Технология CUDA

Выполнили: Саналиев Н., Тунгатаров Б., Кузыров С.
МКМ-15-2

Что такое CUDA?

- CUDA – это архитектура параллельных вычислений от NVIDIA, позволяющая существенно увеличить вычислительную производительность благодаря использованию GPU (графических процессоров).
- На сегодняшний день продажи CUDA процессоров достигли миллионов, а разработчики программного обеспечения, ученые и исследователи широко используют CUDA в различных областях, включая обработку видео и изображений, вычислительную биологию и химию, моделирование динамики жидкостей, восстановление изображений, полученных путем компьютерной томографии, сейсмический анализ, трассировку лучей и многое другое.

Немного о GPU

- Первый вопрос, который должен задать каждый перед применением GPU для решения своих задач — а для каких целей хорош GPU, когда стоит его применять? Для ответа нужно определить 2 понятия:
  Задержка (latency) — время, затрачиваемое на выполнение одной инструкции/операции.
  Пропускная способность — количество инструкций/операций, выполняемых за единицу времени.
  Простой пример: имеем легковой автомобиль со скоростью 90 км/ч и вместимостью 4 человека, и автобус со скоростью 60 км/ч и вместимостью 20 человек. Если за операцию принять перемещение 1 человека на 1 километр, то задержка легкового автомобиля — 3600/90=40с — за столько секунд 1 человек преодолеет расстояние в 1 километр, пропускная способность автомобиля — 4/40=0.1 операций/секунду; задержка автобуса — 3600/60=60с, пропускная способность автобуса — 20/60=0.3(3) операций/секунду.
  Так вот, CPU — это автомобиль, GPU — автобус: он имеет большую задержку но также и большую пропускную способность. Если для вашей задачи задержка каждой конкретной операции не настолько важна как количество этих операций в секунду — стоит рассмотреть применение GPU.

Базовые понятия и термины

- **Устройство (device)** — GPU. Выполняет роль «подчиненного» — делает только то, что ему говорит CPU.
- **Хост (host)** — CPU. Выполняет управляющую роль — запускает задачи на устройстве, выделяет память на устройстве, перемещает память на/с устройства. И да, использование CUDA предполагает, что как устройство так и хост имеют свою отдельную память.
- **Ядро (kernel)** — задача, запускаемая хостом на устройстве.
- При использовании CUDA вы просто пишете код на своем любимом языке программирования (список поддерживаемых языков, не учитывая C и C++), после чего компилятор CUDA сгенерирует код отдельно для хоста и отдельно для устройства. Небольшая оговорка: код для устройства должен быть написан только на языке C с некоторыми 'CUDA-расширениями'.

Базовые понятия и термины

Ядра

- Рассмотрим более детально процесс написания кода для ядер и их запуска. Важный принцип — **ядра пишутся как (практически) обычные последовательные программы** — то-есть вы не увидите создания и запуска потоков в коде самих ядер. Вместо этого, для организации параллельных вычислений **GPU запустит большое количество копий одного и того же ядра в разных потоках** — а точнее, вы сами говорите сколько потоков запустить. И да, возвращаясь к вопросу эффективности использования GPU — чем больше потоков вы запускаете (при условии что все они будут выполнять полезную работу) — тем лучше.
  Код для ядер отличается от обычного последовательного кода в таких моментах:
  Внутри ядер вы имеете возможность узнать «идентификатор» или, проще говоря, позицию потока, который сейчас выполняется — используя эту позицию мы добиваемся того, что одно и то же ядро будет работать с разными данными в зависимости от потока, в котором оно запущено. Кстати, такая организация параллельных вычислений называется SIMD (Single Instruction Multiple Data) — когда несколько процессоров выполняют одновременно одну и ту же операцию но на разных данных.
- В некоторых случаях в коде ядра необходимо использовать различные способы синхронизации.

Базовые понятия и термины

- Каким же образом мы задаем количество потоков, в которых будет запущено ядро? Поскольку GPU это все таки **Graphics** Processing Unit, то это, естественно, повлияло на модель CUDA, а именно на способ задания количества потоков:
  Сначала задаются размеры так называемой сетки (grid), в 3D координатах: *grid_x, grid_y, grid_z*. В результате, сетка будет состоять из *grid_x*grid_y*grid_z* блоков.
- Потом задаются размеры блока в 3D координатах: *block_x, block_y, block_z*. В результате, блок будет состоять из *block_x*block_y*block_z* потоков. Итого, имеем *grid_x*grid_y*grid_z*block_x*block_y*block_z* потоков. Важное замечание — максимальное количество потоков в одном блоке ограничено и зависит от модели GPU — типичны значения 512 (более старые модели) и 1024 (более новые модели).
- Внутри ядра доступны переменные **threadIdx** и **blockIdx** с полями **x, y, z** — они содержат 3D координаты потока в блоке и блока в сетке соответственно. Также доступны переменные **blockDim** и **gridDim** с теми же полями — размеры блока и сетки соответственно.

Основные этапы CUDA-программы

- Хост выделяет нужное количество памяти на устройстве.
- Хост копирует данные из своей памяти в память устройства.
- Хост стартует выполнение определенных ядер на устройстве.
- Устройство выполняет ядра.
- Хост копирует результаты из памяти устройства в свою память.

Аппаратное обеспечение GPU

- CUDA-совместимый GPU состоит из нескольких (обычно десятков) **streaming multiprocessors** (потоковых мультипроцессоров), далее **SM**.
- Каждый **SM**, в свою очередь, состоит из нескольких десятков **simple/streaming processors (SP)** (обычных/потоковых процессоров), или выражаясь более точно, **CUDA cores** (ядер CUDA). Эти ребята уже больше похожи на привычный CPU — имеют свои регистры, кэш и т.д. Каждый **SM** также имеет свою собственную **shared memory** (общую память) — эдакий дополнительный кэш, который доступен всем SP, и может использоваться как в роли кэша для часто используемых данных, так и для «общения» между потоками одного блока CUDA.

Аппаратное обеспечение GPU

- Согласно модели CUDA, программист разбивает задачу на блоки, а блоки на потоки. Каким же образом выполняется сопоставление этих программных сущностей с выше описанными аппаратными блоками GPU?

  Каждый блок будет полностью выполнен на выделенном ему **SM**.
- Распределением блоков по **SM** занимается GPU, не программист.
- Все потоки блока *X* будут разбиты на группы, называемые **warps** (обычно так и говорят — варпы), и выполнены на **SM**. Размер этих групп зависит от модели GPU, например для моделей с микроархитектурой Fermi он равен 32. Все потоки из одного варпа выполняются одновременно, занимая определенную часть ресурсов **SM**. Причем они либо выполняют одну и ту же инструкцию (но на разных данных), либо простаивают.
-

  Исходя из всего этого, CUDA предоставляет следующие гарантии:
  Все потоки в определенном блоке будут выполнены на каком-то одном **SM**.
- Все потоки определенного **ядра** будут выполнены до начала выполнения следующего ядра.
-

  CUDA **не гарантирует** что:
  Какой-то блок *X* будет выполнятся до/после/одновременно с каким-то блоком *Y*.
- Какой-то блок *X* будет выполнен на каком-то конкретном **SM** *Z*.

Синхронизация в CUDA

- **Барьер** — точка в коде ядра, по достижению которой поток может «пройти» дальше, только если все потоки **из его блока** достигли этой точки. Еще раз: барьер позволяет синхронизировать только потоки **одного блока**, а не все потоки в принципе! Ограничение довольно естественное, ведь количество блоков, заданное программистом, может значительно превышать количество доступных **SM**.
- **__threadfence** — не совсем примитив синхронизации: при достижении этой инструкции поток может продолжить выполнение только после того, как все его манипуляции с памятью станут видны другим потокам — по-сути, заставляет поток выполнить flush кэша.

Принципы эффективного использования CUDA

- Принцип увеличение соотношения (время полезной работы) / (время операций с памятью) — обсуждался в предыдущей статье. Значение дроби можно увеличить двумя способами — увеличить числитель, уменьшить знаменатель: то-есть нужно либо делать больше работы, либо тратить меньше времени на операции с памятью. Кроме очевидного решения — по возможности уменьшить количество обращений к памяти, используют следующие принципы эффективной работы с памятью: Перемещение часто используемых данных в более быструю память: локальная память потока > общая память блока >> общая память устройства >> память хоста. Таким образом, если в одном блоке несколько потоков используют одни и те же данные — скорее всего есть смысл переместить их в общую память блока.
- Последовательный доступ к памяти: так как потоки в блоках на самом деле выполняются в группах-варпах, то при условии, что потоки в одном варпе будут работать с данными, расположенными в памяти последовательно, CUDA сможет считать один большой кусок памяти за одну инструкцию. В противном же случае — если потоки в варпе будут обращаться к данным, разбросанным в памяти, количество обращений к памяти возрастает.