**О РАСПАРАЛЛЕЛИВАНИИ АЛГОРИТМОВ ПРИ**

**ПРОГРАММИРОВАНИИ ОБЩЕГО НАЗНАЧЕНИЯ**

**НА ГРАФИЧЕСКИХ ПРОЦЕССОРАХ**

**Основные положения**

Графические процессоры (GPU – Graphics Processing Unit) являются довольно мощными и широко используемыми многоядерными вычислительными устройствами. Например, NVIDIA GeForce GTX 860M (здесь и далее будем, если не оговорено другое, будем иметь в виду GeForce GTX 860M с архитектурой Maxwell), оснащенная 640 ядрами (версия с архитектурой Kepler имеет 1 152 ядер), имеет пиковую производительность 1 300 Гфлопс. Вычислительные блоки GPU называются потоковыми мультипроцессорами (SM – stream multiprocessor); будем считать, что вычислительное ядро и потоковый процессор – это одно и то же.

Универсальные вычисления на GPU компании NVIDIA обеспечивает технология программирования CUDA (Compute Unified Device Architecture); расшифровка аббревиатуры теперь не используется. CUDA предоставляет набор расширений для упрощенных языков C, С++, Fortran, позволяющих выражать параллелизм.

Графический процессор одновременно выполняет множество потоков вычислений. Потоки объединяются в блоки (block), а блоки объединяются в сетку (grid). Множество блоков должно быть разбито на множества, содержащие независимые друг от друга блоки; несколько блоков могут выполняться на мультипроцессоре одновременно. Вычисления потоков всех независимо выполняемых блоков называют ядром.

GeForce GTX 860M имеет 5 мультипроцессоров, каждый состоит из 128 процессоров, Tesla T10 имеет 30 мультипроцессоров, каждый состоит из 8 процессоров, Tesla T20 имеет 16 мультипроцессоров, каждый состоит из 32 процессоров. Заметим, что Tesla T10 имеет архитектуру Tesla, а Tesla T20 – архитектуру Fermi.

Память организована следующим образом (объем памяти приводится для GeForce GTX 860M):

* Регистры (65 536 32-разрядных регистров на каждый мультипроцессор). Делятся поровну между потоками. Если в блоке используется, например, 512 потоков, то на один поток приходится 128 регистров. Максимальное количество регистров, доступных одному потоку, равно 255. Потокам не доступны регистры других потоков, но свои регистры на протяжении выполнения ядра доступны как на чтение, так и на запись.
* Разделяемая память (49 152 байта на каждый мультипроцессор). Организована в виде 16 банков (в Fermi – в виде 32 банков), доступна всем потокам в блоке, обладает высокой скоростью доступа. Часть разделяемой памяти может быть отдана для организации кэша, доступного для всех мультипроцессоров. Если один блок занимает всю разделяемую память, то два блока не могут выполняться на мультипроцессоре одновременно.
* Глобальная память – оперативная память GPU (2 048 MB, 4 295 MB для архитектуры Kepler). Единственный вид памяти, в ячейки которой можно производить запись из любого мультипроцессора. Взаимодействие CPU и GPU происходит с использованием глобальной памяти (память CPU недоступна потокам напрямую). Обладает невысокой скоростью доступа.
* Константная память (64 KB) с мультипроцессорным кэшем (одним на весь мультипроцессор). Из устройства константная память доступна только для чтения (запись возможна до запуска ядра). Константная память очень удобна в использовании: в ней можно размещать данные любого типа и читать их при помощи простого присваивания.
* Текстурная память с мультипроцессорным кэшем. Физически не отделена от глобальной памяти, поэтому под текстурную память можно отвести часть (сотни мегабайт) глобальной памяти. Из устройства текстурная память доступна только для чтения. Для каждого мультипроцессора имеется один, общий для всех задач внутри блока, кэш. Текстурные кэши оптимизированы для двумерных массивов.
* Локальная память. Автоматически используется компилятором, если имеющихся регистров не хватает для размещения локальных (в пределах потока) данных. Физически не отделена от глобальной памяти и такая же медленная как глобальная память.
* С появлением архитектуры Fermi стало возможным увеличить размер памяти мультипроцессора и сделать небольшой L1-кэш, а также разместить на кристалле общий для всех мультипроцессоров L2-кэш размером 768 Кб. (Подробнее – в самом конце файла.)

Приведем еще характеристики графического ускорителя Tesla K80 (открытое решение [https://colab.research.google.com](https://colab.research.google.com/)):

| **Характеристика** | **Kepler GK210** |
| --- | --- |
| **SM** | **15** |
| **Stream processors** | **2496** |
| **Threads per Warp** | **32** |
| **Max Warps per SM** | **64** |
| **Max Threads per SM** | **2048** |
| **Max Thread Blocks per SM** | **16** |
| **32-bit Registers per SM** | **128K** |
| **Max Registers per Thread Block** | **64K** |
| **Max Registers per Thread** | **255** |
| **Max Threads per Thread Block** | **1024** |
| **Shared Memory Configurations** | **128KB** |
| **Max Shared Memory per Thread Block** | **48KB** |

Приведем основные положения для распараллеливания алгоритмов при программировании общего назначения на GPU (GPGPU) под платформу CUDA. Программирование под другие платформы также должно учитывать архитектурные особенности GPU. Поэтому аналогичные положения должны быть и при программировании под другие платформы.

1. *Блоки вычислений.* Операции исходного алгоритма объединяются в потоки (нити) вычислений, потоки объединяются в блоки. Блоки организованы в одномерную, двумерную или трехмерную сетку. Максимальные параметры размера (grid dimension) трехмерной сетки большие: 65 535×65 535×65 535 (GeForce GT 540M) или даже 2 147 483 647× 65 535×65 535 (GeForce GTX 860M). Нумерация блоков вычислений по каждой координате сетки начинается с нуля. Синхронизация между потоками разных блоков не предусмотрена. Каждый блок потоков выполняется атомарно на одном из мультипроцессоров; один мультипроцессор способен одновременно выполнять до восьми (шестнадцати для Tesla ) блоков. Должны быть указаны блоки, которые могут выполняться мультипроцессорами одновременно и независимо друг от друга; очередность выполнения независимых блоков определяет CUDA. Очень желательно, чтобы ячеек для записи при реализации блока вычислений не использовалось много, они должны вмещаться в разделяемую память или регистры.

Небольшой объем внутренней памяти мультипроцессора и большие затраты времени при обращении к внешней памяти не позволяют на одном мультипроцессоре эффективно организовать параллельные вычисления подзадачи большого размера. Этим обусловлена организация вычислений блоками: можно выбрать подходящие размеры блока (здесь еще важно учитывать occupancy – см. ниже).

2. *Потоки вычислений.* Блоки объединяют потоки (нити) вычислений. Потоки одного блока помечаются как узлы одномерной, двумерной или трехмерной сетки. Максимальные параметры размера (block dimension) блока могут быть, например, 1024×1024×64. Нумерация потоков вычислений по каждой координате сетки начинается с нуля. Максимальное количество потоков в одном блоке: 1024. Рекомендации по числу потоков на блок могут быть разные: 64, 196, 512, 1024; для архитектуры Fermi – 128-256 потоков на блок).

Все потоки выполняют одну программу, каждый поток использует свои индексы для вычислений. Имеется два механизма взаимодействия потоков внутри блока: разделяемая память и синхронизация (установка барьеров).

3. *Варпы. Occupancy.* Потоки одного блока выполняются на мультипроцессоре частями-пулами, называемыми варп (дословно: warp – канат). Потоки в блоке разбиваются на варпы автоматически согласно своему номеру: 32 подряд идущих потока образуют варп (два полуварпа по 16 потоков). Внутри пула варп потоки выполняются синхронно в режиме SIMT (Single Instruction Multiple Threads – в один момент времени каждый поток выполняет одну и ту же команду над разными данными). С точки зрения аппаратной части, минимальной единицей исполнения является не поток, а варп. Потоки варпа выполняются одновременно (с логической точки зрения, если процессоров в мультипроцессоре меньше чем 32). Управление пулами варп прозрачно (т.е. осуществляется самим GPU).

В каждом блоке параллельно (логически) выполняются все варпы. В CUDA каждый мультипроцессор содержит аппаратный блок планирования варпов, поэтому смена одного варпa на другой происходит практически без накладных расходов. Заметим, что для большинства CPU-архитектур верно, что для оптимального быстродействия число ядер должно совпадать с количеством запущенных потоков; связано это с долгим процессом переключения ядра контекста процессора с одного потока на другой.

Отношение числа активных, то есть всех реально одновременно выполняемых на мультипроцессоре, варпов к максимально возможному числу варпов для одновременного выполнения мультипроцессором, называется occupancy (дословно – занятость). Значениеoccupancy характеризует то, насколько эффективно используются доступные ресурсы графического процессора при решении той или иной задачи. Максимально возможное число активных варпов рассчитывается по формуле , где  – максимально возможное число потоков (тредов) на один мультипроцессор. Например, для Tesla .

Считается, что значение occupancy примерно 0,65 (65%) уже является вполне достаточным. Другая рекомендация: очень важно, чтобы occupancy было как можно больше, т.е. число активных (всех одновременно выполняемых на мультипроцессоре блоков) варпов было как можно больше. Пусть, например, в блоке 32×32 потоков, . Тогда, так как все 32 варпа могут выполняться параллельно, достаточно двух одновременно выполняемых блока; не надо тратить разделяемую память и регистры для возможности одновременного выполнения восьми или шестнадцати блоков.

Варп образуют 32 подряд идущих потока. Если блоки двумерные (2D) или трехмерные (3D), то рекомендуется число потоков по координате, определяющей подряд идущие потоки, выбирать кратным 32. Пусть, например, варпы формируются из потоков Thr(*i*,*j*) при идущих подряд *i* и фиксированных *j*. Тогда рекомендуется число потоков по первой координате выбирать кратным 32, а по остальным координатам не слишком мало. Если разделяемая память и регистры не позволяют сделать такой выбор, то рекомендуется 32 заменить на 16. Например, для 3D-блоков размер (16,8,8) предпочтительнее размера (32,5,6).

4. *Ядро.* Множество операций потоков всех независимо выполняемых блоков называют ядром. Для запуска ядра на GPU требуется задать размерность и размер сетки блоков, а также размерность и размер блока. Пока ядро выполняется на GPU, CPU может продолжать выполнять свой код, а может просто ждать, пока GPU закончит. Вычислительное ядро не включает в себя вспомогательные операции, такие, например, как операции по подготовке данных (которые исполняются центральным процессором вне GPU).

5. *Обмены между блоками вычислений.* Мультипроцессоры обмениваются информацией посредством оперативной памяти, называемой глобальной памятью. Глобальная память не кэшируется, работает медленно. Время ожидания варпом доступа к памяти может быть использовано для выполнения других варпов. Для оптимального использования глобальной памяти желательно, чтобы количество потоков в блоке было кратным 64 и не менее 192.

6. *Обмены между потоками одного блока вычислений.* Потоковые процессоры мультипроцессора взаимодействуют посредством быстрой разделяемой между этими процессорами памяти. Множественное обращение к банку разделяемой памяти порождает конфликты. Конфликты возникают на уровне полуварпа, задачи в разных полуварпах не конфликтуют по разделяемой памяти. В Fermi разделяемая память организована в виде 32 банков, конфликты возникают на уровне варпа.

**Локальность реализаций на GPU**

Считается, что при реализации алгоритмов на GPU использование локальности (локальность – вычислительное свойство алгоритма, отражающее степень использования памяти с быстрым доступом) еще важнее, чем при реализации на CPU.

При вычислениях на GPU быстрым является процесс обращения к регистрам (самый быстрый вид памяти), разделяемой памяти мультипроцессора и кэшам, но не обращение к глобальной и локальной видам памяти GPU. Особенно эффективно используется регистровая память, если элемент массива приватизирован потоком вычислений. Под приватизацией понимается использование элемента массива в вычислениях только одного потока блока вычислений. Если элемент массива приватизирован, то он может быть размещен в регистрах, выделенных потоку.

Отметим еще некоторые особенности.

* Если несколько потоков из одного полуварпа обращаются к одному и тому же банку разделяемой памяти, то происходит конфликт. Но конфликта не происходит, если все 16 потоков обращаются к одному слову (broadcast).
* Одно чтение из константной памяти может быть передано (broadcast) другим потокам полуварпа: если каждый поток полуварпа запрашивает одно и то же данное из константной памяти, то это данное передается только один раз (broadcast) и трафик сокращается в 16 раз. Но если потоки полуварпа запрашивают одновременно разные данные, то константную память лучше не использовать. Для архитектуры Fermi происходит объединение запросов в память для 32 нитей; кроме того, на Fermi есть возможность кэшировать константные значения для блока из глобальной памяти.
* Текстурная память может быть использована эффективно, если каждый поток полуварпа запрашивает близко расположенные в памяти данные, т.е. если доступ к памяти характеризуется пространственной локальностью.

**Двухуровневый тайлинг для распараллеливания алгоритмов**

**при программировании общего назначения**

**на графических процессорах**

Получать алгоритмы для реализации на графическом процессоре можно применением техники двухуровневого тайлинга. Множество операций алгоритма разбить на блоки вычислений можно путем обычного тайлинга (тайлинг первого уровня). После этого требуется указать блоки, которые можно выполнять независимо друг от друга. Разбить блоки вычислений на потоки вычислений можно путем повторного применения тайлинга (тайлинг второго уровня). Разбиения должны производиться таким образом, чтобы потоки одного блока могли выполняться параллельно (не обязательно независимо, возможно взаимодействие посредством разделяемой памяти и синхронизации).

В лекции «Тайлинг» рассмотрен алгоритм перемножения матриц, применен двухуровневый тайлинг и получен алгоритм, множество операций которого разбито на блоки и потоки вычислений. Алгоритм перемножения матриц обладает естественным параллелизмом, поэтому непосредственно после двухуровневого тайлинга блоки и потоки вычислений могут выполняться независимо. В общем случае для явного задания параллелизма требуется выполнить дополнительные преобразования.

Более сложный пример – алгоритм прямого хода метода Гаусса (Отдельный файл).

В дальнейшем подробно рассмотрим еще один пример (файл «Флойд-Уоршелл»).

Утверждения, предположения, вопросы

Если в одном блоке вычислений 1024 потока, то один мультипроцессор может одновременно обрабатывать (если память позволяет) только 2 блока (так как есть ограничение – 2048 потоков на мультипроцессор).

Время чтения из раделяемой памяти несколько меньше, чем время чтения из кэшей константной или текстурной памяти. Есть мнение: раделяемая память размещена на кристалле GPU и поэтому работает со скоростью регистров процессора.

Время записи массива данных из глобальной памяти в разделяемую память намного меньше, чем в константную или текстурной памяти.

Если потоки вычислений помечены как узлы одномерной сетки (*i*), то варпы формируются из потоков Thr(*i*) при идущих подряд *i*.

Если потоки вычислений помечены как узлы двумерной сетки (*i*,*j*), то варпы формируются из потоков Thr(*i*,*j*) при фиксированном *j*, и идущих подряд *i* (? или из потоков Thr(*i*,*j*) при фиксированном *i*, и идущих подряд *j*).

При новом запуске ядра содержимое глобальной памяти по умолчанию сохраняется. Содержимое константной памяти (запись возможна до запуска ядра) сохраняется? Содержимое текстурной памяти сохраняется?

Для архитектуры Fermi при чтении из глобальной памяти данные помещаются в L1-кэш. При чтении из глобальной (константной, текстурной?) памяти (вопрос: какой объем перемещаемых за одно обращение данных?) данные помещаются в регистры потока? Или нужно как-то указавать куда поместить?

Для оптимального использования глобальной памяти желательно, чтобы количество нитей в блоке было кратным 64 и не менее 192.

Пространственная локальность в потоках происходит в том случае, когда подряд идущие потоки одного полувара или варпа (в зависимости от технологии) запрашивают одновременно подряд идущие (близко расположенные?) в памяти данные.

Непрерывный участок памяти: 64 bytes – каждый поток читает 32-битное слово: int, float, … 128 bytes – каждый поток читает 64-битное слово: int2, float2, double… 256 bytes – каждый поток читает 128-битное слово: int4, float4, … Ограничения: Начальный адрес должен быть выровнен по размеру, *k*-й поток полуварпа должен обращаться к *k*-му элементу.

Тенденция: Выравнивание более не играет такой роли, но сильно хаотичные чтение/запись все еще медленные.

Ветвление. Если происходит ветвление внутри варпа, то разные ветви исполнения сериализуются. Если ветвление происходит между варпами, то штраф минимальный.

**Список использованных источников**

* 1. Боресков А.В., Харламов А.А. Основы работы с технологией CUDA. М.: ДМК Пресс, 2010. 232 с.
  2. Фролов В. Введение в технологию CUDA. // Компьютерная графика и мультимедиа. 2008. выпуск № 6 (1). cgm.computergraphics.ru/issues/issue16/cuda
  3. Baskaran M., Bondhugula U, Krishnamoorthy S., Ramanujam J., Rountev A., Sadayappan, P. Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories // Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. February 2008.
  4. Baskaran M., Ramanujam J., Sadayappan P. Automatic C-to- CUDA code generation for affine programs // Proceedings of the Compiler Construction, 19th International Conference. Part of the Joint European Conferences on Theory and Practice of Software. Paphos, Cyprus, March 20–28, 2010.
  5. Buluc A., Gilberta J.R., Budak C. Solving path problems on the GPU // Parallel Computing. 2010. Vol. 36, № 5-6. P. 241–253.
  6. Rawat P.S., Hong C., Ravishankar M., Grover V., Pouchet L.-N., Sadayappan P. Effective resource management for enhancing performance of 2D and 3D stencils on GPUs // Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit, March 2016.

**L1-кэш**

Он служит, в первую очередь, для буферизации. Даже если нить запрашивает из глобальной памяти 4 байта, транзакция все равно осуществляется минимум с 64 байтами. Загруженные данные могут понадобиться вскоре этой или другим нитям.

Теперь размер разделяемой памяти мультипроцессора составляет 64 Кб и её можно сконфигурировать программно, оставить 16 Кб разделяемой и 48 Кб нового L1-кэша. Это полезно для старых CUDA-программ, которые не знают о том, что объем разделяемой памяти, доступной блоку нитей, увеличился. Зато они могут выиграть от большего размера кэша. Или сконфигурировать память, в пропорции 48 Кб разделяемой, 16 Кб кэша. То есть, дать возможность программе самой решить, что загружать в память мультипроцессора. Но это тоже будет способствовать росту производительности старых CUDA-программ, так как большее количество блоков и нитей сможет одновременно выполняться на мультипроцессоре. Допустим, если раньше один блок из 256 нитей требовал 16 Кб разделяемой памяти и на мультипроцессоре мог выполняться только один блок, то сейчас – 3, так как им всем хватит памяти.

Также, L1-кэш очень поможет для быстрого свопа, то есть, временного хранения регистров. Сейчас, при программировании CUDA-приложения, желательно использовать как можно меньше регистров мультипроцессора, чтобы на одном мультипроцессоре можно было запустить больше нитей. И, крайне нежелательно, временно сбрасывать содержимое регистров в медленную глобальную память. На это иногда жалуются CUDA-программисты, что им не хватает регистров. А если хватает, то запускается меньше нитей, которые меньше скрывают различные задержки.

**L2-кэш**

Первая функция общего кэша второго уровня – это радикально ускорить и упростить синхронизацию всех исполняющихся нитей. Нити, выполняющейся на одном мультипроцессоре, не могут видеть содержимое L1-кэшей других мультипроцессоров, но когда данные сбрасываются в L2-кэш, например, принудительными синхронизирующими инструкциями, они сразу становятся видимыми для всех. Раньше для синхронизации между блоками использовалась сверхмедленная глобальная память.

И, конечно, L2 выступает неким общим пулом загружаемых из глобальной памяти данных. GT200 имеет 256 Кб кэша второго уровня для текстурной памяти на все мультипроцессоры, сейчас размер кэша утроился до 768 Кб. И он стал полноценным, а не только для чтения данных.