

Программно-аппаратный стек CUDA. Иерархия памяти в CUDA. Глобальная память

⌘ Лекторы:

☑ Боресков А.В. (ВМиК МГУ)

☑ Харламов А.А. (NVidia)

Примеры многоядерных систем



⌘ На первой лекции мы рассмотрели

- ☑ Intel Core 2 Duo

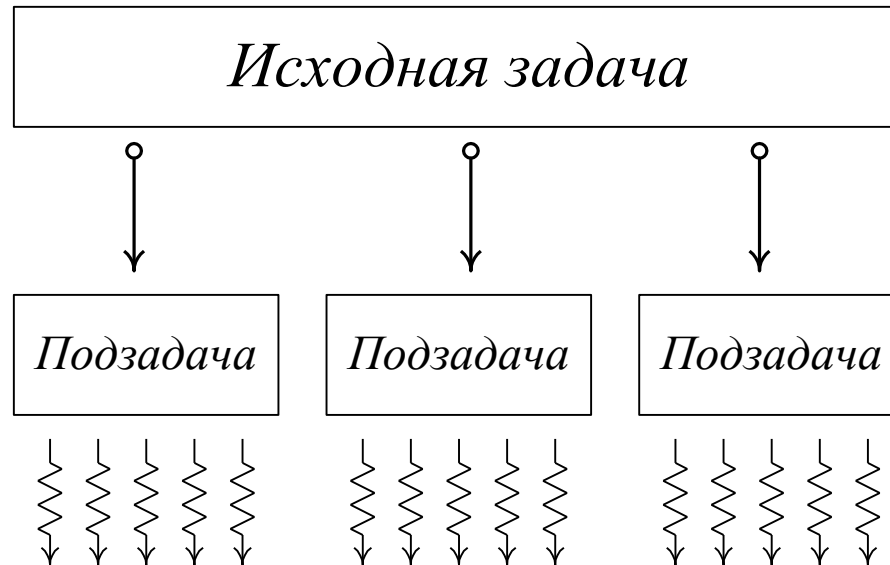
- ☑ SMP

- ☑ Cell

- ☑ BlueGene/L

- ☑ G80 / Tesla / Fermi

Подход CUDA



Исходная задача разбивается на подзадачи, которые можно решать независимо друг от друга.

Каждая из этих подзадач решается набором взаимодействующих между собой нитей

SIMT (Single Instruction, Multiple Threads)



- ⌘ Параллельно на каждом SM выполняется большое число отдельных нитей (*threads*)
- ⌘ Нити подряд разбиваются на *warp*'ы (по 32 нити) и SM управляет выполнением *warp*'ов
- ⌘ Нити в пределах одного *warp*'а выполняются физически параллельно
- ⌘ Большое число *warp*'ов покрывает латентность

Технические детали



⌘ **RTM CUDA Programming Guide**

⌘ **Run CUDAHelloWorld**

☑ Печатает аппаратно зависимые параметры

☒ Размер shared памяти

☒ Кол-во SM

☒ Размер warp'a

☒ Кол-во регистров на SM

☒ Т.д.

Программная модель CUDA



- ⌘ Код состоит как из последовательных, так и из параллельных частей
- ⌘ Последовательные части кода выполняются на CPU
- ⌘ Массивно-параллельные части кода выполняются на GPU как ядра

Программная модель CUDA



- ⌘ GPU (*device*) это вычислительное устройство, которое:
 - ☑ Является сопроцессором к CPU (*host*)
 - ☑ Имеет собственную память (DRAM)
 - ☑ Выполняет одновременно **очень много** нитей

Программная модель CUDA



- ⌘ Последовательные части кода выполняются на CPU
- ⌘ Массивно-параллельные части кода выполняются на GPU как ядра
- ⌘ Отличия нитей между CPU и GPU
 - ⏏ Нити на GPU очень «легкие»
 - ⏏ HW планировщик задач
 - ⏏ Для полноценной загрузки GPU нужны тысячи нитей
 - ⏏ Для покрытия латентностей операций чтения / записи
 - ⏏ Для покрытия латентностей sfu инструкций

Программная модель CUDA



- ⌘ Параллельная часть кода выполняется как большое количество нитей (*threads*)
- ⌘ Нити группируются в блоки (*blocks*) фиксированного размера
- ⌘ Блоки объединяются в сеть блоков (*grid*)
- ⌘ Ядро выполняется на сетке из блоков
- ⌘ Каждая нить и блок имеют свой уникальный идентификатор

Программная модель CUDA



⌘ Десятки тысяч потоков

```
for ( int ix = 0; ix < nx; ix++ )
{
    pData[ix] = f(ix);
}
```

```
for ( int ix = 0; ix < nx; ix++ )
    for ( int iy = 0; iy < ny; iy++ )
    {
        pData[ix + iy * nx] = f(ix) * g(iy);
    }
```

```
for ( int ix = 0; ix < nx; ix++ )
    for ( int iy = 0; iy < ny; iy++ )
        for ( int iz = 0; iz < nz; iz++ )
        {
            pData[ix + (iy + iz * ny) * nx] = f(ix) * g(iy) * h(iz);
        }
```

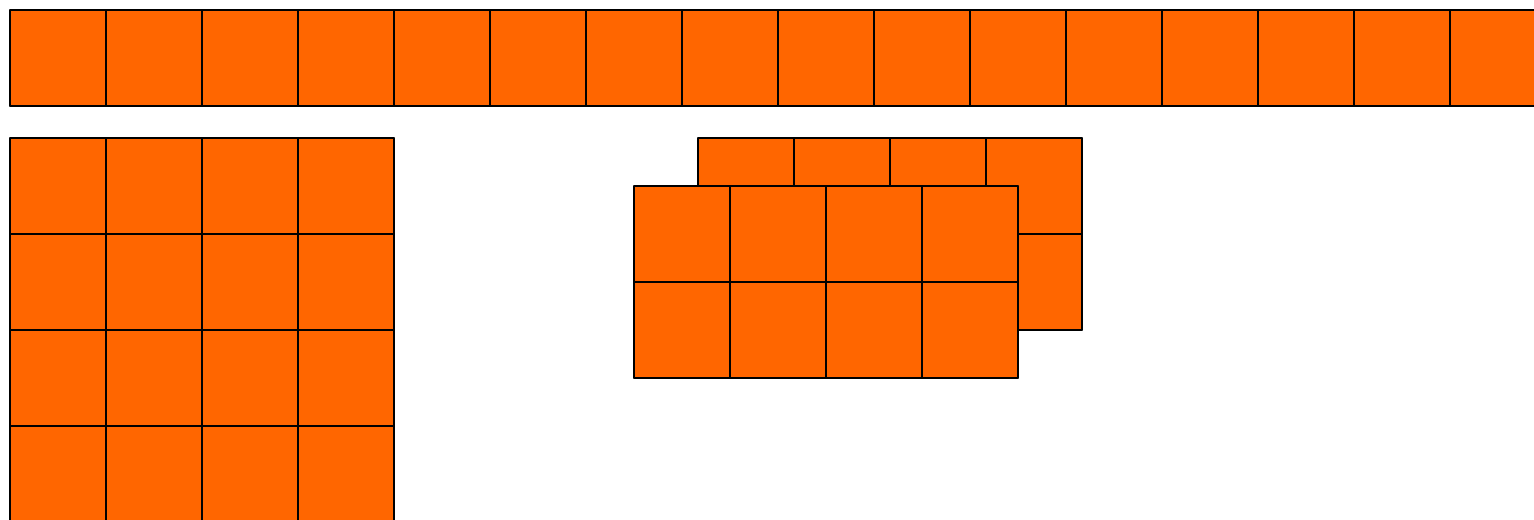
Программная модель CUDA

⌘ Потоки в CUDA объединяются в блоки:

☑ Возможна 1D, 2D, 3D топология блока

☑ Общее кол-во потоков в блоке ограничено

☑ В текущем HW это 512 потоков



Программная модель CUDA



⌘ Потоки в блоке могут разделять ресурсы со своими соседями

```
float data[N];
```

```
for ( int ix = 0; ix < nx; ix++ )  
    data[ix] = f(ix, data[ix / n]);
```

Программная модель CUDA

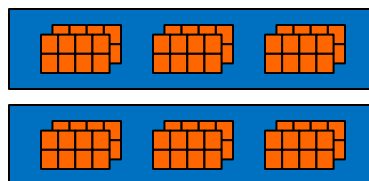
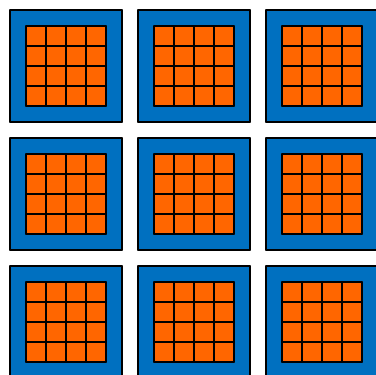
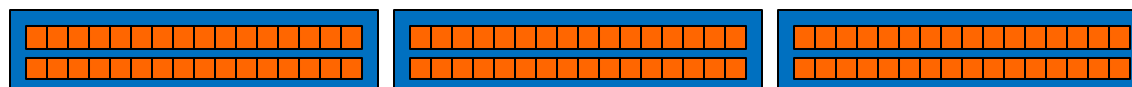


- ⌘ Блоки могут использовать *shared* память
 - ☑ Т.к. блок целиком выполняется на одном SM
 - ☑ Объем *shared* памяти ограничен и зависит от HW
- ⌘ Внутри Блока потоки могут синхронизоваться
 - ☑ Т.к. блок целиком выполняется на одном SM

Программная модель CUDA

⌘ Блоки потоков объединяются в сетку
(*grid*) потоков

☑ Возможна 1D, 2D топология сетки блоков
потоков



Синтаксис CUDA

- ⌘ CUDA – это расширение языка C/C++
 - ☑ [+] спецификаторы для функций и переменных
 - ☑ [+] новые встроенные типы
 - ☑ [+] встроенные переменные (внутри ядра)
 - ☑ [+] директива для запуска ядра из C кода
- ⌘ Как скомпилировать CUDA код
 - ☑ [+] **nvcc** компилятор
 - ☑ [+] .cu расширение файла

Синтаксис CUDA

Спецификаторы

⌘ Спецификатор функций

Спецификатор	Выполняется на	Может вызываться из
<code>__device__</code>	device	device
<code>__global__</code>	device	host
<code>__host__</code>	host	host

⌘ Спецификатор переменных

Спецификатор	Находится	Доступна	Вид доступа
<code>__device__</code>	device	device	R
<code>__constant__</code>	device	device / host	R / W
<code>__shared__</code>	device	block	RW / <code>__syncthreads()</code>

Расширения языка C



- ⌘ Спецификатор `__global__` соответствует ядру
 - ⌘ Может возвращать только `void`
- ⌘ Спецификаторы `__host__` и `__device__` могут использоваться одновременно
 - ⌘ Компилятор сам создаст версии для CPU и GPU
- ⌘ Спецификаторы `__global__` и `__host__` не могут быть использованы одновременно

Расширения языка C



Ограничения на функции, выполняемые на GPU:

- ⌘ Нельзя брать адрес (за исключением `__global__`)
- ⌘ Не поддерживается рекурсия
- ⌘ Не поддерживаются `static`-переменные внутри функции
- ⌘ Не поддерживается переменное число входных аргументов

Расширения языка C



Ограничения на спецификаторы переменных:

- ⌘ Нельзя применять к полям структуры или `union`
- ⌘ Не могут быть `extern`
- ⌘ Запись в `__constant__` может выполнять только CPU через специальные функции
- ⌘ `__shared__` - переменные не могут инициализироваться при объявлении

Расширения языка C

Новые типы данных:

⌘ 1/2/3/4-мерные вектора из базовых
ТИПОВ

⌘ (u)char, (u)int, (u)short, (u)long, longlong

⌘ float, double

⌘ dim3 – uint3 с нормальным
конструкторов, позволяющим задавать
не все компоненты

⌘ Не заданные инициализируются единицей

Расширения языка C

```
int2    a = make_int2 ( 1, 7 );  
float4  b = make_float4 ( a.x, a.y, 1.0f, 7 );  
float2  x = make_float2 ( b.z, b.w );  
dim3    grid    = dim3 ( 10 );  
dim3    blocks  = dim3 ( 16, 16 );
```

Для векторов не определены
покомпонентные операции

Для **double** и **longlong** возможны только
вектора размера 1 и 2.

Синтаксис CUDA

Встроенные переменные

⌘ Сравним CPU код vs CUDA kernel:

```
float * data;  
for ( int i = 0; i < n; i++ )  
{  
    data [x] = data[i] + 1.0f;  
}
```

Пусть $n_x = 2048$
Пусть в блоке 256
ПОТОКОВ

→ кол-во блоков =
 $2048 / 256 = 8$



```
__global__ void incKernel ( float * data )  
{  
    [ 0 .. 7 ]      [ == 256 ]      [ 0 .. 255 ]  
    int idx = blockIdx.x * blockDim.x + threadIdx.x;  
    data [idx] = data [idx] + 1.0f;  
}
```

Синтаксис CUDA

Встроенные переменные

⌘ В любом CUDA kernel'е доступны:

⊡ `dim3` `gridDim;`

⊡ `uint3` `blockIdx;`

⊡ `dim3` `blockDim;`

⊡ `uint3` `threadIdx;`

⊡ `int` `warpSize;`

`dim3` – встроенный тип,
который используется для
задания размеров kernel'а
По сути – это `uint3`.

Синтаксис CUDA

Директивы запуска ядра

⌘ Как запустить ядро с общим кол-во
тредов равным nx?

```
float * data;
```

```
dim3 threads ( 256 );
```

Неявно предполагаем,

```
dim3 blocks ( nx / 256 );
```

что nx кратно 256

```
incKernel<<<blocks, threads>>> ( data );
```

<<< , >>> угловые скобки, внутри которых задаются
параметры запуска ядра:

- Кол-во блоке в сетке
- Кол-во потоков в блоке
- ...

Расширения языка C

Общий вид команды для запуска ядра

```
incKernel<<<bl, th, ns, st>>> ( data );
```

⌘ *bl* – число блоков в сетке

⌘ *th* – число нитей в сетке

⌘ *ns* – количество дополнительной shared-памяти, выделяемое блоку

⌘ *st* – поток, в котором нужно запустить ядро

Как скомпилировать CUDA код

⌘ NVCC – компилятор для CUDA

☒ Основными опциями команды **nvcc** являются:

☒ **-deviceemu** - компиляция в режиме эмуляции, весь код будет выполняться в многопоточном режиме на CPU и можно использовать обычный отладчик (хотя не все ошибки могут проявиться в таком режиме)

☒ **--use_fast_math** - заменить все вызовы стандартных математических функций на их быстрые (но менее точные) аналоги

☒ **-o <outputFileName>** - задать имя выходного файла

⌘ CUDA файлы обычно носят расширение .cu

Основы CUDA host API



Два API

- ⌘ Низкоуровневый driver API (cu*)
- ⌘ Высокоуровневый runtime API (cuda*)
 - ⌘ Реализован через driver API
- ⌘ Не требуют явной инициализации
- ⌘ Все функции возвращают значение типа `cudaError_t`
 - ⌘ `cudaSuccess` в случае успеха

Основы CUDA API



Многие функции API асинхронны:

- ⌘ Запуск ядра
- ⌘ Копирование при помощи функций *Async
- ⌘ Копирование device <-> device
- ⌘ Инициализация памяти

CUDA Compute Capability



Возможности GPU обозначаются при помощи *Compute Capability*, например 1.1

- ⌘ Старшая цифра соответствует архитектуре
- ⌘ Младшая – небольшим архитектурным изменениям
- ⌘ Можно получить из полей *major* и *minor* структуры ***cudaDeviceProp***

Получение информации о GPU

```
int main ( int argc, char * argv [] )
{
    int                deviceCount;
    cudaDeviceProp devProp;

    cudaGetDeviceCount ( &deviceCount );
    printf              ( "Found %d devices\n", deviceCount );

    for ( int device = 0; device < deviceCount; device++ )
    {
        cudaGetDeviceProperties ( &devProp, device );
        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 );
        printf ( "Shared memory per block: %d\n", devProp.sharedMemPerBlock );
        printf ( "Registers per block      : %d\n", devProp.regsPerBlock );
        printf ( "Warp size                : %d\n", devProp.warpSize );
        printf ( "Max threads per block   : %d\n", devProp.maxThreadsPerBlock );
        printf ( "Total constant memory    : %d\n", devProp.totalConstMem );
    }
    return 0;
}
```

Compute Capability

GPU	Compute Capability
Tesla S1070	1.3
GeForce GTX 260	1.3
GeForce 9800 GX2	1.1
GeForce 9800 GTX	1.1
GeForce 8800 GT	1.1
GeForce 8800 GTX	1.0

Compute Capability



⌘ Compute Caps. – доступная версия CUDA

☒ Разные возможности HW

☒ Пример:

☒ В 1.1 добавлены атомарные операции в global memory

☒ В 1.2 добавлены атомарные операции в shared memory

☒ В 1.3 добавлены вычисления в double

⌘ Узнать доступный Compute Caps. можно через `cudaGetDeviceProperties()`

☒ См. `CUDAHelloWorld`

⌘ Сегодня Compute Caps:

☒ Влияет на правила работы с глобальной памятью

Компиляция программ



- ⌘ Используем утилиту make/nmake, явно вызывающую nvcc
- ⌘ Используем MS Visual Studio
 - ⌘ Подключаем cuda.rules
 - ⌘ Используем CUDA Wizard
(<http://sourceforge.net/projects/cudawizard>)

Типы памяти в CUDA

Тип памяти	Доступ	Уровень выделения	Скорость работы
Регистры	R/W	Per-thread	Высокая(on-chip)
Локальная	R/W	Per-thread	Низкая (DRAM)
Shared	R/W	Per-block	Высокая(on-chip)
Глобальная	R/W	Per-grid	Низкая (DRAM)
Constant	R/O	Per-grid	Высокая(L1 cache)
Texture	R/O	Per-grid	Высокая(L1 cache)

Типы памяти в CUDA



- ⌘ Самая быстрая – *shared* (on-chip) и регистры
- ⌘ Самая медленная – глобальная (DRAM)
- ⌘ Для ряда случаев можно использовать кэшируемую константную и текстурную память
- ⌘ Доступ к памяти в CUDA идет отдельно для каждой половины warp'a (*half-warp*)

Работа с памятью в CUDA

Основа оптимизации – оптимизация работы с памятью:

- ⌘ Максимальное использование *shared*-памяти
- ⌘ Использование специальных паттернов доступа к памяти, гарантирующих эффективный доступ
 - ☑ Паттерны работают независимо в пределах каждого *half-warp'a*

Работа с глобальной памятью в CUDA

Пример работы с глобальной памятью

```
float * devPtr;                // pointer device memory
                                // allocate device memory
cudaMalloc ( (void **) &devPtr, 256*sizeof ( float ) );

                                // copy data from host to device memory
cudaMemcpy ( devPtr, hostPtr, 256*sizeof ( float ),
cudaMemcpyHostToDevice );

                                // process data

                                // copy results from device to host
cudaMemcpy ( hostPtr, devPtr, 256*sizeof( float ),
cudaMemcpyDeviceToHost );

                                // free device memory
cudaFree   ( devPtr );
```

Работа с глобальной памятью в CUDA

Функции для работы с глобальной памятью

```
cudaError_t cudaMalloc          ( void ** devPtr, size_t size );
cudaError_t cudaMallocPitch    ( void ** devPtr, size_t * pitch,
                                size_t width, size_t height );

cudaError_t cudaFree           ( void * devPtr );
cudaError_t cudaMemcpy         ( void * dst, const void * src,
                                size_t count,
                                enum cudaMemcpyKind kind );

cudaError_t cudaMemcpyAsync    ( void * dst, const void * src,
                                size_t count,
                                enum cudaMemcpyKind kind,
                                cudaStream_t stream );

cudaError_t cudaMemset         ( void * devPtr, int value,
                                size_t count );
```

Пример: умножение матриц



- ⌘ Произведение двух квадратных матриц A и B размера $N \times N$, N кратно 16
- ⌘ Матрицы расположены в глобальной памяти
- ⌘ По одной нити на каждый элемент произведения
- ⌘ 2D блок – 16×16
- ⌘ 2D *grid*

Умножение матриц.

Простейшая реализация.

```
#define BLOCK_SIZE 16

__global__ void matMult ( float * a, float * b, int n, float * c )
{
    int    bx  = blockIdx.x;
    int    by  = blockIdx.y;
    int    tx  = threadIdx.x;
    int    ty  = threadIdx.y;
    float  sum = 0.0f;
    int    ia  = n * BLOCK_SIZE * by + n * ty;
    int    ib  = BLOCK_SIZE * bx + tx;
    int    ic  = n * BLOCK_SIZE * by + BLOCK_SIZE * bx;

    for ( int k = 0; k < n; k++ )
        sum += a [ia + k] * b [ib + k*n];

    c [ic + n * ty + tx] = sum;
}
```

Умножение матриц.

Простейшая реализация.

```
int          numBytes = N * N * sizeof ( float );
float        * adev, * bdev, * cdev ;
dim3         threads ( BLOCK_SIZE, BLOCK_SIZE );
dim3         blocks ( N / threads.x, N / threads.y);

cudaMalloc   ( (void**) &adev, numBytes ); // allocate DRAM
cudaMalloc   ( (void**) &bdev, numBytes ); // allocate DRAM
cudaMalloc   ( (void**) &cdev, numBytes ); // allocate DRAM
                                                // copy from CPU to DRAM

cudaMemcpy   ( adev, a, numBytes, cudaMemcpyHostToDevice );
cudaMemcpy   ( bdev, b, numBytes, cudaMemcpyHostToDevice );

matMult<<<blocks, threads>>> ( adev, bdev, N, cdev );

cudaThreadSynchronize();
cudaMemcpy   ( c, cdev, numBytes, cudaMemcpyDeviceToHost );
                                                // free GPU memory

cudaFree     ( adev );
cudaFree     ( bdev );
cudaFree     ( cdev );
```

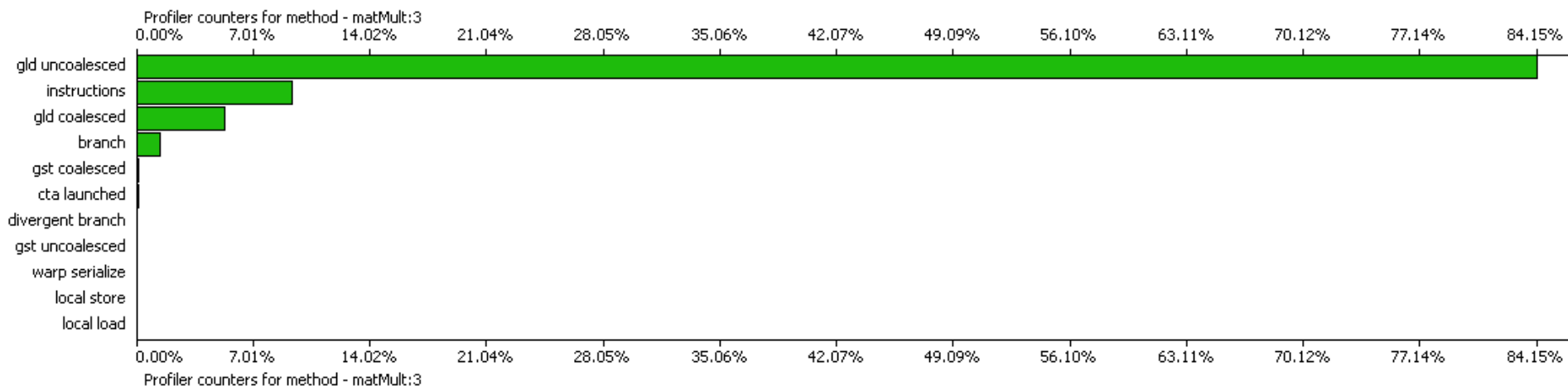
Простейшая реализация.



- ⌘ На каждый элемент
 - ⌘ $2 \cdot N$ арифметических операций
 - ⌘ $2 \cdot N$ обращений к глобальной памяти
- ⌘ *Memory bound* (тормозит именно доступ к памяти)

Используем CUDA Profiler

Profiler Counter Plot



⌘ Легко видно, что основное время (84.15%) ушло на чтение из глобальной памяти

⌘ Непосредственно вычисления заняли всего около 10%

Оптимизация работы с глобальной памятью.



- ⌘ Обращения идут через 32/64/128-битовые слова
- ⌘ При обращении к $t[i]$
 - ⊞ $sizeof(t[0])$ равен 4/8/16 байтам
 - ⊞ $t[i]$ выровнен по $sizeof(t[0])$
- ⌘ Вся выделяемая память всегда выровнена по 256 байт

Использование выравнивания.

```
struct vec3
{
    float x, y, z;
};
```

- ⌘ Размер равен 12 байт
- ⌘ Элементы массива не будут выровнены в памяти

```
struct __align__(16) vec3
{
    float x, y, z;
};
```


- ⌘ Размер равен 16 байт
- ⌘ Элементы массива всегда будут выровнены в памяти

Объединение запросов к глобальной памяти.



- ⌘ GPU умеет объединять ряд запросов к глобальной памяти в один блок (транзакцию)
- ⌘ Независимо происходит для каждого *half-warps*
- ⌘ Длина блока должна быть 32/64/128 байт
- ⌘ Блок должен быть выровнен по своему размеру

Объединение (coalescing) для GPU с CC 1.0/1.1



⌘ Нити обращаются к

☒ 32-битовым словам, давая 64-байтовый блок

☒ 64-битовым словам, давая 128-байтовый блок

⌘ Все 16 слов лежат в пределах блока

⌘ k -ая нить *half-warp*'а обращается к k -му слову блока

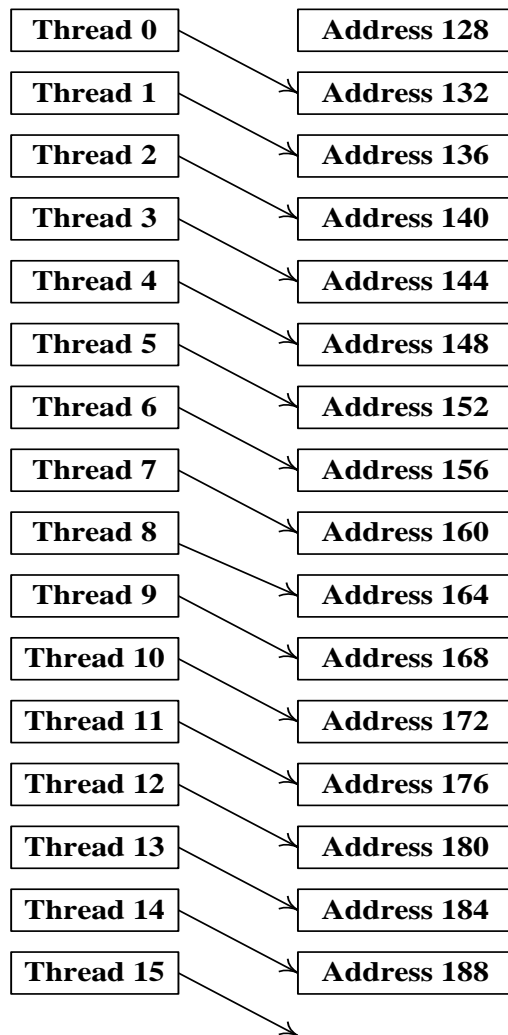
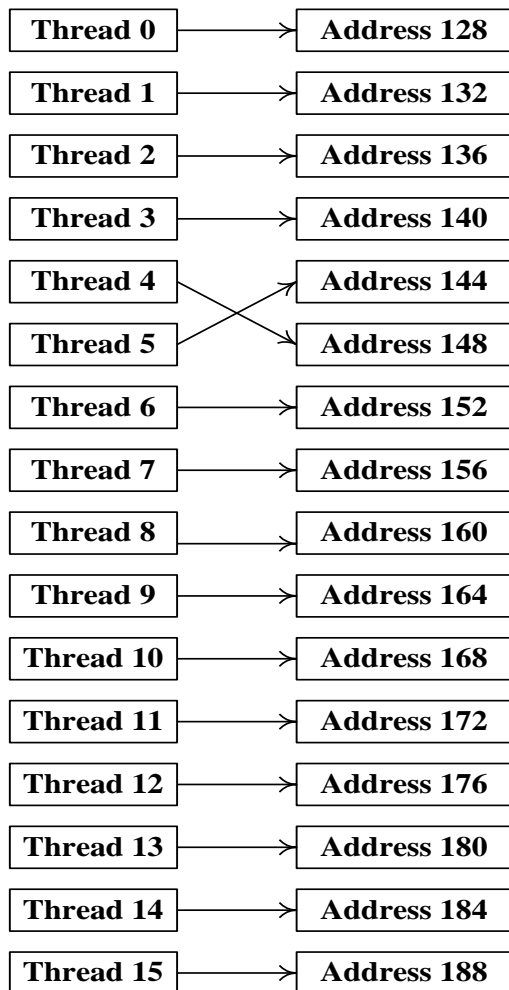
Объединение (coalescing) для GPU с CC 1.0/1.1

Coalescing

Thread 0	→	Address 128
Thread 1	→	Address 132
Thread 2	→	Address 136
Thread 3	→	Address 140
Thread 4	→	Address 144
Thread 5	→	Address 148
Thread 6	→	Address 152
Thread 7	→	Address 156
Thread 8	→	Address 160
Thread 9	→	Address 164
Thread 10	→	Address 168
Thread 11	→	Address 172
Thread 12	→	Address 176
Thread 13	→	Address 180
Thread 14	→	Address 184
Thread 15	→	Address 188


Thread 0	→	Address 128
Thread 1		Address 132
Thread 2	→	Address 136
Thread 3	→	Address 140
Thread 4		Address 144
Thread 5		Address 148
Thread 6	→	Address 152
Thread 7	→	Address 156
Thread 8	→	Address 160
Thread 9	→	Address 164
Thread 10	→	Address 168
Thread 11	→	Address 172
Thread 12		Address 176
Thread 13	→	Address 180
Thread 14	→	Address 184
Thread 15	→	Address 188

Объединение (coalescing) для GPU с CC 1.0/1.1



Not Coalescing

Объединение (coalescing) для GPU с CC 1.2/1.3



⌘ Нити обращаются к

☒ 8-битовым словам, дающим один 32-байтовый сегмент

☒ 16-битовым словам, дающим один 64-байтовый сегмент

☒ 32-битовым словам, дающим один 128-байтовый сегмент

⌘ Получающийся сегмент выровнен по своему размеру

Объединение (coalescing)

⌘ Если хотя бы одно условие не выполнено

☑ 1.0/1.1 – 16 отдельных транзакций

☑ 1.2/1.3 – объединяет их в блоки (2,3,...) и для каждого блока проводится отдельная транзакция

⌘ Для 1.2/1.3 порядок в котором нити обращаются к словам внутри блока не имеет значения (в отличии от 1.0/1.1)

Объединение (coalescing)



- ⌘ Можно добиться заметного увеличения скорости работы с памятью
- ⌘ Лучше использовать не массив структур, а набор массивов отдельных компонент — это позволяет использовать *coalescing*

Использование отдельных массивов

```
struct vec3
{
    float x, y, z;
};
vec3 * a;
```

```
float x = a [threadIdx.x].x;
float y = a [threadIdx.x].y;
float z = a [threadIdx.x].z;
```

```
float * ax, * ay, * az;
```

```
float x = ax [threadIdx];
float y = ay [threadIdx];
float z = az [threadIdx];
```

Не можем использовать
coalescing при чтении
данных

Поскольку нити
одновременно обращаются к
последовательно лежащим
словам памяти, то будет
происходить *coalescing*

Ресурсы нашего курса

⌘ CUDA.CS.MSU.SU

- ☒ Место для вопросов и дискуссий
- ☒ Место для материалов нашего курса
- ☒ Место для ваших статей!
 - ☒ Если вы нашли какой-то интересный подход!
 - ☒ Или исследовали производительность разных подходов и знаете, какой из них самый быстрый!
 - ☒ Или знаете способы сделать работу с CUDA проще!

⌘ <http://steps3d.narod.ru>

⌘ www.nvidia.ru

