Содержание

Об авторе

Доктор Бриан Тоуманен работал с CUDA и программированием GPU с 2014 г. Он получил степень бакалавра по специальности «Электроинженерия» в университете Вашингтона в Сиэтле, затем некоторое время являлся разработчиком программного обеспечения, после чего продолжил обучение в области математики. В университете Миссури (Колумбия) Бриан защитил кандидатскую диссертацию по математике, где впервые столкнулся с программированием GPU для решения научных задач. Доктор Тоуманен был приглашен в исследовательскую лабораторию министерства армии США по программированию GPU для выяснения вопросов общего назначения и не так давно руководил интеграцией GPU и разработкой стартапа в Мэриленде. Сейчас он работает в качестве специалиста по машинному обучению (Azure CSI) в компании Microsoft в Сиэтле.

Я бы хотел поблагодарить профессора Мишеллу Беччи из отдела NCSU ECE и ее студента Эндрю Тодда за помощь, оказанную мне как начинающему программисту GPU в 2014 г. Также хочу выразить особую признательность моему редактору в Packt Акшаде Иер за содействие в написании данного труда и, наконец, профессору Андреасу Клекнеру за составление великолепной библиотеки PyCUDA, которую я активно использовал в своей книге.

О рецензенте

Вандане Шах была присуждена степень бакалавра по специальности «Электроэнергетика и электротехника» в 2001 г. Она также приобрела навыки MBA «Управление персоналом» и магистерскую степень по электронике со специализацией в области разработки VLSI. Также ею была представлена работа на получение кандидатской степени по специальности «Электроника», в частности в области обработки изображений и глубокого обучения для диагностики опухоли мозга. Областями интересов Ванданы являются обработка изображений, а также встраиваемые системы. Имеет более 13 лет опыта в исследованиях, а также в обучении и подготовке студентов по дисциплине «Электроника и связь». Ею было опубликовано множество работ в уважающих себя журналах, таких как *IEEE*, *Springer* и *Interscience*. Она также получила государственный грант для проведения исследований в области обработки изображений в MRI. Кроме того что Вандана прекрасно разбирается в вопросах техники, она неплохо владеет искусством индийского танца катхак.

Я благодарю членов моей семьи за их поддержку во всем.

Предисловие

Приветствую вас и поздравляю! Эта книга является вводным курсом по программированию GPU с использованием Python и CUDA. GPU обычно обозначает Graphics Programming Unit (Графический процессор), но здесь пойдет речь не о программировании графики – она является введением в программирование общего вида на GPU или программирование GPGPU (General Purpose GPU). За последнее десятилетие стало очевидным, что GPU хорошо подходят для вычислений не только графических изображений, но параллельно с этим для тех, которые параллельно требуют высокой пропускной способности. Для этого NVIDIA выпустила CUDA Toolkit, благодаря чему область программирования GPGPU стала более доступной практически для любого человека, хоть немного знакомого с некоторыми знаниями языка программирования.

Целью книги «Программирование GPU при помощи Python и CUDA» является огромное желание как можно быстрее ввести читателя в мир технологий GPGPU. Я старался подать материал так, чтобы в каждой главе можно было встретить не только интересные примеры и упражнения, но и прочесть о некоторых веселых случаях. В частности, вы можете набирать приведенные упражнения и выполнять их в вашей любимой среде Python (могут подойти Spyder, Jupiter и PyCharm). Таким образом, вы с легкостью запомните все необходимые функции и команды и одновременно получите некоторый опыт, как при помощи интуиции могут быть написаны программы для GPGPU.

Поначалу параллельное программирование для GPGPU кажется довольно сложным и несколько обескураживающим, особенно если ранее вы соприкасались только с программированием CPU. Вам придется выучить так много новых понятий и соглашений, что иногда будет казаться, словно вы видите это в первый раз. В такие моменты лучше не отчаиваться, а верить, что все приложенные усилия по освоению данной области не напрасны. Обещаю, что при некоторой любознательности и соблюдении дисциплины столь загадочная область по мере того, как вы дойдете до конца книги, станет как бы вашей второй сущностью.

Для кого эта книга

Эта книга предназначена прежде всего для одного особенного человека – меня в 2014 г., когда я пытался написать программу для моделирования при помощи GPU для своей кандидатской диссертации по математике. Мне приходилось часами просиживать над многочисленными книгами и руководствами по программированию GPU, пытаясь отыскать хоть малейший смысл во всем этом;

большинство книг, казалось, представляло собой сплошной парад аппаратных схем и непонятных слов на каждой странице, в то время как само программирование оставалось где-то на заднем плане.

Также моя книга адресована тем, кто на самом деле мечтает по-серьезному заняться программированием GPU, но без влезания в многочисленные технические детали и схемы аппаратуры. Мы будем программировать GPU на C/C++ (CUDA C), но при этом использовать Python при помощи модуля PyCUDA. PyCUDA позволяет писать лишь действительно необходимый низкоуровневый код для GPU, причем данный модуль призван самостоятельно отвечать за всю необходимую компиляцию, линковку и запуск кода на GPU для нас.

О чем рассказывает эта книга?

Глава 1 «*Почему программирование GPU?*» объясняет, для чего нам необходимо разбираться в данных вопросах и как правильно применить закон Амдала для оценки потенциального выигрыша в быстродействии от перевода последовательной программы на GPU.

Глава 2 «*Настройка окружения для программирования GPU*» объясняет, как настроить соответствующее окружение для Python и C++ для использования CUDA под Windows и Linux.

Глава 3 «*Начало работы с CUDA*» описывает технические навыки, которые вам понадобятся для программирования GPU при помощи Python. В частности, мы увидим, как копировать данные в и из GPU при помощи класса gpuarray и компилировать простейшие ядра при помощи функции PyCUDA ElementwiseKernel.

Глава 4 «*Ядра, нити, блоки и сетки*» обучит основам написания эффективных ядер CUDA, которые являются параллельными функциями, выполняемыми на GPU. Мы увидим, как писать выполняемые на GPU функции («последовательные» функции, вызываемые непосредственно ядрами CUDA), и познакомимся со структурой сетка/блок CUDA и ее ролью в запуске ядер.

Глава 5 «*Потоки, события, контексты и одновременность*» покрывает такие понятия, как «потоки CUDA», которые позволяют одновременно запускать и синхронизировать на GPU множество ядер. Мы увидим, как использовать события CUDA для замера времени выполнения ядер и как создавать и использовать контексты CUDA.

Глава 6 «*Отладка и профилирование вашего кода на CUDA*» заполнит некоторые пробелы в области чистого CUDA C программирования и покажет, как использовать NVIDIA NSight IDE для отладки и разработки, а также как использовать средства профилирования от NVIDIA.

Глава 7 «*Использование библиотек CUDA вместе со Scikit-CUDA*» дает краткий обзор некоторых важных библиотек CUDA при помощи модуля Scikit-CUDA, включая cuBLAS, cuFFT и cuSOLVER.

Глава 8 «*Библиотеки функций для GPU CUDA и Thrust*» покажет, как использовать cuRAND и функции математического API CUDA в вашем коде, а также как использовать контейнеры CUDA Thrust в коде на C++.

Глава 9 «*Реализация глубокой нейросети*» служит основой, на которой мы построим с самого начала глубокую нейросеть, применяя многие из идей, которые разбирались в книге.

Глава 10 «*Работа с компилированным кодом для GPU*» покажет, как связывать наш код на Python с заранее откомпилированным кодом для GPU при помощи PyCUDA и Ctypes.

Глава 11 «*Оптимизация быстродействия в CUDA*» научит ряду низкоуровневых приемов для CUDA, таким как перестановка в пределах варпа (*warp shuffling*), векторизованный доступ к памяти, использование PTX и атомарные операции.

Глава 12 «*Куда идти далее?*» является обзором некоторых направлений, которые помогут вам развить ваши навыки программирования GPU.

Как получить максимум от этой книги

Это уже техническая тема. Будут сделаны лишь некоторые предположения относительно уровня программирования читателя. Мы будем считать, что:

- у вас средний уровень программирования на Python;
- вы знакомы со стандартными научными пакетами для Python, такими как NumPy, SciPy и Matplotlib;
- вы обладаете средним уровнем в любом C-подобном языке (C, C++, Java, Rust, Go и т. п.);
- вы понимаете динамическое выделение памяти в C (в частности, как использовать функции `malloc` и `free`).

Программирование GPU применимо в различных научных областях, где чаще всего требуются знания математики, поэтому при приведении множества (если не большинства) примеров будут использоваться ее основы. Я предполагаю, что читатель знаком с программой первого или второго курса высшей математики, включая:

- тригонометрию (такие функции, как sin, cos, tg...);
- вычисления (интегралы, производные, градиенты);
- статистику (равномерное и нормальное распределение);
- линейную алгебру (векторы, матрицы, векторные пространства, размерность).

Не беспокойтесь, если вы не выучили некоторые из этих тем или проходили их очень давно, по ходу книги мы будем рассматривать основные понятия программирования и математические понятия.

Также хочется сделать еще одно предположение. Если вы помните, в начале книги я говорил, что мы будем работать только с CUDA, которая является про-

приетарным языком программирования для оборудования NVIDIA. Так что для начала вам необходимо наличие соответствующего оборудования. Поэтому я считаю, что у читателя есть доступ к:

- 64-битовому компьютеру на базе процессора от Intel/AMD;
- не менее 4 Гб оперативной памяти;
- GPU NVIDIA GTX 1050 или более усовершенствованная.

Читателю также следует знать, что многие прежние версии GPU, скорее всего, будут пригодны для большинства, если не для всех, примеров, приведенных в книге, но тем не менее все они были проверены на GTX 1050 под Windows 10 и GTX 1070 под Linux. Конкретные инструкции по настройке и конфигурации приводятся в *главе 2* «*Настройка окружения для программирования GPU*».

Используемые соглашения

На протяжении всей книги будут использоваться следующие соглашения.

`CodeInText` обозначает текст программы, имена таблиц баз данных, имена каталогов, файлов, расширения файлов, пути, URL, ввод пользователя и т. п. Например: «Теперь мы можем использовать функцию `cublasSaxpy`».

Блок кода выглядит следующим образом:

```
cublas.cublasDestroy(handle)
print 'cuBLAS returned the correct value: %s' % np.allclose(np.dot(A,x),
y_gpu.get())
```

Когда мне хочется привлечь ваше внимание к определенному участку кода, я соответствующие строки выделяю жирным шрифтом:

```
def compute_gflops(precision='S'):

if precision=='S':
    float_type = 'float32'
elif precision=='D':
    float_type = 'float64'
else:
    return -1
```

Любые команды показываются следующим образом:

```
$ run cublas_gemm_flops.py
```

Полужирное выделение обозначает новое понятие, важное слово или слова, которые вы видите на экране. Например, слова в меню или диалоговых окнах показываются таким образом.

Предупреждения или важные замечание показываются таким образом.

Подсказки или приемы показываются таким образом.

Отзывы и пожелания

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

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

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

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте **www.dmkpress.com** на странице с описанием соответствующей книги.

Пакет кода для книги также размещен на GitHub на **https://github.com/PacktPublishing/Generative-Adversarial-Networks-Projects**.

В случае обновления кода он будет обновлен в существующем репозитории GitHub.

У нас есть другие комплекты кода из нашего богатого каталога книг и видео, доступных по адресу: **http://github.com/PacktPublishing/**. Проверьте их!

Скачивание цветных изображений

Вам также предоставляется PDF-файл, который содержит цветные изображения/диаграммы, используемые в книге. Скачать его вы можете на сайте **http://www.packt.com/sites/default/files/downloads/9781788993913_ColorImages.pdf**.

Список опечаток

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

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

Нарушение авторских прав

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

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

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

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 количество 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:

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

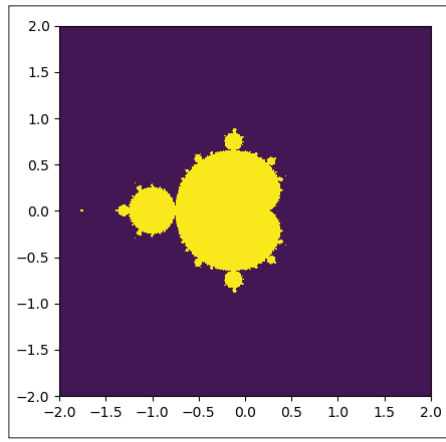
Подводя итоги вышесказанному, хочется упомянуть, что закон Амдала – простая формула, которая позволяет нам примерно (*очень примерно*) оценить потенциальное ускорение от распараллеливания программы. Это может помочь нам определить, стоит ли писать параллельную версию конкретной последовательной программы 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, – алгоритму 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 прописать 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 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 это 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?