



Иерархия памяти **CUDA.**

Глобальная память.

Параллельные решения задач  
умножения матриц и решения  
СЛАУ.

● Лекторы:

- Боресков А.В. (ВМиК МГУ)
- Харламов А.А. (Nvidia)

# 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)
- 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

# Умножение матриц



- Произведение двух квадратных матриц  $A$  и  $B$  размера  $N*N$ ,  $N$  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

cudaMemcpy  ( adev, a, numBytes, cudaMemcpyHostToDevice ); // from CPU to DRAM
cudaMemcpy  ( bdev, b, numBytes, cudaMemcpyHostToDevice ); // from CPU to DRAM

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

cudaMemcpy  ( c, cdev, numBytes, cudaMemcpyDeviceToHost );

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

# Простейшая реализация.



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



# 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


# Device 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

# Device Compute Capability

<b>GPU</b>	<b>Compute Capability</b>
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

# Объединение запросов к глобальной памяти.

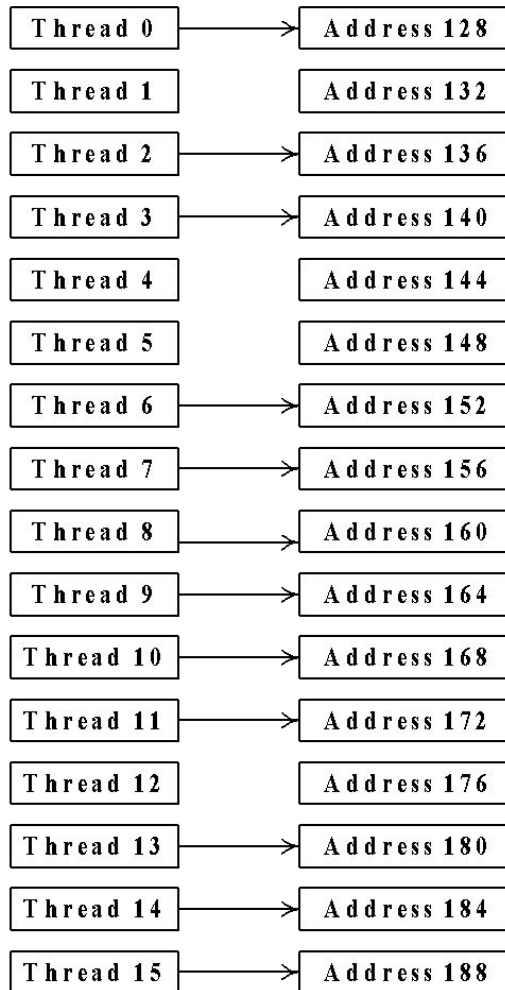
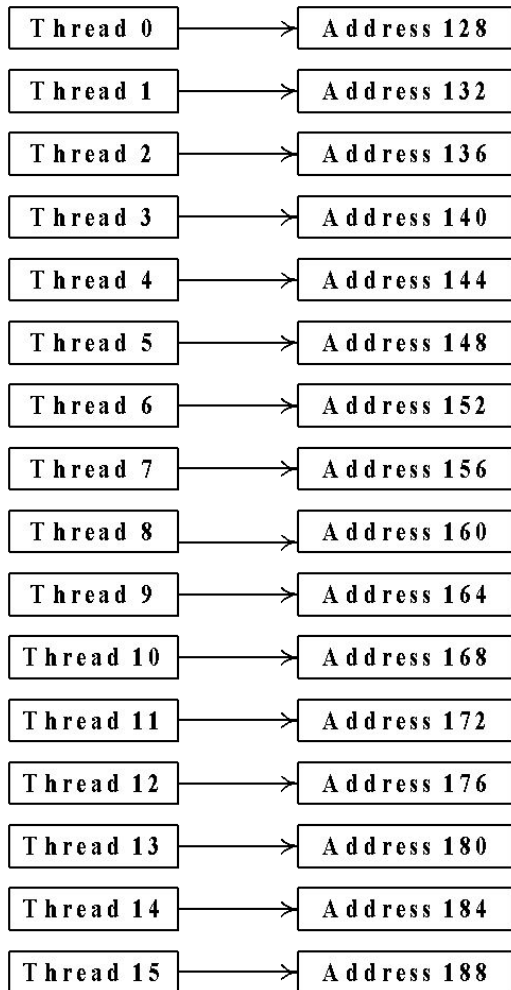


- GPU умеет объединять ряд запросов к глобальной памяти в один блок (транзакцию)
- Независимо происходит для каждого half-warp'a
- Длина блока должна быть 32/64/128 байт
- Блок должен быть выровнен по своему размеру

# Объединение (**coalescing**) для GPU с CC 1.0/1.1

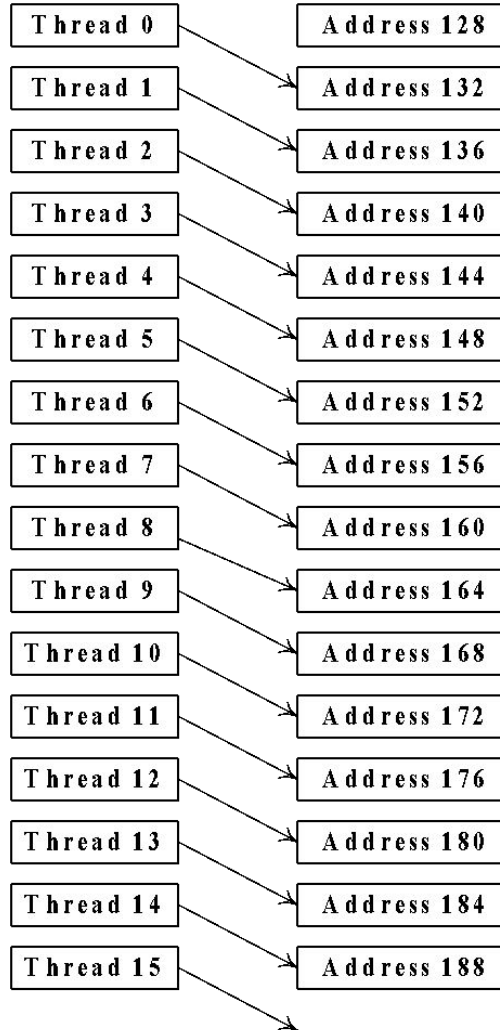
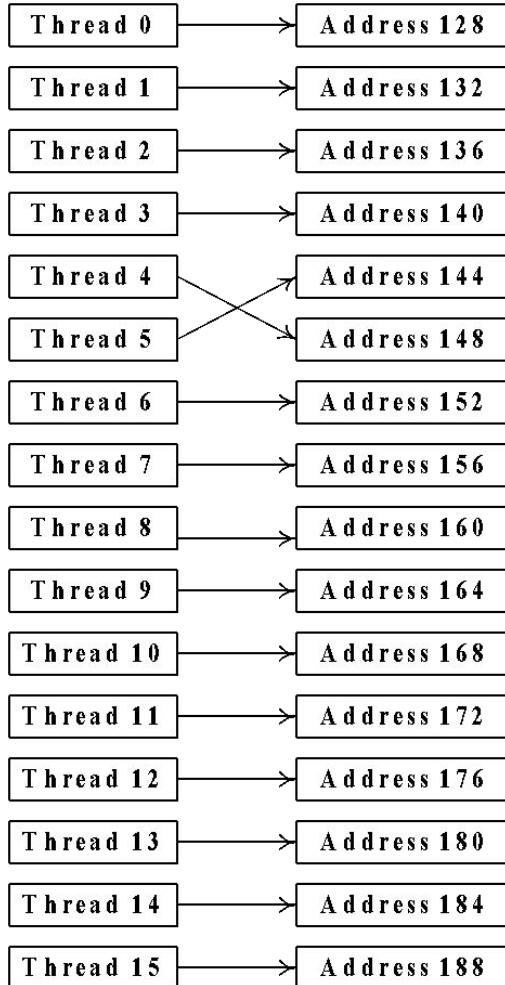
- Нити обращаются к
  - 32-битовым словам, давая 64-байтовый блок
  - 64-битовым словам, давая 128-байтовый блок
- Все 16 слов лежат в пределах блока
- $k$ -ая нить half-warps'a обращается к  $k$ -му слову блока

# Объединение (**coalescing**) для GPU с CC 1.0/1.1



**Coalescing**


# Объединение (**coalescing**) для GPU с CC 1.0/1.1



**Not Coalescing**



# Объединение (**coalescing**) для **GPU** с **СС 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 транзаций
  - 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**

# Решение системы линейных алгебраических уравнений

$$Ax=f,$$

$A$  – матрица размера  $N*N$ ,

$f$  – вектор размера  $N$

- Традиционные методы ориентированы на последовательное вычисление элементов и нам не подходят
- Есть еще итеративные методы

# Итеративные методы

$$x^{k+1} - x^k = \alpha \cdot (A \cdot x^k - f)$$

- Эффективны когда
  - Матрица  $A$  сильно разрежена
  - Параллельные вычисления
- В обоих случаях цена (по времени) одной итерации  $O(N)$

# СХОДИМОСТЬ

$$Ax^* = f,$$

$$d^{k+1} = x^{k+1} - x^*,$$

$$d^{k+1} = \alpha \cdot Ad^k,$$

$$\|d^{k+1}\| \leq |\alpha| \cdot \|A\| \cdot \|d^k\|,$$

$$|\alpha| \cdot \|A\| < 1$$

- Если есть сходимость, то только к решению системы
- Записав уравнения для погрешности получаем достаточное условие сходимости
- За счет выбора достаточно малого значения параметра получаем сходимость

# Код на **CUDA**

```
//  
// one iteration  
//  
__global__ void kernel ( float * a, float * f, float alpha,  
                        float * x0, float * x1, int n )  
{  
    int  idx = blockIdx.x * blockDim.x + threadIdx.x;  
    int  ia  = n * idx;  
    float sum = 0.0f;  
  
    for ( int I = 0; i < n; i++ )  
        sum += a [ia + I] * x0 [I];  
  
    x1 [idx] = x0 [idx] + alpha * (sum - f [idx] );  
}
```



# Ресурсы нашего курса



- [CUDA.CS.MSU.SU](https://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!

- [www.steps3d.narod.ru](http://www.steps3d.narod.ru)

- [www.nvidia.ru](http://www.nvidia.ru)

# Вопросы

---

