):
    i = -1
    z = complex(0, 0)
    max_iter = 200
    while abs(z) < 2:
        i += 1
        if i == max_iter:
            break
        z = z**2 + c
    return 255 - (255 * i) // max_iter
```

Для вычисления множества Мандельброта нам необходимо подготовить матрицу, в каждой ячейке которой будет располагаться двумерная позиция, закодированная в виде комплексного числа. В предыдущей главе мы использовали показанный ниже код:

```
def prepare_pos_array(size, start, end, pos_array):
    size = pos_array.shape[0]
    startx, starty = start
    endx, endy = end
    for xp in range(size):
        x = (endx - startx)*(xp/size) + startx
        for yp in range(size):
            y = (endy - starty)*(yp/size) + starty
            pos_array[yp, xp] = complex(x, y)
```

А вы знаете, что происходит в этой строке?

В теории этот код работает, но на практике это будет тихий ужас. Обратите внимание, что в последней строке происходит не сохранение позиции в ячейке массива. Фактически здесь создается задача в графе задач для вычисления результата. Чтобы пояснить это, давайте создадим маленькое изображение 3×3, т. е. с размером блока, равным трем:

```
size = 3
pos_array = da.empty((size, size), dtype=np.complex128)
prepare_pos_array(3, start, end, pos_array)
pos_array.visualize("10-size3.png", rankdir="LR")
```

На рис. 10.11 показан получившийся в итоге граф задач. Здесь вы видите девять задач, каждая из которых состоит в обновлении отдельной ячейки. Это решение приемлемо только для небольших изображений. При размере изображения 1000×1000 мы получим 1 млн задач.

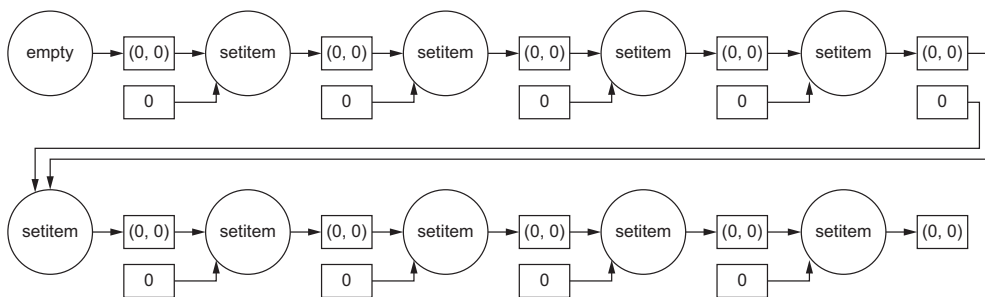


Рис. 10.11. Граф задач для запуска кода инициализации для девяти пикселей

В качестве альтернативы мы могли бы создать локальный массив NumPy, локально же инициализировать его и после этого выполнить разброс. Этот подход будет работать до тех пор, пока массив NumPy будет помещаться в память компьютера. В то же время это условие идет вразрез с основным предназначением библиотеки Dask, заключающимся в работе с большими структурами данных, не уместяющимися в памяти.

Таким образом, в качестве более реалистичной альтернативы Dask предлагает нам осуществлять вычисления с каждой секцией структуры данных независимо друг от друга, что позволит существенно уменьшить количество задач:

```
import dask.array as da
```

```
size = 1000
range_array = da.arange(0, size*size).reshape(size, size)
range__array = pos_array.rechunk(size // 2, size // 2)
range__array.visualize("10-rechunk.png", rankdir="TB")
range_array = range_array.persist()
```

Разбиваем массив на четыре блока размером (500, 500)

Теперь мы будем использовать изображение размером 1000×1000 . Мы инициализируем массив последовательными числами, с помощью которых сможем вычислить двумерные координаты. Начали мы с одномерного массива размером 1000×1000 , а затем изменили его форму на $(1000, 1000)$ с помощью метода `reshape`.

После этого мы разбили массив на порции $(500, 500)$, что в результате дало нам четыре блока. Наконец, мы сохранили блоки с помощью метода `persist`, чтобы подготовиться к вычислению двумерных позиций.

Подготовим массив с позициями. Здесь имеется в виду создание массива с двумерными позициями, закодированными в виде комплексных чисел:

```
def block_prepare_pos_array(size, pos_array):
    nrows, ncols = pos_array.shape
    ret = np.empty(shape=(nrows,ncols), dtype=np.complex128)
    startx, starty = start
    endx, endy = end
    for row in range(nrows):
        x = (endx - startx) * ((pos_array[row, 0] // size) / size) +
startx
        for col in range(ncols):
            y = (endy -
starty
                starty) * ((pos_array[row, col] % size) / size) +
starty
            ret[row, col] = complex(x, y)
    return ret
```

Эта функция преобразовывает массив последовательных чисел в массив позиций на основе значений в исходных ячейках. Об алгоритме беспокоиться не стоит, он просто конвертирует одномерную координату в двумерную. Главное здесь другое – давайте взглянем на код ниже и узнаем, что будет выведено:

```
pos_array = da.blockwise(
    lambda x: block_prepare_pos_array(size, x),
    'ij', range_array, 'ij', dtype=np.complex128)
pos_array.visualize("10-blockwise.png", rankdir="TB")
```

Здесь мы говорим Dask применить код инициализации из функции `block_prepare_pos_array` к каждому из четырех блоков. В качестве входного параметра мы указали наш массив `range_array`. Обратите также внимание на два параметра `ij` (т. е. `i` и `j`), которые указывают на связь между формами входного и выходного параметров (здесь они одинаковые).

В результате запуска этого кода будет создано всего четыре задачи, как показано на рис. 10.12. Если бы мы запустили исходный код, задач был бы миллион.

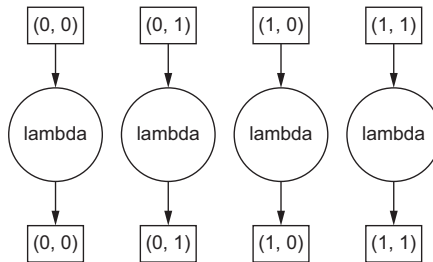


Рис. 10.12. Граф задач для кода инициализации с использованием блочной обработки

Пришло время применять генератор Мандельброта к нашей матрице:

```
from PIL import Image

u_compute_point = da.frompyfunc(compute_point, 1, 1)

image_arr = u_compute_point(pos_array)
image = Image.fromarray(image_np, mode="P")
image.save("mandelbrot.png")
```

Мы воспользовались функцией *frompyfunc*, позволяющей преобразовать родную функцию Python в универсальную функцию NumPy. Затем мы применили полученную функцию к нашей матрице *pos_array*.

Давайте выполним базовое профилирование нашего кода. Наша цель – увидеть прирост производительности при работе с большими изображениями. Ниже приведен код, который поможет нам произвести замер быстродействия:

```
from time import time

def time_scenario(size, persist_range, persist_pos, chunk_div=10):
    start_time = time()
    size = size
    range_array = da.arange(0, size*size).reshape(size, size).persist()
    range_array = range_array.rechunk(size // chunk_div, size // chunk_div)
    range_array = range_array.persist() if persist_range else range_array
    pos_array = da.blockwise(
        lambda x: block_prepare_pos_array(size, x),
        'ij', range_array, 'ij', dtype=np.complex128)
    pos_array = pos_array.persist() if persist_pos else pos_array
    image_arr = u_compute_point(pos_array)
    image_arr.visualize("task_graph.png", rankdir="TB")
    image_arr.compute()
    return time() - start_time
```

Наша функция запускает генератор Мандельброта, что позволяет нам параметризовать размер изображения. Также мы ввели

два параметра, говорящие о том, нужно ли применять метод `persist` к двум промежуточным массивам. Функция возвращает количество секунд, потребовавшихся для ее выполнения, что позволяет нам грубо судить о быстродействии нашего кода.

Давайте запустим наш код для изображения размером 500×500 с делителем на порции, равным двум (т. е. с разбивкой матрицы на четыре блока):

```
size = 500
time_scenario(size, False, False, 2)
```

На рис. 10.13 показан граф задач с четырьмя блоками и четырьмя узлами вычислений (`lambda` и `frompyfunc`).

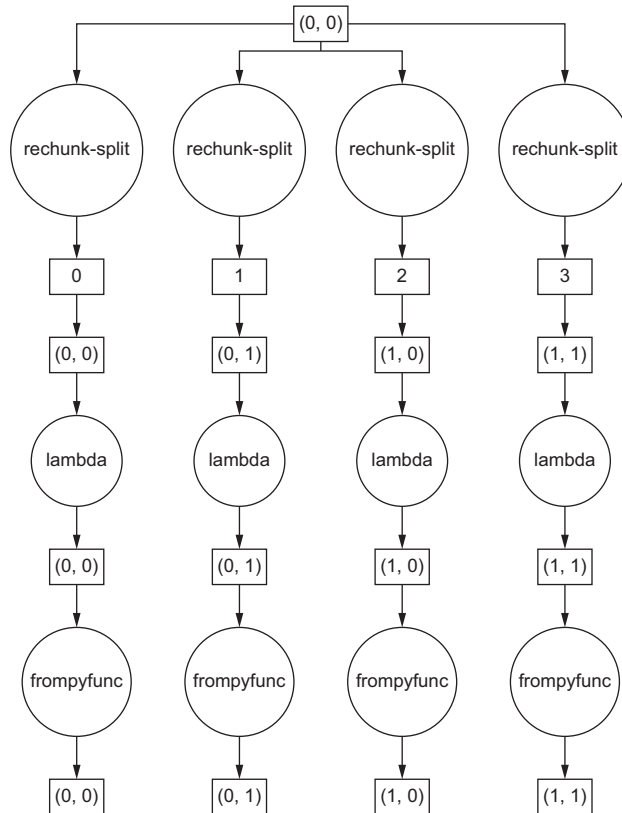


Рис. 10.13. Граф задач для вычисления множества Мандельброта с использованием четырех порций

Теперь давайте попробуем запустить код для изображения размером 5000×5000 с делителем на порции, равным десяти. Это приведет к разбивке массива на 100 блоков. Но, перед тем как сделать это, откроем в браузере адрес <http://127.0.0.1:8787>, по которому располагается наш дашборд:

```
size = 5000
client = client.restart()
time_scenario(size, True, True, 10)
```

После запуска этой строки обновите страницу в браузере, чтобы произвести очистку содержимого

При запуске функции `time_scenario` вы увидите в браузере анимированный процесс вычислений в реальном времени. Видео посредством книги передать трудно, но на рис. 10.14 я показал скриншот, сделанный в процессе выполнения задачи. Основной дашборд содержит пять диаграмм. Помните, что у нас запущено четыре обработчика, и топология графа задач у нас будет такая же, как на рис. 10.13, но с сотней колонок вместо четырех:

- маленький график слева сверху показывает количество байтов, сохраненных обработчиками;
- второй график слева отражает объем памяти, занимаемый каждым обработчиком, так что это просто более детализированный вид верхнего графика;
- на диаграмме слева внизу производится подсчет количества задач для каждого обработчика;
- на основной диаграмме (справа сверху) мы видим временную ось по горизонтали и количество обработчиков по вертикали. Каждый блок соответствует задаче из графа. При этом разным типам задач присвоены свои цвета;
- ну и справа внизу показаны статусы всех задач.

Я советую вам пройтись по всем пунктам меню на дашборде. К примеру, на странице `Profile` вы увидите визуализацию `SnakeViz` с профилировочной информацией, а на вкладке `Graph` – статусы всех задач из графа в реальном времени.



Рис. 10.14. Основная страница дашборда Dask

Ну и, наконец, посмотрим, как Dask справляется с наборами данных, размер которых превышает объем доступной памяти.

10.3.3. Работа с наборами данных, превышающими по объему доступную память

Вы помните, что библиотека Dask позволяет работать с наборами данных, превышающими по объему доступную память. При использовании нескольких компьютеров и, соответственно, большего объема памяти, Dask обладает возможностью распределять структуры данных по этим компьютерам.

Крайней мерой при работе с данными, не помещающимися в памяти, является их временное сохранение на диск. Однако в этом случае, как вы догадываетесь, об эффективности решения речи быть не может.

Мы запустим код генератора Мандельброта для большой матрицы размером 10 000×10 000. В одном случае мы будем использовать метод `persist` для временного хранения данных, а во втором нет:

```
size = 10000
print(size, False, False, time_scenario(size, False, False))
print(size, True, True, time_scenario(size, True, True))
```

Вывод на моем компьютере получился таким:

```
10000 False False 696
10000 True True 752
```

Вторая версия оказалась более медленной как раз из-за того, что память, необходимая для временного хранения массивов и производимых вычислений, превышает объем памяти, доступной для обработчиков. Эту проблему легко обнаружить на дашборде. На рис. 10.15 показаны два верхних левых графика на дашборде. По подписи к первому из них понятно, что в данном случае выполняется запись на диск. Кроме того, на графиках используются разные цвета для обозначения различий по месту хранения данных.

Необходимость сбрасывать данные на диск может привести к снижению производительности решения по сравнению с хранением их в памяти на несколько порядков. Здесь вы можете рассмотреть разные альтернативы, наиболее очевидной из которых является добавление памяти или расширение парка задействованных машин. В любом случае, если вы видите, что у вас происходит сохранение данных на диск, либо попытайтесь устранить это, либо убедитесь, что это не слишком сильно сказывается на производительности.

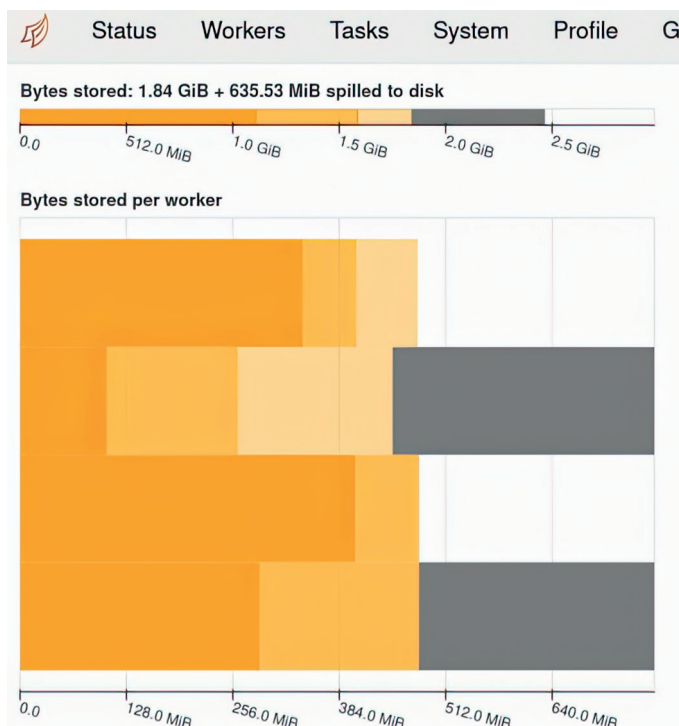


Рис. 10.15. На дашборде отображен процесс записи данных на диск

В этой главе мы познакомились с базовыми концепциями Dask. Заручившись этой информацией, вы сможете легче понять фундаментальные аспекты, влияющие на производительность при использовании Dask, и соответствующим образом применять их в различных архитектурах с учетом их особенностей и узких мест.

Заключение

- Библиотека Dask позволяет выполнять распределенные вычисления на нескольких машинах.
- Dask дает возможность работать с объектами вроде датафреймов и массивов, требующих больше места в памяти, чем доступно.
- В библиотеке Dask реализовано подмножество API таких распространенных библиотек, как pandas и NumPy, но с другой семантикой, обусловленной по большей части ленивыми вычислениями.
- Работая с Dask, вы можете анализировать графы задач еще до их исполнения, что позволяет лучше понять, какие вычисления предстоит сделать, и в некоторых случаях выработать план по оптимизации.

- Dask позволяет осуществить тонкую настройку процесса вычислений. Например, вы можете запросить у узлов локальное сохранение промежуточных вычислений для их дальнейшего использования.
- Вы можете рассекционировать данные по узлам, если это поможет ускорить будущие вычисления. Но этот процесс всегда связан с потерей эффективности в связи с необходимостью передавать данные между узлами.
- В своей основе Dask полагается на библиотеки `pandas` и `NumPy`, которые вы можете использовать напрямую.
- Dask позволяет воспользоваться интерфейсом вроде `concurrent.futures`, с которым мы познакомились в главе 3.
- Базовые алгоритмы для анализа данных должны принимать в расчет способность Dask использовать секционирование данных. Эффективность некоторых из них в подобных условиях может существенно снизиться по сравнению с последовательными версиями, доступными в `pandas` или `NumPy`.
- Библиотека Dask предлагает на выбор несколько планировщиков, среди которых есть распределенный планировщик, позволяющий диверсифицировать вычисления в рамках разных архитектур: от одной машины до огромных кластеров и облачных ресурсов.
- Планировщик `dask.distributed` располагает удобным дашбордом, который может быть использован при анализе и профилировании распределенных приложений.

Приложение А.

Настройка окружения

В данном приложении мы рассмотрим следующие темы:

- установка Anaconda;
- установка дистрибутива Python;
- установка Docker;
- вопросы, связанные с аппаратным обеспечением.

В данном приложении мы дадим некоторые рекомендации по настройке вашего рабочего окружения. В качестве основной версии интерпретатора будем использовать Python 3.10.

Вы можете использовать любую операционную систему на ваше усмотрение. В наше время большинство рабочих окружений строится на базе операционной системы Linux, но вы также можете использовать Windows или Mac OS X. Между Mac и Linux в этом отношении не такая большая разница. В то же время при использовании Windows могут возникнуть некоторые трудности. Если это ваш вариант, я бы посоветовал вам установить инструменты Unix, такие как оболочку Bash, либо же вы можете воспользоваться подсистемой Linux для Windows. Среда Cygwin также подойдет.

В качестве альтернативы на всех операционных системах вы можете воспользоваться контейнеризатором Docker с образом, включающим все необходимое программное обеспечение. Если вы решите пойти по этому пути, не забудьте установить сам Docker для своей операционной системы. В сопроводительных материалах я предлагаю базовый образ для Docker, а в главах, требующих установки до-

полнительного программного обеспечения, ссылаюсь на конкретные образы, которые также можно загрузить из репозитория.

Полный список программных средств, используемых в книге, приведен в соответствующих файлах `Dockerfile`. Этот список может быть полезен и без использования контейнеризатора Docker. Репозиторий этой книги располагается по адресу <https://github.com/tiagoantao/python-performance>.

A.1. Установка Anaconda Python

Anaconda Python является, пожалуй, наиболее распространенным дистрибутивом в области науки о данных и дата-инжиниринге. Лично я рекомендую пользоваться им для запуска кода из этой книги. После установки Anaconda создайте окружение для этой книги следующим образом:

```
conda create -n python-performance python=3.10 ipython=8.3
```

```
conda install pandas numpy requests snakeviz line_profiler blocs
```

В некоторых главах вам потребуется установить дополнительное программное обеспечение. В этих случаях я рекомендую клонировать оригинальное окружение и создать копию для каждой главы. Это можно сделать с помощью следующей команды:

```
conda create --clone python-performance -n NEW_NAME
```

После выполнения клонирования вы можете установить любое необходимое вам программное обеспечение в новое окружение без влияния на исходное окружение.

Я также советую создать отдельное окружение для каждой главы во избежание конфликтов между отдельными пакетами и библиотеками. Управлять пакетами бывает непросто даже при наличии хорошего диспетчера пакетов, такого как `conda`, так что лучше будет держать все окружения отдельно друг от друга.

Обновление окружений conda

Если вы уже давно работаете с Anaconda, возможно, будет лучше, если вы создадите новое окружение с помощью следующей команды:

```
conda create -n python-performance python=3.10
conda activate python-performance
```

Обновление старых окружений может занять немало времени и даже завершиться аварийно. Если вы только начали работать с Anaconda, вам лучше создать отдельное окружение для книги, а еще лучше – для каждой главы.

A.2. Установка дистрибутива Python

Вы можете выбрать дистрибутив Python исходя из собственных предпочтений, но я настоятельно рекомендую использовать Anaconda Python, фактически являющийся стандартом в области науки о данных и высокопроизводительных вычислений. При установке Anaconda (не сокращенной версии Miniconda) вы получите в свое распоряжение большую часть программного обеспечения, которое пригодится вам при чтении книги. На протяжении всей книги я исхожу из предположений о том, что у вас установлена Anaconda. Если вы используете другой дистрибутив, вам, возможно, придется адаптировать инструкции по установке дополнительных инструментов в соответствующих главах. Также я рекомендовал бы воспользоваться инструментом Poetry (<https://python-poetry.org>), который облегчит вам процесс управления пакетами.

Кроме того, вам понадобятся некоторые стандартные библиотеки, такие как NumPy и SciPy. Для построения графиков мы воспользовались библиотекой matplotlib. В разных главах присутствуют свои требования для установки нужных библиотек, таких как Cython, Numba, Apache Arrow или Apache Parquet. Если вы не используете ни conda, ни Poetry, вам необходимо будет прибегнуть к помощи утилиты pip для установки нужных библиотек и инструментов.

A.3. Использование Docker

Если вы хотите избежать необходимости устанавливать разные пакеты или используете Windows и желаете оставить свое рабочее окружение нетронутым, вы можете воспользоваться готовыми образами Docker, в которых интегрировано все необходимое для работы. Эти образы Docker предлагают окружение Linux вне зависимости от существующей операционной системы.

Базовый образ может быть запущен с помощью следующей команды:

```
docker run -v PATH_TO_THE_REPOSITORY:/code -ti tiagoantao/python-performance
```

При первом запуске будет загружен образ, на что может потребоваться определенное время. У вас будет своя оболочка, а код вы найдете в директории /code. Для глав с особыми требованиями я предложил отдельные образы Docker.

A.4. Вопросы, касающиеся аппаратного обеспечения

В книге мы используем довольно стандартное программное обеспечение, но некоторые настройки могут оказаться непростыми и оказывать влияние на процесс оптимизации:

- способ компилирования и подключения библиотеки может оказывать серьезное влияние на производительность (см. главу 4 про NumPy). При использовании рекомендованного дистрибутива значительно повышаются шансы на то, что у вас окажется наиболее производительная версия. В противном случае, а также если вы собираете библиотеку самостоятельно, прочитайте заключительную секцию главы 4;
- если вы используете образ Docker и работаете внутри виртуального окружения, очень трудно понять, полностью ли вы контролируете все, что происходит на компьютере, что затрудняет процесс профилирования и особенно управления кешем центрального процессора;
- те же ограничения могут возникать и при работе на локальном компьютере, на котором одновременно запущено большое количество других задач;
- экземпляры в облаке могут страдать от тех же проблем, если вам не обеспечен полный доступ к выделенной физической машине, что возможно, но достаточно дорого. Обычно вы используете физическую машину в распределенном режиме;
- разные конфигурации аппаратного обеспечения могут демонстрировать разную производительность. К примеру, скорость чтения с устройства памяти SSD обычно гораздо выше, чем с физического накопителя. Такая изменчивость характерна также для центральных процессоров, кеша, внутренних шин, памяти, дисков и сетевых устройств – особенно в части профилирования и задач, связанных с кешированием (центральный процессор, диск или сеть);
- в противовес предыдущим пунктам можно отметить, что лучшую производительность все описанные в этой книге приемы оптимизации покажут на хорошо настроенном выделенном сервере, не загруженном лишними задачами;
- в главе 9, касающейся использования в вычислениях графического процессора, мы предполагаем, что у вас установлен GPU от NVIDIA с микроархитектурой Pascal и выше.

В книге мы очень подробно обсуждаем все перечисленные моменты. Но важно понимать, что примеры, рассматриваемые в книге, хоть и имеют самостоятельную ценность, должны восприниматься как часть пути к оптимизированному решению конкретных проблем, связанных с производительностью. Следующий этап предполагает глубокое понимание реальных задач. Практические примеры – это лишь средство, но вовсе не цель.

Приложение Б. Использование Numba для создания эффективного низкоуровневого кода

Numba представляет собой фреймворк для автоматического преобразования кода на Python в низкоуровневый код, который должен выполняться на центральном или графическом процессоре. С точки зрения процессора это альтернатива расширению Cython. Причина того, почему мы целую главу посвятили именно Cython, а не Numba, состоит в желании показать, как именно работают такие инструменты, а не продемонстрировать сам факт их работы. А Numba, хоть и является очень полезным инструментом, в педагогическом плане не представляет никакой ценности, поскольку работает как чистая магия.

Для решения реальных задач компилятор Numba подходит едва ли не лучше, чем расширение Cython, поскольку требует меньшего участия от разработчика и приводит к похожим результатам. С точки зрения удобства я бы также порекомендовал Numba в качестве альтернативы Cython. В общем, как ни крути, но компилятор Numba можно расценивать как первого кандидата, когда необходимо преобразовать код на Python в низкоуровневый машинный код.

Numba принимает на вход функцию Python и динамически трансформирует ее код в оптимизированный машинный код

прямо во время запуска функции. Именно этим своим свойством Numba обязана своему званию динамического компилятора, или *JIT-компилятора* (just-in-time (JIT) compiler).

В данном приложении мы разработаем пример для выполнения на центральном процессоре. Вы можете использовать эти сведения в качестве подготовительного материала для главы 9, где мы говорим о применении в вычислениях графического процессора, или как дополнительный обучающий контент для изучения Numba применительно к вычислениям с помощью центрального процессора.

Для запуска приведенного здесь кода вам необходимо установить компилятор Numba. Если вы используете conda, вы можете ввести команду `conda install numba`. Образ Docker с установленными пакетами находится в репозитории `tiagoantao/python-performance-numba`.

В своем примере мы будем рассчитывать множество Мандельброта двумя способами: с помощью Python и посредством компилятора Numba, чтобы можно было затем сравнить быстродействие. Возможно, вы уже встречались с этим культовым изображением, одна из вариаций которого показана на рис. Б.1. Множество Мандельброта вычисляется в комплексном пространстве – мы будем использовать комплексные числа – и описывает поведение повторений уравнения $z = z^2 + c$, где c – это точка в пространстве, а z начинается в $(0, 0)$. Это вычисление на самом деле проще, чем кажется. Давайте взглянем на код для лучшего понимания деталей:

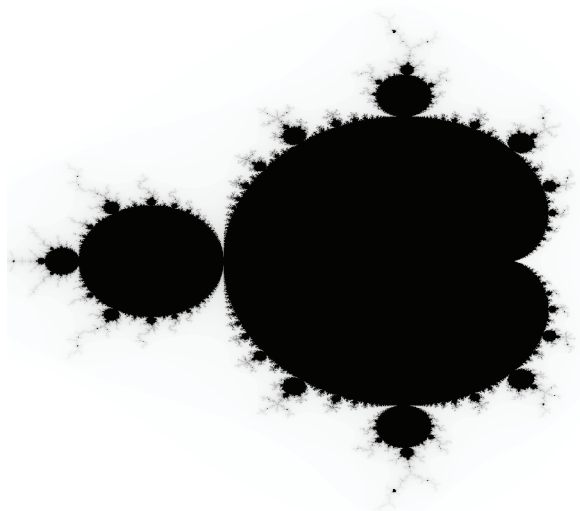


Рис. Б.1. Черно-белая отрисовка множества Мандельброта

В Python
реали-
зована
поддерж-
ка ком-
плексных
чисел

```
def compute_point(c, max_iter=200):
    num_iter = -1
    z = complex(0, 0)
    while abs(z) < 2:
        num_iter += 1
        if num_iter == max_iter:
            break
        z = z**2 + c
    return 255 - (255 * num_iter) // max_iter
```

Нам нужно указать максималь-
ный предел для количества
итераций, поскольку они могут
длиться бесконечно

Уравнение Мандельброта

На вход мы получаем точку в пространстве – c . Нас интересует количество итераций до того момента, когда модуль уравнения $z = z^2 + c$ с начальным z в точке $(0, 0)$ не превысит двух. Количество итераций определяет цвет пикселя для позиции c . Мы устанавливаем максимальное количество итераций как точки, близкие к нулю, но не превышающие 2, так что количество итераций будет бесконечным.

Таким образом, мы завершаем итерации, когда расстояние до z превысит 2. Точки, располагающиеся очень далеко от центра, останавливаются после первой итерации, а для близких точек итерации длятся вечно. Во избежание бесконечного количества вычислений мы задаем максимальное число итераций в виде параметра `max_iter`. Близко к границам количество итераций меняется хаотично (серые полутона на рис. Б.1). На изображении показано преобразование количества итераций в оттенок серого с максимальным числом итераций 255 в комплексном пространстве между $1,5 - 1,3i$ и $0,5 + 1,3i$.

Как видите, основная функция расчета множества Мандельброта достаточно проста. Версия, приведенная здесь, немного усложнена по сравнению с обычной из-за необходимости масштабировать выходные значения от 0 до 255 вне зависимости от количества итераций. Такое масштабирование облегчает вывод черно-белого 8-битного изображения (я использовал черно-белую гамму из-за ограничений печати).

Б.1. Создание оптимизированного кода с помощью Numba

Теперь мы создадим версию функции с использованием компилятора Numba с помощью декоратора `@jit`:

```
from numba import jit
compute_point_numba = jit()(compute_point)
```

Пусть вас не сбивают с толку декораторы – они являются не чем иным, как синтаксическим сахаром. Поскольку мы хотим одновременно поддерживать обе версии кода – родной и с использованием Numba – и выполнять их сравнение, применить декоратор будет

более удобно, поскольку синтаксис с применением символа @ позволил бы использовать только версию с Numba.

В связи с тем, что Numba представляет собой JIT-компилятор, первый вызов функции приведет к ее компиляции в представление LLVM (Low Level Virtual Machine – виртуальная машина низкого уровня), и эта операция будет выполнена лишь раз. Мы сделаем пробный вызов, чтобы в дальнейшем при сравнении производительности эта операция не учитывалась:

```
compute_point_numba(complex(4,4))
```

Вы должны соблюдать осторожность при работе с функциями, которые могут иметь побочные эффекты. Убедитесь, что пробный вызов не приведет ни к каким непредвиденным последствиям. В большинстве рабочих сценариев, где вам нет необходимости выполнять сравнение производительности, вы можете просто проигнорировать этот шаг.

Итак, у нас есть две версии нашей функции: родная (compute_point) и с использованием Numba (compute_point_numba). Нам нужно будет вызвать эти функции для каждой точки, которую мы хотим нарисовать. На вход мы подадим начальный и конечный углы, а также размерность, одинаковую для координат X и Y:

```
def do_all(size, start, end, img_array, compute_fun):
    startx, starty = start
    endx, endy = end
    for xp in range(size):
        x = (endx - startx)*(xp/size) + startx # из-за проблем с точностью
        for yp in range(size):
            y = (endy - starty)*(yp/size) + starty # из-за проблем с точностью
            img_array[yp, xp] = compute_fun(complex(x,y))
```

В этой простой функции мы проходим по всем точкам. Если вы подумали, что здесь было бы неплохо применить векторизацию, вы совершенно правы, и мы вернемся к этому вопросу чуть позже. Параметр size отвечает за количество пикселей в каждом измерении, start и end представляют позиции в комплексном пространстве, img_array – это выходной массив, а compute_fun – функция, которую мы будем использовать для расчета значений в каждой позиции.

Здесь есть небольшой нюанс, касающийся вычисления координат x и y. В теории мы могли бы добавить дельту к текущей позиции следующим образом:

```
x = startx
deltax = (endx - startx) / size
for xp in range(size):
    ....
    x += deltax
```

Проблема с таким чуть более быстрым подходом заключается в ошибке, связанной с точностью, которая будет накапливаться от итерации к итерации. В итоге это может привести к ошибочным результатам. Именно поэтому мы остановились на более дорогом вычислении $x = (\text{endx} - \text{startx}) * (xp / \text{size}) + \text{startx}$.

Для создания изображения зададим следующие параметры:

```
size = 2000
start = -1.5, -1.3
end = 0.5, 1.3

img_array = np.empty((size, size), dtype=np.uint8)
```

Нам также необходимо инициализировать массив, который мы будем использовать для вывода.

Теперь давайте сравним время, потребовавшееся для запуска обычной версии функции и версии с использованием Numba. В IPython это можно сделать так:

```
In [2]: %timeit do_all(size, start, end, img_array, compute_point_numba)
4.71 s ± 105 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
In [3]: %timeit do_all(size, start, end, img_array, compute_point)
50.4 s ± 2.94 s per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

На своем компьютере я получил десятикратное увеличение скорости выполнения.

Как видно на этом примере, компилятор Numba достаточно хорошо справился с задачей оптимизации кода Python, но, как и в случае с расширением Cython, здесь вас также могут ждать ловушки, если вы не сможете до конца избавиться от зависимостей исходного кода от интерпретатора CPython. Давайте искусственно воссоздадим эту проблему. Хотя в нашей задаче Numba в автоматическом режиме прекрасно справился со своей задачей, мы можем заставить компилятор вставить в итоговое решение инструкции CPython и посмотреть, как это скажется на производительности:

```
compute_point_numba_forceobj = jit(forceobj=True)(compute_point)
```

Вывод получился таким:

```
In [2]: %timeit do_all(size, start, end,
img_array, compute_point_numba_forceobj)
1min 46s ± 2.46 s per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Наш результат – 1 мин 46 с. Обратите внимание, что это самое медленное решение. Иногда Numba может оптимизировать часть кода, даже если ему не под силу выполнить полную оптимизацию.

Для вынужденной компиляции Python необходимо добавить к декоратору опцию `nopython=True`. Если Numba не удастся скомпилировать функцию, вы можете обратиться к документации компилятора по адресу <https://numba.readthedocs.io/en/stable/user/5minguide.html> и узнать, какой функционал Python поддерживается. Мы не будем здесь проходить по инструкции, поскольку она может претерпевать изменения.

Б.2. Написание параллельных функций в Numba

Компилятор Numba также позволяет вам писать параллельный многопоточный код с обходом ограничений, накладываемых GIL:

```
from numba import prange
```

```
@jit(nopython=True, parallel=True, nogil=True) ←
```

```
def pdo_all(size, start, end, img_array, compute_fun):
```

```
    startx, starty = start
```

```
    endx, endy = end
```

```
    for xp in prange(size): ← Используем функцию prange
```

```
        x = (endx - startx)*(xp/size) + startx
```

```
        for yp in range(size): # использовать ли здесь prange?
```

```
            y = (endy - starty)*(yp/size) + starty
```

```
            b = compute_fun(complex(x, y))
```

```
            img_array[yp, xp] = b
```

Для параллельного вычисления указываем опцию `parallel=True`, а для отказа от GIL – `nogil=True`

Используя функцию `prange`, мы просим компилятор Numba распараллелить итерации цикла. Параллельное вычисление стало возможным из-за отказа от ограничений GIL после того, как мы избавились от CPython. Результат запуска получился таким:

```
In [3]: %timeit pdo_all(size, start, end, img_array, compute_point_numba)
1.41 s ± 35.6 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Как видите, быстродействие повысилось в три раза по сравнению с последовательной версией скрипта. Очевидно, что это неплохое улучшение, но не пропорциональное увеличению количества задействованных ядер до восьми. В некоторых случаях, когда удастся полностью избавиться от влияния интерпретатора Python, функции Numba могут генерировать действительно параллельный код.

Б.3. Написание кода с использованием NumPy в Numba

Теперь, когда мы научились преобразовывать код на Python с помощью компилятора Numba, давайте рассмотрим версию с использо-

ванием универсальной функции NumPy, поскольку с этой библиотекой тесно работают все приложения в области науки о данных. Функции Numba могут быть преобразованы в универсальные функции NumPy, что является распространенной практикой в сфере науки о данных. Процесс преобразования достаточно прост:

```
from numba import vectorize

compute_point_ufunc = vectorize(
    ["uint8(complex128,uint64)"],
    target="parallel")(compute_point)
```

Мы воспользовались функцией *vectorize* в качестве обертки для функции *compute_point*. Здесь мы указываем, что функция может быть запущена и является параллельной. Также мы должны предоставить список типов сигнатур функций. Опциональные аргументы, такие как *max_iter*, становятся обязательными.

Этот код мы используем по-другому – путем передачи матрицы позиций, для которых необходимо вычислить результат:

```
size = 2000
start = -1.5, -1.3
end = 0.5, 1.3

def prepare_pos_array(start, end, pos_array):
    size = pos_array.shape[0]
    startx, starty = start
    endx, endy = end
    for xp in range(size):
        x = (endx - startx)*(xp/size) + startx
        for yp in range(size):
            y = (endy - starty)*(yp/size) + starty
            pos_array[yp, xp] = complex(x, y)

pos_array = np.empty((size, size), dtype=np.complex128)
prepare_pos_array(start, end, pos_array)
```

Функция *prepare_pos_array* просто подготавливает входной массив со всеми позициями координат для расчета. Недостатком такого способа является то, что нам потребуется дополнительная память для хранения как массива позиций, так и результатов.

Давайте произведем замер времени:

```
%timeit img_array = compute_point_ufunc(pos_array, 200)
```

Вывод на моей машине получился таким:

```
In [2]: %timeit img_array = compute_point_ufunc(pos_array, 200)
539 ms ± 7.17 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```


Результат оказался втрое лучшим по сравнению с параллельной версией без использования NumPy, и нам не пришлось прилагать для этого ровным счетом никаких усилий.

Это приложение может стать вашим проводником в мир компилятора Numba. Если вам интересно, как можно генерировать код для исполнения на графическом процессоре с использованием Numba, вы можете обратиться к главе 9.

Предметный указатель

Symbols

@ 137
Рассекционирование 334
__array__interface__ 120
@cuda.jit 303
%cython 160
%timeit 58, 142

A

Алгоритм сжатия 202
Аннотации 162
Асинхронный итератор 89

Б

Бессерверные вычисления 38
Блок потоков 300
Бродкастинг 135
Будущий объект 96

B

Векторизованная функция 141
Векторизованные операции 138
Вертикальное
 масштабирование 322
Взаимные блокировки 32
Временная сложность 63
Время доступа 36
Высокопроизводительный
 вычислительный кластер 323
Вытеснение 80
Вытесняющая многозадачность 80, 85
Вычислительная архитектура 33

Г

Генератор 73
Гиперпоток 102
Глобальная блокировка
 интерпретатора 40, 98
Гонки данных 32
Горизонтальное масштабирование 321
Граф задач 326
Графический процессор 33, 291
Графический процессор общего
 назначения 292
Группа строк 265

Д

Датафрейм 220
Дерево 62
Динамическая оперативная память 196

Ж

Жесткий диск 198

З

Закон Мура 29
Закон Эдхольма 29

И

Интернет-протокол 37
Исключение 86

К

Кеш первого уровня 196

Кеш процессора 35
Кодирование длин серий 267
Конкурентность 40
Конкурентные вычисления 79
Кооперативная многозадачность 80, 85
Копирование при записи 273
Кортеж 233
Корутина 85
Кратковременная память 196

Л

Ленивые вычисления 73

М

Массив 69
Менеджер контекста 89
Многомерный массив 117
Многопроцессная обработка 41
Множество 60
Множество Мандельброта 315
Модель выполнения 322

О

Облачные инфраструктуры 38
Обработчик 94, 340
Общие вычисления на графических процессорах 34
Объект future 96
Операции ввода-вывода 51
Отображение в памяти 271
Ошибка сегментации 185

П

Память 63
Параллелизм 40, 79
План выполнения 326
Планировщик 80, 338
Порционирование 105, 273
Последовательные вычисления 79
Поток 300
Потоковый мультипроцессор 298
Потоковый процессор 298
Представление 119
Представление памяти 173
Прихотливая индексация 130
Программирование на основе массивов 133
Протокол TCP 83
Профилирование 47, 50
Пул соединений 214

Р

Разделяемая память 109
Распределенный планировщик 339
Ребро 326
Регистр центрального процессора 196

С

Секционирование 269
Сериализация данных 88
Сетевой стек 37
Сеть 36
Сжатая файловая система 35
Сингулярное разложение матриц 146
Синхронное программирование 81
Словарь 61
Сопрограмма 85
Сопроцессор 297
Список 59

Т

Твердотельный накопитель 198
Тензорное ядро 298
Точность 226
Транслирование 135
Третичная память 36

У

Узел 326
Универсальная функция 141, 177
Упорядоченный список 60
Управляющая программа 85

Ф

Функция Мандельброта 306
Функция обратного вызова 96
Функция ядра 300

Х

Хеш 61
Хеш-функция 61

Ц

Центральный процессор 33, 77
Цепочка URL 261
Циклический переход 184

Ш

Шардирование 31

Я

Ядро CUDA 298

Ядро ЦП 77

A

add_signal_handler 114

aiohttp 89

Apache Arrow 241

apply 234

array 70

assert 303

as_strided 133

async 84

async def 85

asyncio 84

await 84, 85, 86

B

base 122

BLAS 146

blocksize 328

Blosc 199

C

cdef 166

cdivision 240

chdir 262

chunksize 274

close 104

concurrent.futures 93, 100

contains 253

ConvertOptions 244

cpdef 167

cProfile 50

CPU 33

CPython 39

ctypes 158

CUDA 294, 298

cuda.blockDim 305

cuda.blockIdx 305

cuda.grid 303

cuda.threadIdx 305

cuDNN 311

CuPy 294, 311

Cython 156

CYTHON_TRACE 171

D

Dask 322

dask.array 344

del 229

df.join 232

DHCP 37

DNS 37

E

eval 237

F

fromarray 121

frompyfunc 347

fsspec 257

G

Game of Life 179

get 253

get_scheduler 339

getsizeof 64

GIL 40, 98

GithubFileSystem 258

GPGPU 34, 292

GPU 33, 291

H

HTTPS 37, 209

I

id 66

imap 103

in 59

index 62, 225

inplace 229

Intel MKL 146

IP 37

IronPython 40, 99

is 122

iterrows 233

itertuples 233

J

join 104

JSON 209

Jython 40, 99

K

kernprof 56

L

LAPACK 146

line_profiler 56, 169

LineProfiler 171
linetrace 170
list 92
LZ4 203

M

map 90
map_async 103
marshal 88
may_share_memory 123
memoryview 123, 173

N

NA 225
nanny 340
nbytes 120
NestedDirectoryStore 283
Node.js 41
nogil 176
not_equal 246
np.array 120
np.dot 137
np.empty 135
np.matmul 137
np.memmap 272
np.minimum 140
np.shares_memory 123
np.vectorize 141
Nsight Systems 317
NumExpr 204
NumPy 117

O

OpenBLAS 146, 149
OpenMP 189
os.cpu_count 102
os.sched_getaffinity() 102

P

pandas 219
Parquet 263
ParquetWriter 275
partial 114, 286
persist 330
pickle 88
Pillow 119
Pipe 113
Plasma 250
Pool 102
prange 189
profile 50

pstats 54
put_nowait 112
PyPy 39, 99
Pyrex 159
Python 28

Q

qsize 113
Queue 110
QUIC 214

R

range 61
read_csv 230
reduce 90
repartition 335
repeat 315
REST 209
RLE 267
rot90 132
rpy2 248

S

SciPy 147
self_destruct 246
Series 235
set_index 336
settimeout 213
shape 120
signal 114
signal.SIG_IGN 114
signal.SIGINT 114
sleep 95
SnakeViz 54
Snappy 267
socket 210
socketserver 211
strides 127
submit 95
SWIG 158
sync 200
sys 64

T

T 131
TCP 209
TCP-coker 37
terminate 104
timeit 58
tkinter 186
TLS 209

to_bytes 88
to_numpy 236

U

UDP 37, 210
uint8 120

V

vectorize 363

W

walk 259
write_table 264

X

xz 221

Y

yield 74

Z

Zarr 276
zipfile 259
Zstandard 203
ZSTD 267

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «КТК Галактика» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, пр. Андропова д. 38 оф. 10.

При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.

Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине:

www.galaktika-dmk.com.

Оптовые закупки: тел. (499) 782-38-89.

Электронный адрес: **books@aliens-kniga.ru**.

Тиаго Антао

Сверхбыстрый Python

Главный редактор *Мовчан Д. А.*

dmkpress@gmail.com

Зам. главного редактора *Сенченкова Е. А.*

Перевод *Гинько А. Ю.*

Корректор *Абросимова Л. А.*

Верстка *Луценко С. В.*

Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.

Гарнитура «NewBaskervilleC». Печать цифровая.

Усл. печ. л. 30,06. Тираж 100 экз.

Веб-сайт издательства: www.dmkpress.com

Быстрый код на чистом Python, оптимизированные библиотеки и аппаратное обеспечение, позволяющее использовать все преимущества параллельной обработки данных, — это цена входа в мир машинного обучения и анализа больших данных. Книга, которую вы держите в руках, предлагает уникальные техники ускорения с акцентом на большие данные и станет вашим надежным проводником в мир оптимизации решений на базе Python. Вы узнаете, как оптимизировать работу со встроенными структурами данных и ускорить решения за счет конкурентного выполнения, а также научитесь сокращать объем занимаемой данными памяти без ущерба для их точности.

Ознакомившись с тщательно проработанными примерами, вы узнаете, как добиться большей производительности популярных библиотек, таких как NumPy и pandas, и как эффективно обрабатывать и хранить данные. В книге используется целостный подход к повышению эффективности решений, так что вы научитесь оптимизировать и масштабировать целые системы — начиная от кода и заканчивая архитектурой.

В числе рассматриваемых тем:

- написание эффективного кода на чистом Python;
- оптимизация использования библиотек NumPy и pandas;
- внедрение кода на Cython для критически важных фрагментов;
- разработка оптимальных структур для хранения данных;
- оптимизация кода с учетом разных архитектур;
- разработка решений на Python для вычисления на GPU.

Книга предназначена для разработчиков Python, знакомых с основами языка и принципами конкурентных вычислений.

Тиаго Антао — один из соавторов Biopython, популярного пакета для биоинформатики на Python.

Интернет-магазин: www.dmkpress.com

Оптовая продажа: КТК «Галактика»
books@aliens-kniga.ru



«Отличная книга для изучения методик написания эффективного кода на Python».

*Ор Голан,
Qedma Quantum Computing*

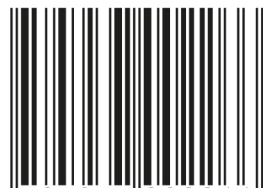
«Если вам кажется, что Python слишком медленный, эта книга для вас! Параллелизм, векторизация, использование Cython и Numba для компиляции кода в C, решения, задействующие GPU. Подарите эту книгу каждому дата-сайентисту в вашем офисе!»

Джеймс Луи, Mediaocean

«Время, которое вы потратите на чтение этой книги, с лихвой окупится при разработке эффективных приложений на Python».

Рууд Гуйсен, Simbeyond

ISBN 978-5-93700-226-6



9 785937 002266 >