С. А. Полетаев

ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ НА ГРАФИЧЕСКИХ ПРОЦЕССОРАХ

ВВЕДЕНИЕ

Видеочипы в параллельных математических расчётах пытались использовать довольно давно. Самые первые попытки такого применения были крайне примитивными и ограничивались использованием некоторых аппаратных функций, таких, как растеризация и Z-буферизация. Но в нынешнем веке, с появлением шейдеров, появилась необходимость ускорять вычисления матриц. В 2003 году на SIGGRAPH отдельная секция была выделена под вычисления на GPU, и она получила название GPGPU (General-Purpose Computation on GPU) – универсальные вычисления на GPU).

Наиболее известен BrookGPU – компилятор потокового языка программирования Brook, созданный для выполнения неграфических вычислений на GPU. До его появления разработчики, использующие возможности видеочипов для вычислений, выбирали один из двух распространённых API: Direct3D или OpenGL. Это серьёзно ограничивало применение GPU, ведь в 3D графике используются шейдеры и текстуры, о которых специалисты по параллельному программированию знать не обязаны, они используют потоки и ядра. Brook: смог помочь в облегчении их задачи. Эти потоковые расширения к языку C, разработанные в Стэндфордском университете, скрывали от программистов трёхмерный API, и представляли видеочип в виде параллельного сопроцессора. Компилятор обрабатывал файл .br с кодом C++ и расширениями, производя код, привязанный к библиотеке с поддержкой DirectX, OpenGL или x86.

Естественно, у Brook было множество недостатков, о которых мы подробнее поговорим далее. Но даже просто его появление вызвало значительный прилив внимания тех же NVIDIA и ATI к инициативе вычислений на GPU, так как развитие этих возможностей серьёзно изменило рынок в дальнейшем, открыв целый новый его сектор – параллельные вычислители на основе видеочипов.

В дальнейшем некоторые исследователи из проекта Brook влились в команду разработчиков NVIDIA, чтобы представить программно-аппаратную стратегию параллельных вычислений, открыв новую долю

рынка. И главным преимуществом этой инициативы NVIDIA стало то, что разработчики детально знают все возможности своих GPU, и в использовании графического API нет необходимости, а работать с аппаратным обеспечением можно напрямую при помощи драйвера. Результатом усилий этой команды стала NVIDIA CUDA (Compute Unified Device Architecture) – новая программно-аппаратная архитектура для параллельных вычислений на NVIDIA GPU, которой посвящена эта статья.

1. РАЗНИЦА МЕЖДУ CPU И GPU В ПАРАЛЛЕЛЬНЫХ РАСЧЁТАХ

Рост частот универсальных процессоров упёрся в физические ограничения и высокое энергопотребление, и увеличение их производительности всё чаще происходит за счёт размещения нескольких ядер в одном чипе. Продаваемые сейчас процессоры содержат лишь до четырёх ядер (дальнейший рост не будет быстрым) и они предназначены для обычных приложений, используют MIMD – множественный поток команд и данных. Каждое ядро работает отдельно от остальных, исполняя разные инструкции для разных процессов.

Специализированные векторные возможности (SSE2 и SSE3) для четырехкомпонентных (одинарная точность вычислений с плавающей точкой) и двухкомпонентных (двойная точность) векторов появились в универсальных процессорах, из-за возросших требований графических приложений, в первую очередь. Именно поэтому для определённых задач применение GPU выгоднее, ведь они изначально сделаны для них.

Например, в видеочипах NVIDIA основной блок – это мультипроцессор с восемью-десятью ядрами и сотнями ALU в целом, несколькими тысячами регистров и небольшим количеством разделяемой общей памяти. Кроме того, видеокарта содержит быструю глобальную память с доступом к ней всех мультипроцессоров, локальную память в каждом мультипроцессоре, а также специальную память для констант.

Самое главное – эти несколько ядер мультипроцессора в GPU являются SIMD (одиночный поток команд, множество потоков данных) ядрами. И эти ядра исполняют одни и те же инструкции одновременно. Такой стиль программирования является обычным для графических алгоритмов и многих научных задач, но требует специфического программирования. Зато такой подход позволяет увеличить количество исполнительных блоков за счёт их упрощения.

Итак, перечислим основные различия между архитектурами CPU и GPU. Ядра CPU созданы для исполнения одного потока последовательных инструкций с максимальной производительностью, а GPU проектируются для быстрого исполнения большого числа параллельно выполняемых потоков инструкций. Универсальные процессоры оптимизированы для достижения высокой производительности единственного потока команд, обрабатывающего и целые числа и числа с плавающей точкой. При этом доступ к памяти случайный.

Разработчики CPU стараются добиться выполнения как можно большего числа инструкций параллельно, для увеличения производительности. Для этого, начиная с процессоров Intel Pentium, появилось суперскалярное выполнение, обеспечивающее выполнение двух инструкций за такт, а Pentium Pro отличился внеочередным выполнением инструкций. Но у параллельного выполнения последовательного потока инструкций есть определённые базовые ограничения, и увеличением количества исполнительных блоков кратного увеличения скорости не добиться.

У видеочипов работа простая и распараллеленная изначально. Видеочип принимает на входе группу полигонов, проводит все необходимые операции, и на выходе выдаёт пиксели. Обработка полигонов и пикселей независима, их можно обрабатывать параллельно, отдельно друг от друга. Поэтому из-за изначально параллельной организации работы в GPU используется большое количество исполнительных блоков, которые легко загрузить, в отличие от последовательного потока инструкций для CPU. Кроме того, современные GPU также могут исполнять больше одной инструкции за такт (dual issue). Так, архитектура Tesla в некоторых условиях запускает на исполнение операции MAD+MUL или MAD+SFU одновременно.

GPU отличается от CPU и по принципам доступа к памяти. В GPU он связанный и легко предсказуемый – если из памяти читается тексель текстуры, то через некоторое время придёт время и для соседних текселей. При записи происходит то же самое – пиксель записывается во фреймбуфер, и через несколько тактов будет записываться пиксель расположенный рядом с ним. Поэтому организация памяти отличается от той, что используется в CPU. И видеочипу, в отличие от универсальных процессоров, просто не нужна кэш-память большого размера, а для текстур требуются лишь несколько (до 128–256 в нынешних GPU) килобайт.

Да и сама по себе работа с памятью у GPU и CPU несколько отличается. Так, не все центральные процессоры имеют встроенные контроллеры памяти, а у всех GPU обычно есть по несколько контроллеров, вплоть до восьми 64-битных каналов в чипе NVIDIA GT200. Кроме того, на видеокартах при-

меняется более быстрая память, и в результате видеочипам доступна в разы большая пропускная способность памяти, что также весьма важно для параллельных расчётов, оперирующих с огромными потоками данных.

В универсальных процессорах большие количества транзисторов и площадь чипа идут на буферы команд, аппаратное предсказание ветвления и огромные объёмы начиповой кэш-памяти. Все эти аппаратные блоки нужны для ускорения исполнения немногочисленных потоков команд. Видеочипы тратят транзисторы на массивы исполнительных блоков, управляющие потоками блоки, разделяемую память небольшого объёма и контроллеры памяти на несколько каналов. Вышеперечисленное не ускоряет выполнение отдельных потоков, но позволяет чипу обрабатывать несколько тысяч потоков, одновременно исполняющихся чипом и требующих высокой пропускной способности памяти.

Рассмотрим отличия в кэшировании. Универсальные центральные процессоры используют кэш-память для увеличения производительности за счёт снижения задержек доступа к памяти, а GPU используют кэш или общую память для увеличения полосы пропускания. CPU снижают задержки доступа к памяти при помощи кэш-памяти большого размера, а также предсказания ветвлений кода. Эти аппаратные части занимают большую часть площади чипа и потребляют много энергии. Видеочипы обходят проблему задержек доступа к памяти при помощи одновременного исполнения тысяч потоков – в то время, когда один из потоков ожидает данных из памяти, видеочип может выполнять вычисления другого потока без ожидания и задержек.

Есть множество различий и в поддержке многопоточности. CPU исполняет 1-2 потока вычислений на одно процессорное ядро, а видеочипы могут поддерживать до 1024 потоков на каждый мультипроцессор, которых в чипе несколько штук. И если переключение с одного потока на другой для CPU стоит сотни тактов, то GPU переключает несколько потоков за один такт.

Кроме того, центральные процессоры используют SIMD (одна инструкция выполняется над многочисленными данными) блоки для векторных вычислений, а видеочипы применяют SIMT (одна инструкция и несколько потоков) для скалярной обработки потоков. SIMT не требует, чтобы разработчик преобразовывал данные в векторы, и допускает произвольные ветвления в потоках.

Вкратце можно сказать, что в отличие от современных универсальных CPU, видеочипы предназначены для параллельных вычислений с большим количеством арифметических операций. И значительно большее число

транзисторов GPU (Рис. 1) работает по прямому назначению – обработке массивов данных, а не управляет исполнением (flow control) немногочисленных последовательных вычислительных потоков. Это схема того, сколько места в CPU и GPU занимает разнообразная логика:

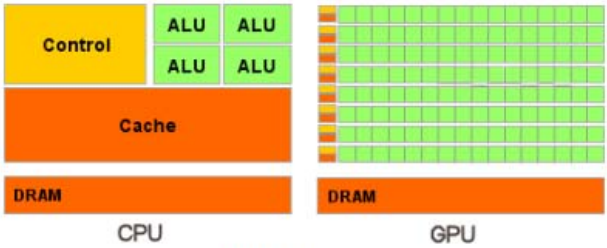

Рисунок 1

В итоге основой для эффективного использования мощи GPU в научных и иных неграфических расчётах является распараллеливание алгоритмов на сотни исполнительных блоков, имеющихся в видеочипах. Например – множество приложений по молекулярному моделированию отлично приспособлено для расчётов на видеочипах, расчеты требуют больших вычислительных мощностей и поэтому удобны для параллельных вычислений. Использование нескольких GPU даёт ещё больше вычислительных мощностей для решения подобных задач.

Выполнение расчётов на GPU показывает отличные результаты в алгоритмах, использующих параллельную обработку данных. То есть, когда одну и ту же последовательность математических операций применяют к большому объёму данных. При этом лучшие результаты достигаются, если отношение числа арифметических инструкций к числу обращений к памяти достаточно велико. Это предъявляет меньшие требования к управлению исполнением (flow control), а высокая плотность математики и большой объём данных отменяет необходимость в больших кэшах, как на CPU.

В результате всех описанных выше отличий, теоретическая производительность видеочипов значительно превосходит производительность CPU. На рис. 2 показан опубликованный компанией NVIDIA приводит такой график роста производительности CPU и GPU за последние несколько лет.

Естественно, эти данные не без доли лукавства. Ведь на CPU гораздо проще на практике достичь теоретических цифр, да и цифры приведены для одинарной точности в случае GPU, и для двойной – в случае CPU. В любом случае, для части параллельных задач одинарной точности хватает, а раз-

ница в скорости между универсальными и графическими процессорами весьма велика.

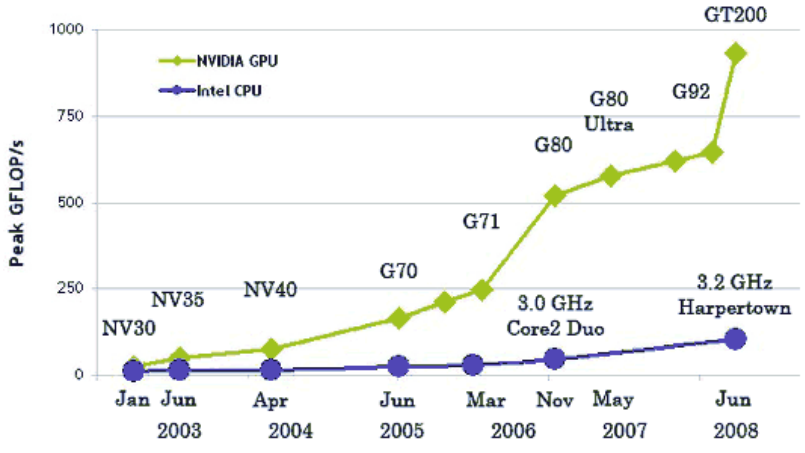

Рисунок 2

2. API CUDA

Успеха Brook оказалось достаточно, чтобы привлечь внимание ATI и NVIDIA, у них зародился интерес к подобной инициативе, поскольку она могла бы расширить рынок, открыв для компаний новый немаловажный сектор.

Исследователи, изначально вовлечённые в проект Brook, быстро присоединились к командам разработчиков в Санта-Кларе, чтобы представить глобальную стратегию для развития нового рынка. Идея заключалась в создании комбинации аппаратного и программного обеспечения, подходящего для задач GPGPU. Поскольку разработчики NVIDIA знают все секреты своих GPU, то на графическое API можно было и не опираться, а связываться с графическим процессором через драйвер. Хотя, конечно, при этом возникают свои проблемы. Итак, команда разработчиков CUDA (Compute Unified Device Architecture) создала набор программных уровней для работы с GPU(рис. 3).

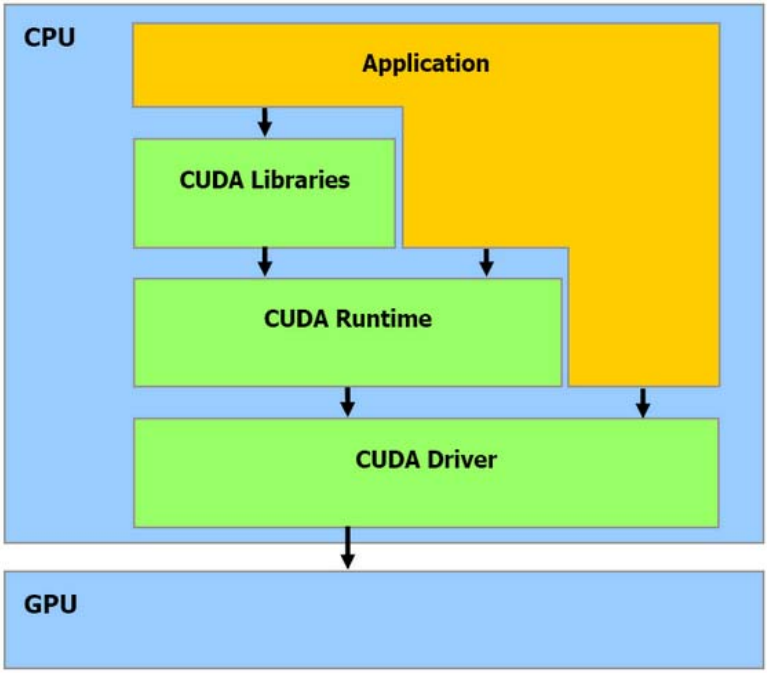

Рисунок 3

Как можно видеть на диаграмме, CUDA обеспечивает два API:

- высокоуровневый API: CUDA Runtime API;
- низкоуровневый API: CUDA Driver API.

Поскольку высокоуровневый API реализован над низкоуровневым, каждый вызов функции уровня Runtime разбивается на более простые инструкции, которые обрабатывает Driver API. Обратите внимание, что два API взаимно исключают друг друга: программист может использовать один или другой API, но смешивать вызовы функций двух API не получится. Вообще, термин «высокоуровневый API» относителен. Даже Runtime API таков, что многие сочтут его низкоуровневым; впрочем, он всё же предоставляет функции, весьма удобные для инициализации или управления контекстом. Но не ожидайте особо высокого уровня абстракции вам всё равно нужно

обладать хорошим набором знаний о NVIDIA GPU и о том, как они работают.

С Driver API работать ещё сложнее; для запуска обработки на GPU вам потребуется больше усилий. С другой стороны, низкоуровневый API более гибок, при необходимости предоставляя программисту дополнительный контроль. Два API способны работать с ресурсами OpenGL или Direct3D (только девятая версия). Польза от такой возможности очевидна CUDA может использоваться для создания ресурсов (геометрия, процедурные текстуры и т.д.), которые можно передать на графическое API или, наоборот, можно сделать так, что 3D API будет отсылать результаты рендеринга программе CUDA, которая, в свою очередь, будет выполнять пост-обработку. Есть много примеров таких взаимодействий, и преимущество заключается в том, что ресурсы продолжают храниться в памяти GPU, их не требуется передавать через шину PCI Express, которая по-прежнему остаётся «узким местом».

Впрочем, следует отметить, что совместное использование ресурсов в видеопамяти не всегда проходит идеально и может привести к некоторым «головным болям». Например, при смене разрешения или глубины цвета, графические данные приоритетны. Поэтому, если требуется увеличить ресурсы в кадровом буфере, то драйвер без проблем сделает это за счёт ресурсов приложений CUDA, которые попросту «вылетят» с ошибкой. Конечно, не очень элегантно, но такая ситуация не должна случаться очень уж часто. Если вы хотите использовать несколько GPU для приложений CUDA, то вам нужно сначала отключить режим SLI, иначе приложения CUDA смогут «видеть» только один GPU.

Наконец, третий программный уровень отдан библиотекам.

- **CUBLAS** – CUDA вариант BLAS (Basic Linear Algebra Subprograms), предназначенный для вычислений задач линейной алгебры и использующий прямой доступ к ресурсам GPU;
- **CUFFT** – CUDA вариант библиотеки Fast Fourier Transform для расчёта быстрого преобразования Фурье, широко используемого при обработке сигналов. Поддерживаются следующие типы преобразований: complex-complex (C2C), real-complex (R2C) и complex-real (C2R).

CUBLAS – это переведённые на язык CUDA стандартные алгоритмы линейной алгебры; на данный момент поддерживается только определённый набор основных функций CUBLAS. Библиотеку очень легко использовать: нужно создать матрицу и векторные объекты в памяти видеокарты, заполнить их данными, вызвать требуемые функции CUBLAS и загрузить

результаты из видеопамяти обратно в системную. CUBLAS содержит специальные функции для создания и уничтожения объектов в памяти GPU, а также для чтения и записи данных в эту память. Поддерживаемые функции BLAS: уровни 1, 2 и 3 для действительных чисел, уровень 1 CGEMM для комплексных. Уровень 1 – это векторно-векторные операции, уровень 2 – векторно-матричные операции, уровень 3 – матрично-матричные операции.

CUFFT – CUDA вариант функции быстрого преобразования Фурье – широко используемой и очень важной при анализе сигналов, фильтрации и т.п. CUFFT предоставляет простой интерфейс для эффективного вычисления FFT на видеочипах производства NVIDIA без необходимости в разработке собственного варианта FFT для GPU. CUDA вариант FFT поддерживает 1D, 2D, и 3D преобразования комплексных и действительных данных, пакетное исполнение для нескольких 1D трансформаций в параллели, размеры 2D и 3D трансформаций могут быть в пределах [2, 16384], для 1D поддерживается размер до 8 миллионов элементов.

3. ФУНДАМЕНТАЛЬНЫЕ ПОНЯТИЯ

Перед тем, как мы погрузимся в CUDA, позвольте определить ряд терминов, разбросанных по документации NVIDIA. Компания выбрала весьма специфическую терминологию, к которой трудно привыкнуть. Прежде всего, отметим, что **поток (thread)** в CUDA имеет далеко не такое же значение, как поток CPU. Потоком GPU в данном случае является базовый набор данных, которые требуется обработать. В отличие от потоков CPU, потоки CUDA очень «лёгкие», то есть переключение контекста между двумя потоками – отнюдь не ресурсоёмкая операция.

Второй термин, часто встречающийся в документации CUDA **варп (warp)**. Термин взят из текстильной промышленности, где через основную пряжу (warp yarn), которая растянута на станке, протягивается уточная пряжа (weft yarn) (рис. 4).

Варп в CUDA представляет собой группу из 32 потоков и является минимальным объёмом данных, обрабатываемых SIMD-способом в мультипроцессорах CUDA.

Рисунок 4

Но подобная «зернистость» не всегда удобна для программиста. Поэтому в CUDA вместо работы с варпами напрямую можно работать с **блоками (block)**, содержащими от 64 до 512 потоков (Рис. 5).

Наконец, эти блоки собираются вместе в **сетки (grid)**. Преимущество подобной группировки заключается в том, что число блоков, одновременно обрабатываемых GPU, тесно связано с аппаратными ресурсами, как мы увидим ниже. Группировка блоков в сетки позволяет полностью абстрагироваться от этого ограничения и применить ядро (kernel) к большему числу потоков за один вызов, не думая о фиксированных ресурсах. За всё это отвечают библиотеки CUDA. Кроме того, подобная модель хорошо масштабируется. Если GPU имеет мало ресурсов, то он будет выполнять блоки последовательно. Если число вычислительных процессоров велико, то блоки могут выполняться параллельно. То есть, один и тот же код может работать на GPU как начального уровня, так и на топовых и даже будущих моделях. Есть ещё пара терминов в CUDA API, которые обозначают CPU (**хост/host**) и GPU (**устройство/device**).

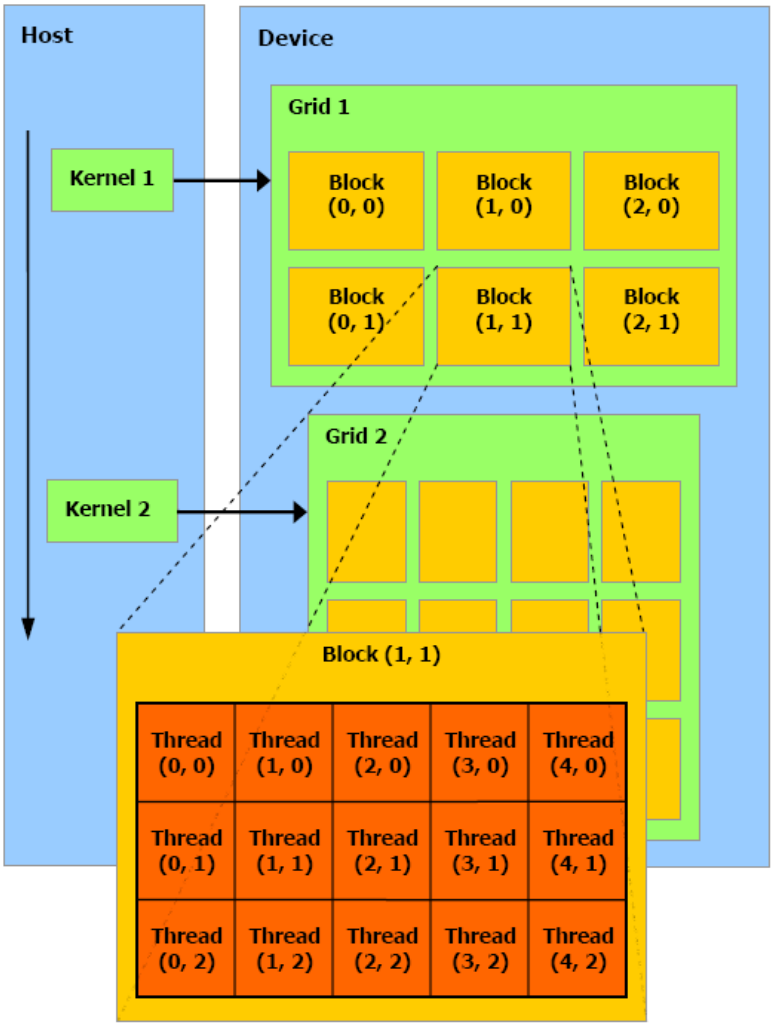
Host
Device
Grid 1
Kernel 1
Block (0, 0)
Block (1, 0)
Block (2, 0)
Block (0, 1)
Block (1, 1)
Block (2, 1)
Grid 2
Kernel 2
Block (1, 1)
Thread (0, 0)
Thread (1, 0)
Thread (2, 0)
Thread (3, 0)
Thread (4, 0)
Thread (0, 1)
Thread (1, 1)
Thread (2, 1)
Thread (3, 1)
Thread (4, 1)
Thread (0, 2)
Thread (1, 2)
Thread (2, 2)
Thread (3, 2)
Thread (4, 2)

Рисунок 5

4. CUDA С АППАРАТНОЙ ТОЧКИ ЗРЕНИЯ

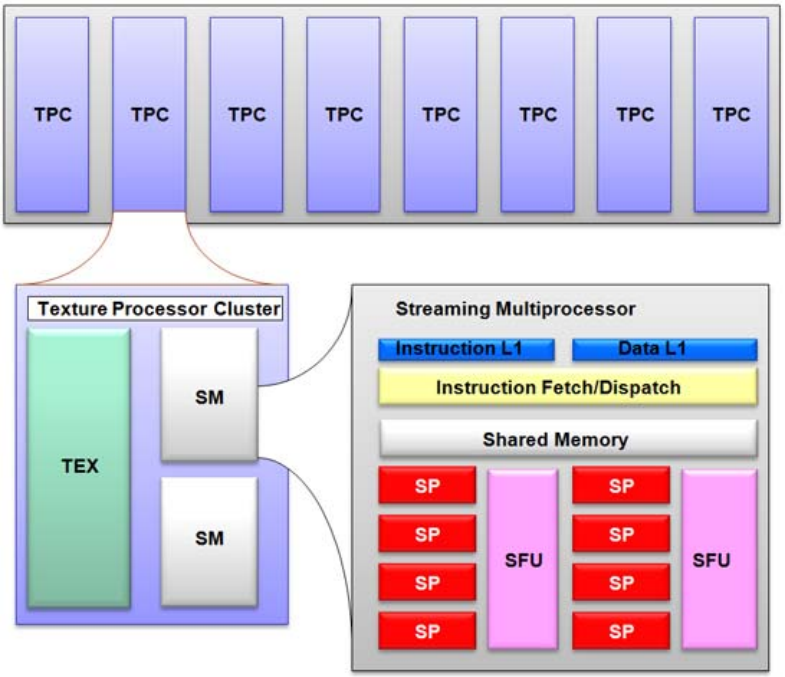

Рисунок 6

Как можно понять по иллюстрации выше на рис. 6, ядро шейдеров NVIDIA состоит из нескольких кластеров текстурных процессоров **(Texture Processor Cluster, TPC)**. Видеокарта 8800 GTX, например, использовала восемь кластеров, 8800 GTS шесть и т.д. Каждый кластер, по сути, состоит из текстурного блока и двух **потоковых мультипроцессоров (streaming multiprocessor)**. Последние включают начало конвейера (front end), выполняющее чтение и декодирование инструкций, а также отсылку их на выполнение, и конец конвейера (back end), состоящий из восьми вычислительных устройств и двух суперфункциональных устройств **SFU (Super Function Unit)**, где инструкции выполняются по принципу SIMD, то есть одна инструкция применяется ко всем потокам в варпе. NVIDIA называет такой способ выполнения **SIMT** (single instruction multiple threads, одна инструкция, много потоков). Важно отметить, что конец конвейера работает на частоте в два раза превосходящей его начало. На практике это означа-

ет, что данная часть выглядит в два раза «шире», чем она есть на самом деле (то есть как 16-канальный блок SIMD вместо восьмиканального). Потоковые мультипроцессоры работают следующим образом: каждый такт начало конвейера выбирает варп, готовый к выполнению, и запускает выполнение инструкции. Чтобы инструкция применилась ко всем 32 потокам в варпе, концу конвейера потребуется четыре такта, но поскольку он работает на удвоенной частоте по сравнению с началом, потребуется только два такта (с точки зрения начала конвейера). Поэтому, чтобы начало конвейера не простаивало такт, а аппаратное обеспечение было максимально загружено, в идеальном случае можно чередовать инструкции каждый такт, классическая инструкция в один такт и инструкция для SFU в другой.

Каждый мультипроцессор обладает определённым набором ресурсов, в которых стоит разобраться. Есть небольшая область памяти под названием **«Общая память/Shared Memory»**, по 16 кбайт на мультипроцессор (рис. 7). Это отнюдь не кэш-память: программист может использовать её по своему усмотрению. То есть перед нами что-то близкое к Local Store у SPU на процессорах Cell. Данная деталь весьма любопытная поскольку она подчёркивает, что CUDA это комбинация программных и аппаратных технологий. Данная область памяти не используется для пиксельных шейдеров.

Данная область памяти открывает возможность обмена информацией между потоками *в одном блоке*. Важно подчеркнуть это ограничение: все потоки в блоке гарантированно выполняются одним мультипроцессором. Напротив, привязка блоков к разным мультипроцессорам вообще не оговаривается, и два потока из разных блоков не могут обмениваться информацией между собой во время выполнения. То есть пользоваться общей памятью не так и просто. Впрочем, общая память всё же оправданна за исключением случаев, когда несколько потоков попытаются обратиться к одному банку памяти, вызывая конфликт. В остальных ситуациях доступ к общей памяти такой же быстрый, как и к регистрам.

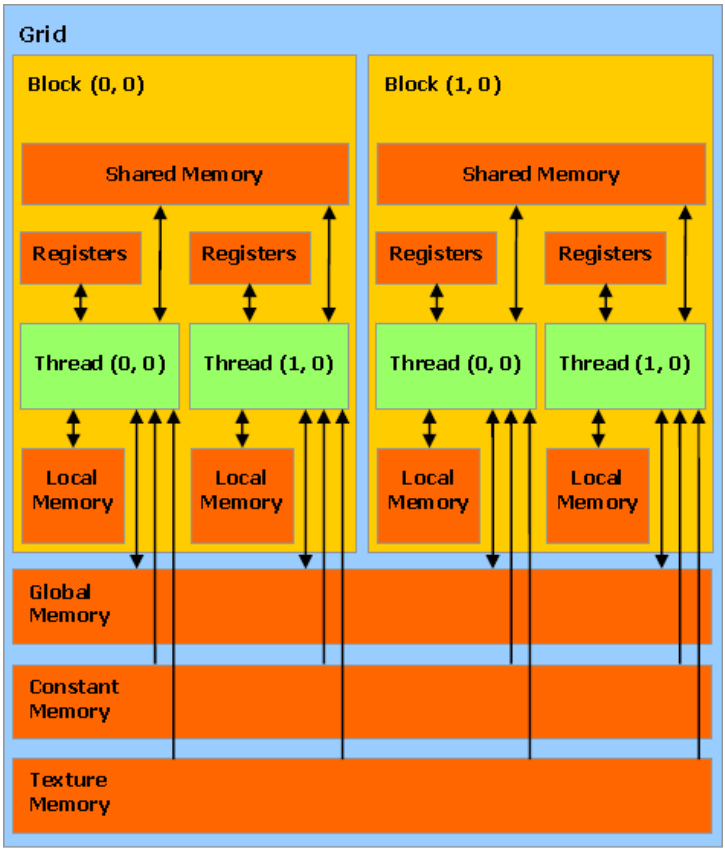
Grid
Block (0, 0)
Block (1, 0)
Shared Memory
Shared Memory
Registers
Registers
Registers
Registers
Thread (0, 0)
Thread (1, 0)
Thread (0, 0)
Thread (1, 0)
Local Memory
Local Memory
Local Memory
Local Memory
Global Memory
Constant Memory
Texture Memory

Рисунок 7

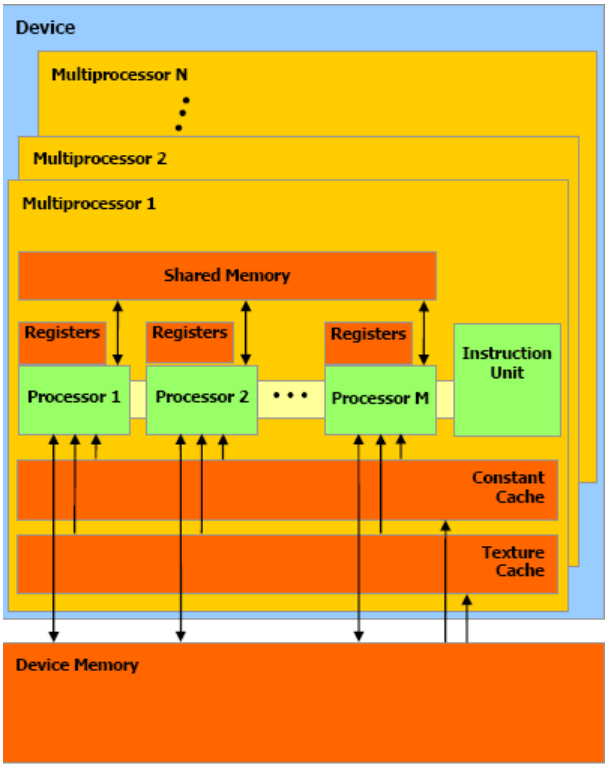

Рисунок 8

Общая память не единственная, к которой могут обращаться мультипроцессоры. Они могут использовать видеопамять, но с меньшей пропускной способностью и большими задержками. Поэтому, чтобы снизить частоту обращения к этой памяти, NVIDIA оснастила мультипроцессоры кэшем (примерно 8 кбайт на мультипроцессор), хранящим константы и текстуры.

Мультипроцессор имеет 8192 регистра, которые общие для всех потоков всех блоков, активных на мультипроцессоре. Число активных блоков на мультипроцессор не может превышать восьми, а число активных варпов ограничено 24 (768 потоков). Поэтому 8800 GTX может обрабатывать до 12 288 потоков в один момент времени. Все эти ограничения стоило упомянуть, поскольку они позволяют оптимизировать алгоритм в зависимости от доступных ресурсов (Рис. 8).

Оптимизация программы CUDA, таким образом, состоит в получении оптимального баланса между количеством блоков и их размером. Больше потоков на блок будут полезны для снижения задержек работы с памятью, но и число регистров, доступных на поток, уменьшается. Более того, блок из 512 потоков будет неэффективен, поскольку на мультипроцессоре может быть активным только один блок, что приведёт к потере 256 потоков. Поэтому NVIDIA рекомендует использовать блоки по 128 или 256 потоков, что даёт оптимальный компромисс между снижением задержек и числом регистров для большинства ядер/kernel.

5. CUDA С ПРОГРАММНОЙ ТОЧКИ ЗРЕНИЯ

С программной точки зрения CUDA состоит из набора расширений к языку C, что напоминает BrookGPU, а также нескольких специфических вызовов API. Среди расширений присутствуют спецификаторы типа, относящиеся к функциям и переменным. Важно запомнить ключевое слово **__global__**, которое, будучи приведённым перед функцией, показывает, что последняя относится к ядру/kernel эту функцию будет вызывать CPU, а выполняться она будет на GPU. Префикс **__device__** указывает, что функция будет выполняться на GPU (который CUDA и называет "устройство/device") но она может быть вызвана только с GPU (иными словами, с другой функции __device__ или с функции __global__). Наконец, префикс **__host__** обозначает функцию, которая вызывается CPU и выполняется CPU - другими словами, обычную функцию.

Есть ряд ограничений, связанных с функциями __device__ и __global__: они не могут быть рекурсивными (то есть вызывать самих себя) и не могут иметь переменное число аргументов. Наконец, поскольку функции __device__ располагаются в пространстве памяти GPU, вполне логично, что получить их адрес не удастся.

API CUDA содержит функции для работы с памятью в VRAM: cudaMalloc для выделения памяти, cudaFree для освобождения и cudaMemcpy для копирования памяти между RAM и VRAM и наоборот.

Компиляция выполняется в несколько этапов (Рис. 9). Сначала извлекается код, относящийся к CPU, который передаётся стандартному компилятору. Код, предназначенный для GPU, сначала преобразовывается в промежуточный язык PTX. Он подобен ассемблеру и позволяет изучать код в поисках потенциальных неэффективных участков. Наконец, последняя фаза

заключается в трансляции промежуточного языка в специфические команды GPU и создании двоичного файла.

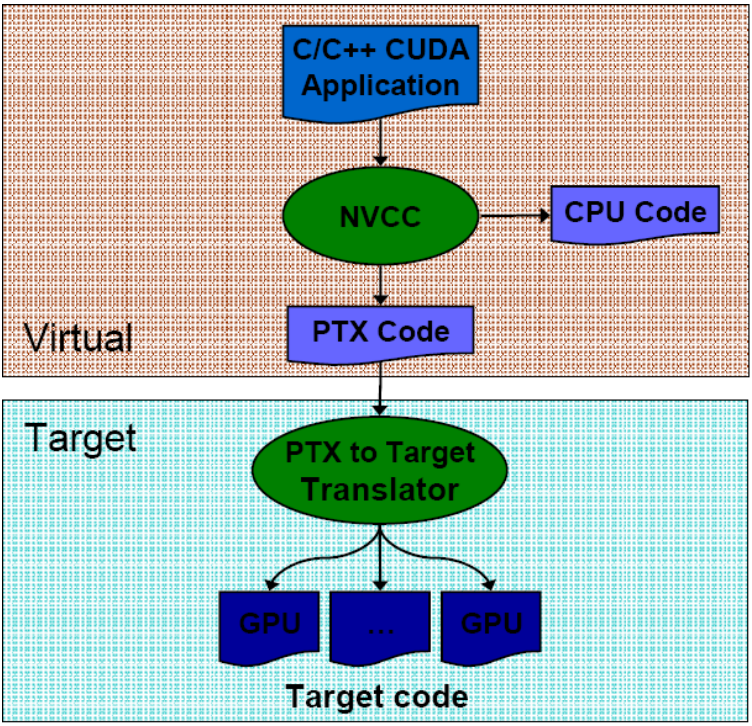

Рисунок 9

Переменные тоже имеют ряд квалификаторов, которые указывают на область памяти, где они будут храниться. Ниже перечислен список предусмотренных CUDA API префиксов:

- **__device__** описывает переменную, создающуюся в памяти GPU.
- **__constant__** опционально используется вместе с **__device__**, объявляет переменную как:
  1. Существующую в константном пространстве памяти
  2. Обладающую временем жизни приложения.
  3. Доступную из всех потоков сетки и с CPU через библиотеки.
- **__shared__** опционально используется вместе с **__device__**, объявляет переменную как:

1. Существующую в памяти блока потоков.
2. Обладающую временем жизни блока потоков.
3. Доступную всем потокам блока.

Только после выполнения **__syncthreads**() записанные данные в **__shared__** переменные будут гарантированно видны другим потокам блока.

Данные определители не могут быть применены к членам структур (**struct**) и объединений (**union**), формальным параметрам и локальным переменным функций которые выполняются на CPU.

- **__shared__** и **__constant__** хранятся статически.
- **__device__**, **__shared__** и **__constant__** не могут быть определены как глобальные переменные, т.е. имеющие идентификатор **extern**.
- **__device__** и **__constant__** действуют только в границах одного файла.
- **__shared__** не может быть объявлено при инициализации.

Переменные, объявляющиеся в функциях, исполняемых на видеоадаптере без данных идентификаторов, обычно хранятся в регистрах. Однако, в некоторых случаях, компилятор может разместить их в локальной памяти потока. Это обычно случается с большими массивами данных занимающих очень много регистровой памяти.

Для любого вызова **__global__** функции должна быть описана исполняемая конфигурация. Она определяет измерения сетки и блоков, которые будут выполнять функцию на устройстве. Задается конфигурирование выражением **<<<Dg, Db, Ns, S >>>** между именем функции и списком аргументов:

- **Dg** имеет тип **dim3** и описывает измерение и размер сетки, такие что **Dg.x * Dg.y** эквивалентны числу блоков которые будут запущены; **Dg.z** не используется.
- **Db** имеет тип **dim3** и описывает измерение и размер каждого блока, такой, что **Db.x * Db.y * Db.z** эквивалентно числу потоков на блок.

- **Ns** имеет типа **size_t** и описывает число байт в общей памяти, которое динамически выделено на блок; эта динамически выделенная память используемая любыми переменными объявленными как внешний массив; **Ns** опциональный аргумент который по умолчанию равен 0.
- **S** имеет **cudaStream_t** тип и описывает поток; **S** опциональный аргумент который по умолчанию равен 0.

Функция, объявленная следующим образом:

__global__ void Func(float* parameter);

Должна вызываться как:

Func<<< Dg, Db[, Ns, S]>>>(parameter);

Аргументы конфигурации вычисляются перед аргументами функции. Вызов функции может завершиться неудачно, если **Db** или **Dg** больше, чем максимальный размер доступный для установленного оборудования или если **Ns** больше чем максимальное количество общей памяти доступной на устройстве минус количество памяти, требуемое для размещения статических объектов, аргументов функций, и конфигураций.

5.1. Предопределенные переменные

- **gridDim** – тип переменной **dim3**, содержит размерность сетки.
- **blockIdx** – тип переменной **uint3**, содержит индекс блока в сетке.
- **blockDim** – тип переменной **dim3**, содержит размерность блока.
- **threadIdx** - тип переменной **uint3**, содержит индекс потока в блоке.
- **warpSize** - тип переменной **int**, содержит размер оболочки в потоках.

Данные переменные не позволяют взятие их адреса и построение указателей и предоставлены только для чтения.

5.2. Предопределенные типы данных

char1, **uchar1**, **char2**, **uchar2**, **char3**, **uchar3**, **char4**, **uchar4**, **short1**, **ushort1**, **short2**, **ushort2**, **short3**, **ushort3**, **short4**, **ushort4**, **int1**, **uint1**, **int2**, **uint2**, **int3**, **uint3**, **int4**, **uint4**, **long1**, **ulong1**, **long2**, **ulong2**, **long3**, **ulong3**, **long4**, **ulong4**, **float1**, **float2**, **float3**, **float4**, **double2 –** все это встроенные типа данных наследованные от фундаментальных аналогичных типов языка С. Они

структурированы по 1, 2, 3 – 4 компонентам по доступности полей **x, y, z, w**.

Их можно конструировать с помощью функции **make<type name>**, например:

int2 make_int2(int x, int y);

данная функция создает вектор типа **int2** со значениями **(x, y).**

5.3. Инициализация CUDA

Перед вызовом любой первой **__global__** функции необходимо инициализировать CUDA устройство. Для этого существуют специальные функции в CUDA API. Такие как: **cudaGetDeviceCount** и **cudaGetDeviceProperties**.

- **cudaGetDeviceCount(int *deviceCount)** – возвращает количество устройств поддерживающих технологию CUDA.
- **cudaGetDeviceProperties(cudaDeviceProp devProp, int devIndex)** – позволяет получить свойства устройства с индексом devIndex, в виде заполненной структуры cudaDeviceProp.
- **cudaSetDevice(int device) –** используется для выбора устройства связанного с данным потоком CPU.

Таким образом, простейшая процедура инициализации CUDA будет выглядеть следующим образом:

```
bool InitCUDA(void)
{
  int count = 0;
  int i = 0;
  cudaGetDeviceCount(&count);
  if(count == 0) {
        fprintf(stderr, " CUDA  устройств нет.\n");
        return false;
  }
  for(i = 0; i < count; i++) {
        cudaDeviceProp prop;
        if(cudaGetDeviceProperties(&prop, i) == cudaSuccess) {
```

```
                if(prop.major >= 1) {
                    break;
                }
            }
        }
        if(i == count) {
            fprintf(stderr, " CUDA  устройств нет.\n");
            return false;
        }
        cudaSetDevice(i);
        printf("CUDA initialized.\n");
        return true;
    }
```

В данном листинге мы видим, что с помощью функции **cudaGetDeviceCount** мы получаем количество доступных поддерживающих технологию CUDA устройств и проверяем свойства каждого устройства на пример **major** версии. Как только мы нашли устройство с версией больше либо равной единице устанавливаем его как активное, для данного потока, с помощью функции **cudaSetDevice**, иначе, если таких устройств нет, выдаем оповещение, что нет устройств, поддерживающих CUDA, и возвращаем из функции значение **false**.

Мажорная и минорная версии обозначают вычислительную мощность устройства. Устройства с мажорными версиями имеют одинаковую архитектуру ядра. Минорная версия соответствует возрастанию улучшений ядра, добавлению новых особенностей и свойств.

5.4. Управление памятью

Как уже упоминалось выше, для выделения памяти на устройстве можно использовать функцию **cudaMalloc**, а для освобождения - **cudaFree**. В данной главе мы рассмотрим еще некоторые очень полезные функции для управления памятью. Первая из них это **cudaMallocPitch(void **devPtr, int *pitch, size_t width, size_t height)** – функция рекомендуется для выделения памяти под двумерный массив для того, чтобы память была соответствующим образом выровнена. Следовательно, будет уверенность в улучшении производительности при доступе к памяти и при копировании между двумерными массивами или другими регионами памяти устройства. Следую-

щий код пример демонстрирует выделение памяти под двумерный массив размерности **width*heigh** и показывает, как перебрать все элементы массива:

```
//Код, выполняемый на хосте(CPU)
float* devPtr;
int pitch;
cudaMallocPitch((void**)&devPtr, &pitch, width * sizeof(float), height);

myKernel<<<100, 512>>>(devPtr, pitch);
```

```
//Код, выполняемый на GPU
__global__ void myKernel(float* devPtr, int pitch)
{
    for (int r = 0; r < height; ++r)
    {
        float* row = (float*)((char*)devPtr + r * pitch);
        for (int c = 0; c < width; ++c)
            float element = row[c];
    }
}
```

Под массивы память можно выделить функцией **cudaMallocArray** и освободить функцией **cudaFreeArray**. **cudaMallocArray** требует описание формата созданного массива созданное функцией **cudaCreateChannelDesc.**

Следующий код, выделяет память для массива размером **width×height** - содержащего 32 битные числа с плавающей точкой:

```
cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc<float>();
cudaArray* cuArray;
cudaMallocArray(&cuArray, &channelDesc, width, height);
```

Функция **cudaGetSymbolAddress** используется, чтобы получить адрес переменной расположенной в глобальной памяти. Размер выделенной памяти доступен через функцию **cudaGetSymbolAddress.**

Существуют различные функции для копирования регионов памяти, например, для копирования двумерного массива, для массива показанного в предыдущем примере используется следующая функция:

```
cudaMemcpy2DToArray(cuArray, 0, 0, devPtr, pitch, width * sizeof(float), height, cudaMemcpyDeviceToDevice);
```

Чтобы скопировать в константную память из памяти, хоста нужно сделать следующее:

```
__constant__ float constData[256];
float data[256];
cudaMemcpyToSymbol(constData, data, sizeof(data));
```

Каждый поток копирует свою порцию данных входного массива **hostPtr** в массив **inputDevPtr** в памяти устройства, **inputDevPtr** обрабатывается на устройстве посредством вызова **myKernel**, и копирует результат **outputDevPtr** назад в **hostPtr**.

6. УПРАВЛЕНИЕ ПРОИЗВОДИТЕЛЬНОСТЬЮ

Для того, чтобы обработать инструкцию для варпа потоков, мультипроцессор должен:

- прочитать операнды инструкций для каждого потока варпа;
- выполнить инструкцию;
- записать результат для каждого потока варпа.

Следовательно, эффективная производительность инструкций зависит от номинальной производительности, так же как задержка памяти и пропускная способность. Производительность можно максимизировать посредством:

- минимизации использования инструкций с низкой производительностью,

- максимизации использования доступной полосы пропускания памяти для каждой категории памяти.
- позволения планировщику памяти перекрывать транзакции памяти с математическими вычислениями настолько, насколько это возможно, требуют чтобы:
  - программа, выполняющаяся посредством потоков с высокой арифметической интенсивностью, имела высокое число арифметических операций на одну операцию с памятью;
  - существовало много активных потоков на один мультипроцессор.

1. Производительность инструкций.

   Чтобы выдать одну инструкцию, мультипроцессор занимает:
   - 4 такта для:
     - операций добавления с одинарной точностью и плавающей точкой, умножения, и многократного добавления,
     - целочисленного добавления,
     - битовых операций, сравнения, инструкций преобразования типов.
   - 16 тактов для обратных операций, квадратного корня, **__logf(x).**

32 – битное целочисленное умножение занимает 16 тактов, но **__mul24** и **__umul24** обеспечивает знаковое и беззнаковое 24-битное умножение в 4 такта. На будущих архитектурах, однако, **__[u]mul24** будет медленнее, чем 32- битное целочисленное умножение, рекомендуется обеспечить два ядра («**kernels**»), одно использует **__[u]mul24**, а другое общее 32-битное умножение и вызывать их соответствующим образом.

Целочисленное деление и операция взятия модуля являются очень дорогостоящими и должны быть обойдены, если это возможно, либо заменены битовыми операциями: если n –степень двойки, тогда **i/n** эквивалентно **i >> log2(n)** и **i % n** эквивалентно **(i & (n - 1))**. Другие функции занимают больше тактов, так как они являются комбинацией нескольких инструкций.

__sinf(x), **__cosf(x)**, **__expf(x)** занимают 32 такта, **sinf(x)**, **cosf(x)**, **tanf(x), sincosf(x)** являются более дорогостоящими операциями, если абсолютное значение **x** больше чем 48039. Более того, в этом случае, для уменьшения аргумента код использует локальную память, которая может влиять на производительность больше в силу медленного доступа и малой пропускной способности.

Иногда компилятор может вставлять преобразовывающие инструкции, которые уменьшают количество дополнительных циклов. Это случается, когда:

- Функция оперирует на **char** или **short,** чьи операнды, как правило, должны быть преобразованы к **int**.
- Константы с плавающей точкой имеющие двойную точность(определенные без каких либо типовых суффиксов) используются как ввод для вычислений с плавающей точкой имеющих одинарную точность.

  Последнего пункта можно избежать использованием:
- Констант с плавающей точкой одинарной точности, определенных с суффиксом **f,** таких как **3.141592653589793f, 1.0f, 0.5f.**
- Версий функций с одинарной точностью, имеющих суффикс **f – sinf(), logf(), expf()**.

Для операций с одинарной точностью строго рекомендуется использовать тип **float** и версии функций с одинарной точностью.

2. Контроль потока инструкций

Любой поток управляющих инструкций (**if, switch, do, for, while**) может значительно ударить по эффективности производительности инструкций посредством расщепления потоков в одном варпе, что приводит к различным путям выполнения. Если это случилось, потоки будут упорядочены, что вызовет выполнение дополнительных инструкций и затраты времени. Чтобы получить лучшую производительность в случае, когда контроль потока зависит от **id** потока, управляющие условия должны быть написаны так, чтобы минимизировать число различных разветвлений.

Иногда компилятор может развернуть циклы или оптимизировать использование **if** или switch операторов посредством ветвления предикатов. Это можно контролировать, используя **#pragma unroll** директиву.

3. Глобальная память

Глобальная память не кэшируется, а это очень важно для получения максимальной производительности. Ниже описаны два способа, помогающих ускорить процесс работы с глобальной памятью:

Первое устройство способно читать 32, 64, 128 битные слова из глобальной памяти в регистры в одну инструкцию. Чтобы иметь такое определение:

```
__device__ type device[32];
type data = device [tid];
```

компилируемое в выполняемое в одну инструкцию. **type** должен быть такой, чтобы **sizeof(type)** был эквивалентен 4, 8 или 16 байтам, и переменные типа type должны быть выравнены на **sizeof(type)** байт.

Для структур требования выравнивания и размера могут быть выполненными компилятором посредством использования директивы **__align__(8)** или **__align__(16)**, например:

```
struct __align__(16) {
float a;
float b;};
```

Для структур больше 16 байт компилятор генерирует различные инструкции загрузки. Для того чтобы убедиться что будет генерироваться минимальное количество инструкций мы можем определить структуру с директивой **__align__(16)**. Тогда компилятор будет использовать выравнивание по максимальному размеру.

Любой адрес переменной, живущей в глобальной памяти или возвращающийся одной операцией выделения памяти из драйвера или во время выполнения, всегда выровнен по наименьшим 256 байтам.

Второе, пропускная способность глобальной памяти используется более эффективно когда одновременный доступ к памяти потоков в половине варпа может быть объединен в одну транзакцию, размер которой может быть также 32, 64 или 128 байт.

7. ОБЪЕДИНЕНИЕ НА УСТРОЙСТВАХ С COMPUTE CAPABILITY 1.0 И 1.1

Доступ к глобальной памяти всеми потоками половины варпа объединяется в одну или две транзакции, если это удовлетворяет следующим трем условиям:

- Потоки должны получить доступ:
  - Любым 32-битным словам, выливающийся в 64-байтовую транзакцию.

  - Или 64-битным словам, выливающийся в 128-байтовую транзакцию.
  - Или 128-битным словам, выливающийся в 128-байтовую транзакцию
- Все 16 слов должны лежать в одном сегменте имеющего размер эквивалентный размеру транзакции.
- Потоки должны получить доступ к словам в последовательности: **k**-й поток обращается к **k**-му слову.

Если эти условия не будут выполняться, тогда транзакции будут осуществляться для каждого потока, что значительно повлияет на производительность системы. Рис. 10 показывает некоторые примеры объединенного доступа к памяти, в то время, как рис. 11 и рис. 12 показывают примеры не объединенного доступа.

8. ОБЪЕДИНЕНИЕ НА УСТРОЙСТВАХ С COMPUTE CAPABILITY 1.2 И ВЫШЕ

Доступ к глобальной памяти всеми потоками половины варпа объединяется в одну или две транзакции как только слова доступные всем потокам лежат в одном сегменте размером:

- 32 байта, если доступ к 8 – битным словам.
- 64 байта, если доступ к 16 битным словам.
- 128 байт, если доступ к 32 или 64 битным словам.

Объединение будет возможно для любых запрошенных образцов адресов, включая образцы, когда много потоков обращаются к одному адресу. Если потоки адресуют слова в n различных сегментах, тогда будет осуществлено n транзакций памяти, по транзакции на каждый сегмент, тогда как устройства с меньшей версией вычислительной способности выпускали бы 16 транзакций, как только n было бы больше 1. В частности, если поток обращается к 128 – битному слову, это вызвало бы менее чем 2 транзакции.

Неиспользуемые слова в транзакции так же читаются, хотя они не нужны и уменьшают производительность. Необходимо добиться как можно меньшего наличия таких слов с целью увеличения полосы пропускания памяти.

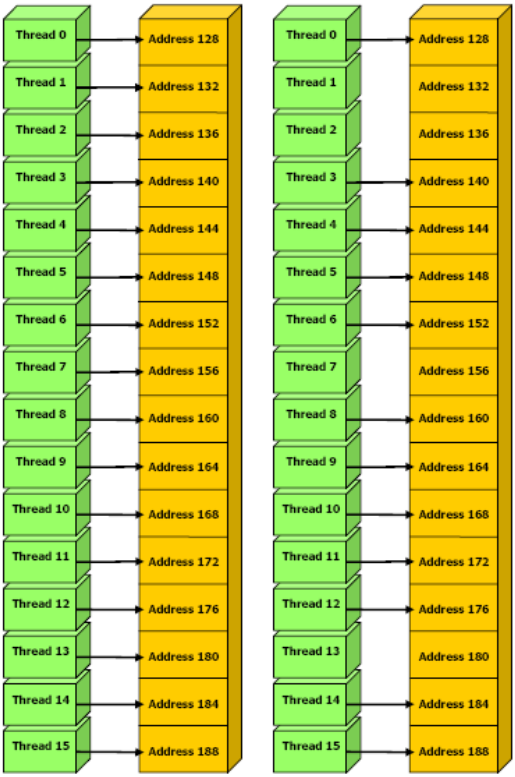

Рисунок 10
Левая колонка: объединенный float доступ к памяти, результат 1 транзакция.
Правая колонка: объединенный float доступ к памяти(расходящиеся потоки), результат 1 транзакция.

Ниже показан алгоритм, который используется для осуществления транзакции памяти:

- Найти сегмент памяти, который содержит в себе адрес, запрошенный наименьшим числом активных потоков. Размер сегмента в 32-байта для 8 битных данных, 64 – байта для 16 – битных данных, 128 байт для 32-, 64- и 128 битных данных.
- Найти все другие активные потоки, чьи адреса лежат в одном сегменте.
- Уменьшить размер транзакции, если возможно.
- Если размер транзакции 128 байт и только нижняя или верхняя половина использовалась, уменьшить размер до 64 байт.

Рисунок 11
Левая колонка: непоследовательный доступ к памяти, результат 16 транзакций.
Правая колонка: доступ с невыровненным стартовым адресом, результат 16 транзакций.

- Если размер транзакции 64 байт и только нижняя или верхняя половина использовалась, уменьшить размер до 32 байт.
- Выполнить транзакцию и пометить обслуживаемые потоки как неактивные.
- Повторить пока все потоки не будут обслужены.

Рис. 13 показывает некоторые примеры доступа к глобальной памяти для устройств с Compute Capability 1.2.

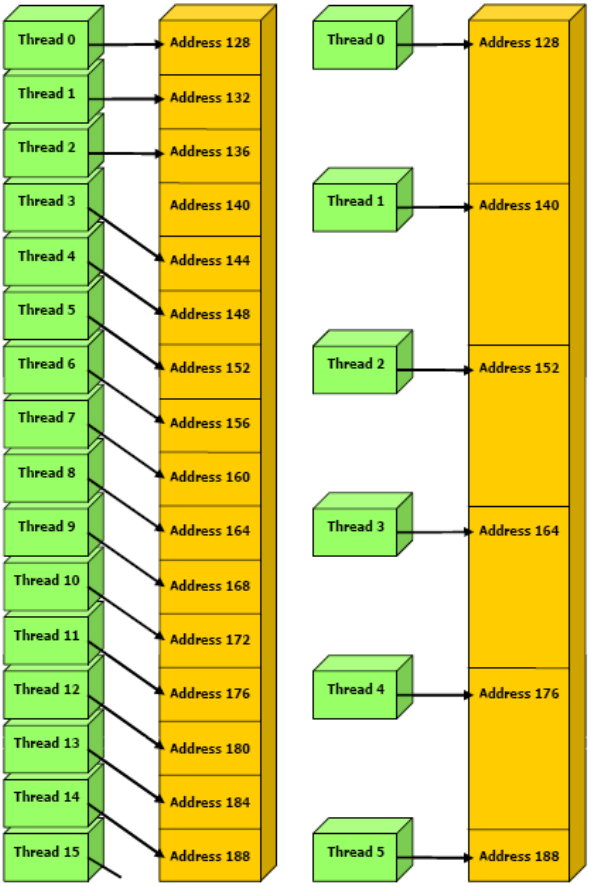

Рисунок 12
Левая колонка: невыровненный доступ, результат 16 транзакций.
Правая колонка: необъединенный доступ, результат 16 транзакций.

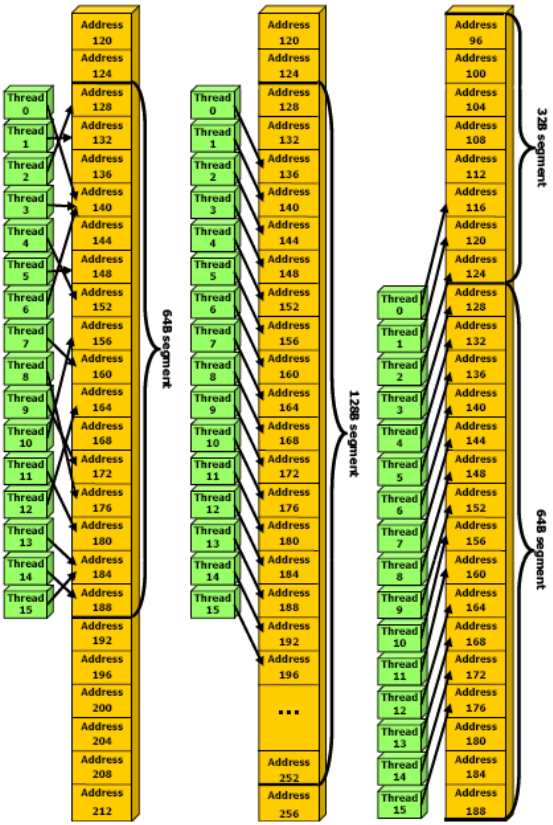

Рисунок 13
Слева: произвольный доступ в пределах 64 байтового сегмента, результат 1 транзакция.
Центр: невыровненный доступ, результат 1 транзакция.
Справа: невыровненный доступ, результат 2 транзакции.

СПИСОК ЛИТЕРАТУРЫ

1. Берилло А. NVIDIA CUDA – неграфические вычисления на графических процессорах // Информационный ресурс сети интернет IXBT.com, 23.09.2008 г. – http://www.ixbt.com/video3/cuda-1.shtml.
2. Чеканов Д. NVIDIA CUDA: вычисления на видеокарте или смерть CPU? // Tom’s Hardware Guide, 22.07.2008 г. – http://www.thg.ru/graphic/nvidia_cuda/index.html.
3. CUDA 2.0 Programming Guide // NVIDIA Corporation, 2007–2008.