

Бриан Тоуманнен

Программирование GPU при помощи Python и Cuda

ДМК
ИЗДАТЕЛЬСТВО

УДК 004.438Python
ББК 32.973.22
Т50

Тоуманен Б.
Т50 Программирование GPU при помощи Python и CUDA / пер. с анг. А. В. Борескова. – М.: ДМК Пресс, 2020. – 254 с.: ил.

ISBN 978-5-97060-821-0

Книга предлагает быстрое погружение в программирование GPU. Вы узнаете, как применять закон Амдала, использовать профилировщик для определения узких мест в коде на Python, настроить окружения для программирования GPU. По мере чтения вы будете запускать свой код на GPU и писать полноценные ядра и функции на CUDA C, научитесь отлаживать код при помощи NSight IDE и получите представление об известных библиотеках от NVIDIA, в частности cuFFT и cuBLAS. Вооружившись этими знаниями, вы сможете написать с нуля глубокую нейронную сеть, использующую GPU, и изучить более основательные темы.

Книга предназначена для разработчиков и специалистов по обработке данных, которые хотят познакомиться с основами эффективного программирования GPU для улучшения быстродействия, используя программирование на Python. Желательно общее знакомство с базовыми понятиями математики и физики, а также опыт программирования на Python и любом основанном на C языке программирования.

УДК 004.438Python
ББК 32.973.22

Authorized Russian translation of the English edition of Hands-On GPU Programming with Python and CUDA ISBN 9781789136678 © 2018 Packt Publishing.

This translation is published and sold by permission of Packt Publishing, which owns or controls 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 978-1-78899-391-3 (анг.)
ISBN 978-5-97060-821-0 (рус.)

© 2018 Packt Publishing
© Оформление, издание, перевод, ДМК Пресс, 2020

Содержание

Об авторе 10

О рецензенте 11

Предисловие 12

Глава 1. Почему программирование GPU? 18

Технические требования 19
Параллелизация и закон Амдала 19
  Использование закона Амдала 21
  Множество Мандельброта 22
Профилировка вашего кода 25
  Использование модуля cProfile 25
Резюме 26
Вопросы 27

Глава 2. Настройка окружения для программирования GPU 28

Технические требования 29
Убедитесь, что у вас есть требуемое оборудование 29
  Проверка вашего оборудования (Linux) 30
  Проверка вашего оборудования (Windows) 31
Установка драйверов для GPU 33
  Установка драйверов GPU (Linux) 33
  Установка драйвера GPU (Windows) 35
Установка окружения для программирования на C++ 35
  Настройка GCC, Eclipse IDE и графических зависимостей (Linux) 35
  Установка Visual Studio (Windows) 36
  Установка CUDA Toolkit 38
Установка окружения Python для программирования GPU 39
  Установка PyCUDA (Linux) 40
  Создание скрипта для настройки окружения (Windows) 40
  Установка PyCUDA (Windows) 41
  Проверка PyCUDA 42
Резюме 42
Вопросы 43

Глава 3. Начало работы с PyCUDA 44
Технические требования 44
Опрос вашего GPU 45
Опрос вашего GPU при помощи PyCUDA 46
Использование класса `gpuarray` модуля PyCUDA 49
Перенос данных в и из GPU при помощи `gpuarray` 49
Использование основных поэлементных операций через методы `gpuarray` 50
Использование ElementWiseKernel из PyCUDA для выполнения поэлементных операций 55
Возвращаемся к множеству Мандельброта 58
Краткая вылазка в функциональное программирование 61
Основа параллельного сканирования и редуцирования 63
Резюме 64
Вопросы 65

Глава 4. Ядра, нити, блоки и сетки 66
Технические требования 67
Ядра 67
Функция SourceModule из PyCUDA 67
Нити, блоки и сетки 70
Игра «Жизнь» Джона Конвея 70
Синхронизация и взаимодействие нитей 77
Использование функции устройства `__syncthreads()` 77
Использование разделяемой памяти 80
Алгоритм параллельной префиксной суммы 82
Алгоритм наивный параллельной префиксной суммы 82
Исключающая префиксная сумма и включающая префиксная сумма 85
Эффективный алгоритм параллельной префиксной суммы 85
Эффективный алгоритм параллельной префиксной суммы (реализация) 87
Резюме 89
Вопросы 90

Глава 5. Потоки, события, контексты и одновременность 91
Технические требования 92
Синхронизация устройства CUDA 92
Использование класса stream из PyCUDA 93
Параллельная игра «Жизнь» Конвея при помощи потоков CUDA 97
События 100
События и потоки 102
Контексты 103
Синхронизация в текущем контексте 104

Создание контекста ... 105
Многопроцессность и многонитиевость на стороне хоста ... 106
Различные контексты для параллельности на стороне хоста ... 107
Резюме ... 110
Вопросы ... 111

Глава 6. Отладка и профилирование вашего кода на CUDA ... 112
Технические требования ... 113
Использование **`printf`** внутри ядер CUDA ... 113
Использование **`printf`** для отладки ... 115
Заполняем пробелы в CUDA C ... 119
Использование NSight IDE для разработки и отладки кода на CUDA C ... 124
Использование NSight с Visual Studio IDE под Windows ... 125
Использование NSight с Eclipse под Linux ... 128
Использование NSight для понимания варпа в CUDA ... 131
Использование профайлера nvprof и Visual Profiler ... 134
Резюме ... 136
Вопросы ... 136

Глава 7. Использование библиотек CUDA вместе со Scikit-CUDA ... 137
Технические требования ... 138
Установка Scikit-CUDA ... 139
Базовая линейная алгебра при помощи cuBLAS ... 139
Функции 1-го уровня AXPY в cuBLAS ... 139
Другие функции cuBLAS 1-го уровня ... 141
GEMV 2-го уровня в cuBLAS ... 142
Функции 3-го уровня GEMM в cuBLAS для измерения производительности GPU ... 144
Быстрое преобразование Фурье при помощи cuFFT ... 147
Простое одномерное FFT ... 148
Использование FFT для свертки ... 149
Использование cuFFT для двумерной свертки ... 150
Использование cuSolver из Scikit-CUDA ... 155
Сингулярное разложение (SVD) ... 155
Использование SVD для анализа методом главных компонент (PCA) ... 156
Резюме ... 158
Вопросы ... 158

Глава 8. Библиотеки функций для GPU CUDA и Thrust ... 159
Технические требования ... 160
Библиотека функций GPU cuRAND ... 160

Оценка π при помощи метода Монте-Карло ... 161
CUDA Math API ... 165
Краткий обзор определенных интегралов ... 165
Вычисление определенного интеграла при помощи метода Монте-Карло ... 166
Пишем тесты ... 172
Библиотека CUDA Thrust ... 174
Использование функторов в Thrust ... 176
Резюме ... 178
Вопросы ... 178

Глава 9. Реализация глубокой нейросети ... 180

Технические требования ... 181
Искусственные нейроны и нейросети ... 181
Реализация плотного слоя искусственных нейронов ... 182
Реализация слоя мягкого максимума ... 187
Реализация функции потери перекрестной энтропии ... 189
Реализация последовательной сети ... 189
Реализация методов вывода ... 191
Градиентный спуск ... 193
Подготовка и нормализация данных ... 197
Данные Iris ... 197
Резюме ... 200
Вопросы ... 200

Глава 10. Работа с компилированным кодом для GPU ... 201

Запуск откомпилированного кода при помощи Ctypes ... 202
Снова возвращаемся к вычислению множества Мандельброта ... 202
Компиляция и запуск PTX-кода ... 208
Написание «оберток» для CUDA Driver API ... 209
Использование CUDA Driver API ... 213
Резюме ... 216
Вопросы ... 217

Глава 11. Оптимизация быстродействия в CUDA ... 218

Динамический параллелизм ... 219
Быстрая сортировка при помощи динамического параллелизма ... 220
Векторные типы данных и доступ к памяти ... 222
Потокобезопасные атомарные операции ... 224
Перестановки в пределах варпа ... 225
Вставка PTX-ассемблера прямо в код ... 228

Оптимизированная по быстродействию версия суммирования элементов массива ... 232
Резюме ... 235
Вопросы ... 235

Глава 12. Куда идти далее? ... 237
Расширение знаний о CUDA и программировании GPGPU ... 238
Системы из нескольких GPU ... 238
Кластерные вычисления и MPI ... 238
OpenCL PyOpenCL ... 239
Графика ... 239
OpenGL ... 240
DirectX12 ... 240
Vulkan ... 240
Машинное обучение и компьютерное зрение ... 241
Основы ... 241
cuDNN ... 241
Tensorflow и Keras ... 242
Chainer ... 242
OpenCV ... 242
Технология блокчейн ... 242
Резюме ... 243
Вопросы ... 243

Ответы на вопросы ... 244

Предметный указатель ... 250

Глава 1

Почему программирование GPU?

Оказывается, что кроме непосредственно рендеринга графические процессоры (GPU, Graphics Processing, Unit) также предоставляют обычным пользователям возможность заниматься *массивно-параллельными вычислениями* – обычный человек может купить современный GPU за $2000 в магазине электроники, воткнуть в домашний компьютер и сразу же начать использовать всю вычислительную мощность, которая еще 5–10 лет назад была доступна только в лабораториях суперкомпьютерного моделирования крупнейших корпораций и университетов. Доступность GPU в последние годы сделалась намного заметнее, например добытчики применяют эти процессоры для майнинга таких криптовалют, как биткойн. Генетики и биологи при помощи GPU анализируют ДНК и проводят различные исследования, физикам и математикам они необходимы для широкомасштабного моделирования, исследователи в области искусственного интеллекта могут программировать GPU для написания пьес и музыки, крупные интернет-компании, такие как Google и Facebook, используют фермы серверов с GPU для выполнения сверхзадач в машинном обучении… Данный список можно продолжать и продолжать.

Эта книга в первую очередь написана для того, чтобы как можно быстрее познакомить вас с программированием GPU и научить правильно задействовать их вычислительные возможности независимо от того, является ли это конечной целью. Я собираюсь говорить об основах программирования GPU вместо разбора технических деталей о том, как они работают. Ближе к концу книги вам будут предоставлены ссылки на онлайн-ресурсы и дополнительные источники, благодаря чему у вас появится прекрасная возможность применить полученные знания на практике. (Всю необходимую информацию о требуемых технических знаниях и оборудовании вы найдете в конце этого раздела.)

Также мы будем работать с CUDA – библиотекой для расчетов общего назначения на GPU (GPGPU) от NVIDIA, выпустившей ее в 2007 г. Являясь довольно зрелой и устойчивой платформой, которую легко использовать, она предоставляет несравнимый с другими набор первоклассных математических и AI-

библиотек, не сравнимый с другими, крайне простой в установке и интеграции. Кроме того, существуют стандартные и легкодоступные библиотеки для Python, такие как PyCUDA и SciKit-CUDA, которые делают программирование GPU доступным даже для начинающих. Именно по этим причинам мы выбрали использовать CUDA в данной книге.



CUDA *всегда* произносится как *ку-да* и никогда как C-U-D-A! Изначально CUDA было сокращением от *Compute Unified Device Architecture*, но в дальнейшем NVIDIA отказалась от столь длинного названия, и теперь CUDA употребляется в качестве обычного слова, записанного заглавными буквами.

Мы начнем наше путешествие в программирование GPU с обзора **закона Амдала**, который является довольно простым и эффективным способом оценить потенциальный выигрыш от переноса программы или алгоритма на GPU. Он поможет четко определить, стоит ли нам переписывать код для использования GPU. Далее, для того чтобы определить узкие места в программе, мы вкратце рассмотрим профилирование кода на Python при помощи модуля *cProfile*.

В результате изучения этой главы вы сможете:

- понимать закон Амдала;
- применять закон Амдала в контексте вашего кода;
- использовать модуль *cProfile* для базового профилирования кода на Python.

Технические требования

Для этой главы предлагается использовать установленную Anaconda Python 2.7: **https://www.anaconda.com/download/**.

Код доступен на GitHub: **https://github.com/PacktPublishing/Hands-On-GPU-Programming-with-Python-and-CUDA**.



За дополнительной информацией о требованиях обратитесь к предисловию к этой книге; о требованиях к программному обеспечению и оборудованию прочтите README в **https://github.com/PacktPublishing/Hands-On-GPU-Programming-with-Python-and-CUDA**.

Параллелизация и закон Амдала

Прежде чем мы приступим к раскрытию потенциала GPU, нам следует понять, как их вычислительная мощность соотносится с современными процессорами от Intel/AMD – мощность заключается не в более высокой тактовой частоте, как у CPU или более сложных ядер. На самом деле отдельное ядро GPU простое и уступает ядру традиционного CPU, использующего многие хитрые приемы, такие как предсказание ветвления, для уменьшения **латентности** вычислений. Латентность означает время, затраченное на вычисление, от начала и до конца.

Мощность GPU заключается в том, что в нем намного больше ядер, чем в традиционном CPU, что означает мощный прорыв в **пропускной способности** (*throughput*). **Пропускная способность** есть количество вычислений, которые могут быть выполнены одновременно. Давайте рассмотрим аналогию, чтобы лучше понять, что это за процесс. GPU можно сравнить с очень широкой дорогой, предназначенной для огромного количества медленно двигающихся машин одновременно (высокая пропускная способность, высокая латентность), в то время как CPU – это узкий хайвей, на котором разрешено движение ограниченного числа машин только в одну сторону, но каждая из них может ехать очень быстро (низкая пропускная способность, низкая латентность).

Мы можем получить представление об увеличении пропускной способности, всего лишь рассмотрев, сколько ядер вмещает в себя GPU. Обычно CPU от Intel/AMD содержит от двух до восьми ядер, в то время как базовый GPU низкого уровня NVIDIA GTX 1050 – *640 ядер*, а топовый NVIDIA RTX 2080 Ti – *4352 ядра*! Мы можем использовать столь массивную пропускную способность при условии, что нам известно, как правильно **распараллелить** любую программу или алгоритм, которые мы хотим ускорить. Под **распараллеливанием** мы подразумеваем переписывание программы или алгоритма таким образом, чтобы всю нашу работу одновременно можно было выполнить сразу на большом количестве процессоров (ядер). И вновь напрашивается аналогия из обычной жизни.

Представьте, что вы строите дом, проект готов и закуплены все необходимые материалы. Вы нанимаете одного рабочего и оцениваете, что для постройки здания понадобится минимум 100 ч. Теперь представьте, что дом можно возвести таким образом, что вся работа будет равномерно распределяться между нанятыми рабочими – тем самым потребуется 50 ч для двух рабочих, 25 – для четырех, 10 – для десяти. Число часов для постройки дома будет равно 100, поделенное на число рабочих. Это пример **распараллеливаемой задачи**.

Таким образом, мы видим, что постройка будет идти вдвое быстрее для двух и в десять раз быстрее для десяти человек, работающих вместе (т. е. *параллельно*). Вывод такой: если у нас N рабочих, то строительство будет в N раз быстрее. В этом случае N называют **ускорением** от параллелизации по сравнению с последовательной версией задачи.

Прежде чем приступить к программированию параллельной версии алгоритма, как правило, лучше всего начать с оценки *потенциального ускорения*, которое даст параллелизация. Это поможет нам определить, стоит ли тратить ресурсы и время на распараллеливание данной программы. Поскольку реальная жизнь гораздо сложнее приведенного примера, то становится понятным, что мы не сможем распараллелить каждую программу целиком – скорее всего, только часть нашей программы будет хорошо распараллеливаться, в то время как оставшуюся придется выполнять последовательно.

Использование закона Амдала

Теперь мы выведем **закон Амдала**, который является простой арифметической формулой, что используется для оценивания потенциального ускорения от распараллеливания части кода из последовательной программы на ряд процессоров. Мы проделаем это, продолжая нашу предыдущую аналогию со строительством дома.

В прошлый раз мы рассмотрели только процесс реального физического строительства дома как один интервал времени, но сейчас нас интересует еще и время, требуемое для его проектирования как часть строительства. Допустим, что лишь один человек во всем мире может спроектировать ваш дом – вы – и требуется 100 ч для создания генерального плана. Вы прекрасно понимаете, что никто другой на планете не сможет сравниться с вашим архитектурным мастерством, поэтому нет никакого шанса, что часть этой работы может быть поделена со сторонними архитекторами. Разработка проекта вашего дома займет ровно 100 ч, независимо от того, каким количеством ресурсов вы располагаете на данный момент или сколько человек вы можете нанять. Поэтому если в строительстве дома у вас задействован всего лишь один рабочий, то время, затраченное на возведение здания, будет равно 200 ч – 100 ч на проектирование и 100 ч для выполнения одним рабочим всех строительных работ. Если нанять двух рабочих – время, соответственно, увеличится до 100–150 ч на проектирование, но вот само строительство уже займет 50 ч. Таким образом, общее время, затраченное на постройку дома, будет равно 100 + 100/*N*, где *N* – число нанятых рабочих.

Теперь вернемся назад и посмотрим, сколько времени уйдет на строительство, если мы наймем всего лишь одного рабочего – мы будем рассматривать его как основу для вычисления ускорения при найме дополнительных рабочих. Если мы нанимаем одного, то видим, что строительство дома занимает столько же времени, сколько и его проектирование, – 100 ч. Поэтому мы с полной уверенностью можем сказать, что часть времени, затраченная на проектирование, равна 0,5 (50 %), и часть времени, затраченная на строительство, также равна 0,5 (50 %). При этом в сумме эти две части дают 1, т. е. 100 %. Что же произойдет, если нам придется нанять дополнительных рабочих? Когда у нас есть двое рабочих, то время, затраченное на строительство, также уменьшается вдвое. Поэтому по сравнению с последовательной версией нам потребуется 0,5 + 0,5/2 = 0,75 (75 %) времени для решения исходной задачи и 0,75 × 200 ч = 150 ч. То есть мы видим, как это работает на деле. Также мы понимаем, что если наймем *N* рабочих, то сможем вычислить долю времени по формуле 0,5 + 0,5/*N*.

Теперь давайте определим *ускорение*, которое мы получаем, нанимая дополнительных рабочих. Поскольку необходимо 75 % времени, если в нашем распоряжении имеются двое рабочих, то мы возьмем обратное к 0,75 для получения ускорения от нашего распараллеливания. Итак, ускорение будет равно 1/0,75, и процесс постройки дома пойдет примерно в 1,33 раза быстрее, чем если бы

мы располагали силой всего одного рабочего. Таким образом, если у нас есть N рабочих, то ускорение будет $1/(0{,}5 + 0{,}5/N)$.

Мы знаем, что $0{,}5/N$ по мере добавления новых рабочих будет быстро приближаться к нулю. Поэтому можно получить верхнюю границу на ускорение, которое мы получим от распараллеливания данной задачи, – $1/(0{,}5 + 0) = 2$. При делении исходного последовательного времени на полученное максимальное ускорение эта задача займет $200/2 = 100$.

Подход, который только что был нами применен, в параллельном программировании называется **законом Амдала**. Он требует лишь хорошего знания части программы, которая может быть распараллелена в нашей исходной последовательной программе, которую мы обозначим через p, и числа доступных ядер N.



Доля времени выполнения, которая не параллелизуется, в этом случае всегда будет равна $1 - p$.

Мы можем вычислить ускорение при помощи закона Амдала следующим образом:

$$Speedup = 1/((1 - p) + p/N).$$

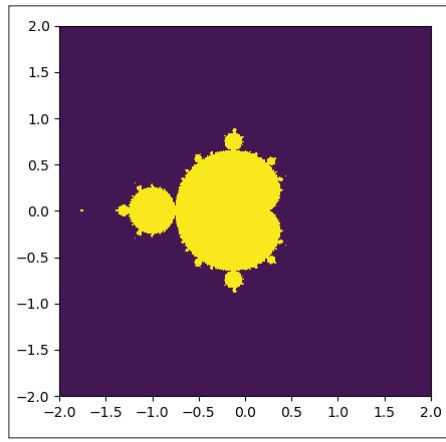
Подводя итоги вышесказанному, хочется упомянуть, что закон Амдала – простая формула, которая позволяет нам примерно (*очень примерно*) оценить потенциальное ускорение от распараллеливания программы. Это может помочь нам определить, стоит ли писать параллельную версию конкретной последовательной программы, при условии что нам известно, какую часть кода можно распараллелить (p) и на каком числе ядер (N) будет выполняться параллельный код.

Множество Мандельброта

Теперь мы готовы рассмотреть простейший пример параллельных вычислений, к которому вернемся в книге немного позднее, – алгоритм для построения изображения *множества Мандельброта*. Для начала давайте определимся, что мы имеем в виду.

Для заданного комплексного числа c мы определим рекурсивную последовательность для $n \leqslant 0$ с $z_0 = 0$ и $z_n = z_{n-1}^2 + c$ для $n > 1$. Если $|z_n|$ остается не больше 2 по мере увеличения n до бесконечности, то в этом случае можно считать, что c принадлежит множеству Мандельброта.

Мы можем представить комплексные числа в виде точек на двухмерной плоскости, при этом x-координата будет соответствовать вещественной части, а y – мнимой. Таким образом, множество Мандельброта выводится при помощи хорошо известной картинки. На ней мы изобразим точки, принадлежащие множеству при помощи более светлого оттенка, а остальные – при помощи более темного следующим образом:



Теперь давайте подумаем над тем, как мы можем построить это множество на Python. Сначала нам нужно рассмотреть следующее. Так как у нас нет возможности проверить каждое комплексное число на принадлежность множеству Мандельброта, то необходимо выбрать определенный диапазон и для него уже определить нужное количество точек (*width*, *height*). Также нам понадобится максимальное число *n*, для которого мы будем проверять $|z_n|$ (`max_iters`). Теперь мы можем подготовиться для написания функции для построения изображения множества Мандельброта – для этого *последовательно* проверим каждую точку.

Мы начнем с импорта NumPy – библиотеки для вычислений, которую будем использовать на протяжении всей книги. Наша реализация – это функция `simple_mandelbrot`. Для начала мы возьмем функцию `linspace` из NumPy для создания решетки точек, которая выступит в качестве дискретной комплексной плоскости (остальной код довольно прямолинеен).

```
import numpy as np

def simple_mandelbrot(width, height, real_low, real_high, imag_low,
                      imag_high, max_iters):
    real_vals = np.linspace(real_low, real_high, width)
    imag_vals = np.linspace(imag_low, imag_high, height)
    # мы будем предствлять члены множества как 1, не члены - как 0.
    mandelbrot_graph = np.ones((height,width), dtype=np.float32)
    for x in range(width):
        for y in range(height):
            c = np.complex64( real_vals[x] + imag_vals[y] * 1j )
            z = np.complex64(0)
            for i in range(max_iters):
                z = z**2 + c
                if(np.abs(z) > 2):
                    mandelbrot_graph[y,x] = 0
                    break
    return mandelbrot_graph
```

Чтобы сохранить множество Мандельброта в файл в формате PNG, необходимо добавить код. Для этого предлагаю вам в начало кода вставить соответствующие заголовки:

```
from time import time
import matplotlib
# Следующий код не позволит «выскочить» картинке.
matplotlib.use('Agg')
from matplotlib import pyplot as plt
```

Теперь добавим код для создания множества Мандельброта и сохранения его в файл и используем функцию `time` для замера затраченного времени:

```
if __name__ == '__main__':
    t1 = time()
    mandel = simple_mandelbrot(512,512,-2,2,-2,2,256, 2)
    t2 = time()
    mandel_time = t2 - t1
    t1 = time()
    fig = plt.figure(1)
    plt.imshow(mandel, extent=(-2, 2, -2, 2))
    plt.savefig('mandelbrot.png', dpi=fig.dpi)
    t2 = time()
    dump_time = t2 - t1
    print 'Потребовалось {} секунд для расчета множества Мандельброта.'.format(mandel_time)
    print 'Потребовалось {} секунд для сохранения графика.'.format(dump_time)
```

После этого запустим полученную программу (она доступна как файл `mandelbrot0.py` в папке `1` в репозитории на GitHub):

```
PS C:\Users\btuom\examples\1> python mandelbrot0.py
It took 14.617000103 seconds to calculate the Mandelbrot graph.
It took 0.110999822617 seconds to dump the image.
```

Потребовалось 14,62 с для расчета множества Мандельброта и 0,11 с для его сохранения на диск. Как вы уже видели, мы рассчитываем множество Мандельброта точка за точкой; нет никаких взаимозависимостей между значениями в различных точках. Таких образом, это легко распараллеливаемая функция. В отличие от нее, код для сохранения изображения не может быть распараллелен.

Теперь проанализируем это в терминах закона Амдала. Какой выигрыш по скорости мы можем получить от распараллеливания данного кода? В сумме оба фрагмента кода заняли на выполнение 14,73 с. Из всего этого можно сделать вывод, что доля кода, который будет распараллелен, равна $p = 14{,}62/14{,}73 = 0{,}99$. Эта программа, распараллеливаемая на 99 %.

Какое ускорение мы можем теоретически получить? Сейчас я работаю на лэптопе с GPU GTX 1050 с 640 ядрами. Поэтому наше $N = 640$.

Ускорение рассчитывается по следующей формуле:

$$Speedup = \frac{1}{0{,}01 + 0{,}99/640} \approx 86{,}6.$$

Все это означает, что наш алгоритм, безусловно, стоит переделать для использования GPU. Имейте в виду, что закон Амдала дает лишь очень приблизительную оценку! При переносе расчетов на GPU будут также учитываться дополнительные моменты, такие как время для CPU на передачу данных на GPU и из них, или тот факт, что алгоритмы, переносимые на GPU, лишь частично распараллеливаемы.

Профилировка вашего кода

Из предыдущего примера мы увидели, что время, затраченное на выполнение различных функций и установку компонентов, можно замерять при помощи функции `time` в Python. Хотя этот подход подошел для нашей небольшой программки, он не всегда будет удобен и возможен для больших программ с многочисленными функциями, некоторые из которых стоит или не стоит распараллеливать или хотя бы просто оптимизировать на CPU. Нашей целью является найти узкие места программы. Даже если мы полны энергии и используем `time` для замера каждой функции, мы легко можем пропустить что-то, или могут быть вызовы системных библиотек, о которых мы даже не подумали, но они здорово замедляют работу программы. Необходимо определить фрагменты кода для переноса на GPU, прежде чем начать думать о переписывании его для GPU. Нам всегда нужно следовать мудрым словам известного американского ученого Дональда Кнута: «Преждевременная оптимизация есть источник всевозможного зла».

Для нахождения узких мест в нашей программе мы будем использовать **профайлер**, который позволит в удобной форме увидеть, где именно программа тратит больше всего времени, и поможет ее оптимизировать в соответствии с полученной информацией.

Использование модуля cProfile

Для проверки нашего кода мы в основном будем использовать модуль *cProfile*. Это стандартная библиотечная функция, которая содержится в каждой установке Python. Его можно вызвать из командной строки при помощи `-m cProfile` и задать способ организации результатов профилировки, которые будут отсортированы в соответствии с затраченным временем, при помощи опции `-s cumtime`. После чего перенаправить вывод в текстовый файл при помощи стандартного оператора `>`.



Это будет работать и под Linux в bash, и под Windows в PowerShell.

Давайте именно сейчас попробуем это сделать.

```
PS C:\Users\btuom\examples\1> python -m cProfile -s cumtime mandelbrot0.py > mandelbrot_profile.txt
PS C:\Users\btuom\examples\1>
```

Теперь можно посмотреть на содержимое текстового файла при помощи своего любимого редактора. Имейте в виду, что вывод программы также будет включен и помещен в начало файла:

```
It took 14.5690000057 seconds to calculate the Mandelbrot graph.
It took 0.136000156403 seconds to dump the image.
         564104 function calls (559254 primitive calls) in 14.965 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.002    0.002   14.966   14.966 mandelbrot0.py:1(<module>)
        1   14.363   14.363   14.572   14.572 mandelbrot0.py:10(simple_mandelbrot)
   263606    0.209    0.000    0.209    0.000 {range}
        1    0.007    0.007    0.134    0.134 __init__.py:101(<module>)
        1    0.003    0.003    0.123    0.123 pyplot.py:17(<module>)
       12    0.017    0.001    0.119    0.010 __init__.py:1(<module>)
        1    0.000    0.000    0.097    0.097 pyplot.py:694(savefig)
        2    0.000    0.000    0.082    0.041 backend_agg.py:418(draw)
    152/2    0.000    0.000    0.081    0.041 artist.py:47(draw_wrapper)
        2    0.000    0.000    0.081    0.041 figure.py:1264(draw)
      4/2    0.000    0.000    0.080    0.040 image.py:120(_draw_list_compositing_images)
```

Итак, поскольку мы не убрали наши вызовы `time`, то сможем прочесть соответствующий вывод в первых двух строчках файла. Далее мы можем увидеть общее число вызовов функций в программе и суммарно потраченное на них время.

Перед нами появится список всех функций, которые ранее вызывались в программе, упорядоченный от потративших наибольшее время к потратившим наименьшее. Первая строка и есть сама программа, а вторая, как и ожидалось, является функцией `simple_mandelbrot`. (Обратите внимание, что указанное для нее время согласуется с результатами замеров при помощи функции `time`.) Далее мы можем увидеть многочисленные библиотечные и системные вызовы, связанные с сохранением множества Мандельброта в файл, которые занимают сравнительно мало времени. Подобный вывод *cProfile* будет использован для определения того, где в нашей программе находятся узкие места.

Резюме

Основным преимуществом использования GPU по сравнению с CPU является увеличенная пропускная способность, которая означает, что мы можем выполнять больше кода параллельно на GPU, чем на CPU. GPU не способен сделать рекурсивные или нераспараллеливаемые алгоритмы быстрее, чем они есть. Мы с вами увидели, что некоторые задачи, например строительство дома, являются лишь частично распараллеливаемыми – отталкиваясь от приведенного примера, нам не удастся ускорить процесс *проектирования* дома (который по своей сути *последователен*), но мы можем ускорить процесс *строительства*,

просто нанимая большее количество рабочих (этот процесс является распараллеливаемым).

Мы использовали эту аналогию для вывода закона Амдала, который является формулой, способной дать грубую оценку возможного ускорения программы, если нам известны доля времени для выполнения той ее части, которая распараллеливаема, и число процессоров, что мы будем использовать для реализации этого кода. Далее мы применили закон Амдала для анализа простой программы, которая вычисляет множество Мандельброта и сохраняет его в файл с изображением, и определили, что она будет хорошим кандидатом для распараллеливания на GPU. Наконец, мы привели краткий обзор профилировки кода при помощи модуля *cProfile*, что позволило нам увидеть узкие места в нашей программе, не прибегая к явным замерам времени выполнения.

Итак, после того как мы сумели разобраться с рядом фундаментальных понятий и получить мотивацию для изучения программирования GPU, следующая глава будет посвящена рассмотрению настройки окружения для программирования GPU на Linux или Windows 10. После этого мы сразу же перейдем к программированию GPU, начав с написания GPU-версии программы для расчета множества Мандельброта, которая была приведена ранее.

Вопросы

1. В примере с расчетом множества Мандельброта из этой главы есть три оператора `for`, но мы можем распараллелить только первые два. Почему мы не можем распараллелить все операторы?
2. Что закон Амдала не учитывает, когда мы переносим последовательный алгоритм на GPU?
3. Допустим, у вас есть исключительный доступ к трем новым сверхсекретным GPU, которые одинаковы, за исключением одного – числа ядер. В первом – 131 072 ядра, во втором – 262 144, и в третьем – 524 288 ядер. Если вы распараллелите и перенесете на эти GPU расчет множества Мандельброта из нашего примера (который строит изображение 512×512), будет ли разница во времени выполнения между первым и вторым GPU? А как насчет второго и третьего?
4. Подумайте о каких-либо задачах или частях кода как распараллеливаемых в связи с законом Амдала.
5. Почему вместо функции `time` лучше всего использовать профилировщик?

Глава 2

Настройка окружения для программирования GPU

Давайте теперь рассмотрим, как настроить подходящее окружение для программирования GPU под Windows и Linux. В обоих случаях необходимо предпринять несколько шагов. Мы их проделаем один за другим, обращая внимание на разницу между Windows и Linux. Можете, конечно, пропустить или проигнорировать любую часть либо комментарии, которые не соответствуют вашему выбору операционной системы.

Советую обратить особое внимание на то, что мы будем рассматривать только две платформы для 64-битовых PC на основе процессоров Intel/AMD – Ubuntu LTS (*long-term support*) и Windows 10. Имейте в виду, что любая Ubuntu LTS система (из таких, как Xubuntu, Kubuntu или Linux Mint) так же хорошо подходит, как и Unity/GNOME Ubuntu.

Предлагаю использовать Python 2.7 вместо Python 3.x. У Python 2.7 имеется стабильная поддержка всех библиотек, которые мы будем использовать в этой книге. Потому каждый пример проверен именно на языке программирования Python 2.7 на Windows и Linux. Пользователи Python 3.x также могут использовать данную книгу, но должны при этом иметь в виду, что между этими двумя языками имеются большие отличия. Некоторые из примеров были проверены на Python 3.7, но требуют стандартных изменений, таких как добавление круглых скобок в вызове функции `print`.



Автор Packt доктор Себастиян Рашка выложил список наиболее важных отличий между Python 2.7 и 3.x по адресу **https://sebastianraschka.com/Articles/2014_python_2_3_key_diff.html**.

Лучше всего для этих целей подойдет дистрибутив Anaconda Python 2.7 и для Windows, и для Linux, поскольку он может легко устанавливаться без администраторских прав, содержит все необходимые модули для анализа и визуализации данных, требуемые в книге, и использует быстрые оптимизированные пакеты NumPy/SciPy, применяющие библиотеку **Math Kernel Library** (MKL) от Intel. (Стандартная установка для Linux `/usr/bin/python` также может быть ис-

пользована для этой книги, но тогда читателю придется вручную устанавливать такие пакеты, как NumPy и Matplotlib.)



Anaconda Python (версий 2.7 и 3.x) может быть скачана для всех платформ по адресу **https://www.anaconda.com/download/**.

Пользователи, работающие с другими поддерживаемыми платформами (например, macOS, Windows 7/8, Windows Server 2016, Red Hat/Fedora, OpenSUSE и CENTOS), должны обратиться к официальной документации по NVIDIA CUDA (**https://docs.nvidia.com/cuda/**) за дополнительными деталями. Кроме того, имеются и иные варианты по выбору необходимой аппаратуры: читателям, заинтересованным во встраиваемых системах, таких как Raspberry Pi, рекомендую начать с использования платы, основанной на ARM-процессорах для разработки NVIDIA Jetson, в то время как тем, кто находит себя в облачных вычислениях или веб-программировании, советую рассмотреть удаленную работу через Azure или AWS. В подобных случаях следует хорошенько изучить официальную документацию для настройки их драйверов, компилятора и CUDA Toolkit, но нужно учитывать, что отдельные шаги из данной главы не всегда могут подойти для всех случаев.

В результате проделанных действий вы:

- сможете убедиться, что имеете необходимое оборудование;
- установите NVDIA GPU драйверы;
- установите соответствующее окружение для программирования на C/C++;
- установите NVIDIA CUDA Toolkit;
- установите окружение для Python для программирования GPU.

Технические требования

Для этой главы предлагается использовать установленную Anaconda Python 2.7: **https://www.anaconda.com/download/**.

Код доступен на GitHub: **https://github.com/PacktPublishing/Hands-On-GPU-Programming-with-Python-and-CUDA**.



За дополнительной информацией о требованиях обратитесь к введению в книгу и файлу README по адресу: **https://github.com/PacktPublishing/Hands-On-GPU-Programming-with-Python-and-CUDA**.

Убедитесь, что у вас есть требуемое оборудование

Мы рекомендуем для практического применения как минимум следующее оборудование:

- 64-битовый компьютер на базе процессора от Intel/AMD;
- 4 Гб памяти (RAM);
- NVIDIA GeForce 1050 GTX (или старше).

Эта конфигурация гарантирует, что вы сможете комфортно изучать программирование GPU, запускать все примеры из книги, а также другие интересные приложения, использующие GPU, такие как TensorFlow от Google (библиотека по машинному обучению) или Vulkan SDK (современная графическая библиотека).



Обратите внимание, что перед началом работы с данной книгой у вас должен быть установлен новый NVIDIA GPU! CUDA Toolkit рассчитан только на NVIDIA GPU, поэтому он не будет взаимодействовать с Intel HD или Radeon GPU.

Как уже отмечалось выше, мы подразумеваем, что вы используете либо Windows 10, либо Ubuntu LTS (*long-term service*).



Версии Ubuntu LTS обычно имеют номера версий вида 14.04, 16.04, 18.04 и т. д.

Ubuntu LTS является одной из наиболее распространенных версий Linux, что обеспечивает наибольшую совместимость программного обеспечения и библиотек. Имейте в виду, что существует множество вариаций Linux, основанных на Ubuntu, таких как Linux Mint, Xubuntu. Они также идеально подходят для осуществления ваших целей. (Я лично обнаружил, что Linux Mint неплохо работает на ноутбуках с установленными GPU.)

Также стоит отметить, что мы считаем, что у вас имеется как минимум GTX 1050 (*Pascal*) GPU или ее эквивалент для любой новой архитектуры. Большинство примеров из книги, скорее всего, будут работать и на старых версиях GPU, но они были проверены автором на GTX 1050 (под Windows) и GTX 1070 (под Linux). Хотя они и не проверялись на старых версиях GPU, таких как Maxwell 2014 г., GTX 750, их должно быть достаточно для достижениях тех целей, что упомянуты в книге.



Если вы располагаете настольным ПК, то перед тем как переходить к дальнейшим действиям, убедитесь, что при установке GPU следовали всем приведенным инструкциям.

Проверка вашего оборудования (Linux)

Сейчас мы произведем несколько базовых проверок в Linux, чтобы полностью убедиться в том, что ваше оборудование идеально походит для осуществления необходимых манипуляций. Откройте терминал и перейдите в командную строку bash – в Ubuntu это можно сделать, просто нажав на сочетание клавиш **Ctrl**+**Alt**+**T**.

Теперь давайте проверим процессор, набрав `lscpu` и нажав клавишу **Enter**. Высветится множество различной информации, однако вам необходимо взглянуть на первую строку, чтобы убедиться в том, что архитектура действительно x86_64: