Параллельное программирование и суперкомпьютерный кодизайн

Смирнов А.В. asmirnov@srcc.msu.ru

Раздел 7. Параллельное программирование с использованием графических процессоров

Раздел 7. План.

Параллельное программирование с использованием графических процессоров

- Теоретические основы и архитектура графических процессоров
- Основы программирования с использованием CUDA
- Ветвления при выполнении программы на CUDA
- Пример превосходства GPU, общая память и инструкции перетасовки
- OpenMP как способ использовать GPU

Раздел 7. План.

Параллельное программирование с использованием графических процессоров

- **Теоретические основы и архитектура графических процессоров**
- Основы программирования с использованием CUDA
- Ветвления при выполнении программы на CUDA
- Пример превосходства GPU, общая память и инструкции перетасовки
- OpenMP как способ использовать GPU

Графические процессоры общего назначения (GPU)

Почему вообще видеокарты?

Видеокарты это ведь про вывод изображения на экран?
Причем тут видеокарты и вычисления?

Почему вообще видеокарты?

Видеокарты это ведь про вывод изображения на экран? Причем тут видеокарты и вычисления?

- Структура видеокарты изначально рассчитана на одновременное проведение одинаковых операций над разными данными (рассчитать каждый пиксель на экране в игре)
- Сами по себе вычислители в видеокарте имеют куда меньшую скорость, чем ядра процессора, но их может быть много в одном устройстве
- Зачастую требуемые вычисления ложатся на схему SIMD – мы это видели и на примере других параллельных технологий

Виды видеокарт

- Графика с разделяемой памятью (Shared graphics, Shared Memory Architecture). Видеопамять в виде специализированных ячеек как таковая отсутствует; вместо этого под нужды видеоадаптера динамически выделяется область основной оперативной памяти компьютера. (интегрированные видеокарты, являющиеся частью северного моста). Преимущества данного решения — низкая цена и малое энергопотребление. Недостатки — невысокая производительность в 3D-графике и отрицательное влияние на пропускную способность памяти.

Виды видеокарт

- Графика с разделяемой памятью (Shared graphics, Shared Memory Architecture). Видеопамять в виде специализированных ячеек как таковая отсутствует; вместо этого под нужды видеоадаптера динамически выделяется область основной оперативной памяти компьютера. (интегрированные видеокарты, являющиеся частью северного моста). Преимущества данного решения — низкая цена и малое энергопотребление. Недостатки — невысокая производительность в 3D-графике и отрицательное влияние на пропускную способность памяти.
- Дискретная графика (Dedicated graphics). Есть своя видеопамять. Дискретная графика обеспечивает наивысшую производительность в трёхмерной графике. (выполнены в виде отдельных плат) Недостатки: более высокая цена и большее энергопотребление.

Виды видеокарт

- Дискретная графика (Dedicated graphics). Есть своя видеопамять. Дискретная графика обеспечивает наивысшую производительность в трёхмерной графике. (выполнены в виде отдельных плат) Недостатки: более высокая цена и большее энергопотребление.
- Гибридная дискретная графика (Hybrid graphics). Комбинация вышеназванных способов, ставшая возможной с появлением шины PCIExpress. Небольшой объём физически распаянной на плате видеопамяти, который может расширяться за счёт использования основной оперативной памяти(ОЗУ)

Виды видеокарт

- Гибридная дискретная графика (Hybrid graphics). Комбинация вышеназванных способов, ставшая возможной с появлением шины PCIExpress. Небольшой объём физически распаянной на плате видеопамяти, который может расширяться за счёт использования основной оперативной памяти(ОЗУ)
- Специализированные видеокарты. Применяются для высокопроизводительных вычислений. Для вывода изображения на монитор не используется.

Программирование на видеокартах

В компьютерных технологиях есть два существенных драйвера прогресса – порно и игры. Утверждают, что текущих скоростей интернет достиг благодаря распространению порно, современным же видеокартам и возможностью вычисления на них мы достигли благодаря играм.

Программирование на видеокартах

В компьютерных технологиях есть два существенных драйвера прогресса – порно и игры. Утверждают, что текущих скоростей интернет достиг благодаря распространению порно, современным же видеокартам и возможностью вычисления на них мы достигли благодаря играм.

- Изначально видеокарты не предоставляли возможности для программирования на них, а лишь давали базовые операции. Но этого оказалось недостаточно для разработчиков игр. Так появилась возможность программировать, то есть создавать программы, **исполняющиеся на видеокартах**

Варианты программирования на видеокартах

- Языки для программирования шейдеров – HLSL, GLSL, CG, – в основном, C-подобные языки, ориентированные на видеоигры

Варианты программирования на видеокартах

- Языки для программирования шейдеров – HLSL, GLSL, CG, – в основном, C-подобные языки, ориентированные на видеоигры

Шейдер (англ. shader «затеняющий») — компьютерная программа, предназначенная для исполнения процессорами видеокарты (GPU)

Изначально, видеокарты подразумевали три типа - вершинный (для эффектов отдельно взятых вершин; например, для создания эффекта волн, отрисовка травы и прочее), геометрический (для небольших примитивов; например, для создания силуэтов) и пиксельный (для фильтров определенной области изображения; например, туман). И, соответственно, было три типа специализированных процессоров в плате. Позже же от подобного деления отказались и все процессоры видеокарт стали универсальными (поддерживают все три типа).

Варианты программирования на видеокартах

- Языки для программирования шейдеров – HLSL, GLSL, CG, – в основном, C-подобные языки, ориентированные на видеоигры

Для исполнения программ с применением шейдеров (в первую очередь, игр) требуется наличие на компьютере установленной библиотеки DirectX подходящей версии или OpenGL

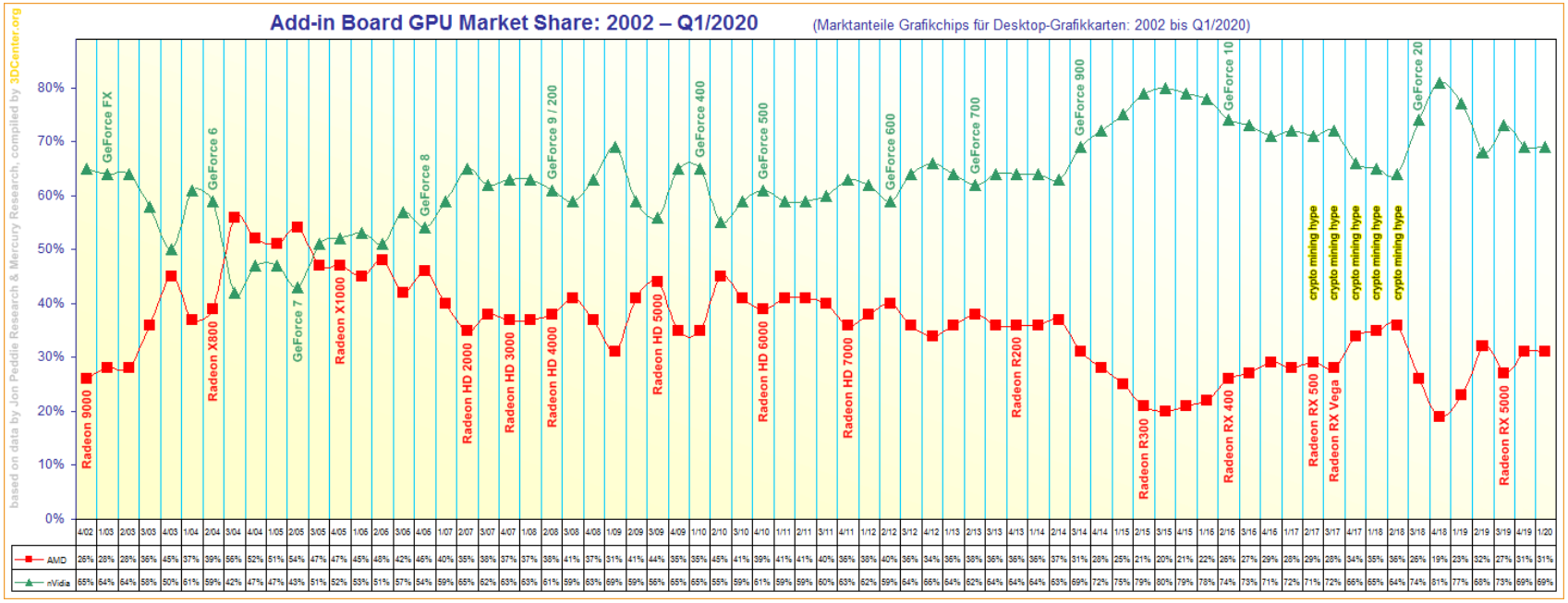
Варианты программирования на видеокартах

- Языки для программирования шейдеров – HLSL, GLSL, CG, – в основном, C-подобные языки, ориентированные на видеоигры
- Языки для вычислений (могут как быть связаны с 3D-графикой, так и совсем нет) – Cuda (Nvidia), Firestream (AMD), OpenCL, ...

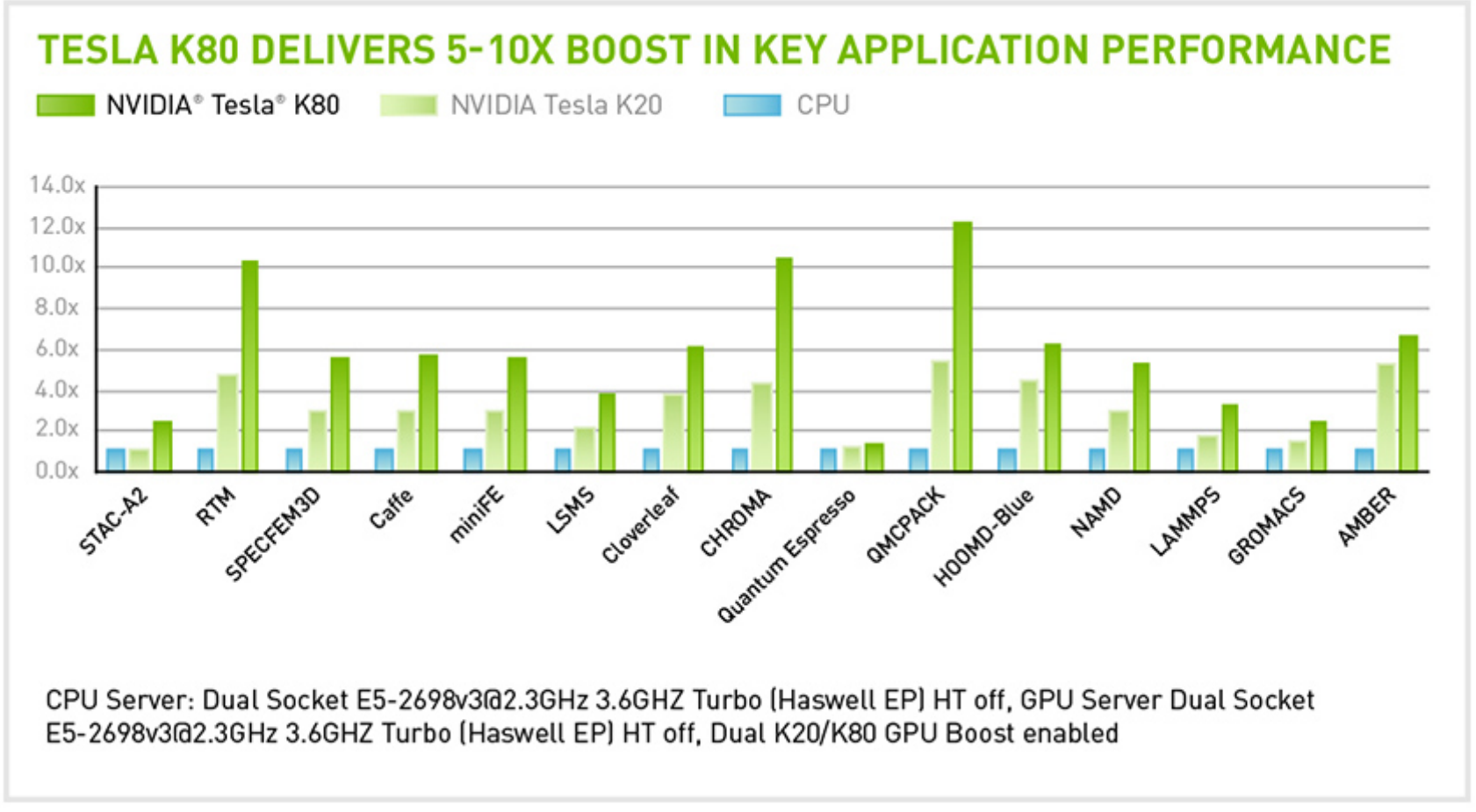
Варианты программирования на видеокартах

- Языки для программирования шейдеров – HLSL, GLSL, CG, – в основном, C-подобные языки, ориентированные на видеоигры
- Языки для вычислений (могут как быть связаны с 3D-графикой, так и совсем нет) – Cuda (Nvidia), Firestream (AMD), OpenCL, ...
- Относительно новый язык от AMD - HIP, имеющий CUDA-похожий синтаксис, претендующий на универсальность. Хотел добавить его в презентации, но возникли большие сложности с установкой...

Рынок видеокарт

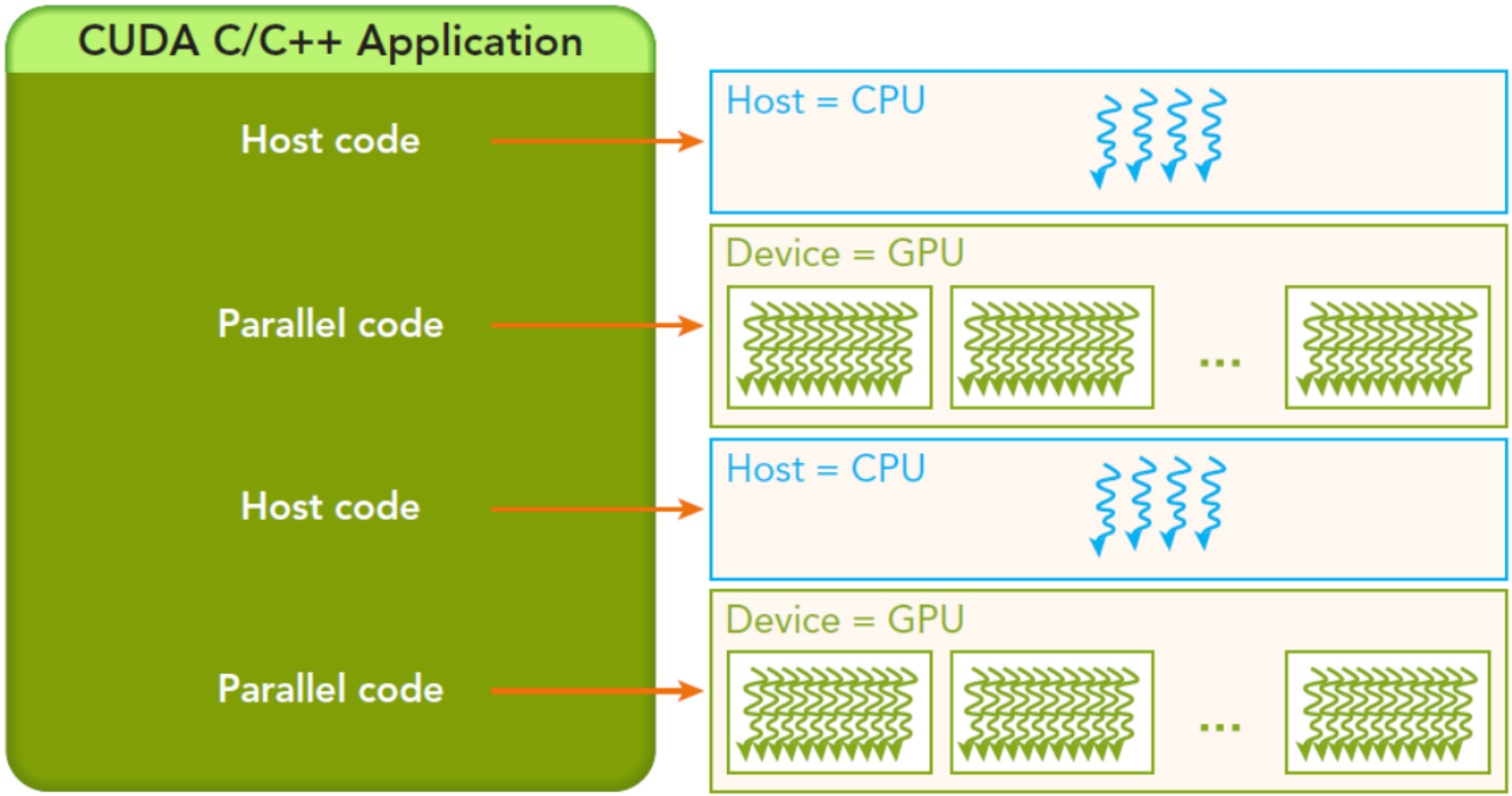


Причины использования GPU для вычислений



Сервер на 24 GPU

Модель выполненения программы на CUDA



CUDA – общие положения

- GPU – сопроцессор для CPU (хоста)

CUDA – общие положения

- GPU – сопроцессор для CPU (хоста)
- У GPU есть собственная память (device memory)

CUDA – общие положения

- GPU – сопроцессор для CPU (хоста)
- У GPU есть собственная память (device memory)
- GPU способен одновременно обрабатывать множество процессов (threads) данных одним и тем же алгоритмом

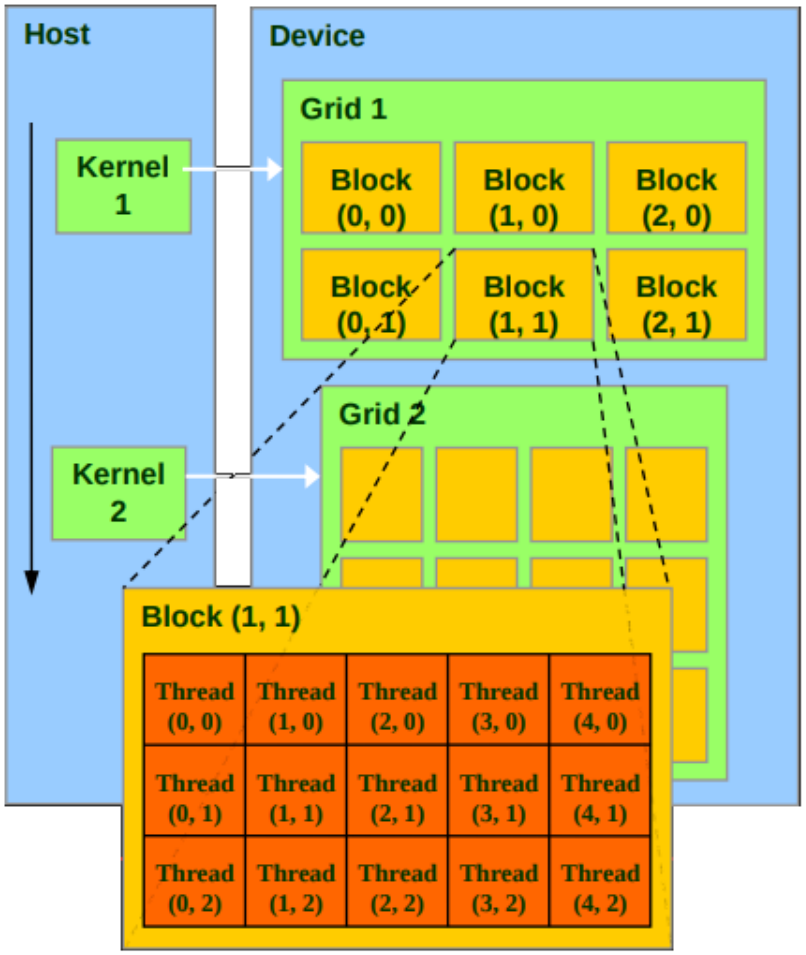
CUDA – общие положения

- GPU – сопроцессор для CPU (хоста)
- У GPU есть собственная память (device memory)
- GPU способен одновременно обрабатывать множество процессов (threads) данных одним и тем же алгоритмом
- Для осуществления осуществления расчётов при помощи GPU хост должен осуществить запуск вычислительного ядра (kernel), который определяет конфигурацию GPU в вычислениях и способ получения результатов (алгоритм)

CUDA – общие положения

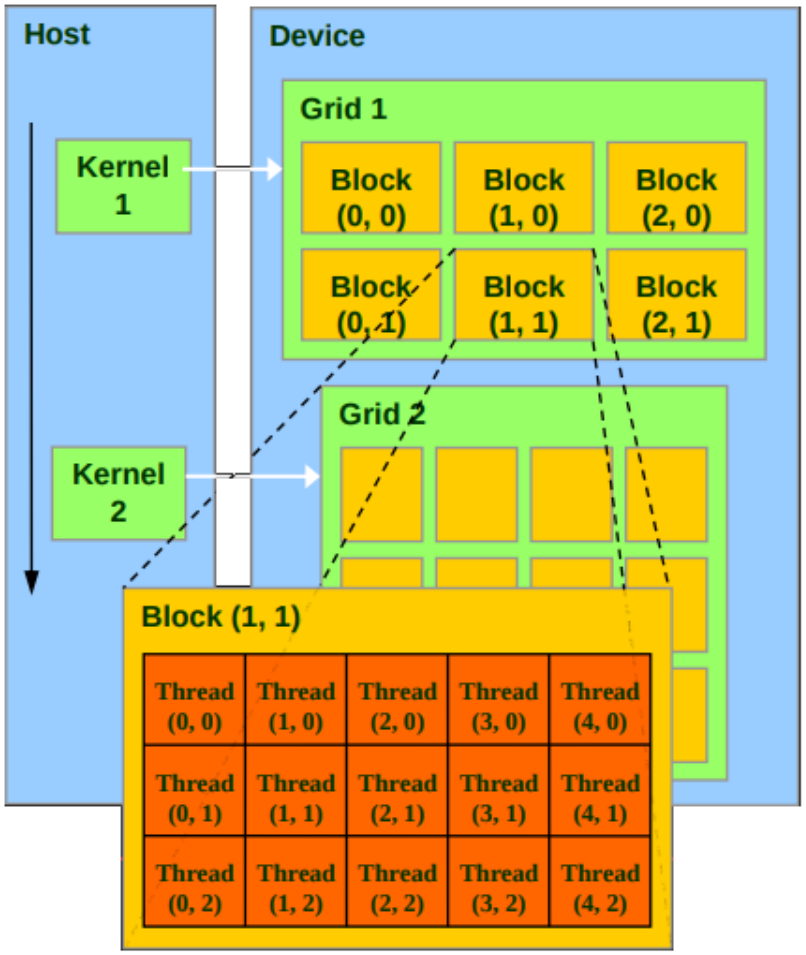
- GPU – сопроцессор для CPU (хоста)
- У GPU есть собственная память (device memory)
- GPU способен одновременно обрабатывать множество процессов (threads) данных одним и тем же алгоритмом
- Для осуществления осуществления расчётов при помощи GPU хост должен осуществить запуск вычислительного ядра (kernel), который определяет конфигурацию GPU в вычислениях и способ получения результатов (алгоритм)
- Процессы GPU (в отличие от CPU) очень просты и многочисленны (порядка 1000 для полной загрузки загрузки GPU)

CUDA – общие положения



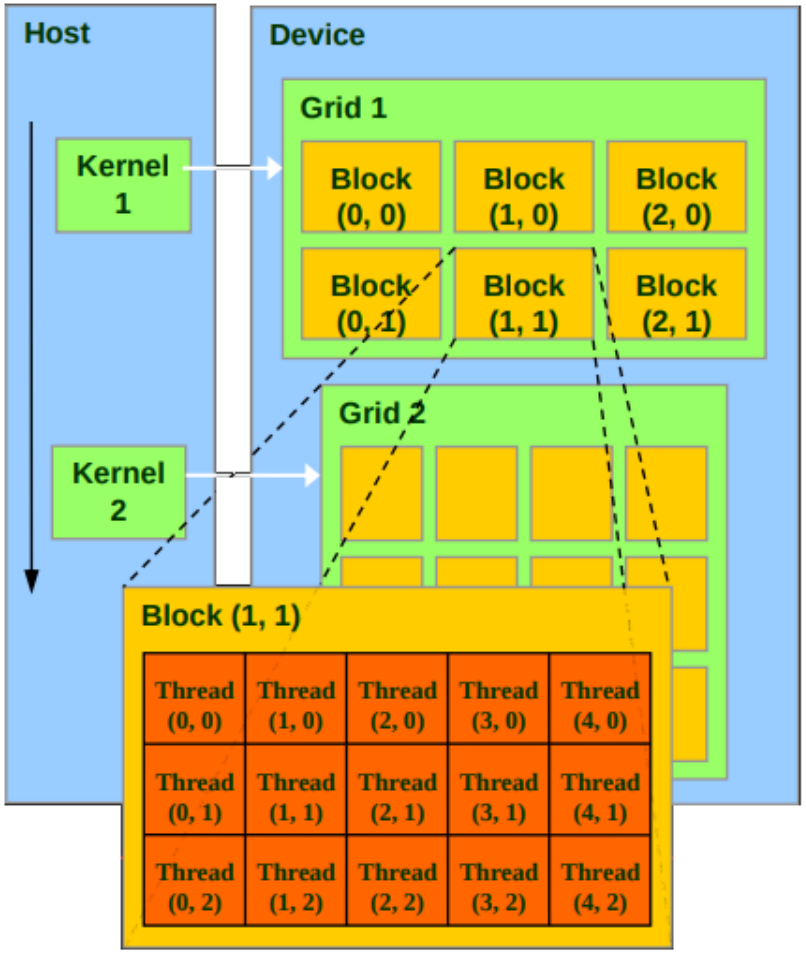
- Процессы объединяются в блоки (blocks), внутри которых они имеют общую память (shared memory) и синхронное исполнение
- Блоки объединяются в сетки (grids).

CUDA – общие положения



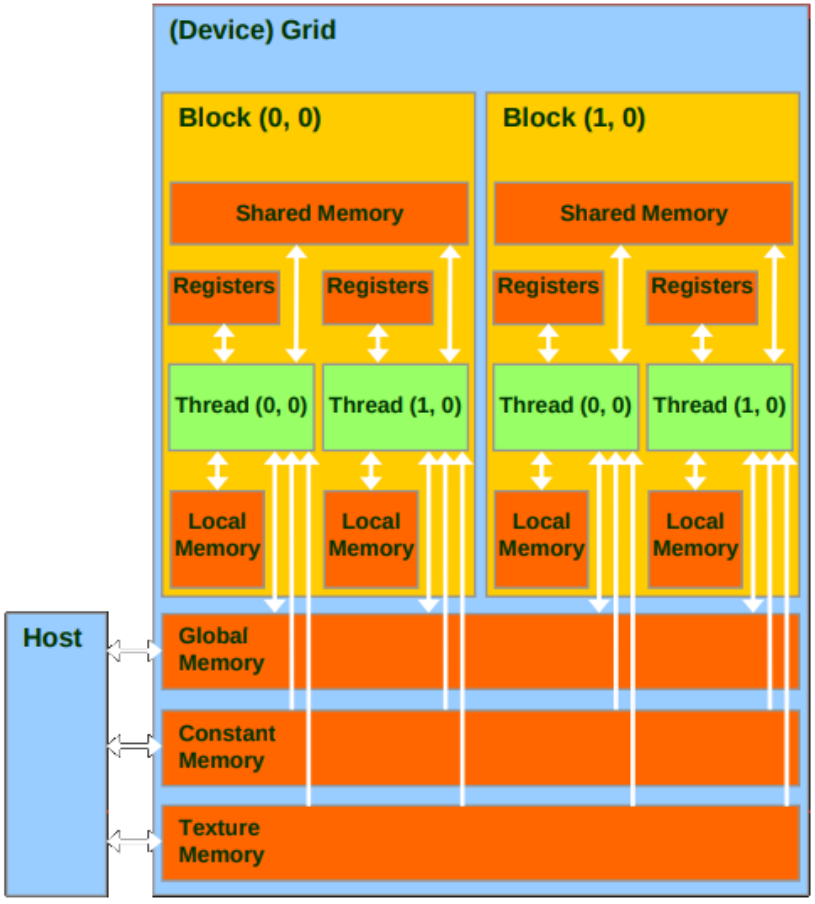
- Процессы объединяются в блоки (blocks), внутри которых они имеют общую память (shared memory) и синхронное исполнение
- Блоки объединяются в сетки (grids).
- Нет возможность предсказать очередность выполнения блоков в гриде.

CUDA – общие положения



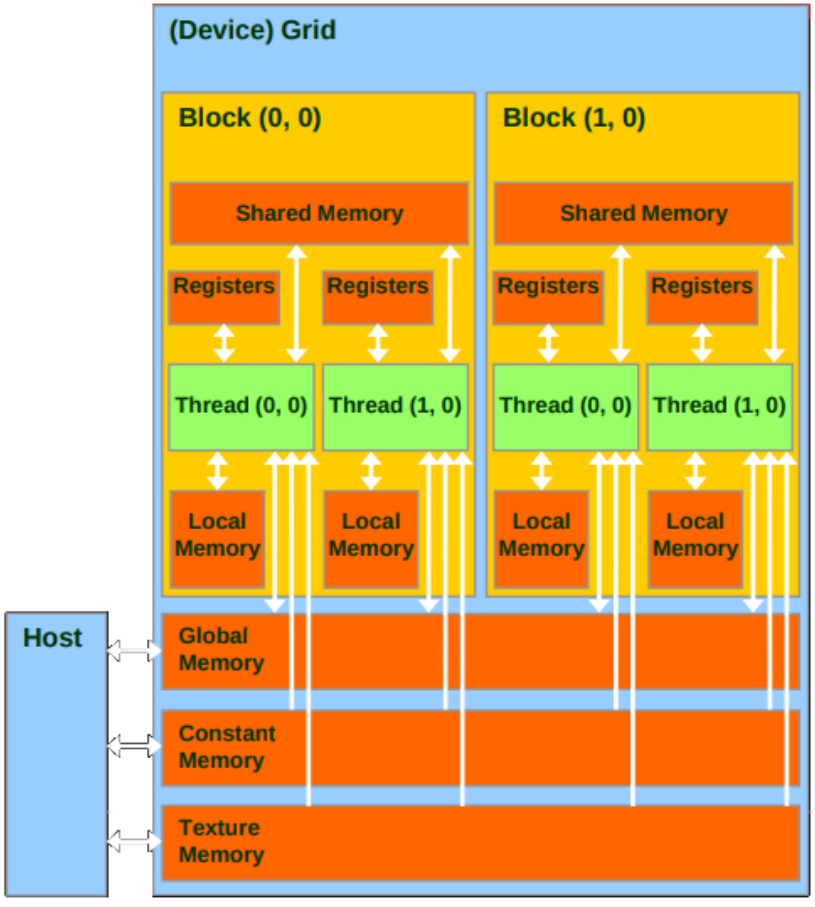
- Процессы объединяются в блоки (blocks), внутри которых они имеют общую память (shared memory) и синхронное исполнение
- Блоки объединяются в сетки (grids).
- Нет возможность предсказать очередность выполнения блоков в гриде.
- Между блоками нет и не может быть общей памяти.

CUDA – модель памяти



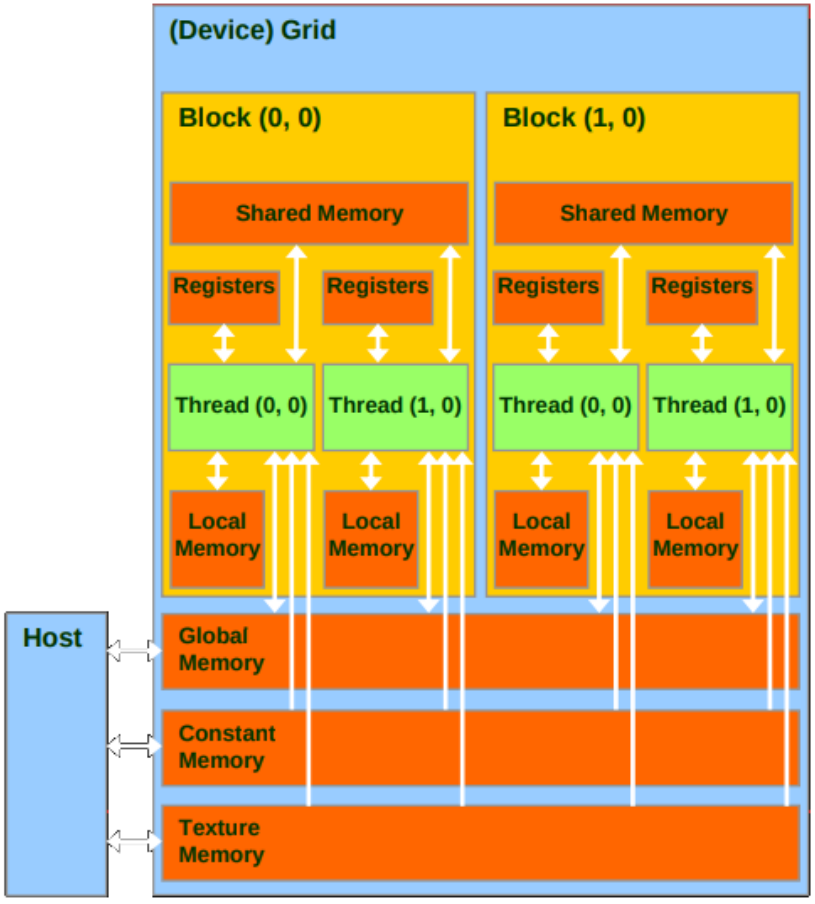
- Хост читает и пишет в глобальную память, констатную и текстурную

CUDA – модель памяти



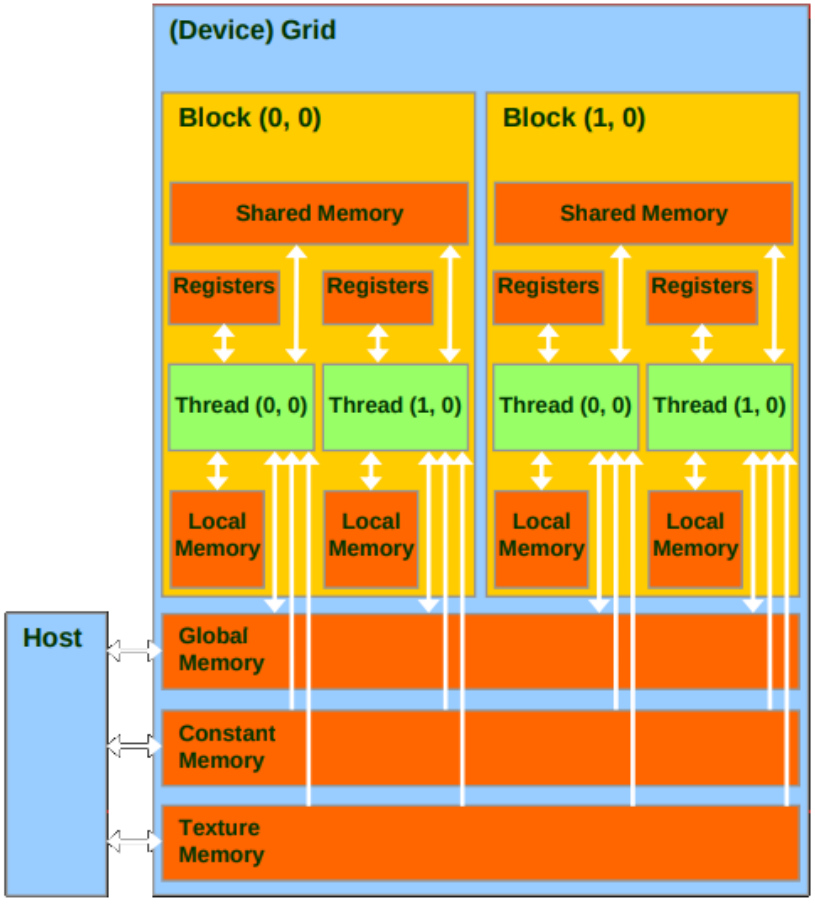
- Хост читает и пишет в глобальную память, константную и текстурную
- GPU может читать и писать в глобальную память, только читать константную и текстурную

CUDA – модель памяти



- Хост читает и пишет в глобальную память, констатную и текстурную
- GPU может читать и писать в глобальную память, только читать констатную и текстурную
- Каждый процесс читает и пишет в свою локальную память и свои регистры

CUDA – модель памяти



- Хост читает и пишет в глобальную память, констатную и текстурную
- GPU может читать и писать в глобальную память, только читать констатную и текстурную
- Каждый процесс читает и пишет в свою локальную память и свои регистры
- Каждый процесс внутри блока читает и пишет в общую память внутри блока

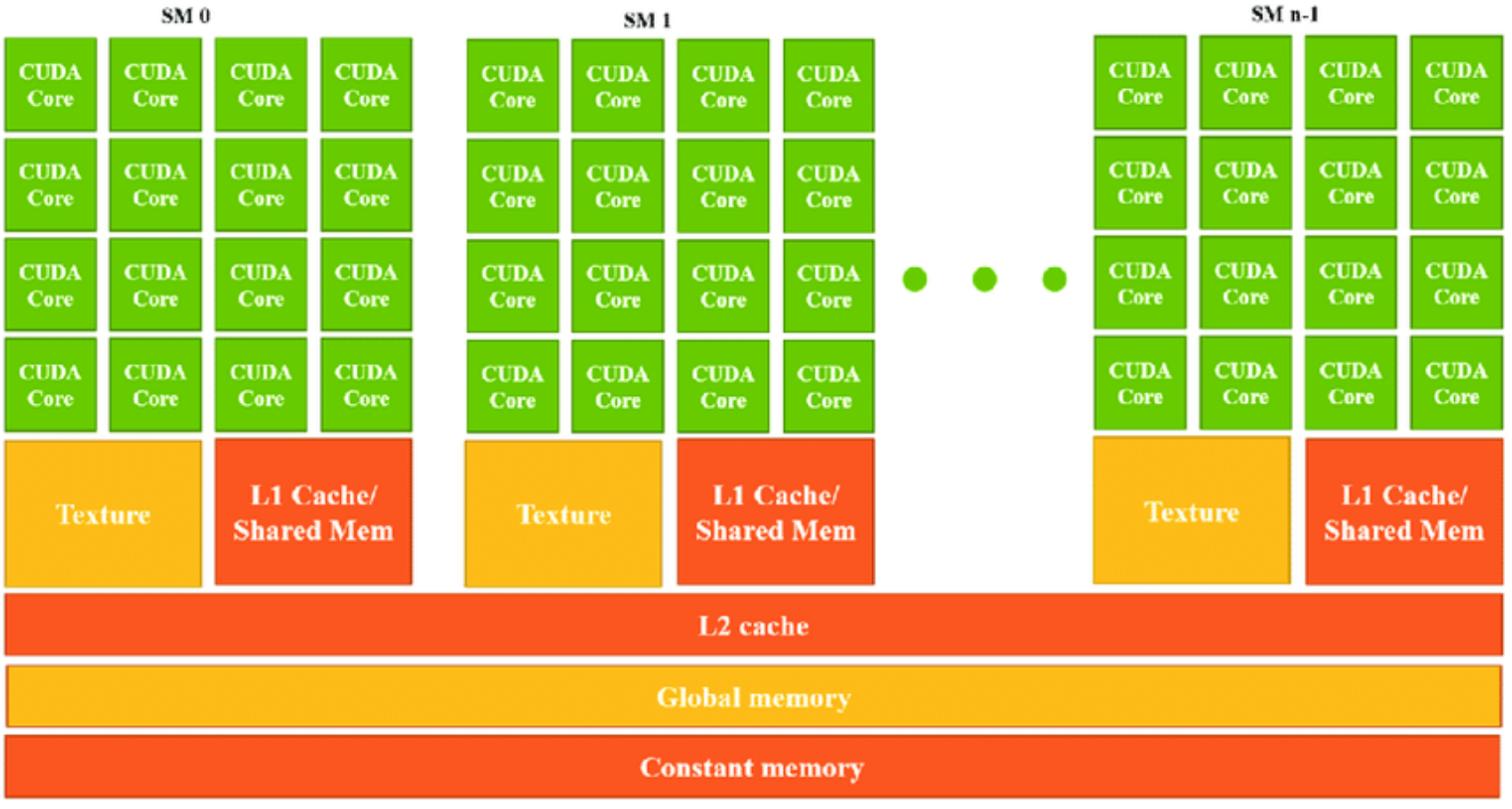
Аппаратная архитектура GPU

SM (Streaming Multiprocessor)

Основной вычислительный блок GPU, включающий:

- множество CUDA-ядер (FP32/INT32 ALU);
- Tensor Cores для ускорения матричных операций;
- планировщики потоков (warp schedulers);
- текстурную память;
- Shared Memory / L1 Cache.

Аппаратная архитектура GPU



Модель выполнения потоков

- Блок потоков распределяется на один из мультипроцессоров карты и выполняется там «до конца».
- Блок разбивается на «варпы» (Warps) по 32 потока.
- Варпы выполняются мультипроцессором в режиме SIMD – потоки в варпе выполняются одновременно, при этом на каждом шаге выполняется одна и та же инструкция разными потоками.

Модель выполнения потоков

- Блок потоков распределяется на один из мультипроцессоров карты и выполняется там «до конца».
- Блок разбивается на «варпы» (Warps) по 32 потока.
- Варпы выполняются мультипроцессором в режиме SIMD – потоки в варпе выполняются одновременно, при этом на каждом шаге выполняется одна и та же инструкция разными потоками.
- Например, если в блоке 80 потоков, то будет выделено 3 варпа. 16 потоков 3-го варпа будут «простаивать», потребляя ресурсы, но не выполняя вычислений.

Модель выполнения потоков

- Блок потоков распределяется на один из мультипроцессоров карты и выполняется там «до конца».
- Блок разбивается на «варпы» (Warps) по 32 потока.
- Варпы выполняются мультипроцессором в режиме SIMD – потоки в варпе выполняются одновременно, при этом на каждом шаге выполняется одна и та же инструкция разными потоками.
- Например, если в блоке 80 потоков, то будет выделено 3 варпа. 16 потоков 3-го варпа будут «простаивать», потребляя ресурсы, но не выполняя вычислений.
- Ветвление (if) внутри одного варпа приводит к тому, что сначала выполняется одна ветка, потом другая

Ключевые параметры GPU для вычислений

Ключевые параметры GPU для вычислений

- Количество мультипроцессоров (SM)! Именно этим числом в первую очередь отличаются друг от друга видеокарты одного поколения

Ключевые параметры GPU для вычислений

- Количество мультипроцессоров (SM)! Именно этим числом в первую очередь отличаются друг от друга видеокарты одного поколения
- Количество Cuda cores на один мультипроцессор (зависит от поколения)

Ключевые параметры GPU для вычислений

- Количество мультипроцессоров (SM)! Именно этим числом в первую очередь отличаются друг от друга видеокарты одного поколения
- Количество Cuda cores на один мультипроцессор (зависит от поколения)
- Тактовая частота ядер и памяти

Ключевые параметры GPU для вычислений

- Количество мультипроцессоров (SM)! Именно этим числом в первую очередь отличаются друг от друга видеокарты одного поколения
- Количество Cuda cores на один мультипроцессор (зависит от поколения)
- Тактовая частота ядер и памяти
- Объем L1 Cache, количество регистров на ядро (здесь в общем чем старше поколение, тем больше)

Аппаратная архитектура GPU

Ключевые параметры GPU для вычислений

- Количество мультипроцессоров (SM)! Именно этим числом в первую очередь отличаются друг от друга видеокарты одного поколения
- Количество Cuda cores на один мультипроцессор (зависит от поколения)
- Тактовая частота ядер и памяти
- Объем L1 Cache, количество регистров на ядро (здесь в общем чем старше поколение, тем больше)
- Объем общей памяти

Количество ядер на SM

- Поколение Fermi – 32 или 48
- Поколение Kepler – 192
- Поколение Maxwell – 128
- Поколение Pascal – 64 или 128
- Поколение Volta и Turing – 64
- Поколение Ampere – 64 или 128

Количество ядер на SM

- Поколение Fermi – 32 или 48
- Поколение Kepler – 192
- Поколение Maxwell – 128
- Поколение Pascal – 64 или 128
- Поколение Volta и Turing – 64
- Поколение Ampere – 64 или 128

Иногда различают отдельно ядра (ALU) для операция FP32 и INT32. Если они реализованы различними блоками в железе (Volta, Ampere), то при смешанной работе повышается производительность. Плюс еще бывают тензорные ядря для произведения матриц 4 на 4

GTX 1080 (Pascal)

SMs	20
cores	2560
base clock	1607 GHz
Boost clock	1733 GHz

Вопросы?

Раздел 7. План.

Параллельное программирование с использованием графических процессоров

- Теоретические основы и архитектура графических процессоров
- **Основы программирования с использованием CUDA**
- Ветвления при выполнении программы на CUDA
- Пример превосходства GPU, общая память и инструкции перетасовки
- OpenMP как способ использовать GPU

Установка Cuda

Внимание! Попытка установить драйверы Cuda может привести к неработоспособности вашей операционной системы

Установка Cuda

Внимание! Попытка установить драйверы Cuda может привести к неработоспособности вашей операционной системы

Методы установки:

- `sudo apt install nvidia-cuda-toolkit` (это не драйвер, а только компилятор! безопасно, но не даст возможность запускать программу)

Установка Cuda

Внимание! Попытка установить драйверы Cuda может привести к неработоспособности вашей операционной системы

Методы установки:

- `sudo apt install nvidia-cuda-toolkit` (это не драйвер, а только компилятор! безопасно, но не даст возможность запускать программу)
- Скачать с сайта Nvidia и установить пакет с компилятором и драйверами

Установка Cuda

Внимание! Попытка установить драйверы Cuda может привести к неработоспособности вашей операционной системы

Методы установки:

- `sudo apt install nvidia-cuda-toolkit` (это не драйвер, а только компилятор! безопасно, но не даст возможность запускать программу)
- Скачать с сайта Nvidia и установить пакет с компилятором и драйверами

Дальнейшие действия: выберите Nvidia драйвер в разделе "дополнительные драйверы"

Установка Cuda

- Далее можно выбирать разные версии драйвера, по-разному устанавливать, гуглить и пробовать разные советы из интернета.
- Если ничего не помогает, используйте универсальный инструмент программиста и системного администратора

Установка Cuda

Компиляция программ с использованием Cuda

- Файлы, содержащие cuda-код, должны иметь расширение .cu

Компиляция программ с использованием Cuda

- Файлы, содержащие cuda-код, должны иметь расширение .cu
- Для компиляции используется специальный компилятор nvcc

Компиляция программ с использованием Cuda

- Файлы, содержащие cuda-код, должны иметь расширение .cu
- Для компиляции используется специальный компилятор nvcc
- При сборке многомодульных программ занесите требуемую часть кода в .cu файл и компилируйте этот файл с опцией `nvcc -c`. Остальные файлы компилируйте, а также линкуйте g++ или иным привычным вам компилятором. Обратите внимание, что nvcc поддерживает не все флаги и опции, известные gcc, например, вы, возможно, не сможете внутри использовать поздние стандарты c++.

Компиляция программ с использованием Cuda

- Файлы, содержащие cuda-код, должны иметь расширение .cu
- Для компиляции используется специальный компилятор nvcc
- При сборке многомодульных программ занесите требуемую часть кода в .cu файл и компилируйте этот файл с опцией `nvcc -c`. Остальные файлы компилируйте, а также линкуйте g++ или иным привычным вам компилятором. Обратите внимание, что nvcc поддерживает не все флаги и опции, известные gcc, например, вы, возможно, не сможете внутри использовать поздние стандарты c++.
- Для наших тестовых целей мы будем использовать программу из одного файла и сразу компилировать и линковать ее с nvcc

Простейшая программ с использованием Cuda

```
#include <stdio.h>
#include <cuda.h>

__global__ void helloFromGPU(/* args */) {
    printf("Hello world from GPU!\n");
}

int main(void){
    printf("Hello World from CPU!\n\n");
    helloFromGPU <<<1, 10>>>();
    cudaDeviceReset();
    return 0;
}
```

Как это работает?

- `__global__` – спецификатор, означающий, что далее пойдет функция выполняемая на устройства (видеокарта), но при этом которую можно вызвать с хоста (процессор). Возможен также спецификатор `__device__`, означающий, что это функция, существующая только на видеокарте, и вызывать ее можно только с видеокарты

Как это работает?

- `__global__` – спецификатор, означающий, что далее пойдет функция выполняемая на устройства (видеокарта), но при этом которую можно вызвать с хоста (процессор). Возможен также спецификатор `__device__`, означающий, что это функция, существующая только на видеокарте, и вызывать ее можно только с видеокарты
- `helloFromGPU < < <1, 10> > >()` – вызов ядра (kernel). Вызов с хоста, выполнение на карте. Первый параметр – число блоков, второй – число потоков в блоке. В данном случае создается один блок из 10 потоков.

Как это работает?

- `__global__` – спецификатор, означающий, что далее пойдет функция выполняемая на устройства (видеокарта), но при этом которую можно вызвать с хоста (процессор). Возможен также спецификатор `__device__`, означающий, что это функция, существующая только на видеокарте, и вызывать ее можно только с видеокарты
- `helloFromGPU < < <1, 10> > >()` – вызов ядра (kernel). Вызов с хоста, выполнение на карте. Первый параметр – число блоков, второй – число потоков в блоке. В данном случае создается один блок из 10 потоков.
- `cudaDeviceSynchronize()` явно дожидается выполнения задач на устройстве. Необходимо, поскольку вызов функций устройства является асинхронным.

Простейшая программ с использованием Cuda

```
#include <stdio.h>
#include <cuda.h>

__global__ void helloFromGPU(/* args */) {
    printf("Hello world from GPU!\n");
}

int main(void){
    printf("Hello World from CPU!\n\n");
    helloFromGPU <<<1, 10>>>();
    cudaDeviceReset();
    return 0;
}
```

Результат выполнения

```
Hello World from CPU!

Hello world from GPU!
Hello world from GPU!
Hello world from GPU!
Hello world from GPU!
Hello world from GPU!
Hello world from GPU!
Hello world from GPU!
Hello world from GPU!
Hello world from GPU!
Hello world from GPU!
```

Определение числа блоков и потоков

```
__global__ void helloFromGPU() {
   printf("threadIdx:(%d, %d, %d) "
   "blockIdx:(%d, %d, %d) "
   "blockDim:(%d, %d, %d) "
   "gridDim:(%d, %d, %d)\n",
      threadIdx.x, threadIdx.y, threadIdx.z,
      blockIdx.x, blockIdx.y, blockIdx.z,
      blockDim.x, blockDim.y, blockDim.z,
      gridDim.x, gridDim.y, gridDim.z);
}
int main(void){
      printf("Hello World from CPU!\n\n");
      helloFromGPU <<<1, 10>>>();
      cudaDeviceSynchronize();
}
```

Результат выполнения

```
Hello World from CPU!

threadIdx:(0, 0, 0) blockIdx:(0, 0, 0)
    blockDim:(10, 1, 1) gridDim:(1, 1, 1)
threadIdx:(1, 0, 0) blockIdx:(0, 0, 0)
    blockDim:(10, 1, 1) gridDim:(1, 1, 1)
threadIdx:(2, 0, 0) blockIdx:(0, 0, 0)
    blockDim:(10, 1, 1) gridDim:(1, 1, 1)
threadIdx:(3, 0, 0) blockIdx:(0, 0, 0)
    blockDim:(10, 1, 1) gridDim:(1, 1, 1)
threadIdx:(4, 0, 0) blockIdx:(0, 0, 0)
    blockDim:(10, 1, 1) gridDim:(1, 1, 1)
...
```

Вычисление размерности блоков

```
int main() {
    printf("Hello World from CPU!\n\n");
    constexpr int nt = 10;
    constexpr int blsize = 3;
    dim3 block(blsize);
    dim3 grid((nt + blsize - 1)/blsize);
    helloFromGPU <<<grid, block>>>();
    cudaDeviceReset();
    return 0;
}
```

Результат выполнения

```
Hello World from CPU!

threadIdx:(0, 0, 0) blockIdx:(1, 0, 0)
    blockDim:(3, 1, 1) gridDim:(4, 1, 1)
threadIdx:(1, 0, 0) blockIdx:(1, 0, 0)
    blockDim:(3, 1, 1) gridDim:(4, 1, 1)
threadIdx:(2, 0, 0) blockIdx:(1, 0, 0)
    blockDim:(3, 1, 1) gridDim:(4, 1, 1)
threadIdx:(0, 0, 0) blockIdx:(3, 0, 0)
    blockDim:(3, 1, 1) gridDim:(4, 1, 1)
```

Результат выполнения

```
threadIdx:(1, 0, 0) blockIdx:(3, 0, 0)
    blockDim:(3, 1, 1) gridDim:(4, 1, 1)
threadIdx:(2, 0, 0) blockIdx:(3, 0, 0)
    blockDim:(3, 1, 1) gridDim:(4, 1, 1)
threadIdx:(0, 0, 0) blockIdx:(0, 0, 0)
    blockDim:(3, 1, 1) gridDim:(4, 1, 1)
threadIdx:(1, 0, 0) blockIdx:(0, 0, 0)
    blockDim:(3, 1, 1) gridDim:(4, 1, 1)
threadIdx:(2, 0, 0) blockIdx:(0, 0, 0)
    blockDim:(3, 1, 1) gridDim:(4, 1, 1)
threadIdx:(0, 0, 0) blockIdx:(2, 0, 0)
    blockDim:(3, 1, 1) gridDim:(4, 1, 1)
threadIdx:(1, 0, 0) blockIdx:(2, 0, 0)
    blockDim:(3, 1, 1) gridDim:(4, 1, 1)
threadIdx:(2, 0, 0) blockIdx:(2, 0, 0)
    blockDim:(3, 1, 1) gridDim:(4, 1, 1)
```

Суммирование векторов на CPU

```
#include <chrono>
int main() {
  constexpr size_t n = 32 * 1024 * 1024;
  // vector declaration and generation here
  auto start=std::chrono::steady_clock::now();
  #pragma omp parallel for num_threads(4)
  for (size_t i = 0; i != n; ++i) {
    c[i] = a[i] + b[i];
  }
  auto end = std::chrono::steady_clock::now();
  int time = std::chrono::duration_cast<
    std::chrono::milliseconds
  >(diff).count();
  std::cout << time << " ms "<< std::endl;
}
```

Суммирование векторов на GPU

```
__global__ void sumArraysOnGPU(double *A,
    double *B, double *C, const int N) {
    int i = blockIdx.x * blockDim.x +
        threadIdx.x;
    if (i < N) C[i] = A[i] + B[i];
}

int main() {
  ...
  double *d_A, *d_B, *d_C;
  size_t nBytes = n * sizeof(double);
  cudaMalloc(static_cast<double**>(&d_A),
    nBytes);
  cudaMalloc(static_cast<double**>(&d_B),
    nBytes);
  cudaMalloc(static_cast<double**>(&d_C),
```

Суммирование векторов на GPU

```
    nBytes);
cudaMemcpy(d_A, a.data(), nBytes,
    cudaMemcpyHostToDevice);
cudaMemcpy(d_B, b.data(), nBytes,
    cudaMemcpyHostToDevice);
constexpr size_t blsize = 1024;
dim3 block(blsize);
dim3 grid((n + blsize - 1)/blsize);
sumArraysOnGPU <<<grid, block>>>
    (d_A, d_B, d_C, n);
cudaDeviceSynchronize();
cudaMemcpy(c.data(), d_C, nBytes,
    cudaMemcpyDeviceToHost);
cudaFree(d_A); cudaFree(d_B);
cudaFree(d_C);
...
```

Сравниваем время

- CPU: 47 ms

Сравниваем время

- CPU: 47 ms
- GPU (GeForce GTX 1650 Mobile): 264 ms. Как так?

Сравниваем время

- CPU: 47 ms
- GPU (GeForce GTX 1650 Mobile): 264 ms. Как так?
- Довольно много времени уходит на первую инициализацию карточки, это не нужно учитывать. До старта замера времени вызовем какой-нибудь метод, например `cudaDeviceSynchronize()`

Сравниваем время

- CPU: 47 ms

Сравниваем время

- CPU: 47 ms
- GPU (GeForce GTX 1650 Mobile): 264 -> 162 ms. Что еще-то?

Сравниваем время

- CPU: 47 ms
- GPU (GeForce GTX 1650 Mobile): 264 -> 162 ms. Что еще-то?
- GPU, замеряем только сложение: 7ms.

Сравниваем время

- CPU: 47 ms
- GPU (GeForce GTX 1650 Mobile): 264 -> 162 ms. Что еще-то?
- GPU, замеряем только сложение: 7ms.
- Вывод: если данные хранятся в памяти процессора, то применимость графической карточки еще более зависит от интенсивности работы с памятью со стороны процессора.

Вопросы?

Раздел 7. План.

Параллельное программирование с использованием графических процессоров

- Теоретические основы и архитектура графических процессоров
- Основы программирования с использованием CUDA
- **Ветвления при выполнении программы на CUDA**
- Пример превосходства GPU, общая память и инструкции перетасовки
- OpenMP как способ использовать GPU

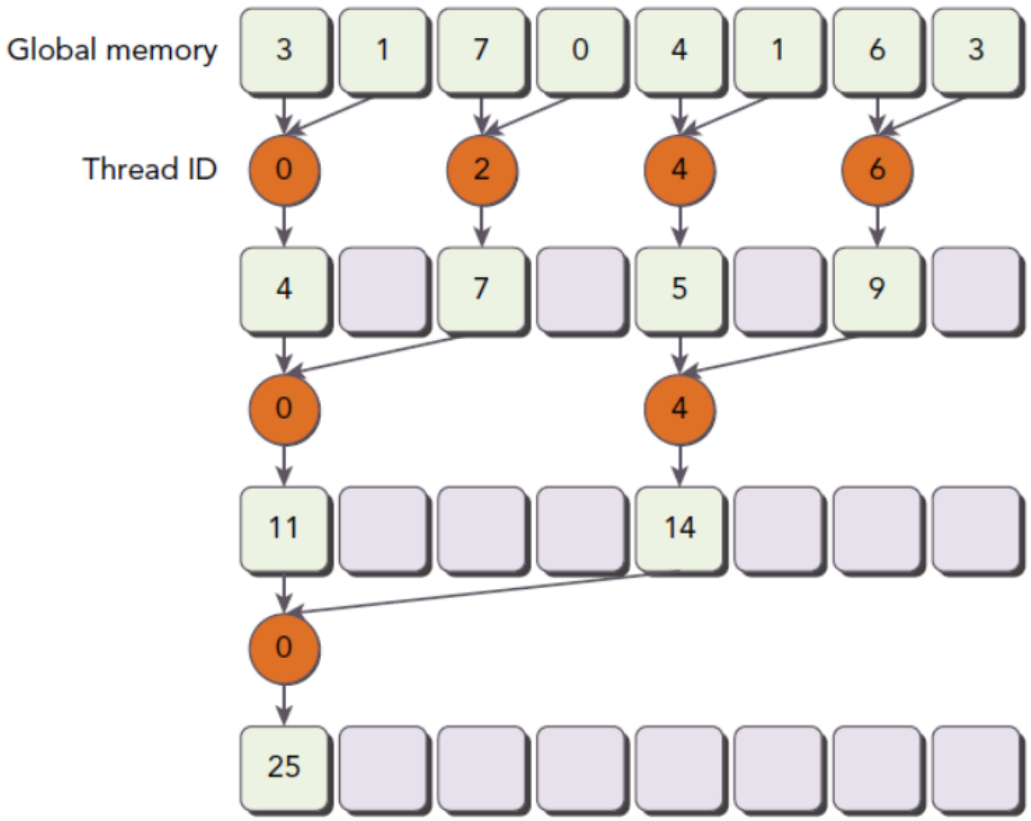
Пример – суммирование вектора

```
double sum_vector(std::vector<double>& a,
      size_t start, size_t end) {
      double res = 0.;
      for (size_t i = start; i!=end; i++) {
            res = res + a[i];
      }
      return res;
}
```

Пример – суммирование вектора

```
int main() {
  constexpr size_t n = 256 * 1024 * 1024;
  std::vector<double> a(n);
  // fill with random here
  auto start =
       std::chrono::steady_clock::now();
  double res = sum_vector(a, 0, n);
  auto end =
       std::chrono::steady_clock::now();
  auto diff = end-start;
  int time = std::chrono::duration_cast<
    std::chrono::milliseconds>(diff).count();
  std::cout << time << " ms" << std::endl;
}
```

Параллелизация суммирования



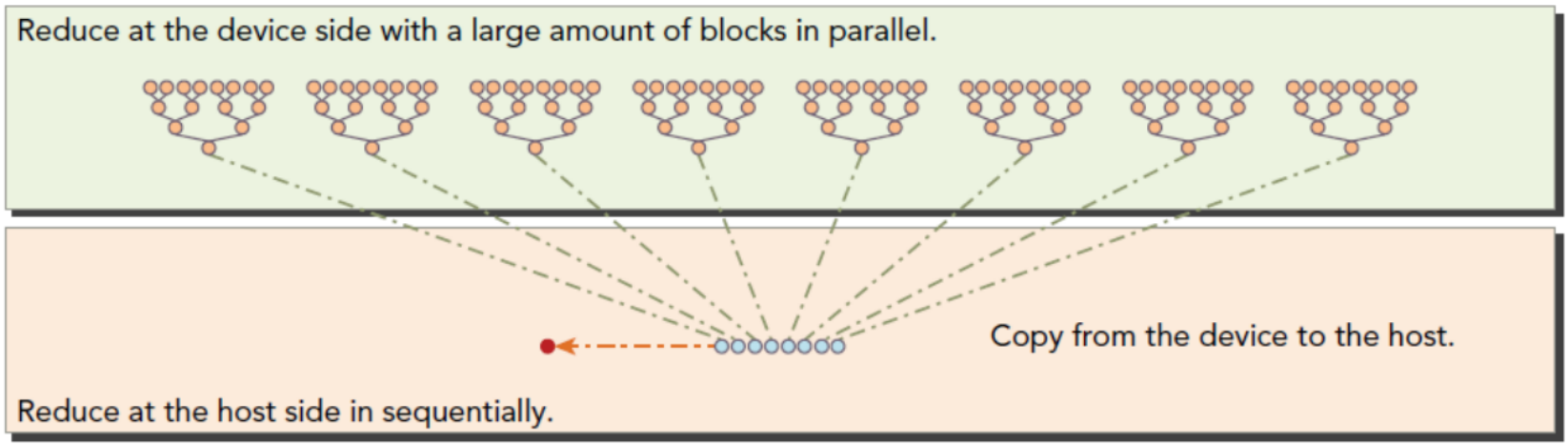
Параллелизация суммирования

```
__global__ void reduce(double *g_idata,
    double *g_odata, unsigned int n) {
    unsigned int tid = threadIdx.x;
    if (tid + blockIdx.x * blockDim.x >= n)
        return;
    double *idata = g_idata + blockIdx.x
        * blockDim.x;
    for (int stride = 1; stride < blockDim.x;
        stride *= 2) {
        if ((tid % (2 * stride)) == 0) {
            idata[tid] += idata[tid + stride];
        }
        __syncthreads();
    }
    if (tid==0) g_odata[blockIdx.x]=idata[0];
}
```

Параллелизация суммирования

```
auto start = std::chrono::steady_clock::now();
double *d_A, *d_C;
size_t nBytes = n * sizeof(double);
cudaMalloc(
    static_cast<double**>(&d_A), nBytes);
cudaMemcpy(d_A, a.data(), nBytes,
    cudaMemcpyHostToDevice);
constexpr size_t blsize = 1024;
dim3 block(blsize);
dim3 grid((n + blsize - 1)/blsize);
cudaMalloc(static_cast<double**>(&d_C),
    grid.x * sizeof(double));
cudaDeviceSynchronize();
auto start2 =std::chrono::steady_clock::now();
reduce <<<grid, block>>>(d_A, d_C, n);
cudaDeviceSynchronize();
```

Параллелизация суммирования



Параллелизация суммирования

```
auto end2 = std::chrono::steady_clock::now();
cudaMemcpy(c.data(), d_C, grid.x *
    sizeof(double), cudaMemcpyDeviceToHost);
cudaFree(d_A); cudaFree(d_C);
double res = 0.;
for (size_t i = 0; i!=grid.x; ++i) {
    res += c[i];
}
auto end = std::chrono::steady_clock::now();
auto diff2 = end2-start2;
int time2 = std::chrono::duration_cast
  <std::chrono::milliseconds>(diff2).count();
auto diff = end-start;
int time = std::chrono::duration_cast
  <std::chrono::milliseconds>(diff).count();
std::cout<< time << "|" << time2 << " ms \n";
```

Параллелизация суммирования

Сравним время исполнения

Параллелизация суммирования

Сравним время исполнения

- 494ms на CPU

Параллелизация суммирования

Сравним время исполнения

- 494ms на CPU
- 449ms на GPU (из них 79 ms) на суммирование
- Слегка обогнали CPU, но на самом деле программа написана плохо

Параллелизация суммирования

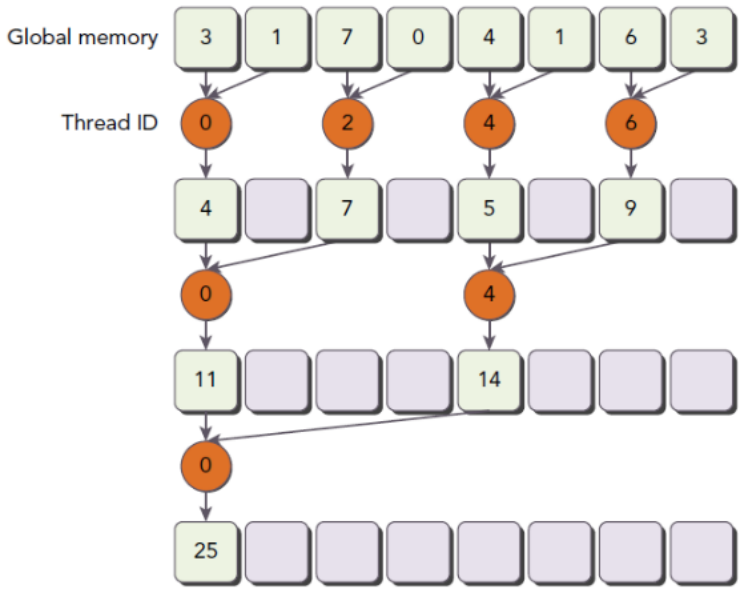
Расхождение потоков

- Условный оператор `if ((tid % (2 * stride)) == 0)` приводит к тому, что половина потоков в варпе не выполняется – это приводит к плохой утилизации ресурсов.

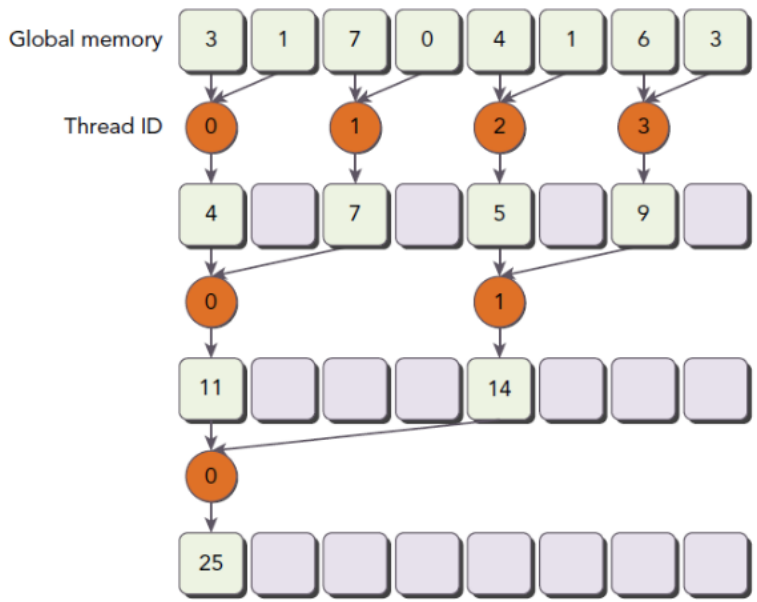
Параллелизация суммирования

Более эффективная схема

Старый вариант



Новый вариант



Параллелизация суммирования

Сравнение схем

Параллелизация суммирования

```
__global__ void reduce (double *g_idata,
    double *g_odata, unsigned int n) {
    unsigned int tid = threadIdx.x;
    if (tid + blockIdx.x * blockDim.x >= n)
        return;
    double *idata = g_idata + blockIdx.x
        *blockDim.x;
    for (int stride = 1; stride < blockDim.x;
        stride *= 2) {
        int index = 2 * stride * tid;
        if (index < blockDim.x)
            idata[index]+=idata[index+stride];
        __syncthreads();
    }
    if (tid==0) g_odata[blockIdx.x]=idata[0];
}
```

Параллелизация суммирования

Сравним время исполнения

- 494ms на CPU
- Было 449ms на GPU (из них 79 ms на суммирование)

Параллелизация суммирования

Сравним время исполнения

- 494ms на CPU
- Было 449ms на GPU (из них 79 ms на суммирование)
- Стало 414ms на GPU (из них 44 ms на суммирование)
- Еще больше обогнали одноядерный CPU, но все равно все проблемы связаны с памятью

Параллелизация суммирования

Сравним время исполнения

- 494ms на CPU
- Было 449ms на GPU (из них 79 ms на суммирование)
- Стало 414ms на GPU (из них 44 ms на суммирование)
- Еще больше обогнали одноядерный CPU, но все равно все проблемы связаны с памятью
- Нужны задачи меньше связанные со взаимодействием с памятью вне GPU. Или нужно все вычисление организовывать в памяти GPU.

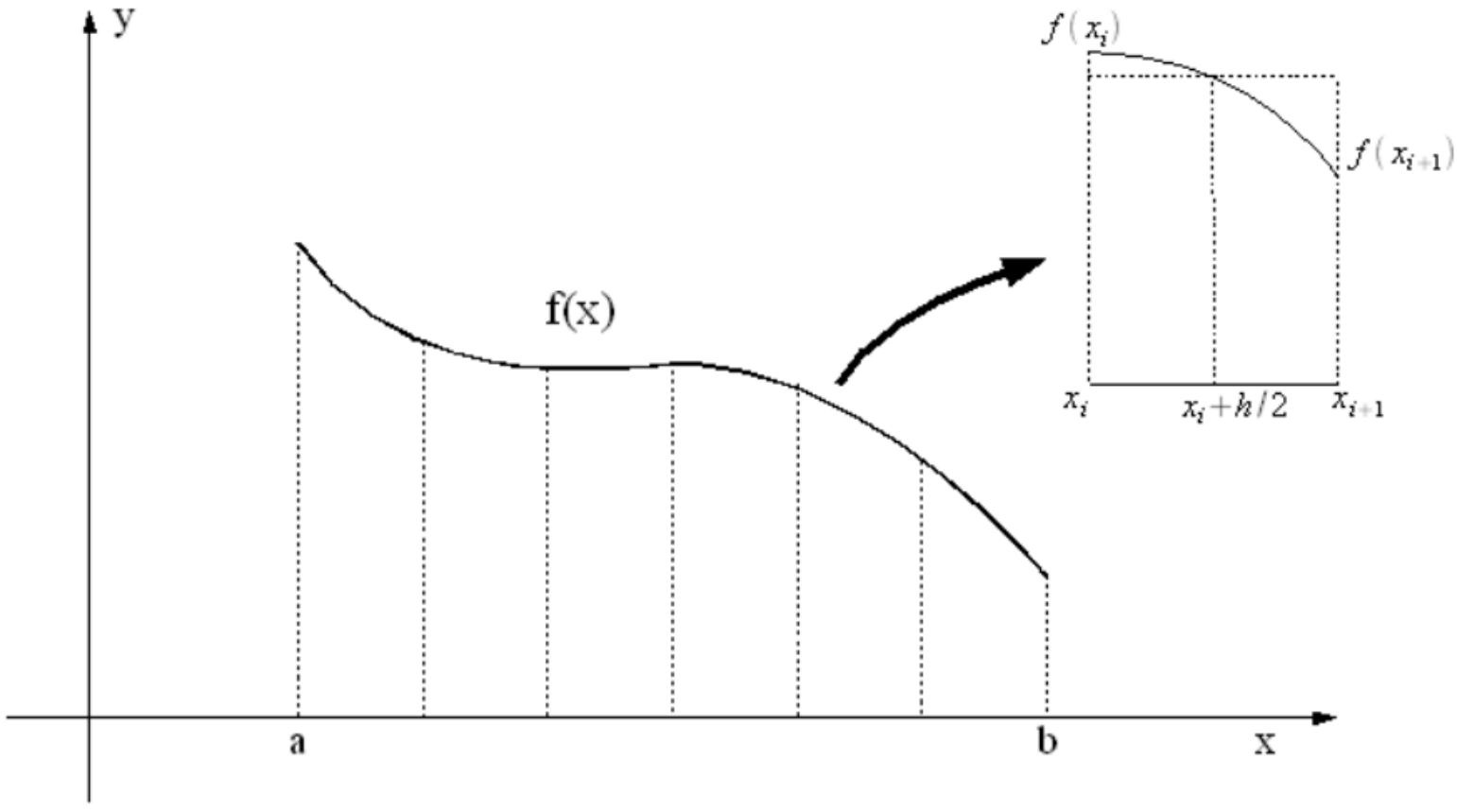
Вопросы?

Раздел 7. План.

Параллельное программирование с использованием графических процессоров

- Теоретические основы и архитектура графических процессоров
- Основы программирования с использованием CUDA
- Ветвления при выполнении программы на CUDA
- **Пример превосходства GPU, общая память и инструкции перетасовки**
- OpenMP как способ использовать GPU

Численное интегрирование



Численное интегрирование

```
#include <chrono>
#include <iostream>

#ifdef GPU
#include <cuda.h>

__device__
#endif
double f(double x) {
    return 4./(1 + x * x);
}

constexpr size_t n = 256 * 1024 * 1024;
```

Численное интегрирование

```
__global__ void integrateGPU (double *g_idata,
    double *g_odata, unsigned int n,
    double a, double h) {
    unsigned int tid = threadIdx.x;
    unsigned int num = tid +
        blockIdx.x * blockDim.x;
    if (num >= n) return;
    double x = a + (num + 0.5) * h;
    g_idata[num] = f(x) * h;
```

Численное интегрирование

```
    double *idata = g_idata +
        blockIdx.x*blockDim.x;
    for (int s = blockDim.x/2; s > 0; s>>=1) {
        if (tid < s) {
            idata[tid] += idata[tid + s];
        }
        __syncthreads();
    }
    if (tid==0) g_odata[blockIdx.x]=idata[0];
}
```

Численное интегрирование

```
int main() {
    double a = 0.0, b = 1.0;
    double h = (b - a) / n;
    cudaDeviceSynchronize();
    auto sta=std::chrono::steady_clock::now();
    double *d_A, *d_C;
    cudaMalloc(static_cast<double**>(&d_A),
        n * sizeof(double));
    constexpr size_t blsize = 1024;
    dim3 block(blsize);
    dim3 grid((n + blsize - 1)/blsize);
    cudaMalloc(static_cast<double**>(&d_C),
        grid.x * sizeof(double));
    integrateGPU <<<grid, block>>>
        (d_A, d_C, n, a, h);
```

Численное интегрирование

```
    std::vector<double> c(grid.x);
    cudaMemcpy(c.data(), d_C, grid.x *
      sizeof(double), cudaMemcpyDeviceToHost);
    cudaFree(d_A); cudaFree(d_C);
    double res = 0.;
    for (size_t i = 0; i!=grid.x; ++i) {
        res += c[i];
    }
    auto end=std::chrono::steady_clock::now();
    auto d = end-sta;
    int time = std::chrono::duration_cast
      <std::chrono::milliseconds>(d).count();
    std::cout << time << " ms "<< std::endl;
    std::cout << res << std::endl;
}
```

Произведем замер времени

- MPI-версия

```
mpirun -np 1 ./mpi
403 milliseconds
mpirun -np 2 ./mpi
210 milliseconds
mpirun -np 3 ./mpi
147 milliseconds
mpirun -np 4 ./mpi
113 milliseconds
mpirun -np 8 ./mpi
107 milliseconds
```

Произведем замер времени

- MPI-версия

```
mpirun -np 1 ./mpi
403 milliseconds
mpirun -np 2 ./mpi
210 milliseconds
mpirun -np 3 ./mpi
147 milliseconds
mpirun -np 4 ./mpi
113 milliseconds
mpirun -np 8 ./mpi
107 milliseconds
```

- CUDA-версия

```
86 milliseconds
```

Как видеокарта обыграла 4 ядра?

AMD Ryzen 7 3750H VS GeForce GTX 1650 Mobile

Как видеокарта обыграла 4 ядра?

AMD Ryzen 7 3750H, 4 cores VS GeForce GTX 1650 Mobile, 16 SM $\times$ 64 = 1024 cuda cores

Как видеокарта обыграла 4 ядра?

AMD Ryzen 7 3750H, 4 cores VS GeForce GTX 1650 Mobile, 16 SM $\times$ 64 = 1024 cuda cores

```
Global memory size : 4101898240
Memory clock rate : 4001000
Number of multiprocessors : 16
Number of cores : 1024
Maximal number of threads per MP : 1024
Clock rate of CUDA cores : 1560000
L2 cache size : 1048576
Shared memory per block : 49152
Registers per block : 65536
Warp size : 32
Maximal number of threads per block : 1024
Constant memory size: 65536
```

Численное интегрирование

Какой существенный недостаток есть у нашего кода на GPU?

Какой существенный недостаток есть у нашего кода на GPU?

```
constexpr size_t n = 256 * 1024 * 1024;
cudaMalloc(static_cast<double**>(&d_A),
    n * sizeof(double));
```

Мы выделили 2 гигабайта памяти, половина от того, что есть на GPU! А если бы точек вычисления функции было больше? Нам не нужно хранить все значения функции!

Какой существенный недостаток есть у нашего кода на GPU?

```
constexpr size_t n = 256 * 1024 * 1024;
cudaMalloc(static_cast<double**>(&d_A),
    n * sizeof(double));
```

Мы выделили 2 гигабайта памяти, половина от того, что есть на GPU! А если бы точек вычисления функции было больше? Нам не нужно хранить все значения функции!

- Чтобы избавиться от этого недостатка нужна возможность хранить лишь часть значений функции, чтобы блок мог его просуммировать. Глобальная память не подходит, поскольку мы не знаем, сколько блоков запущено одновременно. Нужна общая память (shared memory).

Общая память

- Общая память на GPU – общая внутри одного блока

Общая память

- Общая память на GPU – общая внутри одного блока
- Между блоками нет и не может быть общей памяти

Общая память

- Общая память на GPU – общая внутри одного блока
- Между блоками нет и не может быть общей памяти
- Общая память намного быстрей глобальной памяти... как минимум, в теории

Общая память

- Общая память на GPU – общая внутри одного блока
- Между блоками нет и не может быть общей памяти
- Общая память намного быстрей глобальной памяти... как минимум, в теории
- Попробуем ее задействовать

Численное интегрирование

```
__global__ void integrateGPUShared (
        double *g_odata, unsigned int n,
        double a, double h) {
    extern __shared__ double s_idata[];
    unsigned int tid = threadIdx.x;
    unsigned int num = tid +
        blockIdx.x * blockDim.x;
    if (num >= n) return;
    double x = a + (num + 0.5) * h;
    s_idata[tid] = f(x) * h;
    __syncthreads();
```

```
    for (int s = blockDim.x/2; s > 0;
            s >>= 1) {
        if (tid < s) {
            s_idata[tid] +=
                s_idata[tid + s];
        }
        __syncthreads();
    }
    if (tid == 0) g_odata[blockIdx.x] =
        s_idata[0];
}
```

Численное интегрирование

```
int main() {
    ...
    integrateGPUShared <<<grid, block,
        blsize * sizeof(double)>>>
        (d_C, n, a, h);
    ...
}
```

Численное интегрирование

```
int main() {
    ...
    integrateGPUShared <<<grid, block,
        blsize * sizeof(double)>>>
        (d_C, n, a, h);
    ...
}
```

- Третий параметр запуска ядра – размер общей памяти на блок. Возможно, когда в ядре есть ровно одна переменная, объявленная в виде `extern __shared__` без указания явного размера.

Численное интегрирование

```
int main() {
    ...
    integrateGPUShared <<<grid, block,
        blsize * sizeof(double)>>>
        (d_C, n, a, h);
    ...
}
```

- Третий параметр запуска ядра – размер общей памяти на блок. Возможно, когда в ядре есть ровно одна переменная, объявленная в виде `extern __shared__` без указания явного размера.
- Есть и статический вариант задавать размер общей памяти, увидим ниже

Произведем замер времени

- MPI-версия

```
mpirun -np 1 ./mpi
403 milliseconds
mpirun -np 8 ./mpi
107 milliseconds
```

Произведем замер времени

- MPI-версия

```
mpirun -np 1 ./mpi
403 milliseconds
mpirun -np 8 ./mpi
107 milliseconds
```

- CUDA-версия без общей памяти: 86 milliseconds

Произведем замер времени

- MPI-версия

```
mpirun -np 1 ./mpi
403 milliseconds
mpirun -np 8 ./mpi
107 milliseconds
```

- CUDA-версия без общей памяти: 86 milliseconds
- CUDA-версия с общей памятью: 117 milliseconds

Общая память медленней

Общая память расположена на мультипроцессоре, у него в нее прямой доступ

- CUDA-версия без общей памяти: 86 milliseconds

Общая память медленней

Общая память медленней

Общая память расположена на мультипроцессоре, у него в нее прямой доступ

- CUDA-версия без общей памяти: 86 milliseconds
- CUDA-версия с общей памятью: 117 milliseconds

Внимание! В большинстве руководств по разработке в интернете включая сайт NVidia вы найдете информацию о том, что нужно использовать общую память, и что она быстрее

- Она конечно быстрее, но еще есть и кэш, который быстрее общей памяти и не имеет ее недостатков (банки памяти), которые не помещаются в этот курс. Фактически каждый блок работает с кэшем, запись в глобальную память происходит один раз на double параллельно с вычислениями

Общая память медленней

Общая память расположена на мультипроцессоре, у него в нее прямой доступ

- CUDA-версия без общей памяти: 86 milliseconds
- CUDA-версия с общей памятью: 117 milliseconds
- Что делать, чтобы не терять в скорости, но не выделять такие объемы глобальной памяти? Воспользуемся механизмом перетасовки

Инструкции перетасовки

- Напомним, что блок разбивается на варпы. Размер варпа – 32 (не менялся ни разу за время развития CUDA). Начиная с архитектуры Kepler CUDA предлагает обмениваться данными внутри варпа, не прибегая к общей памяти

Инструкции перетасовки

- Напомним, что блок разбивается на варпы. Размер варпа – 32 (не менялся ни разу за время развития CUDA). Начиная с архитектуры Kepler CUDA предлагает обмениваться данными внутри варпа, не прибегая к общей памяти
- Инструкция `__shfl_down_sync` позволяет передать содержимое локальной переменной (регистра) потокам с меньшим номером.

Инструкции перетасовки

- Напомним, что блок разбивается на варпы. Размер варпа – 32 (не менялся ни разу за время развития CUDA). Начиная с архитектуры Kepler CUDA предлагает обмениваться данными внутри варпа, не прибегая к общей памяти
- Инструкция `__shfl_down_sync` позволяет передать содержимое локальной переменной (регистра) потокам с меньшим номером.
- Первый параметр этой инструкции – маска, задающая то, какие процессы передают данные, мы будет ставить полную маску

Инструкции перетасовки

- Напомним, что блок разбивается на варпы. Размер варпа – 32 (не менялся ни разу за время развития CUDA). Начиная с архитектуры Kepler CUDA предлагает обмениваться данными внутри варпа, не прибегая к общей памяти
- Инструкция `__shfl_down_sync` позволяет передать содержимое локальной переменной (регистра) потокам с меньшим номером.
- Первый параметр этой инструкции – маска, задающая то, какие процессы передают данные, мы будет ставить полную маску
- Также бывают инструкции рассылки, передачи вверх и передачи по циклу

Численное интегрирование

```
__inline__ __device__
double warpReduceSum(double val) {
    for (int shift = warpSize/2;
         shift > 0; shift /= 2) {
         val += __shfl_down_sync(
              warpSize - 1, val, shift
         );
    }
    return val;
}
```

Численное интегрирование

```
__inline__ __device__
double blockReduceSum(double val) {
    static __shared__ double shared[32];
    int lane = threadIdx.x % warpSize;
    int wid = threadIdx.x / warpSize;
    val = warpReduceSum(val);
    if (lane==0) shared[wid]=val;
    __syncthreads();
    val = (threadIdx.x < blockDim.x
            / warpSize) ?
        shared[lane] : 0;
    if (wid==0) val = warpReduceSum(val);
    return val;
}
```

Численное интегрирование

```
__global__ void integrateGPUShuffle (
        double *g_odata, unsigned int n,
        double a, double h) {
    unsigned int tid = threadIdx.x;
    unsigned int num = tid +
        blockIdx.x * blockDim.x;
    if (num >= n) return;
    double x = a + (num + 0.5) * h;
    double val = f(x) * h;
    val = blockReduceSum(val);
    if (tid == 0) g_odata[blockIdx.x] = val;
}
```

Численное интегрирование

```
int main() {
    ...
    integrateGPUShuffle
        <<<grid, block>>>
    (d_C, n, a, h);
    ...
}
```

Произведем замер времени

- MPI-версия

```
mpirun -np 1 ./mpi
403 milliseconds
mpirun -np 8 ./mpi
107 milliseconds
```

- CUDA-версия без общей памяти: 86 milliseconds
- CUDA-версия с общей памятью: 117 milliseconds

Произведем замер времени

- MPI-версия

```
mpirun -np 1 ./mpi
403 milliseconds
mpirun -np 8 ./mpi
107 milliseconds
```

- CUDA-версия без общей памяти: 86 milliseconds
- CUDA-версия с общей памятью: 117 milliseconds
- CUDA-версия с перетасовкой: 88 milliseconds

Произведем замер времени

- MPI-версия

```
mpirun -np 1 ./mpi
403 milliseconds
mpirun -np 8 ./mpi
107 milliseconds
```

- CUDA-версия без общей памяти: 86 milliseconds
- CUDA-версия с общей памятью: 117 milliseconds
- CUDA-версия с перетасовкой: 88 milliseconds
- Можно выжать еще немного развернув цикл `warpReduceSum` и сравняться с версией с глобальной памятью

Использование памяти

Примечание: мы все равно используем глобальную память для результатов суммирования по блоку (количество точек разделить на 1024). В реальном использовании от этого нужно избавляться

Использование памяти

Примечание: мы все равно используем глобальную память для результатов суммирования по блоку (количество точек разделить на 1024). В реальном использовании от этого нужно избавляться

- Это можно решить путем запуска ограниченного количества блоков (не более чем фиксированное количество, не зависимо от общего количества точек интегрирования). Тогда у нас будет лимит на использование памяти. Частичные результаты суммируются отдельным ядром или на CPU
- Однако это никак не влияет на производительность и не приносит ничего нового, так что здесь рассматриваться не будет

Выводы

Программирование на графических карточках – прогрессивная технология, позволяющая добиваться существенно превосходящих процессор результатов

(то, что у нас результаты были сравнимые – показатель не особо сильной карточки на моем ноутбуке при современном процессоре)

К сожалению, с преимуществами приходят и сложности

Выводы

Программирование на графических карточках – прогрессивная технология, позволяющая добиваться существенно превосходящих процессор результатов

(то, что у нас результаты были сравнимые – показатель не особо сильной карточки на моем ноутбуке при современном процессоре)

К сожалению, с преимуществами приходят и сложности

- Необходимо особое внимание уделять взаимодействию с памятью, стараться минимизировать пересылки между памятью процессора и видеокарты.

Выводы

Программирование на графических карточках – прогрессивная технология, позволяющая добиваться существенно превосходящих процессор результатов

(то, что у нас результаты были сравнимые – показатель не особо сильной карточки на моем ноутбуке при современном процессоре)

К сожалению, с преимуществами приходят и сложности

- Необходимо особое внимание уделять взаимодействию с памятью, стараться минимизировать пересылки между памятью процессора и видеокарты.
- Написание для GPU весьма чувствительно к выбору механизмов (общая память, текстурная, константная, перетасовки,...). И что осложняет использование, информация в интернете весьма разрозненная и зачастую устаревшая ввиду быстрого развития технологии.

Вопросы?

Раздел 7. План.

Параллельное программирование с использованием графических процессоров

- Теоретические основы и архитектура графических процессоров
- Основы программирования с использованием CUDA
- Ветвления при выполнении программы на CUDA
- Пример превосходства GPU, общая память и инструкции перетасовки
- **OpenMP как способ использовать GPU**

OpenMP как способ использовать GPU

- OpenMP также позволяет писать программы, которые будут запускаться на GPU. Попробуем изучить, как это работает при условии, что у вас есть GPU от NVidia и OpenMP начиная с версии 4.0.

OpenMP как способ использовать GPU

- OpenMP также позволяет писать программы, которые будут запускаться на GPU. Попробуем изучить, как это работает при условии, что у вас есть GPU от NVidia и OpenMP начиная с версии 4.0.
- Завести OpenMP + GPU под g++ крайне сложно, нам потребуется clang

OpenMP как способ использовать GPU

- OpenMP также позволяет писать программы, которые будут запускаться на GPU. Попробуем изучить, как это работает при условии, что у вас есть GPU от NVidia и OpenMP начиная с версии 4.0.
- Завести OpenMP + GPU под g++ крайне сложно, нам потребуется clang
- sudo apt install clang llvm clang-tools libomp-dev

OpenMP как способ использовать GPU

- OpenMP также позволяет писать программы, которые будут запускаться на GPU. Попробуем изучить, как это работает при условии, что у вас есть GPU от NVidia и OpenMP начиная с версии 4.0.
- Завести OpenMP + GPU под g++ крайне сложно, нам потребуется clang
- sudo apt install clang llvm clang-tools libomp-dev
- Теперь вы можете собрать свою программу через

clang++ -fopenmp -fopenmp-targets=nvptx64-nvidia-cuda openmp-gpu.cpp

OpenMP как способ использовать GPU

Напишем простейшую программу

```
int main() {
  int num_devices = omp_get_num_devices();
  printf("Number of devices %d\n", num_devices);
#pragma omp target
  {
    if (omp_is_initial_device()) {
      printf("Running on host\n");
    } else {
      int nteams= omp_get_num_teams();
      int nthreads= omp_get_num_threads();
      printf("%d teams, %d threads in team\n",
        nteams,nthreads);
    }
  }
}
```

OpenMP как способ использовать GPU

Теперь попробуем осуществить суммирование векторов. Опуская генерацию,

```
double* A = a.data();
double* B = b.data();
double* C = c.data();
```

Далее последует Offload на GPU, как это называется в терминах OpenMP.

OpenMP как способ использовать GPU

```
#pragma omp target data map(to: A[0:n], B[0:n])
                        map(from: C[0:n])
{
  #pragma omp target teams distribute parallel for
  for (int i = 0; i < n; ++i) {
    int tid = omp_get_thread_num();
    int gid = omp_get_team_num();
    int num_teams = omp_get_num_teams();
    int num_threads = omp_get_num_threads();
    printf("Team %d/%d\n", gid, num_teams);
    printf("Thread %d/%d\n",tid,num_threads);
    C[i] = A[i] + B[i]; // main command
  }
}
```

OpenMP как способ использовать GPU

На самом деле, команда-то крайне простая. Магия OpenMP в действии...

```
#pragma omp target data map(to: A[0:n], B[0:n])
                                    map(from: C[0:n])
{
  #pragma omp target teams distribute parallel for
  for (int i = 0; i < n; ++i) {
    C[i] = A[i] + B[i]; // main command
  }
}
```

OpenMP как способ использовать GPU

На самом деле, команда-то крайне простая. Магия OpenMP в действии...

```
#pragma omp target data map(to: A[0:n], B[0:n])
                                    map(from: C[0:n])
{
  #pragma omp target teams distribute parallel for
  for (int i = 0; i < n; ++i) {
    C[i] = A[i] + B[i]; // main command
  }
}
```

...если не сравнивать время, которое выходит 259 ms на одном ядре и 359ms на GPU (размер 32*1024*1024).

OpenMP как способ использовать GPU

Но мы прошаренные чуваки, и знаем, что время уходит на копирование, поэтому давайте хотя бы попробуем решить задачу, которую мы более эффективно делали на GPU — суммирование вектора.

```
#pragma omp target data map(to: A[0:n] map(from: sum
{
  #pragma omp target teams distribute parallel for
                                        reduction(:+sum)
  for (int i = 0; i < n; ++i) {
    sum += A[i];
  }
}
```

И, к сожалению, мы все равно проигрываем (взяли вектор 256*1024*1024), получая 494ms на одном ядре и 5491ms на GPU. Победить это какими-либо опциями невозможно.

Вопросы?

Домашнее задание

Внимание! Задание!

- Возьмите остаток деления своего номера на 10 (0 -> 10)

Домашнее задание

Внимание! Задание!

- Возьмите остаток деления своего номера на 10 (0 -> 10)
- Найдите в нижеследующем списке объект или метод, отвечающий этому номеру

Домашнее задание

Внимание! Задание!

- Возьмите остаток деления своего номера на 10 (0 -> 10)
- Найдите в нижеследующем списке объект или метод, отвечающий этому номеру
- Реализуйте простейшую программу, демонстрирующую полезность именно этой конструкции. При этом используйте минимальное количество других конструкций из Cuda

Домашнее задание

Внимание! Задание!

- Возьмите остаток деления своего номера на 10 (0 -> 10)
- Найдите в нижеследующем списке объект или метод, отвечающий этому номеру
- Реализуйте простейшую программу, демонстрирующую полезность именно этой конструкции. При этом используйте минимальное количество других конструкций из Cuda
- Результат сдается на сайте курса – нужен код программы (должна компилироваться) и описание, что вы этим демонстрируете (можно внутри кода в комментариях). Если требуются особые опции компиляции, напишите об этом

Список объектов и методов

1. `cudaMallocPitch`
2. `cudaMalloc3D`
3. `cudaMallocHost`
4. `__syncthreads`
5. `cudaGetSymbolSize`
6. `cudaHostAlloc`
7. `cudaStream_t`
8. `cudaGetDeviceProperties`
9. `cudaEventRecord`
10. `cudaCreateTextureObject`

Вопросы?