



# 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

<b>1</b> <i>Острая нехватка производительности при обработке данных</i> .....	<b>27</b>
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
<b>2</b> <i>Извлечение максимума возможного из встроенных средств Python</i> .....	<b>46</b>
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

<b>4</b>	<b><i>Высокопроизводительный NumPy</i></b> .....	<b>117</b>
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**

<b>5</b>	<b><i>Реализация критически важного кода с помощью Cython ...</i></b>	<b>155</b>
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
<b>6</b>	<b><i>Иерархия памяти, хранение данных и работа с сетью ...</i></b>	<b>192</b>
6.1.	Как современная архитектура аппаратных средств влияет на эффективность кода Python .....	194

6.1.1. Неожиданное 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 Aggow для делегирования задач более эффективным языкам и системам....	247
7.5.1. Предпосылки архитектуры межъязыкового взаимодействия Aggow .....	247
7.5.2. Операции с нулевым копированием с использованием сервера Plasma от Aggow.....	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 PyAggow .....	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
<b>10 Анализ больших данных с использованием библиотеки Dask .....</b>	<b>321</b>
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
<b>Приложение А. Настройка окружения .....</b>	<b>353</b>
A.1. Установка Anaconda Python.....	354
A.2. Установка дистрибутива Python .....	355
A.3. Использование Docker.....	355
A.4. Вопросы, касающиеся аппаратного обеспечения.....	355
<b>Приложение Б. Использование Numba для создания эффективного низкоуровневого кода.....</b>	<b>357</b>
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
<b>Предметный указатель .....</b>	<b>365</b>

# 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. Заменяв 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 беспроигрышной 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.

# 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 Zang, 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 компанией 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 использует

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 авторскую, и он продолжает плодотворно работать над переводом новых.

Помимо перевода книг, Александр ведет свой канал в Telegram ([https://t.me/alexanderginko\\_books](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.

## 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.

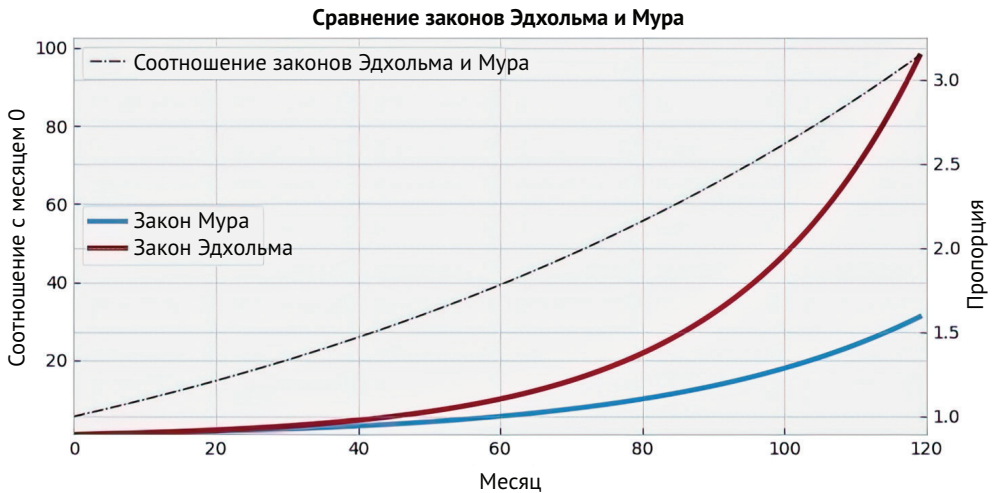
*Python* стал наиболее предпочтительным языком или некоей связующей прослойкой для выполнения всей рутинной работы в условиях этого нескончаемого потока или даже потопа данных. Популярность Python в науке о данных и инженерии данных обусловила рост его востребованности во всех остальных сферах, что сделало 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 поговорим о проблеме, 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?

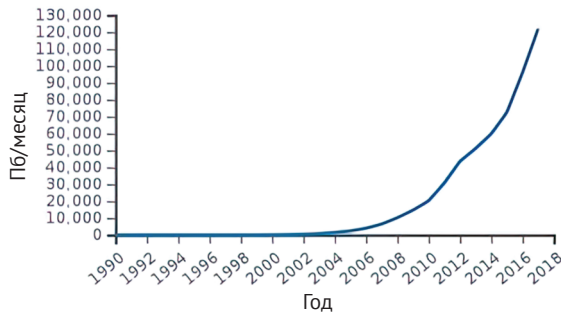
Возможно, вы слышали о существовании двух законов, носящих имена Мура (Moore) и Эдхольма (Edholm), которые в совокупности рисуют очень драматичную картину, связанную с экспоненциальным ростом собираемых данных и отставанием индустрии по обработке этих самых данных. Так, закон Эдхольма гласит, что объем 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 немаломо 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](https://en.wikipedia.org/wiki/Internet_traffic))

Кроме того, по данным сайта [uschamberfoundation.org](http://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 вам 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 умещалось 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 вам 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-

сая 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, построены 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
<b>Центральный процессор</b>		
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
<b>Оперативная память</b>		
DIMM	8 Гб	100 нс
<b>Вторичная память</b>		
SSD	256 Гб	50 мкс
HDD	2 Тб	5 мс
<b>Третичная память</b>		
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

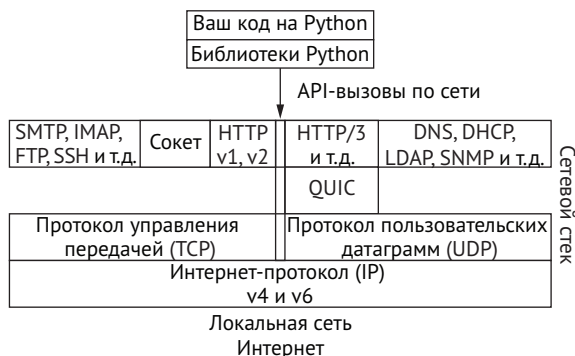
В высокопроизводительных вычислительных системах *сеть* (network) используется как для хранения данных, так и (в особенности) для их обработки путем увеличения вычислительных мощностей. И даже если нам очень хочется обходиться ресурсами одного компьютера, зачастую это бывает невозможно, и требуется привлечение 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 быстродействие 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». Полагаю, 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 быстродействия 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 быстродействием 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/tiagoantao/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 постараемся 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.