Вычисления общего назначения на GPU

GPU уже достигли той точки развития, когда многие приложения реального мира могут с легкостью выполняться на них, причем быстрее, чем на многоядерных системах. Будущие вычислительные архитектуры станут гибридными системами с графическими процессорами, состоящими из параллельных ядер и работающими в связке с многоядерными ЦП

GPUs have evolved to the point where many real-world applications are easily implemented on them and run significantly faster than on multi-core systems. Future computing architectures will be hybrid systems with parallel-core GPUs working in tandem with multi-core CPUs.

Профессор Джек Донгарра (Jack Dongarra)
Директор Innovative Computing Laboratory
Университет штата Теннесси

Вычисления общего назначения на GPU

История графических чипов началась с графических конвейеров с фиксированной функциональностью.

За счет узкой специализации удалось добиться создания устройств с внушительными вычислительными способностями и относительно небольшой стоимостью. В настоящее время вычислительная мощность современных видеоускорителей в десятки и даже сотни раз превышает аналогичные показатели центральных процессоров при примерно равной стоимости.

Долгое время эти мощности были доступны лишь для разработки графических приложений и не могли использоваться для решения прикладных программ.

Однако потенциал в данной области был очевиден, поэтому предпринимались многочисленные попытки приспособить видеоускорители для решения прикладных задач.

История графических чипов началась с графических конвейеров с фиксированной функциональностью.

За счет узкой специализации удалось добиться создания устройств с внушительными вычислительными способностями и относительно небольшой стоимостью. В настоящее время вычислительная мощность современных видеоускорителей в десятки и даже сотни раз превышает аналогичные показатели центральных процессоров при примерно равной стоимости.

Долгое время эти мощности были доступны лишь для разработки графических приложений и не могли использоваться для решения прикладных программ.

Однако потенциал в данной области был очевиден, поэтому предпринимались многочисленные попытки приспособить видеоускорители для решения прикладных задач.

В 1999-2000 годах специалисты в компьютерной области и научные работники в таких сферах, как получение медицинских изображений и электромагнетизм, перешли на **GPU** для вычислительных приложений общего назначения.

Они обнаружили, что высокая производительность вычислений с плавающей точкой графических процессоров значительно ускоряла работу научных приложений.

Это стало началом мощного движения, называющегося **GPGPU** или вычисления общего назначения на **GPU**.

К настоящему времени в корпорациях - лидерах отрасли по производству видеоускорителей (**nVidia** и **ATI**) - разработаны технологии, позволяющие использовать мощности видеоускорителей для проведения расчетов в прикладных задачах. Этими технологиями являются nVidia **CUDA** и ATI **Stream**.

Существуют и другие технологии, семейства **GPGPU**, то есть позволяющие использовать ресурсы видеоускорителей в прикладных программах: Microsoft© **Direct3D 11 - Compute Shader,** а также полностью открытая технология **OpenCL**, разработку которой поддержали такие компании как Apple, nVidia, AMD, IBM, Intel, ARM, Motorola и другие. В версии MATLAB R2010b в компонент Parallel Computing Toolbox была добавлена официальная поддержка вычислений с применением GPU. В настоящий момент объявлено о поддержке только графических процессоров NVidia.

Общим для этих технологий является то, что, затратив определенные усилия на адаптацию прикладных программ к использованию в качестве вычислителей графические процессоры (**GPU**), можно значительно **понизить стоимость вычислений**, получив при этом, **схожую производительность**.

Ниже речь пойдёт о технологии от компании nVidia – **CUDA** (Common Unified Development Architecture).

Аппаратной архитектуре параллельных вычислений **CUDA** сопутствует среда программирования **CUDA**, которая обеспечивает набор абстракций, позволяющих выражать как параллелизм данных, так и параллелизм задач.

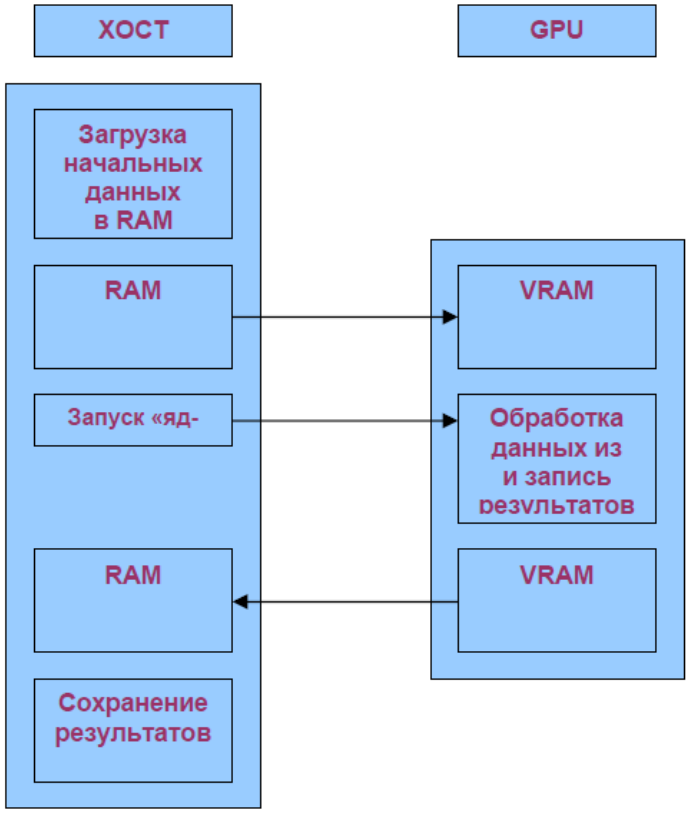
Набор программных инструментов разработки **C** для **CUDA** позволяет программировать **GPU** с помощью этого языка с минимальным набором ключевых слов и расширений.

С программной точки зрения, технология **CUDA** представляет собой расширение языка **Си**, которое можно представить в виде двух взаимно зависимых составляющих :

- Средства программирования для создания кода, выполняемого в рамках обычной программы на центральном процессоре (**CPU**);

- Средства программирования позволяющие выполнять части (фрагменты) программы (проводить часть вычислений) на видеоускорителе (**GPU**).

Первая из указанных составляющих (т.н. «управляющая часть») позволяет производить такие операции как выделение памяти на GPU и копирование данных между GPU и хостом, то есть служит для организации запусков вычислений на GPU [3] - "вводить" начальные данные и получать результаты расчетов.



На рисунке:
RAM – это память центрального процессора,
VRAM – память видеоускорителя.

Согласно изображенной схеме: под управлением центрального процессора осуществляется передача данных из оперативной памяти CPU в память GPU, затем по «команде» из CPU запускаются вычисления на GPU (запуск «ядра»). Результаты вычислений «считываются» из памяти видеоадаптера.

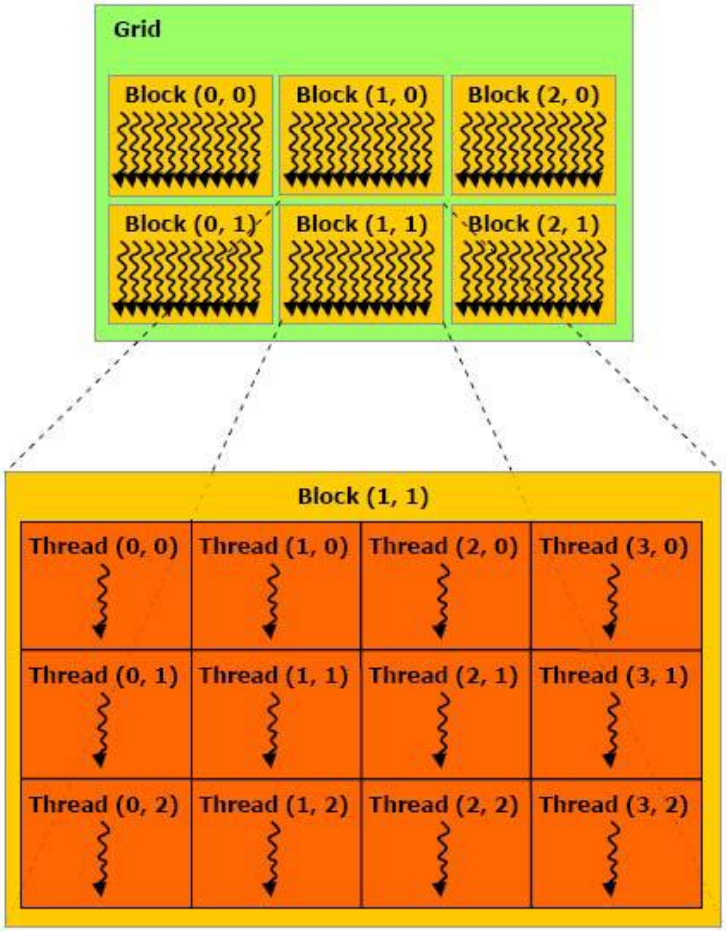
Для описания этой части технологии **CUDA** нам понадобятся следующие определения:

Потоком исполнения или просто потоком (***thread***) называется совокупность логических и арифметических операций (исполняемый код), выполняемых в определённой последовательности на вычислительном устройстве. При наличии множества потоков будем предполагать, что они выполняются параллельно, то есть без предписанного порядка во времени.

Ядро - это совокупность потоков, запускаемая на выполнение с хоста и выполняемая на GPU. Потоки в ядре группируются в так называемые ***блоки (block)***, совокупность блоков представляет собой ***решетку*** (***grid***).

Warp - это совокупность потоков блока, исполняемых одновременно в текущий момент. Число потоков в **warp**'е определяется системой и обычно равно 32.

Потоки в блоке организованы в трёхмерный массив. Блоки в решетке – в двумерный.

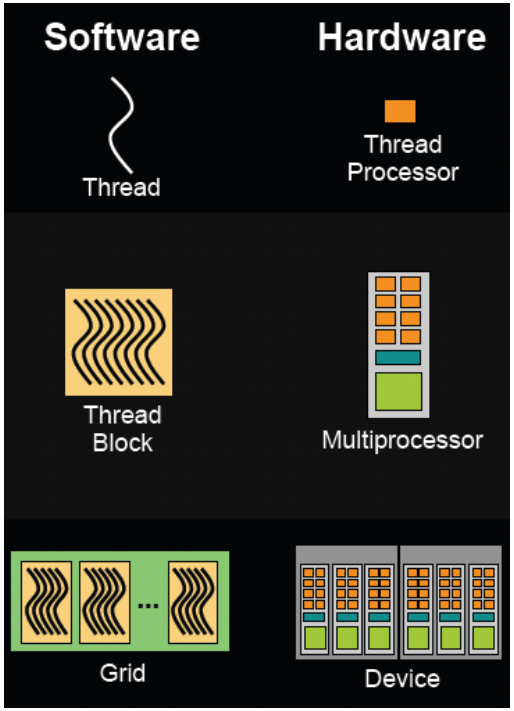


Все потоки имеют идентичный исполняемый код, поэтому, чтобы отличать потоки друг от друга, каждому присвоен номер его блока в решетке (двумерный индекс) и его номер в блоке (трёхмерный индекс).

На рисунке изображена структура решетки, состоящей из набора блоков, и схема блока, состоящего из набора потоков. Для наглядности блок представлен двумерным массивом, то есть размерность блока по третьему индексу равна единице.

При запуске ядра (совокупность программных пакетов) блоки распределяются между мультипроцессорами (при нехватке мультипроцессоров становятся в очередь) и выполняются независимо. С каждым мультипроцессором может быть ассоциирован один и более блоков (в зависимости от ресурсов, используемых блоком).

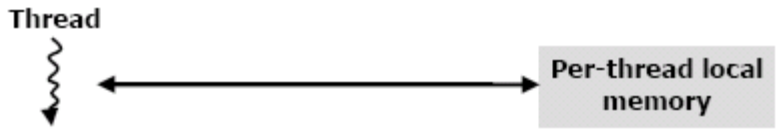
Один блок не может выполняться более чем на одном мультипроцессоре.



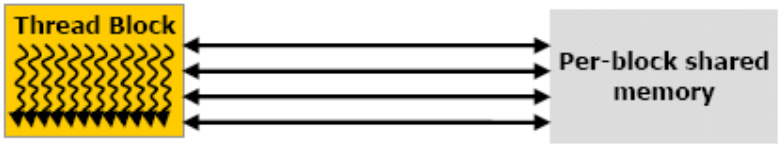
На рис. показано соответствие программных средств и аппаратных устройств при проведении расчетов с использованием графических ускорителей:

- поток выполняется вычислителем (потоковым процессором),
- блок (совокупность потоков) - мультипроцессором,
- решетка (grid) – устройством (видеокарта).

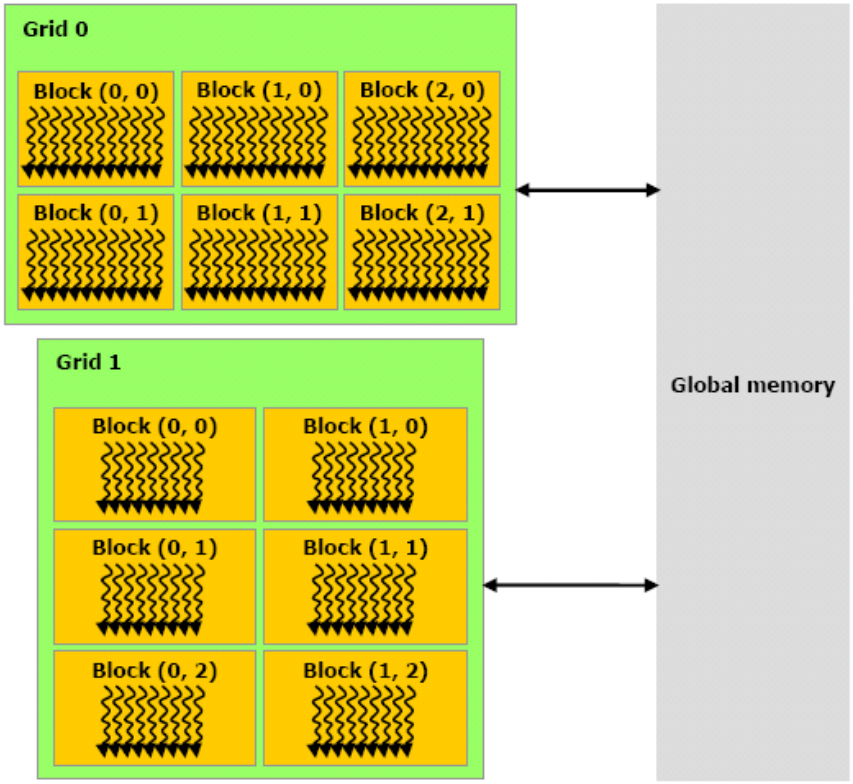
Важной особенностью графического адаптера является наличие трех видов памяти: локальной (**local**), связанной с конкретным потоковым процессором, разделяемой (**shared**) памяти, доступной в рамках мультипроцессора, и глобальной (**VRAM**) – определенной для видеоадаптера. На рис. показаны доступность и «время жизни» данных, находящихся в различных видах памяти.



Так, ***локальная*** память доступна только потоку, который выполняется на данном вычислителе. Время жизни данных в такой памяти равно времени жизни потока.



Разделяемая память доступна всем потокам блока и её время жизни определяется временем жизни блока.



Глобальная память доступна всем потокам во всех блоках решетки.

Время жизни областей глобальной памяти определяется программой: области глобальной памяти могут быть выделены или освобождены только с хоста, то есть только между вызовами ядер. Таким образом, время жизни области в глобальной памяти больше, чем время жизни одного или нескольких ядер (Grid0 и Grid1 на рисунке — решетки различных ядер) и меньше, чем время жизни программы.

Глобальная память, при всём её удобстве, не кэшируется и является относительно медленной по сравнению с разделяемой памятью. Использование в процессе решения задачи разделяемой памяти вместо глобальной позволяет добиться значительного увеличения эффективности использования графических процессоров.

CUDA - приложения фактически являются многопоточными приложениями. Однако, в силу специфики потоков, а именно, большого их количества и крайней легковесности (то есть, крайне малых затрат на создание, уничтожение потоков и переключение между ними и, как следствие, ограниченных возможностей управления потоками), программирование **CUDA** - приложений значительно отличается от программирования обычных многопоточных приложений.

Отметим основные из этих отличий.

Первое. В силу того, что задачам, под которые изначально создавались современные видеоадаптеры, характерна практически абсолютная декомпозиция данных (то есть для параллельно обрабатываемых данных результаты обработки одной их части не зависят от результатов обработки другой), в **CUDA** - приложениях практически отсутствуют методы контроля конкурентного доступа к данным. Существуют минимальные возможности такого контроля для потоков в рамках одного блока, однако их зачастую бывает недостаточно. Такие ограничения требуют введения определённой степени декомпозиции данных в прикладных приложениях. То есть, для параллельно обрабатываемых данных требуется по возможности максимально уменьшить, а то и вовсе исключить, зависимость результатов обработки этих данных друг от друга.

Другая особенность также обусловлена архитектурой аппаратных средств, нацеленных на задачи обработки графики, в которых алгоритмы линейны и не содержат условных переходов.

То есть, условные переходы не поддерживаются аппаратной частью вовсе.

При этом условные переходы и циклы хоть и поддерживаются программными средствами CUDA, тем не менее, являются самыми «тяжелыми» операциями для GPU, поэтому при написании приложений следует старательно избегать таких операций.

При разработке CUDA - приложений для управления и оптимизации доступны два вида памяти графического адаптера: ***глобальная*** и ***разделяемая*** (существует также ***локальная*** память, но её распределением занимается компилятор, и программист не может управлять этим видом памяти).

Глобальная память (**VRAM**). Скорость доступа к этой памяти является достаточно низкой, однако при определённом подходе (см. ниже) к организации доступа, может быть значительно повышена. Такое повышение достигается при помощи «**связывания**» нескольких логических запросов к памяти (запросов от разных потоков) в один аппаратный (физический) запрос.

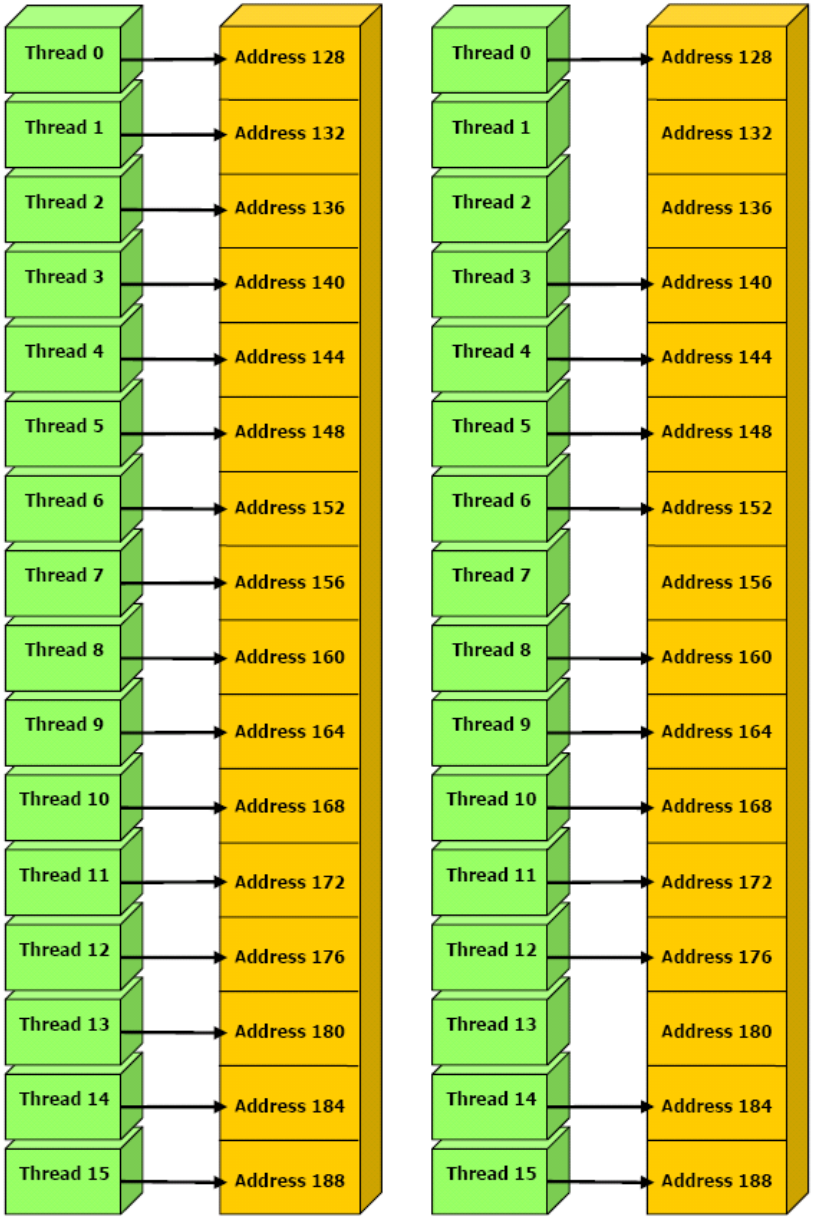
«**Связанным**» (“coalesced”) запросом к ресурсам глобальной памяти (**VRAM**) называется такая совокупность логических запросов, при которой половина потоков ***warp***'а осуществляет чтение/запись за один физический запрос.

Для того чтобы запросы стали **связанными** необходимо выполнение следующих условий (для случая доступа к четырёхбайтным элементам):

1. ячейки памяти, к которым производится доступ, лежат в области памяти размером 64 байта.
2. физический адрес начала области памяти кратен 64 байтам.
3. номер ячейки в области памяти должен совпадать с номером потока, который к этой ячейке обращается (для более современных устройств это условие не обязательно).

В случае, если хотя бы одно из условий нарушается, логические запросы от разных потоков перестают быть связанными, и для каждого из них формируется отдельный физический запрос.

Таким образом, максимальное число запросов, которые можно связать, равно половине размера **warp**'а, то есть максимальная скорость чтения из глобальной памяти может быть в 16 раз выше минимальной скорости.



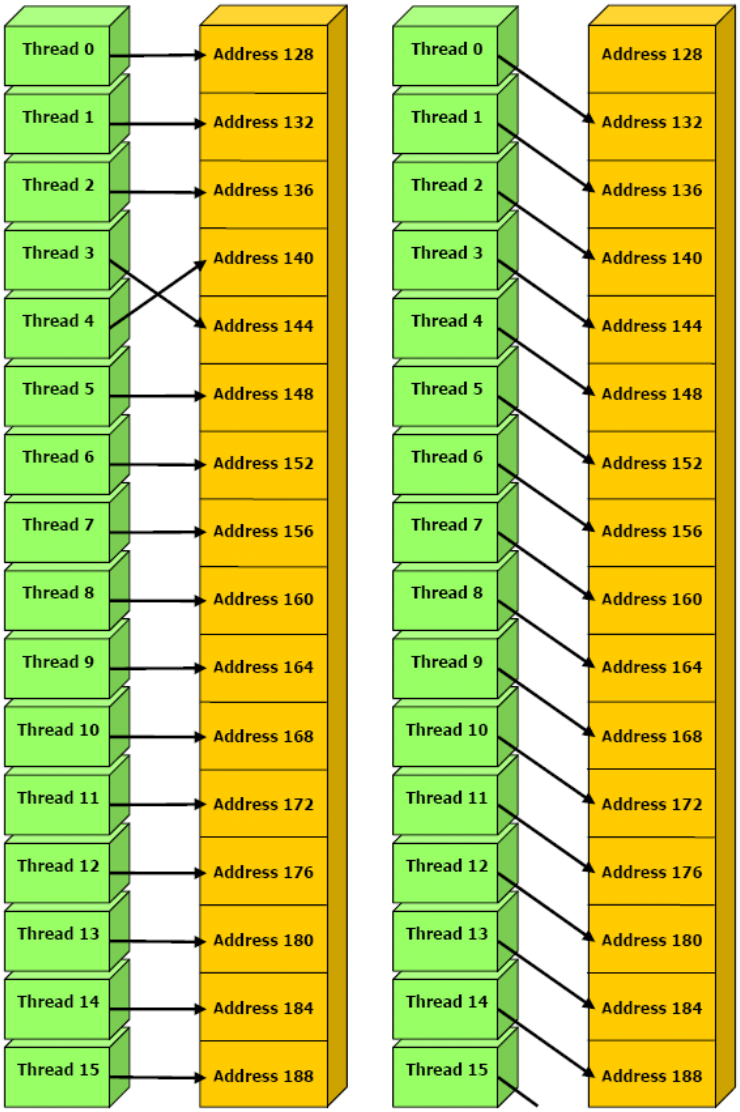
Рассмотрим примеры.

На рис. все потоки читают из ячеек расположенных последовательно.

Номера ячеек соответствуют номеру потока.

В обоих изображенных на рисунке случаях, для получения данных всеми потоками будет сформирован один единственный физический запрос к памяти.

Пример «**связанного**» запроса к памяти.



На рис. изображен пример «**несвязанного**» доступа к памяти.

Слева - непоследовательное обращение к ячейкам, расположенным в области, допускающей «связанные» запросы.
Справа - последовательное обращение к ячейкам, выходящее за область, допускающую «связанные» запросы.

В обоих изображенных на рисунке случаях, для получения данных всеми потоками будут сформированы 16 запросов к памяти

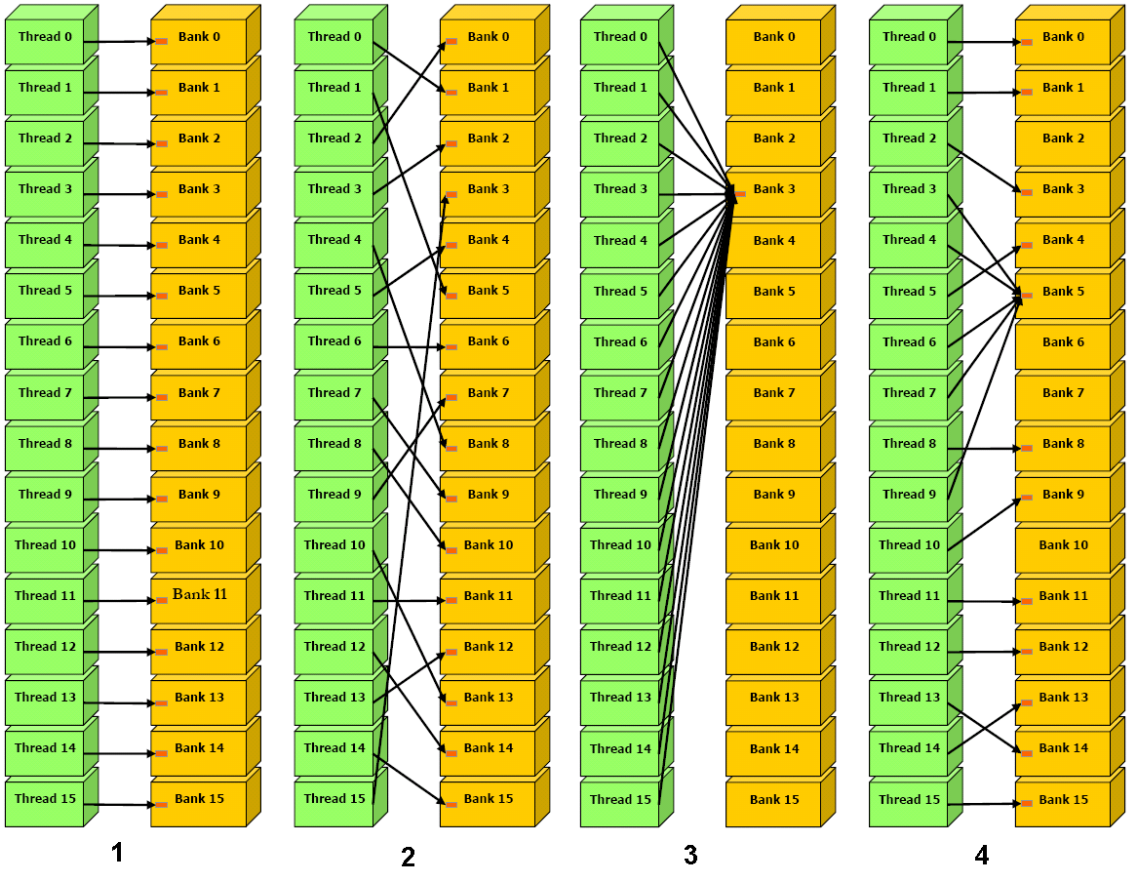
Современные устройства могут частично «исправлять» такие ситуации: обращение слева «связывается» и приводится к единственному запросу к памяти; обращение справа приводится к двум запросам к памяти.

Разделяемая (быстрая) память. Организация этого вида памяти отличается от организации глобальной памяти.

Разделяемая память разбита на так называемые «**банки**». При обращении нескольких потоков к ячейке памяти внутри одного и того же банка, происходит так называемый «банк-конфликт», в результате которого одиночный запрос на чтение превращается в N запросов, где N – максимальное количество потоков, обращающихся одному банку.

Для уменьшения количества банк-конфликтов при осуществлении запроса на чтение, один из банков объявляется системой «**широковещательным**» (обычно это тот, к которому обращается максимальное количество потоков).

Запрос на чтение из «**широковещательного**» банка всегда происходит как одиночный запрос, вне зависимости от количества обращающихся потоков. Тем самым разрешается один банк-конфликт максимального порядка.

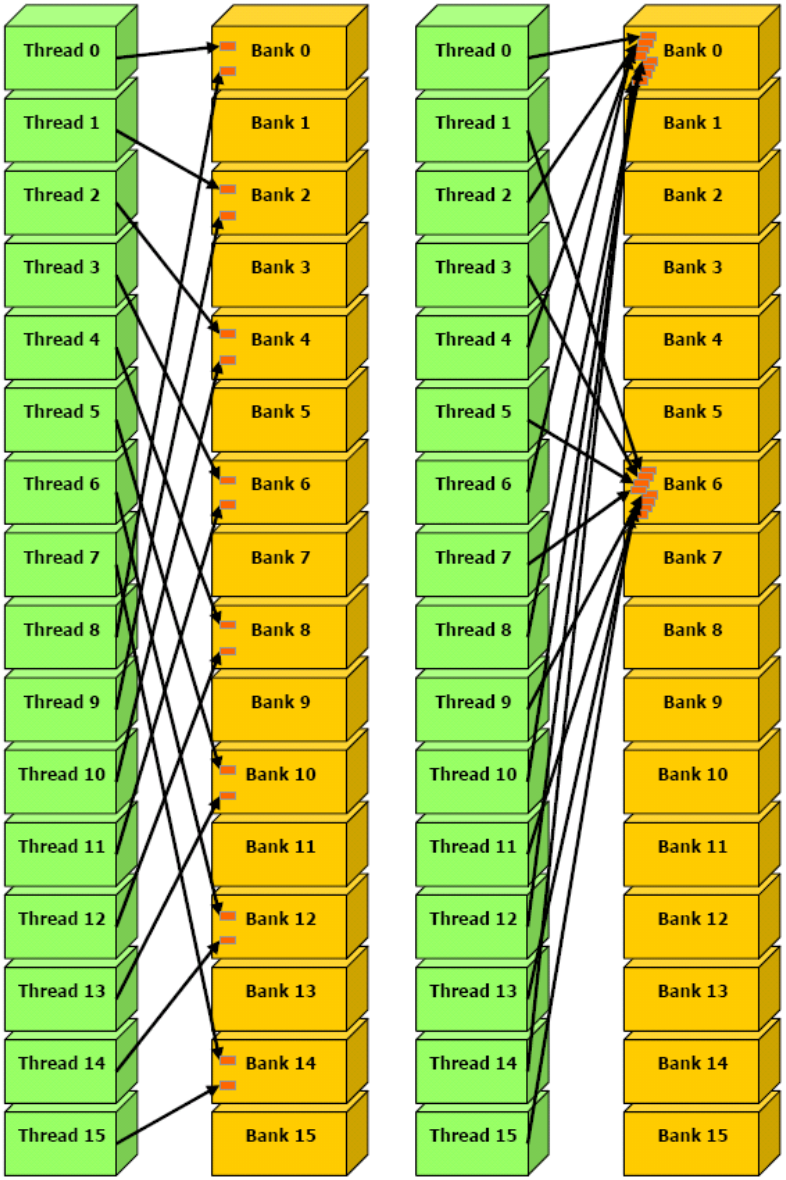


Примеры обращений без банк-конфликтов

На рис. в вариантах 1 и 2 банк-конфликтов не возникнет, так как все потоки обращаются к различным банкам.

В варианте 3 банк-конфликтов также удается избежать, так как Bank 3 будет выбран системой «широковещательным».

В четвёртом варианте банк конфликтов не возникнет лишь в том случае, если Bank 5 будет выбран «широковещательным».



На рис. пример с банк-конфликтами:

слева - второго порядка,
справа – восьмого порядка.

Банк-конфликт N-го порядка приводит к N физическим запросам к памяти, в то время как бесконфликтные обращение всегда проходят за единственный физический запрос.

Пример использования технологии

В качестве примера применения технологии **CUDA** рассмотрим решение задачи о вычислении корней большого числа квадратных уравнений.

Пусть необходимо решить $N \approx 107$ квадратных уравнений $ax_i^2 + b\,x_i^2 + c = 0$, для каждого из них заданы a_i , b_i , c_i , $i = 1,\ldots,N$.

Требуется найти действительные $Re(x_i^1)$ и $Re(x_i^2)$, а также мнимые $Im(x_i^1)$ *и* $Im(x_i^2)$ части корней уравнений.

Рассмотрим сначала однопроцессорную реализацию расчетного алгоритма:

```
void computeGold(eqdata* data)
{
    int i; float d;
    for(i = 0; i < BLOCK_SIZE*GRID_SIZE; i++)
     {
       d = -data[i].b*data[i].b + 4*data[i].a*data[i].c;
       data[i].re_x1 = -data[i].b;
       data[i].re_x2 = -data[i].b;
       if(d > 0)
                    {
                        data[i].re_x1 += sqrtf(d);
                        data[i].re_x2 -= sqrtf(d);
                    }
        else
                   {
                        data[i].im_x1 = sqrtf(-d);
                        data[i].im_x2 = -sqrtf(-d);
                    }
       data[i].re_x1 /= 2*data[i].a;
       data[i].re_x2 /= 2*data[i].a;
       data[i].im_x1 /= 2*data[i].a;
       data[i].im_x2 /= 2*data[i].a;
}  }
```

Где data – массив структур eqdata, состоящий из N элементов

```
struct eqdata
{
//Коэффициенты уравнения
  float a;
  float b;
  float c;
//Действительные части
корней уравнения
  float re_x1;
  float re_x2;
//Мнимые части корней
уравнения
  float im_x1;
  float im_x2;
};
```

Время выполнения этого расчета на AMD Athlon 64 X2 3800+ (2.0Гц) составляет **1038** мс.

Рассмотрим теперь реализацию расчетного алгоритма, использующую технологию **CUDA**:

```
__global__ void compute(eqdata* data)
{
   int tx = threadIdx.x;
   int bx = blockIdx.x;
   int i = tx+bx*BLOCK_SIZE;
   float d;

d = -data[tx].b*data[tx].b +
4*data[tx].a*data[tx].c;
   data[i].re_x1 = -data[tx].b;
   data[i].re_x2 = -data[tx].b;
   data[i].im_x1 = 0;
   data[i].im_x2 = 0;
```

```
if(d > 0)
 {
   data[i].re_x1 += sqrtf(d);
   data[i].re_x2 -= sqrtf(d);
 }
else
{
   data[i].im_x1 += sqrtf(-d);
   data[i].im_x2 -= sqrtf(-d);
}
   data[i].re_x1 /= (2*data[tx].a);
   data[i].re_x2 /= (2*data[tx].a);
   data[i].im_x1 /= (2*data[tx].a);
   data[i].im_x2 /= (2*data[tx].a);
}
```

В данной реализации, каждый поток решает отдельное квадратное уравнение, всего N потоков, они разбиты примерно на 16 000 блоков, по 512 потоков в каждом.

Время выполнения расчета с использованием видеоускорителя Nvidia GeForce 9600GT составляет **118** мс, то есть простой подход с использованием в расчете видеокарты, что называется «в лоб», дает ускорение около 10 раз.

Однако для такого простого примера, где данные для разных уравнений никак не связаны между собой эффективность простого применения рассматриваемой технологии оказалась весьма низкой.

В данном случае дело заключается в крайне неэффективном использовании возможностей GPU. Основной проблемой в данном примере является работа с памятью.

Нетрудно заметить, что в приведенной программе используется только глобальная (медленная) память GPU (__global__ void compute(eqdata* data)), в то время как использование быстрой разделяемой памяти могло бы значительно увеличить производительность.

Перепишем реализацию кода так, чтобы минимизировать работу с глобальной памятью:

```
__global__ void compute(eqdata* data)
{
__shared__ float a[BLOCK_SIZE];
__shared__ float b[BLOCK_SIZE];
__shared__ float c[BLOCK_SIZE];
__shared__ float reX1[BLOCK_SIZE];
__shared__ float reX2[BLOCK_SIZE];
__shared__ float imX1[BLOCK_SIZE];
__shared__ float imX2[BLOCK_SIZE];

int tx = threadIdx.x;
int bx = blockIdx.x;
int i = tx+bx*BLOCK_SIZE;

float d;

a[tx] = data[i].a;
b[tx] = data[i].b;
c[tx] = data[i].c;
```

```
__syncthreads();

d = -b[tx]*b[tx] + 4*a[tx]*c[tx];
reX1[tx] = -b[tx];
reX2[tx] = -b[tx];
imX1[tx] = 0;
imX2[tx] = 0;

if(d > 0)
{
   reX1[tx] += sqrtf(d);
   reX2[tx] -= sqrtf(d);
}
else
{
   imX1[tx] += sqrtf(-d);
   imX2[tx] -= sqrtf(-d);
}
   data[i].re_x1 = reX1[tx]/(2*a[tx]);
   data[i].re_x2 = reX2[tx]/(2*a[tx]);
   data[i].im_x1 = imX1[tx]/(2*a[tx]);
   data[i].im_x2 = imX2[tx]/(2*a[tx]);
}
```

Как видно из приведенного выше фрагмента, в данной версии работа с глобальной памятью происходит лишь в момент загрузки коэффициентов уравнения в быструю память, а также сохранения результатов.

Все остальные операции происходят с использованием быстрой разделяемой памяти.

Во втором варианте кода время выполнения расчета на видеокарте Nvidia GeForce 9600GT составляет **37** мс, что в 3 раза быстрее предыдущей версии. Общее ускорение по сравнению с однопроцессорным вариантом составило при этом около 30 раз. Но и это не предел.

Еще одним недостатком рассматриваемой реализации является то, что организация хранения данных, описанная выше (см. описание структуры eqdata), не позволяет осуществлять «связанные» запросы к памяти (см. выше), так как данные, читаемые различными потоками, расположены в памяти неупорядоченно.

Для оптимизации обращения к памяти изменим организацию хранения данных следующим образом:

```
struct eqdata
{
   float a[N];
   float b[N];
   float c[N];
   float re_x1[N];
   float re_x2[N];
   float im_x1[N];
   float im_x2[N];
};
```

То есть вместо использования массива структур, мы используем структуру, хранящую в себе массивы коэффициентов.

В этом случае «соседние» потоки будут запрашивать данные, располагающиеся в глобальной памяти подряд, что сделает все запросы «связанными», а это, в свою очередь, повысит скорость считывания.

Ниже приведена реализация алгоритма, в которой сделаны соответствующие изменения в организации хранения данных:

```
__global__ void compute(eqdata* data)
{
__shared__ float a[BLOCK_SIZE];
__shared__ float b[BLOCK_SIZE];
__shared__ float c[BLOCK_SIZE];
__shared__ float reX1[BLOCK_SIZE];
__shared__ float reX2[BLOCK_SIZE];
__shared__ float imX1[BLOCK_SIZE];
__shared__ float imX2[BLOCK_SIZE];

int tx = threadIdx.x;
int bx = blockIdx.x;
int i = tx+bx*BLOCK_SIZE;

float d;

a[tx] = data->a[i];
b[tx] = data->b[i];
c[tx] = data->c[i];
```

```
d = -b[tx]*b[tx] + 4*a[tx]*c[tx];
reX1[tx] = -b[tx];
reX2[tx] = -b[tx];
imX1[tx] = 0;
imX2[tx] = 0;

if(d > 0)
{
  reX1[tx] += sqrtf(d);
  reX2[tx] -= sqrtf(d);
}
else
{
   imX1[tx] += sqrtf(-d);
   imX2[tx] -= sqrtf(-d);
}

data->re_x1[i] = reX1[tx]/(2*a[tx]);
data->re_x2[i] = reX2[tx]/(2*a[tx]);
data->im_x1[i] = imX1[tx]/(2*a[tx]);
data->im_x2[i] = imX2[tx]/(2*a[tx]);
}
```

В таком варианте время выполнения расчета составляет около **9** мс, что в 4 раза быстрее предыдущей версии и в 115 раз быстрее однопроцессорного варианта.

Таким образом, «правильная» организация работы с различными видами памяти позволяет более чем на порядок повысить эффективность применения технологии CUDA даже в таком примитивном примере.

При получении коэффициентов ускорения, мы сравнивали различные CUDA-реализации с первоначальной реализацией на CPU. Стоит, однако, отметить, что правильная организация памяти с точки зрения вычислений GPU, не является таковой с точки зрения классических вычислений на CPU и, будучи применена к изначальной CPU-реализации, увеличивает время выполнения на CPU более чем в два раза. Это отличие ещё раз подчеркивает разницу подходов в программировании на GPU и на CPU.

Классические задачи

Умножение матриц

Рассмотрим задачу умножения плотных матриц. При нынешних объемах памяти на графическом процессоре можно умножать матрицы размером в миллионы элементов, и все данные будут размещены в видеоОЗУ.

Реально достигнутая производительность при этом будет достаточно высока: на GPU AMD HD 2900 это 100 ГФлоп/с, на GPU nVidia GeForce 8800GTX – 125 ГФлоп/с.

Следует заметить, что такая производительность достигается за счет использования блочных алгоритмов умножения матриц, а классические прямые реализации оказываются на порядок медленнее.

Преобразование Фурье

Преобразование Фурье, относится к классу алгоритмов, которые на GPU выполняются за время, пропорциональное N•ln(N), где N – это размер задачи.

Алгоритм реализуется при помощи многопроходной схемы, где на каждом проходе применяется ядро типа "бабочки".

И хотя вычислительная сложность подобного ядра относительно невысока, GPU nVidia удается достичь на нем неплохих результатов.

На первых этапах работы весь рабочий набор умещается в статическую память, что позволяет собрать их в один проход и получить порядка 40 ГФлоп/с на графической карте nVidia GeForce 8800GTX.

Обработка изображений

Обработка изображений была одной из первых задач, решаемых на GPU. Наиболее важными алгоритмами здесь являются фильтрация изображений и преобразование Фурье.

Фильтрация, благодаря относительно простой структуре, легко отображается на GPU. Однако коэффициент повторного использования данных значительно меньше, чем у умножения матриц, что и объясняет значительно меньшую эффективность графического процессора в решении данной задачи.

При размере ядра 3x3 производительность кода на GPU nVidia GeForce 8800GTX составляет около 20 ГФлоп/с. Часто используемая на практике сепарабельная фильтрация при помощи фильтра Гаусса показывает худшие результаты ввиду еще более низкого коэффициента повторного использования.

Литература:

1. Жуковский М.Е., Усков Р.В. О применении графических процессоров видеоускорителей в прикладных задачах // Препринты ИПМ им. М.В.Келдыша. 2010. № 2. 23 с. URL: **ttp://library.keldysh.ru/preprint.asp?id=2010-2**