

Технология CUDA для высокопроизводительных вычислений на кластерах с графическими процессорами

Колганов Александр
alexander.k.s@mail.ru

часть 2

Что было в прошлый раз?

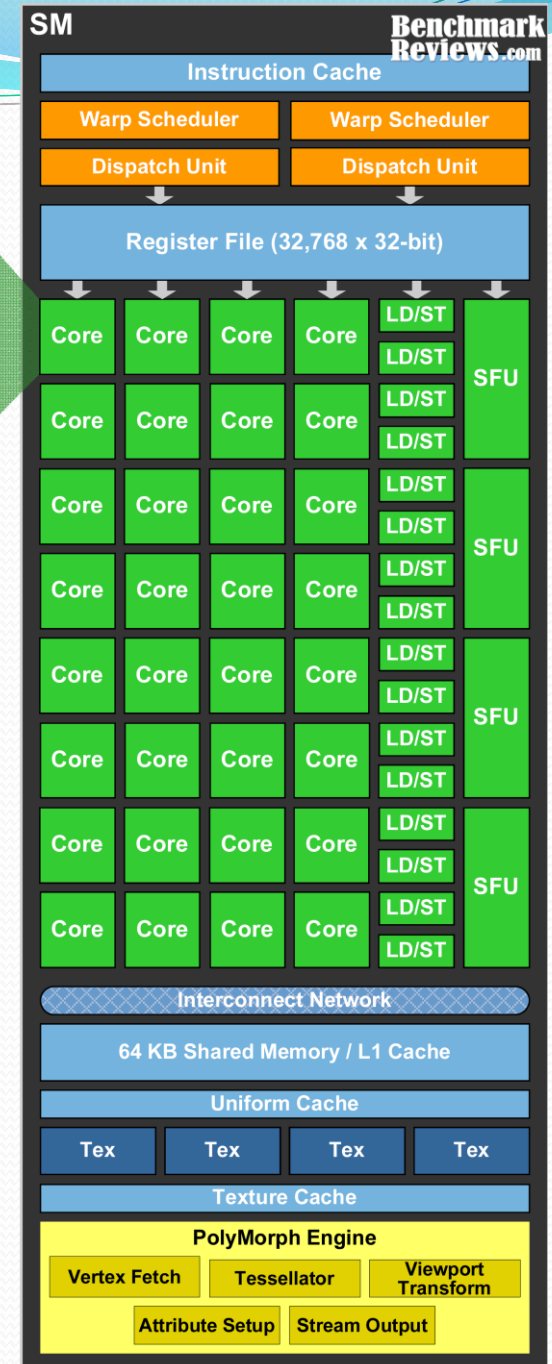
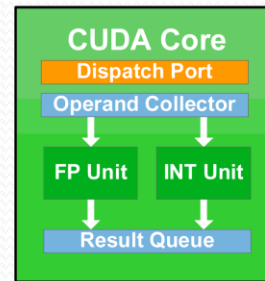
Чип в максимальной конфигурации

- 16 SM
- 512 ядер CUDA Core
- Кеш L2 758KB
- GigaThreadEngine
- Контроллеры памяти DDR5
- Интерфейс PCI

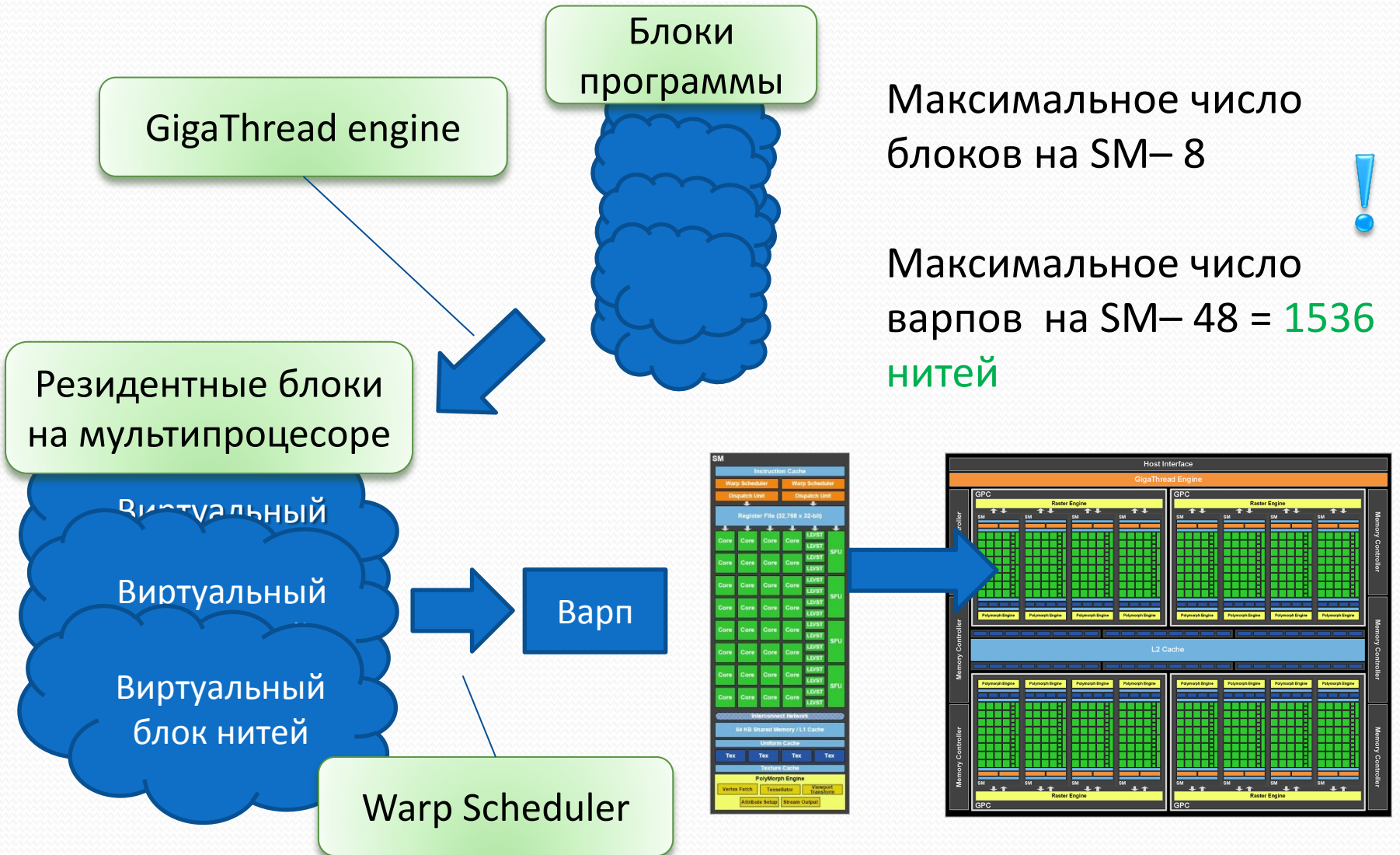


GPU Streaming Multiprocessor (SM)

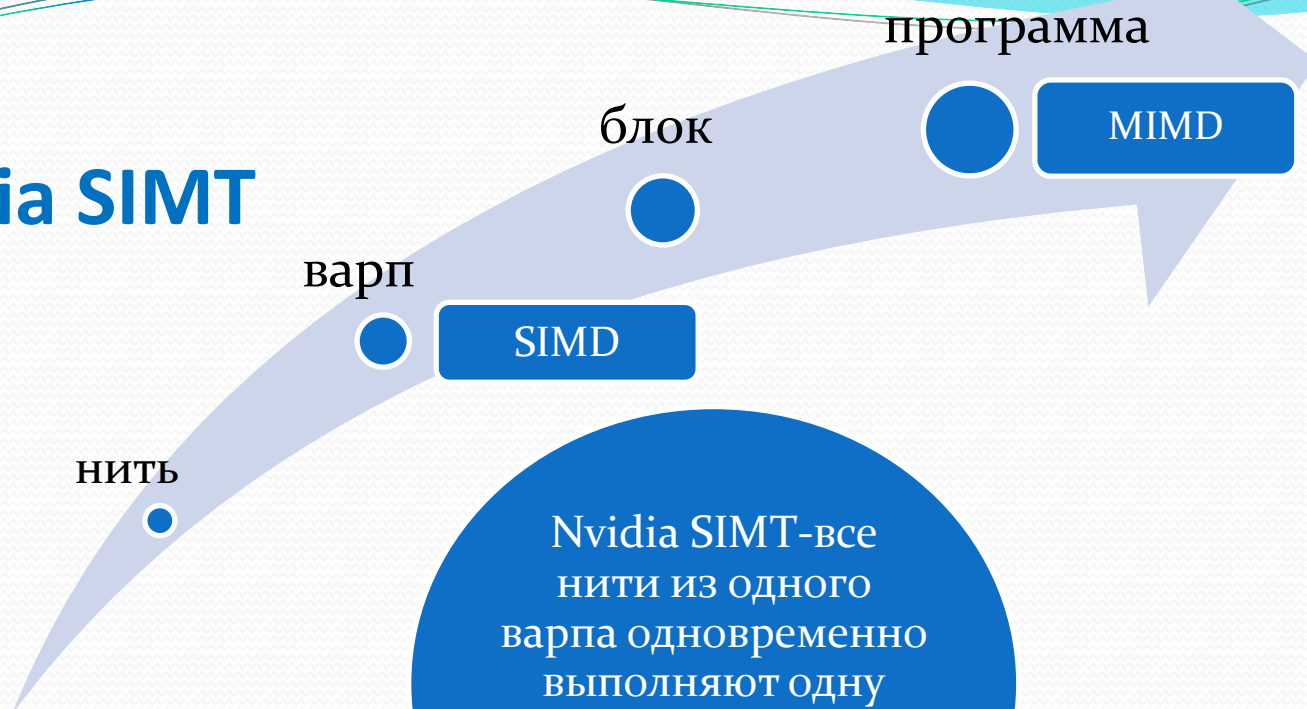
- Поточковый мультипроцессор
- «Единица» построения устройства (как ядро в CPU):
 - 32 скалярных ядра CUDA Core, ~1.5ГГц
 - 2 Warp Scheduler-а
 - Файл регистров, 128KB
 - 3 Кэша – текстурный, глобальный (L1), константный(uniform)
 - PolyMorphEngine – графический конвейер
 - Текстурные юниты
 - 16 x Special Function Unit (SFU) – интерполяция и трансцендентная математика одинарной точности
 - 16 x Load/Store



Модель выполнения



Nvidia SIMT



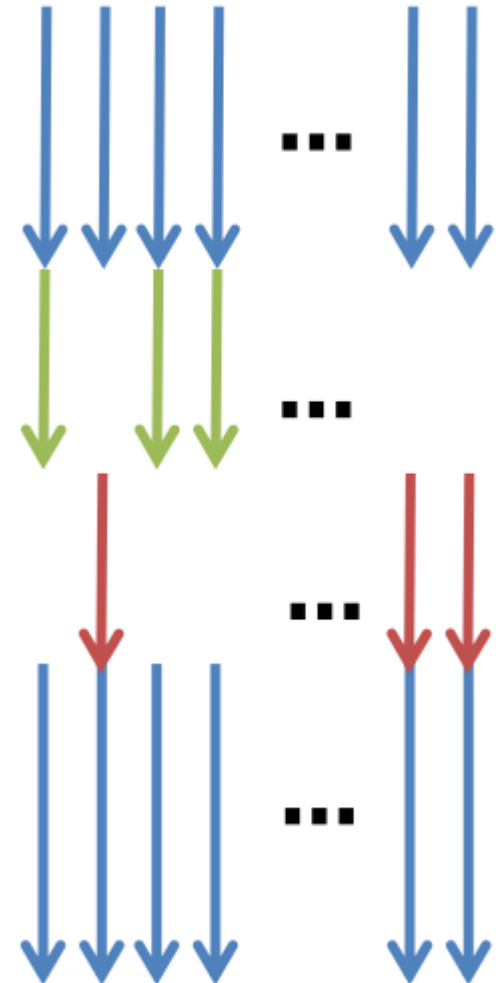
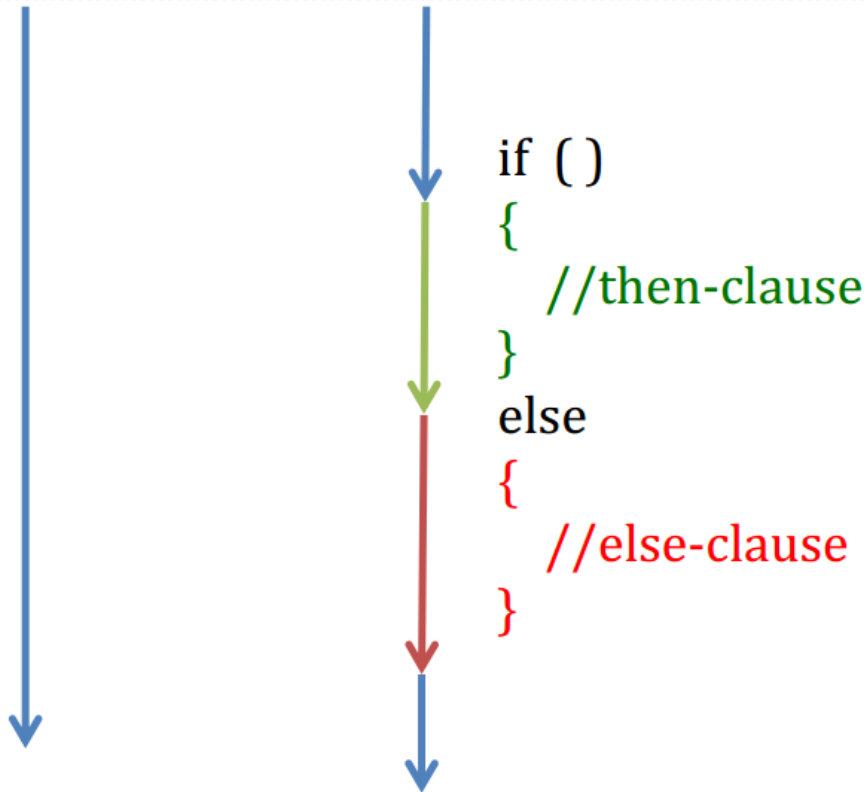
Nvidia SIMT-все нити из одного варпа одновременно выполняют одну инструкцию, варпы выполняются независимо

SIMD – все нити одновременно выполняют одну инструкцию

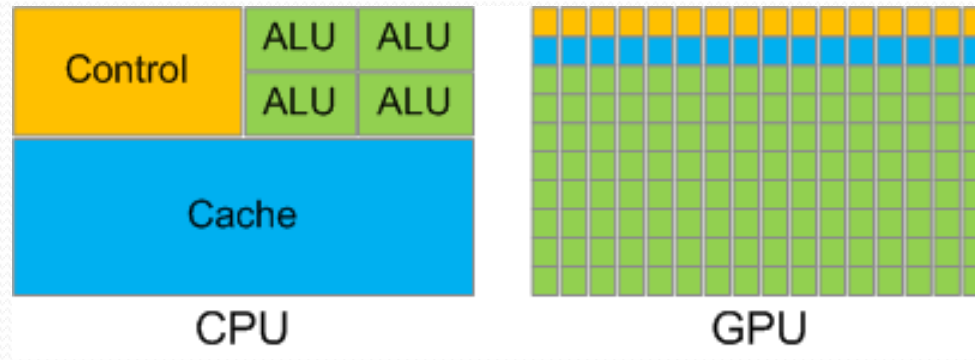
MIMD – каждая нить выполняется независимо от других, SMP – все нити имеют равные возможности для доступа к памяти

Ветвление (branching)

И
н
с
т
р
у
к
ц
и
и



Утилизация латентности памяти



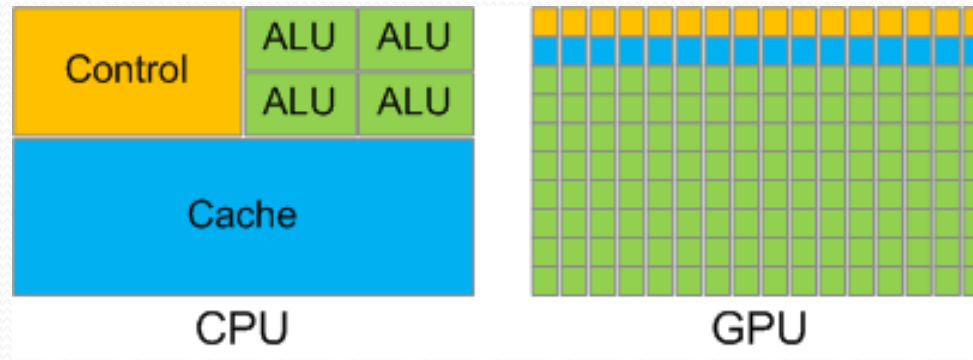
- Цель: эффективно загружать Ядра

Проблема: латентность памяти

Решение:

- CPU: Сложная иерархия кешей
- GPU: Много нитей, покрывать обращения одних нитей в память вычислениями в других за счёт быстрого переключения контекста

Утилизация латентности памяти



- GPU: Много нитей, покрывать обращения одних нитей в память вычислениями в других за счёт быстрого переключения контекста
- За счёт наличия сотен ядер и поддержки миллионов нитей (потребителей) на GPU **легче** утилизировать всю полосу пропускания



SIMT и масштабирование

- Виртуальная
 - GPU может поддерживать миллионы виртуальных нитей
 - Виртуальные блоки независимы
 - Программу можно запустить на любом количестве SM
- Аппаратная
 - Мультипроцессоры независимы
 - Можно «нарезать» GPU с различным количеством SM



CUDA Kernel («Ядро»)

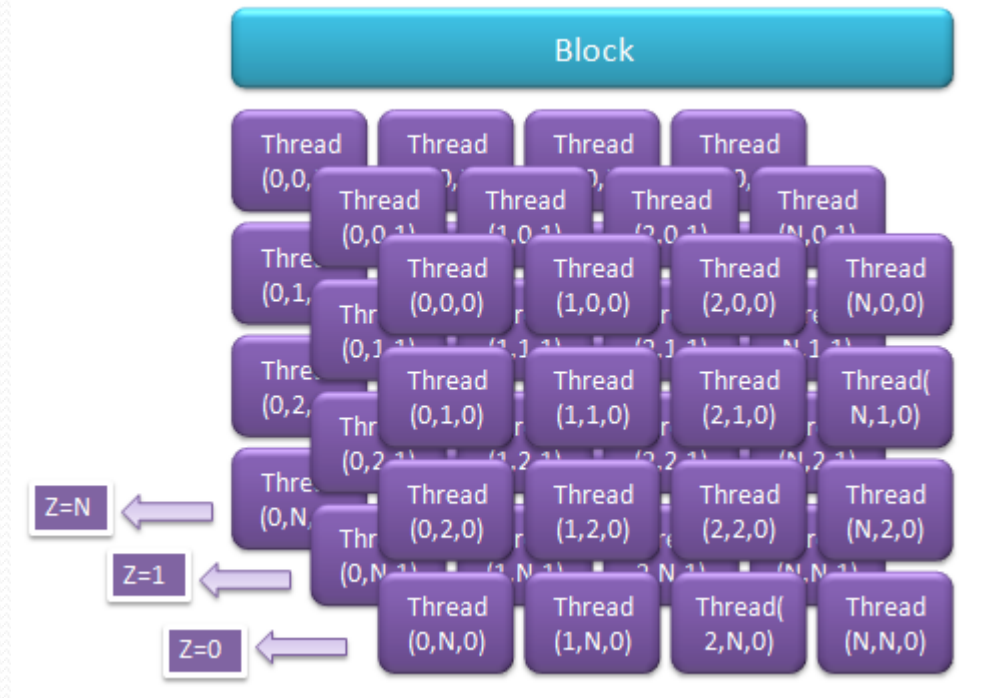
- Специальная функция, являющаяся **входной точкой** для кода на GPU
 - Нет возвращаемого значения (`void`)
 - Выделена атрибутом `__global__`
- Объявления параметров и их использование такое же, как и для обычных функций

```
__global__ void kernel (int * ptr) {  
    ptr = ptr + 1;  
    ptr[0] = 100;  
    ...; //other code for GPU  
}
```

- **Хост** запускает именно «**ядра**», устройство их выполняет

CUDA Grid

- Двумерный грид из трёхмерных блоков
 - Логический индекс по переменной z у всех блоков равен нулю
 - Каждый блок состоит из трёх «слоёв» нитей, соответствующих $z=0,1,2$



Ориентация нити в гриде

- Осуществляется за счёт встроенных переменных:
 - *threaIdx.x threaIdx.y threaIdx.z* - индексы нити в блоке
 - *blockIdx.x blockIdx.y blockIdx.z* – индексты блока в гриде
 - *blockDim.x blockDim.y blockDim.z* – размеры блоков в нитях
 - *gridDim.x gridDim.y gridDim.z* – размеры грида в блоках

- Линейный индекс нити в гриде:

```
int gridSizeX = blockDim.x * gridDim.x;
int gridSizeZ = ... ; gridSizeY = ...;
int gridSizeAll = gridSizeX * gridSizeY * gridSizeZ;
int threadLinearIdx =
    threaIdx.z * gridSizeY + threaIdx.y) * gridSizeX +
threadIdx.x;
```

Пример: сложение векторов

```
__global__ void sum_kernel( int *A, int *B, int *C, int N)
{
    int threadLinearIdx =
        blockIdx.x * blockDim.x + threadIdx.x; //определить свой индекс
    if (threadLinearIdx < N) { // все нити [0 .. N - 1]
        int elemA = A[threadLinearIdx]; //считать нужный элемент A
        int elemB = B[threadLinearIdx]; // считать нужный элемент B
        C[threadLinearIdx] = elemA + elemB; //записать результат
    }
}
```

• Каждая нить

- Получает копию параметров (В данном случае, это адреса вектором на GPU);
- Определяет своё положение в гриде **threadLinearIdx** ;
- Считывает из входных векторов элементы с индексом **threadLinearIdx** и записывает их сумму в выходной вектор по индексу **threadLinearIdx** ;
- рассчитывает один элемент выходного массива.

Host Code

- Выбрать устройство
 - ✓ По умолчанию, устройство с номером 0
- Выделить память на устройстве
- Переслать на устройство входные данные
- Рассчитать грид
 - ✓ Размер грида зависит от размера задачи
- Запустить ядро
- Переслать с устройства на хост результат

Выбор устройства

- `struct cudaDeviceProp`
 - структура с параметрами устройства. Полный список параметров см. в документации
- `cudaError_t cudaGetDeviceCount (int* count)`
 - записывает в `*count` число доступных устройств в системе
- `cudaError_t cudaGetDeviceProperties (cudaDeviceProp * prop, int device)`
 - записывает параметры устройства с индексом `device` в `*prop`
- `cudaError_t cudaSetDevice (int device)`
 - выбрать устройство с индексом `device` для проведения вычислений

Выбор устройства

```
int deviceCount=0, suitableDevice=-1;
cudaDeviceProp devProp; // структура с параметрами устройства

cudaGetDeviceCount( &deviceCount ); // число доступных устройств
printf ( "Found %d devices\n", deviceCount );
for ( int device = 0; device < deviceCount; device++ ) {
    cudaGetDeviceProperties ( &devProp, device ); // получить параметры устройства с
                                                    заданным номером

    printf( "Device %d\n", device );
    printf( "Compute capability      : %d.%d\n", devProp.major, devProp.minor);
    printf( "Name                    : %s\n", devProp.name );
    printf("Total Global Memory      : %d\n", devProp.totalGlobalMem );
    if (ourRequirementsPassed(devProp)) // ищем устройство с нужными параметрами
        suitableDevice = device ;
}
assert(suitableDevice != -1);
cudaSetDevice(suitableDevice); // Выбрать для работы заданное устройство
```

Пример: Копирование памяти

```
int n = getSize(); // размер задачи
```

```
int nb = n * sizeof (float); // размер размер задачи в байтах
```

```
float * inputDataOnHost = (float *)malloc( nb ); // память на хосте для входных данных
```

```
float * resultOnHost = (float *)malloc( nb ); // память на хосте для результата
```

```
float * inputDataOnDevice= NULL , *resultOnDevice = NULL; // память на устройстве
```

```
getInputData(inputDataOnHost); // получить входные данные
```

```
cudaMalloc( (void**)& inputDataOnDevice, nb ); // выделить память на устройстве для входных данных
```

```
cudaMalloc( (void**)& resultOnDevice, nb ); // выделить память на устройстве для хранения результата
```

```
cudaMemcpy(inputDataOnDevice, inputDataOnHost , nb, cudaMemcpyHostToDevice); // переслать на устройство  
входные данные
```

```
//запустить ядро. Выходные данные получим в resultOnDevice
```

```
cudaMemcpy(resultOnHost , resultOnDevice , nb, cudaMemcpyDeviceToHost); // переслать результаты на хост
```

```
cudaFree(inputDataOnDevice) ; // освободить память на устройстве
```

```
cudaFree(resultOnDevice ) ; // освободить память на устройстве
```

Пример: Запуск ядра

```
int n = getSize(); // размер задачи
//определения указателей, получение входных данных на хосте

cudaMalloc( (void**)& inputDataOnDevice, nb); // выделить память на устройстве для входных данных
cudaMalloc( (void**)& resultOnDevice, nb); // выделить память на устройстве для хранения результата

cudaMemcpy(inputDataOnDevice, inputDataOnHost , nb, cudaMemcpyHostToDevice); // переслать на устройство
                                                    входные данные

dim3 blockDim = dim3(512), gridDim = dim3( (n - 1) / 512 + 1 ); // рассчитать конфигурацию запуска
kernel <<< gridDim, blockDim >>> (inputDataOnDevice, resultOnDevice, n); // запустить ядро с рассчитанной
                                                    конфигурацией и параметрами

cudaMemcpy(resultOnHost , resultOnDevice , nb, cudaMemcpyDeviceToHost); // переслать результаты на хост

cudaFree(inputDataOnDevice) ; // освободить память на устройстве
cudaFree(resultOnDevice ) ; // освободить память на устройстве
```

Асинхронность в CUDA

- Чтобы GPU больше времени работало в фоновом режиме, параллельно с CPU, некоторые вызовы являются асинхронными
 - Отправляют команду на устройство и сразу возвращают управление хосту
- К таким вызовам относятся:
 - Запуски ядер (если `CUDA_LAUNCH_BLOCKING` не установлена на 1)
 - Копирование между двумя областями памяти на устройстве
 - Копирование с хоста на устройство менее 64KB
 - Копирования, выполняемые функциями с окончанием `*Async`
 - `cudaMemSet` – присваивает всем байтам области памяти на устройстве одинаковое значение (чаще всего используется для обнуления)

Асинхронность в CUDA

- Почему тогда верно работает код?

```
//запуск ядра (асинхронно)
sum_kernel<<< blocks, threads >>>(aDev, bDev, cDev);
//переслать результаты обратно на хост
cudaMemcpy(cHost, cDev, nb, cudaMemcpyDeviceToHost);
```

- Ведь хост вызывает `cudaMemcpy` до завершения выполнения ядра!

cudaStream

- Последовательность команд для GPU (запуски ядер, копирования памяти и т.д.), **исполняемая строго последовательно, следующая выполняется после завершения предыдущей**
- Команды из разных потоков могут выполняться параллельно, независимо от исполнения команд в других потоках
- Пользователь сам объединяет команды в потоки.
- Результаты взаимодействия команд из разных потоков непредсказуемы
- По умолчанию, все команды помещаются в «Default Stream», равный нулю



Асинхронность в CUDA

- Почему тогда верно работает код?

```
// запуск ядра (асинхронно)
sum_kernel<<< blocks, threads >>>(aDev, bDev, cDev);
//переслать результаты обратно на хост
cudaMemcpy(cHost, cDev, nb, cudaMemcpyDeviceToHost);
```

- Вызов ядра и `cudaMemcpy` попадают в один поток (поток по умолчанию)
 - Устройство гарантирует их последовательное выполнение

Обработка ошибок

- Коды всех возникающих ошибок автоматически записываются в единственную специальную хостовую переменную типа `enum cudaError_t`
 - Эта переменная в каждый момент времени равна коду **последней** ошибки, произошедшей в системе
 - `cudaError_t cudaPeekAtLastError()` – возвращает текущее значение этой переменной
 - `cudaError_t cudaGetLastError()` - возвращает текущее значение этой переменной и присваивает ей `cudaSuccess`
 - `const char* cudaGetErrorString (cudaError_t error)` – по коду ошибки возвращает её текстовое описание

Обработка ошибок

- Простейший способ быть уверенным, что в программе не произошло CUDA-ошибки:
 - В конце main вставить
std::cout << cudaGetErrorString(cudaGetLastError());
- Рекомендуемый:

```
#define SAFE_CALL(err) do \  
{ if (err != 0) \  
  { printf("ERROR [%s] in line %d: %s\n", __FILE__,  
          __LINE__, cudaGetErrorString(err)); \  
    exit(1); \  
  } \  
} while(0)
```

События

- Маркеры, приписываемые точкам программы
 - ✓ Можно проверить произошло событие или нет
 - ✓ Можно замерить время между двумя произошедшими событиями
- Событие происходит когда завершаются все команды, помещённые в поток, к которому приписано событие, **до последнего** вызова **cudaEventRecord** для него
- Если событие приписано потоку по умолчанию (`stream = 0`), то оно происходит в момент завершения **всех** команд, помещённых во **все потоки** до последнего вызова **cudaEventRecord** для него

Измерение времени выполнения

- Расчёт времени выполнения ядра
 - Записать одно событие до вызова ядра, другое – сразу после:

```
cudaEvent_t start, stop;  
float time;  
cudaEventCreate(&start);  
cudaEventCreate(&stop);
```

```
cudaEventRecord( start, 0 ); // В потоке могут быть некоторые команды  
                                // приписываем этой точке событие start  
kernel<<<grid,threads>>> ( params);  
cudaEventRecord( stop, 0 ); // К данному моменту в поток отправлена команда  
                                вызова ядра, приписываем этой точке событие stop  
cudaEventSynchronize( stop ); // Дождаться событие stop, т.е. завершения  
                                выполнения всех команд, в т.ч. запуска ядра  
cudaEventElapsedTime( &time, start, stop ); // Время между двумя событиями  
  
cudaEventDestroy( start );  
cudaEventDestroy( stop );
```

Компиляция и запуск

Особое отношение к .cu

При работе с CUDA используются расширения Си++:

- Конструкция запуска ядра <<< ... >>>
- Встроенные переменные `threadIdx`, `blockIdx`
- Квалификаторы `__global__` `__device__` и т.д.
-
- Эти расширения могут быть обработаны только в `*.cu` файлах!
 - `cudafe` не запускается для файлов с другим расширением
 - В этих файлах можно не делать `#include <cuda_runtime.h>`
- Вызовы библиотечных функций вида `cuda*` можно располагать в `*.cpp` файлах
 - Они будут слинкованы обычным линковщиком из библиотеки `libcudart.so`

Компиляция хост-кода

В файле `test.cpp` :

- Основной хост-код. Т.к. конструкцию запуска ядра в `*.cpp` применять нельзя, вынесем её в отдельную функцию, определённую в каком-нибудь `*.cu`

```
#include <cuda_runtime.h> // здесь объявления функций тулкита
void launchKernel(params); // определить эту функцию в каком-нибудь *.cu
int main() {
    ... // обычный хост-код

    cudaSetDevice(0); // Можно! Обычная функция, потом слинкуется

    ... // обычный хост-код, kernel<<1,1>>(params) здесь нельзя, только в *.cu!

    launchKernel(params); // Внутри этой функции будет вызвано ядро
                          // Определена в некотором *.cu

    ... // обычный хост-код
}
```

Компиляция хост-кода

В файле `test.cpp` :

- Основной хост-код. Т.к. конструкцию запуска ядра в *.cpp применять нельзя, вынесем её в отдельную функцию, определённую в каком-нибудь *.cu
- Компиляция:

```
$g++ -I /toolkit_install_dir/include test.cpp -c -o test.o
```

- Указали путь, по которому следует искать инклюд-файлы для CUDA
- Попросили сделать объектник

Или же через nvcc, без указания директории с инклюдами:

```
$nvcc test.cpp -c -o test.o
```

Компиляция device-кода

В файле `kernel.cu`

- Определяем ядро и функцию для его запуска. В функции запуска рассчитывается конфигурация и дополнительно выводится на экран время работы ядра

```
__global__ void kernel(params) {  
    ...; // код ядра  
}
```

```
void launchKernel(params) {  
  
    ...; // расчёт конфигурации запуска в зависимости от параметров, создание событий  
  
    float time;  
    cudaEventRecord( start, 0 );  
    kernel<<< конфигурация >>> (params); // запуск ядра  
    cudaEventRecord( stop, 0 );  
    cudaEventSynchronize( stop );  
    cudaEventElapsedTime( &time, start, stop ); // Время между двумя событиями  
    printf("%4.4f, time);  
}
```


Компиляция device-кода

В файле `kernel.cu`

- Определяем ядро и функцию для его запуска. В функции запуска рассчитывается конфигурация и дополнительно выводится на экран время работы ядра
- Компиляция:

```
$nvcc -arch=sm_20 -Xptxas -v kernel.cu -c -o kernel.o
```

Линковка проекта

```
$g++ -L/toolkit_install_dir/lib64 -lcudart test.o kernel.o -o test
```

- Попросили слинковаться с `libcudart.so`, указали где она может лежать

```
$nvcc test.o kernel.o -o test
```



- `nvcc -v test.o kernel.o -o test` покажет какая конкретно команда вызвалась

Также можно расположить весь код в `*.cu` файле и не пользоваться `*.cpp` вообще

Запуск

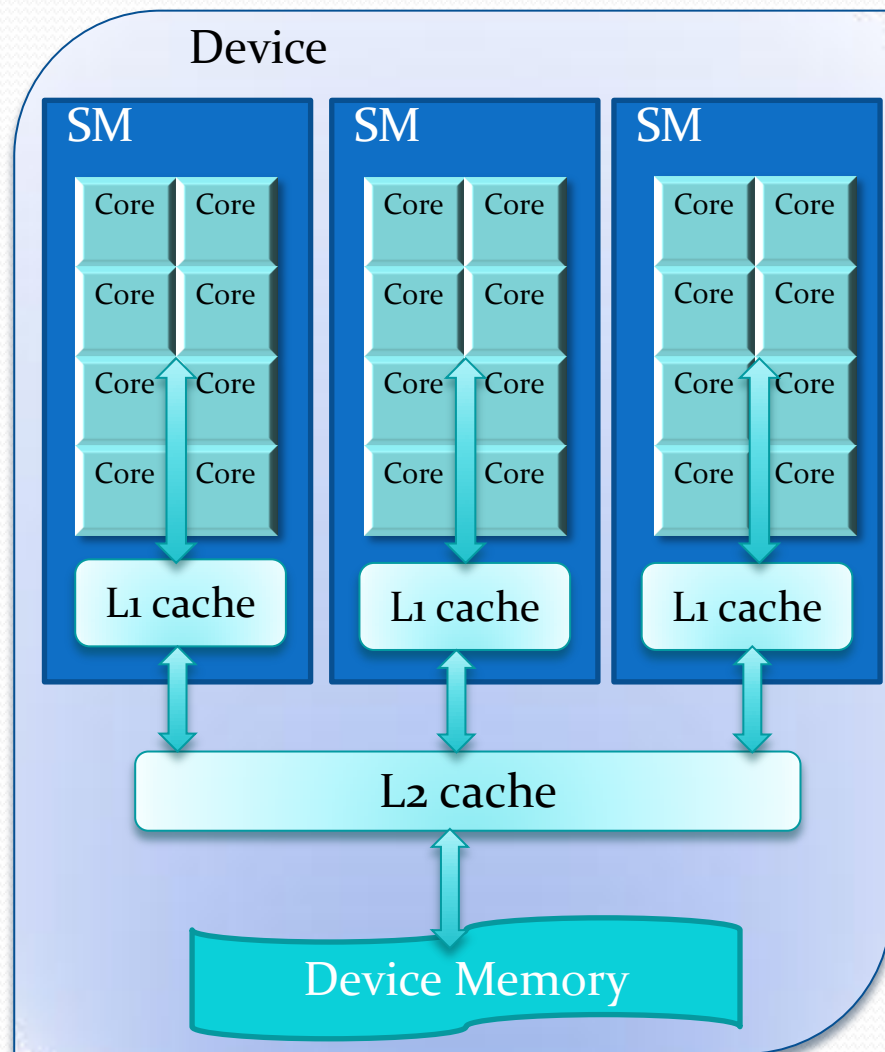
- В результате компиляции и линковки получаем обычный исполняемый файл
- Запускаем из командной строки обычным способом

```
$/test 1024
```

Глобальная память

Глобальная память

- Расположена в **DRAM GPU**
- Объём до 4Gb
 - Параметр устройства **totalGlobalMem**
- **Кешируется** в кешах L1 и L2:
 - L1 – на каждом мультипроцессоре
максимальный размер 48KB
минимальный размер 16KB
 - L2 – на устройстве
максимальный размер 768 KB
Параметр устройства **l2CacheSize**



Глобальная память

Выделение:

- Динамически с хоста через `cudaMalloc()`
- Динамически из ядер через `malloc()`
- Статически - глобальная переменная с атрибутом `__device__`

Динамически с хоста

```
__global__ void kernel(int *arrayOnDevice) {  
    arrayOnDevice[threadIdx.x] = threadIdx.x;  
}
```

```
int main() {  
    size_t size = 0;  
    void *devicePtr = NULL;  
    int hostMem[512];  
    cudaMalloc(&devicePtr, sizeof(hostMem));  
    cudaMemcpy(devicePtr, hostMem, size, cudaMemcpyHostToDevice);  
    kernel<<<1,512>>>(devicePtr);  
}
```

Статически

```
__device__ int arrayOnDevice[512]
__global__ void kernel() {
    arrayOnDevice[threadIdx.x] = threadIdx.x;
}

int main() {
    size_t size = 0;
    void *devicePtr = NULL;
    int hostMem[512];
    cudaGetSymbolSize(&size, arrayOnDevice);
    cudaMemcpyToSymbol(arrayOnDevice, localMem, size);
    kernel<<<1,512>>>(devicePtr);
}
```


Статически

```
__device__ int arrayOnDevice[512]
__global__ void kernel() {
    arrayOnDevice[threadIdx.x] = threadIdx.x;
}

int main() {
    size_t size = 0;
    int hostMem[512];
    void *devicePtr = NULL;
    cudaGetSymbolSize(&size, arrayOnDevice);
    cudaGetSymbolAddress(&devicePtr, arrayOnDevice);
    cudaMemcpy(devicePtr, hostMem, size, cudaMemcpyHostToDevice);
    kernel<<<1,512>>>();
}
```

cuda*Symbol*

- Переменные с атрибутами `__device__` и `__constant__` находятся в глобальной области видимости и хранятся в объектном модуле как отдельные символы
- Память под них выделяется статически при старте приложения, как и под обычные глобальные переменные
- Работать с ними на хосте можно через функции `cudaMemcpyToSymbol`, `cudaMemcpyToSymbolAsync`, `cudaGetSymbolAddress`, `cudaMemcpyFromSymbol`, `cudaMemcpyFromSymbolAsync`, `cudaGetSymbolSize`

Динамически из ядер

```
#include <stdlib.h>
```

```
__global__ void kernel() {  
    size_t size = 1024 * sizeof(int);  
    int *ptr = (int *)malloc(size);  
    memset(ptr, 0, size)  
    free(ptr)  
}
```

```
int main() {  
    cudaDeviceSetLimit(cudaLimitMallocHeapSize, 128*1024*1024);  
    kernel<<<1, 128>>>();  
}
```

Динамически из ядер

- `malloc()` из ядра выделяет память в куче
- Не освобождается между запусками ядер
- Освобождение по `free()` только с устройства
- Компилировать с `-arch=sm_20`
- Доступны `memcpy()`, `memset()`

Динамически из ядер

- Память под кучу выделяется на устройстве при инициализации CUDA runtime и освобождается при завершении программы
 - После создания размер кучи не может быть изменен
 - По-умолчанию 8мб
 - Можно задать до первого вызова ядра с `malloc` через `cudaDeviceSetLimit(cudaLimitMallocHeapSize, N)`

Режимы работы кеша L1

- Кеш может работать в двух режимах: 48KB и 16KB
- Переключение режимов:

- `cudaError_t cudaDeviceSetCacheConfig(cudaFuncCache cacheConfig)`

Устанавливает режим работы кеша `cacheConfig` для всего устройства

- Возможные режимы:
 - `cudaFuncCachePreferNone` – без предпочтений(по умолчанию). Выбирается последняя использованная конфигурация. Начальная конфигурация – 16KB L1
 - `cudaFuncCachePreferShared`: 16KB L1
 - `cudaFuncCachePreferL1`: 48KB L1

- `cudaError_t cudaFuncSetCacheConfig (const void* func, cudaFuncCache cacheConfig)`

Устанавливает режим работы кеша `cacheConfig` для заданной функции `func`

- По умолчанию - `cudaFuncCachePreferNone` - запускать с режимом устройства

Транзакции

- Глобальная память оптимизирована с целью увеличения полосы пропускания
 - Отдать максимум данных за одно обращение

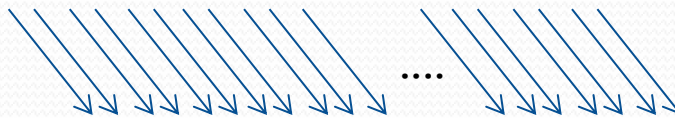
Транзакции

- Транзакция – выполнение загрузки из глобальной памяти сплошного отрезка в 128 байт, с началом кратным 128 (**naturally aligned**)
- Инструкция обращения в память выполняется одновременно для всех нитей варпа (**SIMT**)
 - Выполняется столько транзакций, сколько нужно для покрытия обращений всех нитей варпа
 - Если нужен один байт – все равно загрузится 128



Шаблоны доступа

Обращения нитей варпа



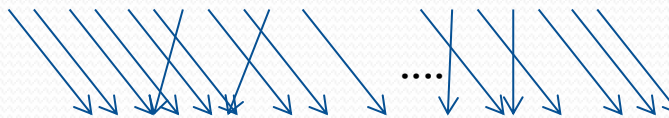
Все обращения
умещаются в одну
транзакцию

512

640

768

Обращения нитей варпа



Порядок не важен,
главное, чтобы попадали
в одну кеш-линию

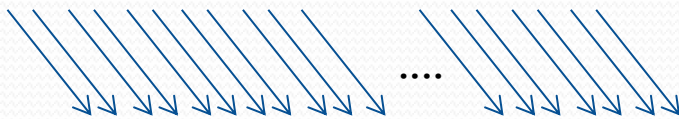
512

640

768

Шаблоны доступа

Обращения нитей варпа



512

640

768

Запрашивается 128 байт, но с не выровненного адреса
2 транзакции - 256 байт

Обращения нитей варпа



512

640

768

Запрашивается 128 байт, но обращения разбросаны в пределах трёх кеш-линий -
3 транзакции – 384 байта

Кеш-линии

- Ядра взаимодействуют не с памятью напрямую, а с кешами
- Транзакция – выполнение загрузки кеш-линии
 - У кеша **L1** кеш-линии **128** байт, у **L2** - **32** байта, **naturally aligned**
 - Кеш грузит из памяти всю кеш-линию, даже если нужен один байт
- Можно обращаться в память минуя кеш L1
 - Транзакции будут по 32 байта

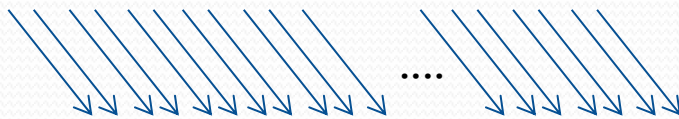


Включение \ отключение L1

- Кеширование в L1 можно отключить **при компиляции**
 - `nvcc -Xptxas -dlcm=ca`
с кешированием в L1 (по умолчанию)
 - `nvcc -Xptxas -dlcm=cg`
В бинарном коде обращения в глобальную память будут транслированы в инструкции, не использующие кеш L1 при выполнении
- Различия именно на уровне бинарного кода – другие инструкции ассемблера

Шаблоны доступа: L1 выключен

Обращения нитей варпа



Запрашивается 128 байт, но с невыровненного адреса. Умещаются в четыре кеш-линии L2

4 транзакции по 32 байта – 128 байт

512

576

640

768

Обращения нитей варпа



Запрашивается 128 байт, но обращения разбросаны по пяти кеш-линиям L2

5 транзакций по 32 – 160 байт

512

608

640

768

Выводы

- Если в ядре не используется общая память (см. далее), то заведомо стоит включить `cudaFuncCachePreferL1`
- Если разреженный доступ – кеширование в L1 отключаем
- В общем случае, стоит проверить производительность работы всех 4-х вариантов:
 - `(-dlcm=ca, -clcm=cg)x(16KB, 48KB)` !

Транзакции: L1 включен

Варп



512

640

768

L1



512

576

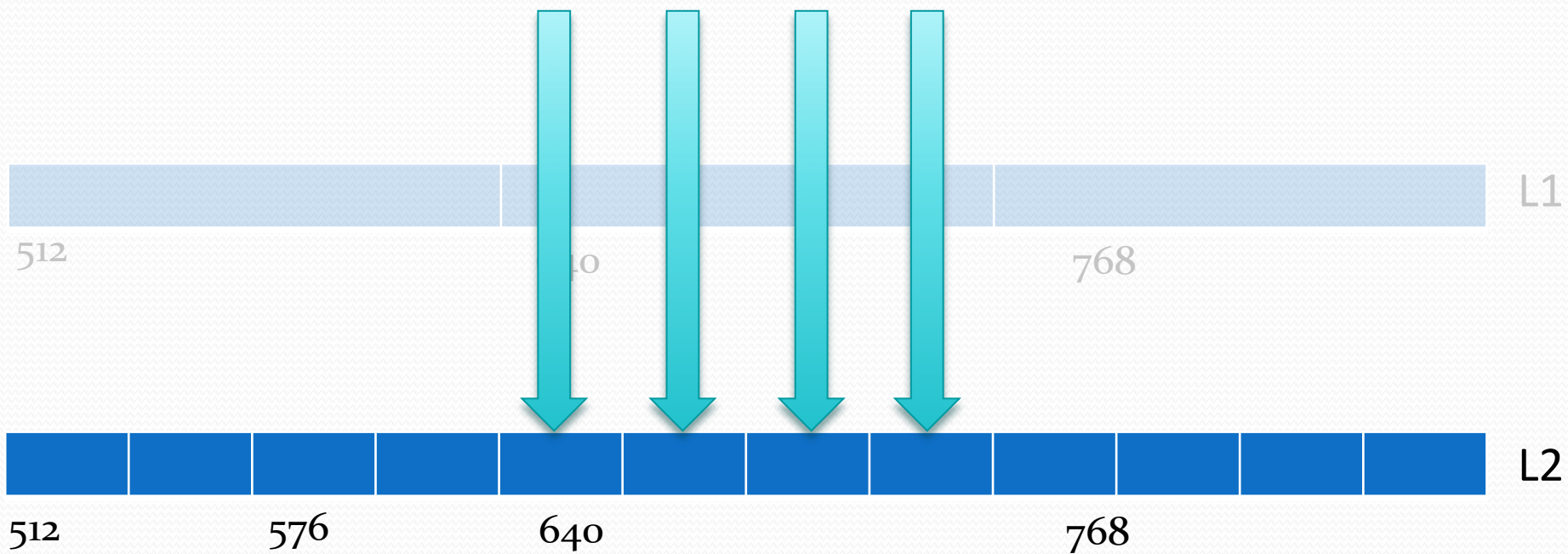
640

768

L2

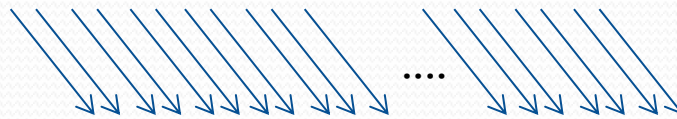
Транзакции: L1 выключен

Варп



Шаблоны доступа: L1 включен

Обращения нитей варпа



Все обращения
умещаются в одну
транзакцию



Обращения нитей варпа

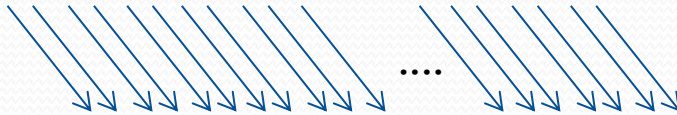


Порядок не важен,
главное, чтобы попадали
в одну кеш-линию



Шаблоны доступа: L1 включен

Обращения нитей варпа



Запрашивается 128 байт, но с не выровненного адреса
2 транзакции - 256 байт

512

640

768

Обращения нитей варпа



Запрашивается 128 байт, но обращения разбросаны в пределах трёх кеш-линий -
3 транзакции - 384 байта

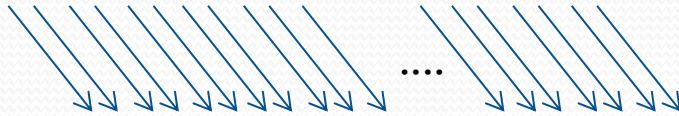
512

640

768

Шаблоны доступа: L1 выключен

Обращения нитей варпа



512

576

640

768

Запрашивается 128 байт, но с невыровненного адреса. Умещаются в четыре кеш-линии L2

4 транзакции по 32 байта – 128 байт

Обращения нитей варпа



512

608

640

768

Запрашивается 128 байт, но обращения разбросаны по пяти кеш-линиям L2

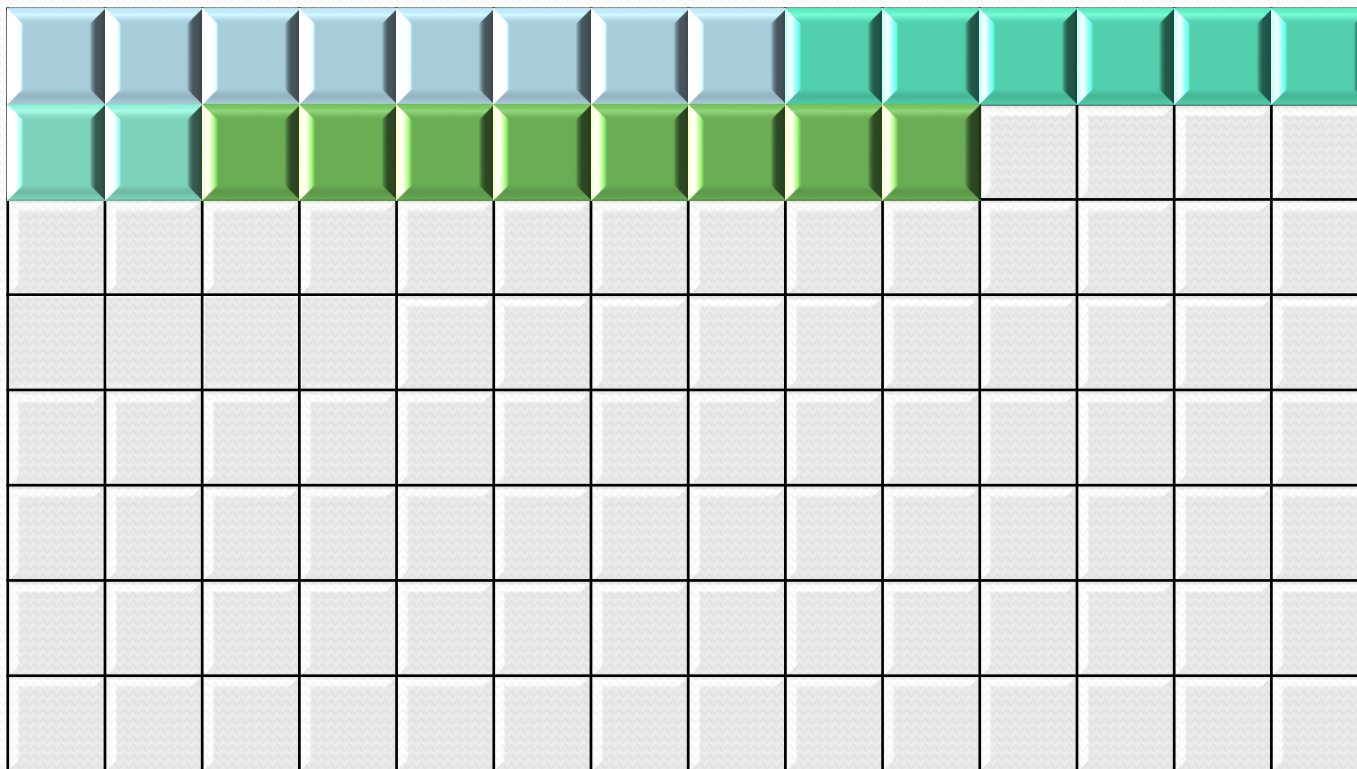
5 транзакций по 32 – 160 байт

Матрицы и глобальная память

Пусть транзакция – $8 \times 4 = 32$ байта, адрес транзакции выровнен по 32 байта

Если ширина матрицы не кратна 32 байтам – большая часть строк не выровнена

0 7 14



Матрицы и глобальная память

- Матрицы хранятся в линейном виде, по строкам
- Пусть длина строки матрицы – 480 байт (120 float)
 - обращение – `matrix[idy*120 + idx]`

Адрес начала каждой строки, кроме первой, не выровнен по 128 –
обращения варпов в память будут выполняться за **2 транзакции**



512

Строка 0

992

Строка 1

1472

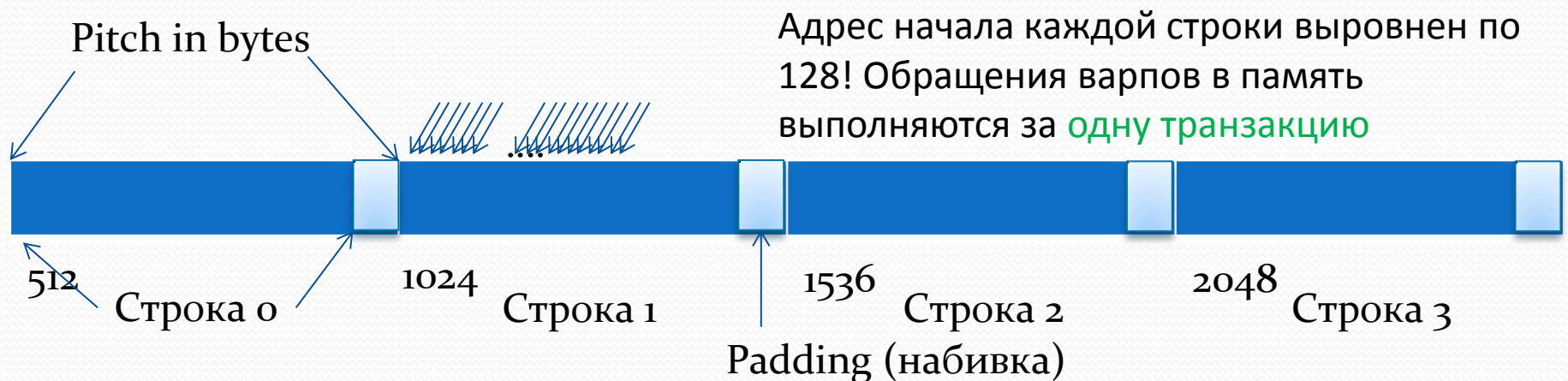
Строка 2

1952

Строка 3

Матрицы и глобальная память

- Дополним каждую строку до размера, кратного 128 байтам – в нашем случае, $480 + 32 = 512$, это наш **pitch** – фактическая ширина в байтах
- Эти байты никак не будут использоваться, т.е. $32/512=6\%$ лишней памяти будет выделено (Но для больших матриц эта доля будет существенно меньше)
- Зато каждая строка будет выровнена по 128 байт
 - Обращение `matrix[idy*128+ idx]`



Выделение памяти с «паддингом»

- `cudaError_t cudaMallocPitch (void ** devPtr, size_t * pitch, size_t width, size_t height)`
 - `width` – логическая ширина матрицы в **байтах**
 - Выделяет не менее `width * height` **байтов**, может добавить в конец строк набивку, с целью соблюдения выравнивания начала строк
 - сохраняет указатель на память в (`*devPtr`)
 - сохраняет фактическую ширину строк в байтах в (`*pitch`)
- Адрес элемента (Row, Column) матрицы, выделенной при помощи `cudaMallocPitch`:

```
T* pElement = (T*) ((char*) devPtr + Row * pitch) + Column;
```



Копирование в матрицу с padding-ом

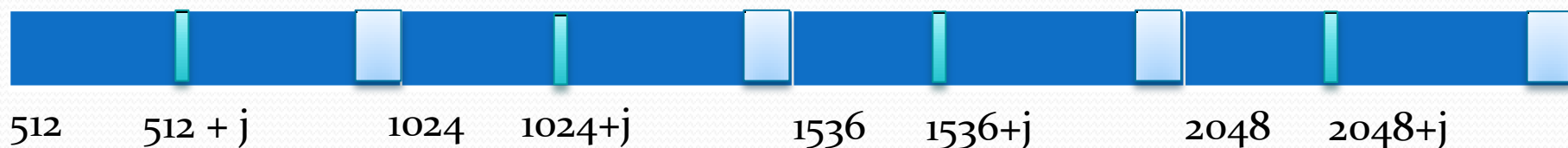
- `cudaError_t cudaMemcpy2D (void* dst, size_t dpitch, const void* src, size_t spitch, size_t width, size_t height, cudaMemcpyKind kind)`
 - `dst` - указатель на матрицу, в которую нужно копировать ,
`dpitch` – *фактическая* ширина её строк в байтах
 - `src` - указатель на матрицу из которой нужно копировать,
`spitch` – *фактическая* ширина её строк в байтах
 - `width` – сколько *байтов* каждой строки нужно копировать
 - `height` – число строк
 - `kind` – направление копирования (как в обычном `cudaMemcpy`)
- Данная функция из начала каждой строки исходной матрицы копируется по `width` байтов. Всего копируется `width*height` байтов, при этом
 - Адрес строки с индексом `Row` определяется по фактической ширине:
`(char*)src + Row* spitch` – в матрице-источнике
`(char*)dst + Row* dpitch` – в матрице-получателе

Обращение к матрице по столбцам?

- Матрица расположена по строкам, а обращение идёт по столбцам

Обращения нитей варпа

Каждая нить варпа обращается в свою строку к элементу в столбце j



Если матрица имеет размер больше 128 байт, то эти обращения ни за что не «влезут» в одну транзакцию!

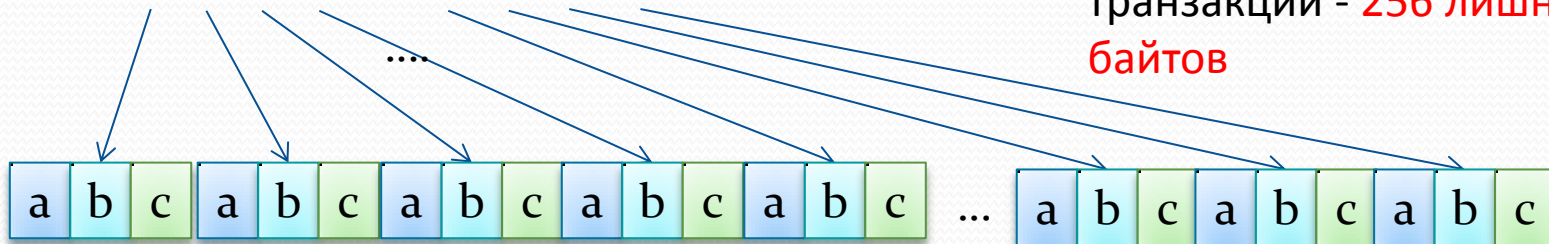
Транспонировать!

- Решение – хранить матрицу в транспонированном виде!
 - В этом случае обращения по столбцам превратятся в обращения к последовательным адресам
 - Выделять память под транспонированную матрицу также через `cudaMallocPitch`

Массивы структур?

```
struct example {  
    int a;  
    int b;  
    int c;  
}  
  
__global__ void kernel(example * arrayOfExamples) {  
    int idx = threadIdx.x + blockIdx.x * blockDim.x;  
    arrayOfExamples[idx].c =  
        arrayOfExamples[idx].b + arrayOfExamples[idx].a;  
}
```

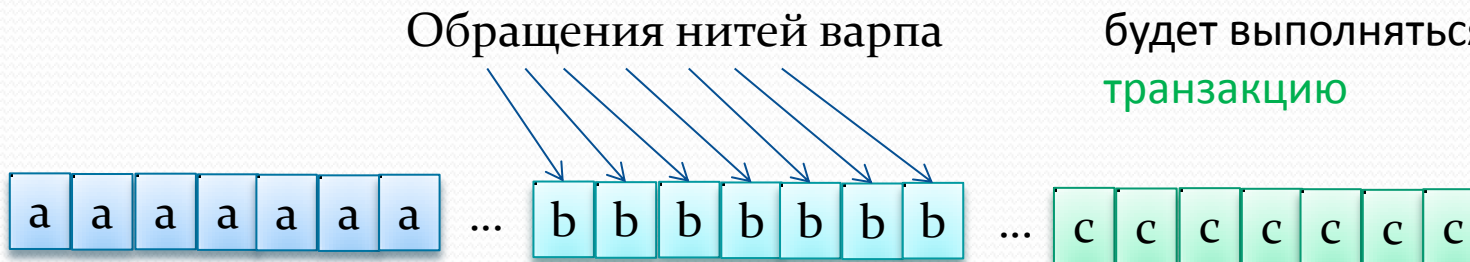
Обращения нитей варпа



Обращение варпа в память
будет выполняться в три
транзакции - **256 лишних
байтов**

Структура с массивами!

```
struct example {  
    int *a;  
    int *b;  
    int *c;  
}  
  
__global__ void kernel(example arrayOfExamples) {  
    int idx = threadIdx.x + blockIdx.x * blockDim.x;  
    arrayOfExamples.c[idx] =  
        arrayOfExamples.b[idx] + arrayOfExamples.a[idx];  
}
```

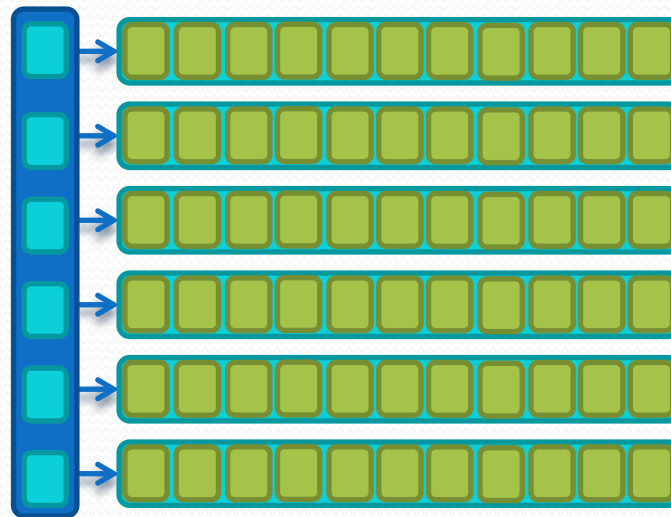


Обращение варпа в память
будет выполняться за **одну**
транзакцию

Проблема Косвенной адресации

- Использование косвенной адресации нежелательно, поскольку требует двух чтений из памяти (сначала $A[i]$, потом $A[i][j]$)

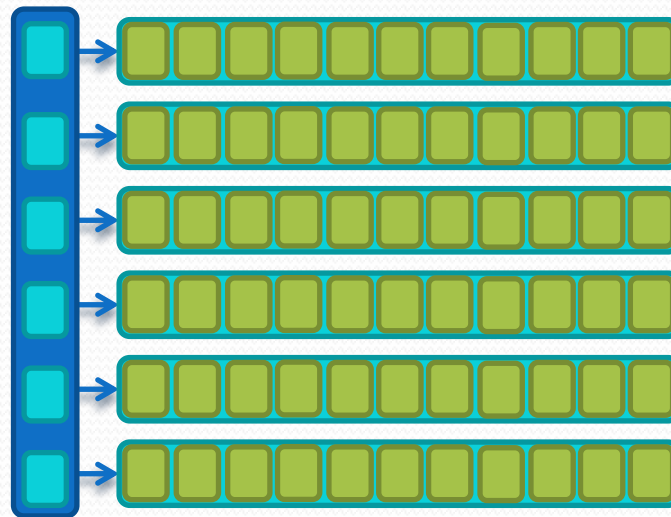
```
float **A;  
A[i][j] = 1;
```



Проблема Косвенной адресации

- $A[i]$ для разных нитей варпа скорее всего будут в одной кеш-линии -> одно обращение к кешу
- Но $A[i,j]$ в общем случае могут быть «разбросаны»

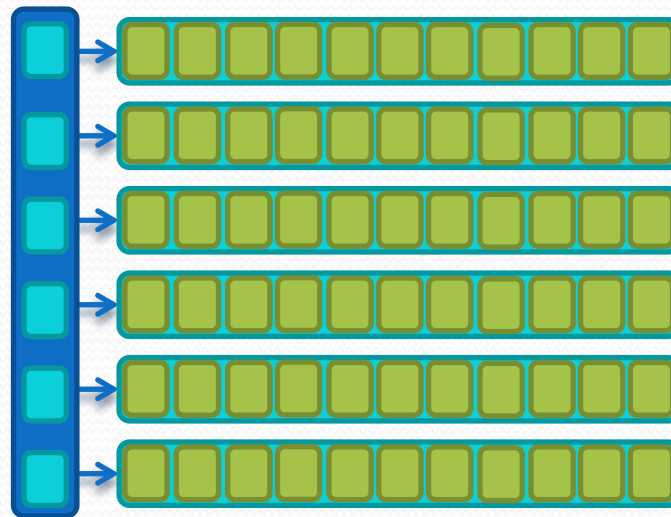
```
float **A;  
A[i][j] = 1;
```



Проблема Косвенной адресации

- Принципиального решения нет!
- Скорее всего придется переработать алгоритм

```
float **A;  
A[i][j] = 1;
```



Выводы

- Обращения нитей варпа в память должны быть пространственно-локальными
- Начала строк матрицы должны быть выровнены
- Массивы структур -> структура с массивами
- 16KB vs 48KB L1
- Избегаем косвенной адресации
- Избегаем обращений нитей варпа к столбцу матрицы
- В случае сильно разреженного доступа проверяем работу с отключенным кешем