Лекция 11
Технология CUDA

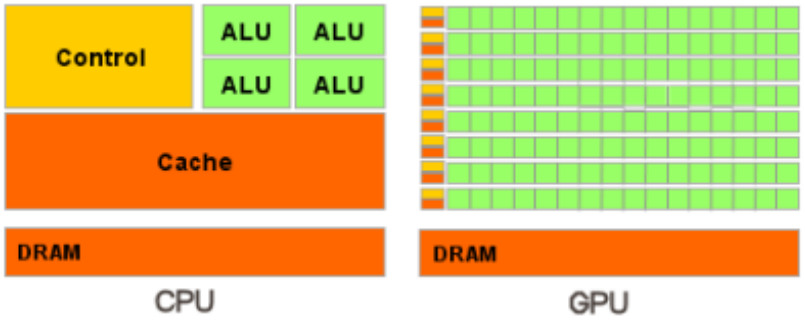
Курносов Михаил Георгиевич

E-mail: mkurnosov@gmail.com
WWW: www.mkurnosov.net

Курс «Высокопроизводительные вычислительные системы»
Сибирский государственный университет телекоммуникаций и информатики (Новосибирск)
Осенний семестр, 2015

GPU – Graphics Processing Unit

- **Graphics Processing Unit (GPU)** – графический процессор, специализированный многопроцессорный ускоритель с общей памятью
- Большая часть площади чипа занята элементарными ALU/FPU/Load/Store модулями
- Устройство управления (control unit) относительно простое по сравнению с CPU
- GPU управляется с CPU: копирование данных между оперативной памятью узла и GPU, запуск программ и др.

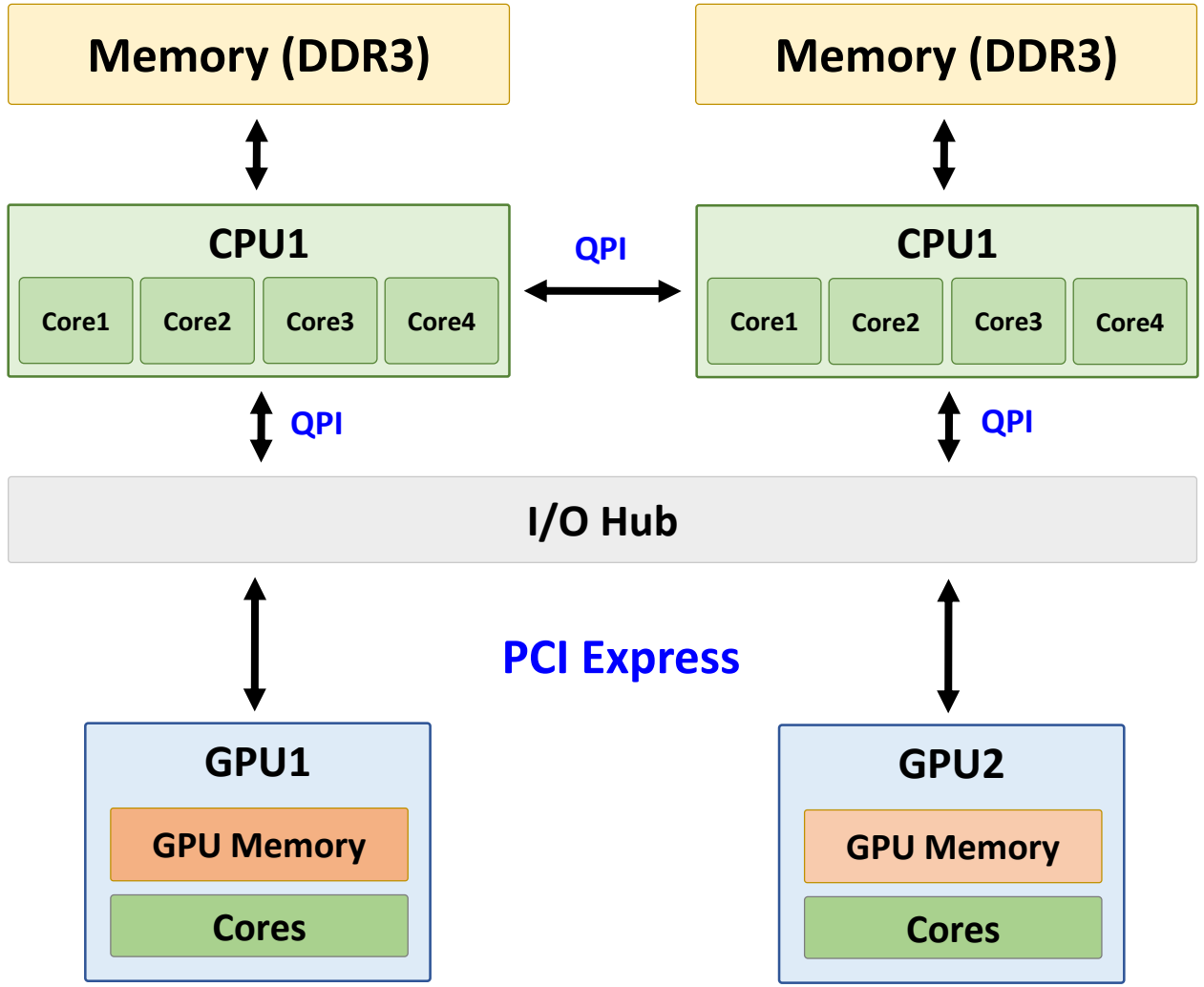


NVIDIA GeForce GTX 780
(Kepler, **2304 cores**, GDDR5 3 GB)

AMD Radeon HD 8970
(**2048 cores**, GDDR5 3 GB)

Гибридные вычислительные узлы

- Несколько GPU
- CPU управляет GPU через драйвер
- Узкое место – передача данных между памятью CPU и GPU
- GPU не является bootable-устройством



74 системы из Top500 (Nov. 2014)
оснащены ускорителями (NVIDIA, ATI, Intel Xeon Phi, PEZY SC)

NVIDIA CUDA

- **NVIDIA CUDA** – программно-аппаратная платформа для организации параллельных вычислений на графических процессорах

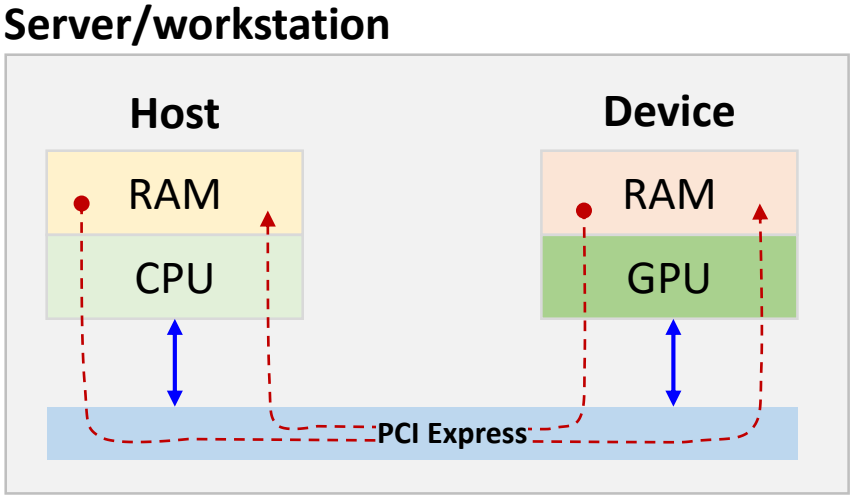
- **NVIDIA CUDA SDK**:
  - архитектура виртуальной машины CUDA
  - компилятор C/C++
  - драйвер GPU

- **ОС**: GNU/Linux, Apple Mac OS X, Microsoft Windows
- **2006** – CUDA 1.0
- **2015 – CUDA 7.5** (Unified memory, Multi-GPU)
- **Микроархитектуры**: **Tesla** (GeForce 8), **Fermi** (GeForce 400, 500), **Kepler** (GeForce 600, 700), **Maxwell** (GeForce 700, 800, 900)

http://developer.nvidia.com/cuda

Основные понятия NVIDIA CUDA

- **Хост (host)** – узел с CPU и его память
- **Устройство (device)** – графический процессор и его память
- **Ядро (kernel)** – это фрагмент программы, предназначенный для выполнения на GPU



- CUDA-программа и ее входные данные находятся в памяти хоста
- Программа компилируется и запускается на хосте
- В CUDA-программе выполняются следующие шаги:

1. Копирование данных из памяти хоста в память устройства
2. Запуск ядра (kernel) на устройстве
3. Копирование данных из памяти устройства в память хоста

CUDA Hello World

```
/*
 * hello.cu:
 *
 */

#include <stdio.h>

__global__ void mykernel()
{

}

int main()
{
    /* Запуск ядра на GPU – один поток */
    mykernel<<<1,1>>>();
    printf("Hello, CUDA World!\n");
    return 0;
}
```

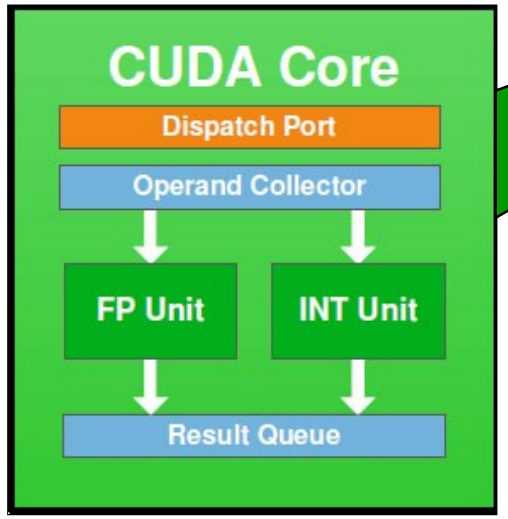
Спецификатор **__global__** сообщает компилятору, что функция предназначена для выполнения на GPU

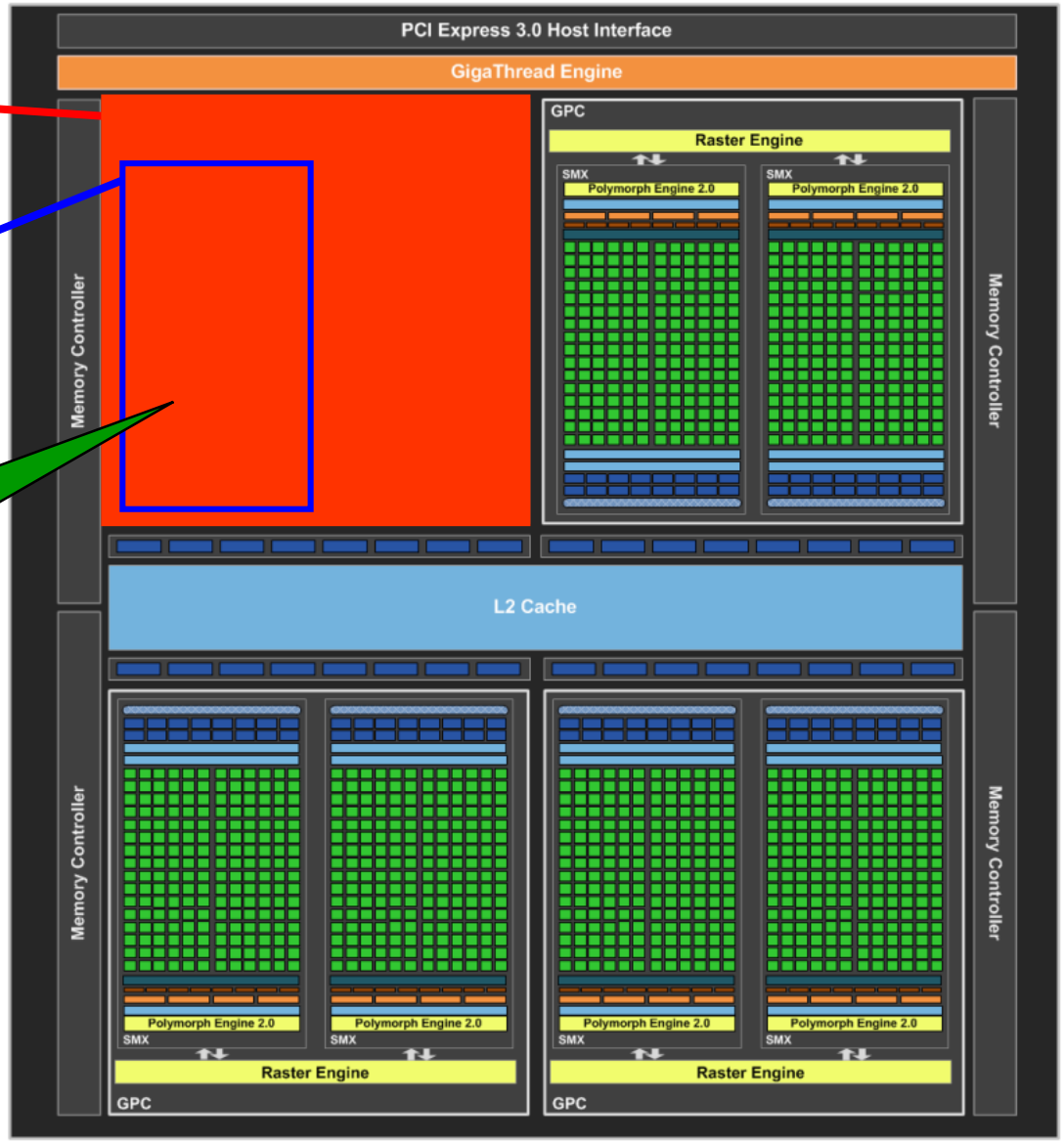
“<<< >>>” – запуск заданного числа потоков на GPU

NVIDIA GeForce 680 (GK104, Kepler microarch.)

4 Graphics Processing Clusters (GPC)
2 Streaming Multiprocessor (SMX)

Streaming Multiprocessor (SMX)
192 cores, 32 special function units,
32 LD/ST units, 4 warp schedulers





http://www.nvidia.com/content/PDF/product-specifications/GeForce_GTX_680_Whitepaper_FINAL.pdf

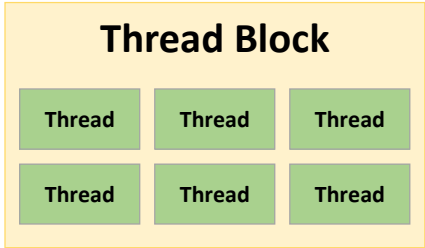
Информация об устройстве

```
./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce GTX 680"
  CUDA Driver Version / Runtime Version          7.5 / 7.5
  CUDA Capability Major/Minor version number:    3.0
  Total amount of global memory:                 2048 MBytes (2147287040 bytes)
  ( 8) Multiprocessors, (192) CUDA Cores/MP:     1536 CUDA Cores
  GPU Max Clock rate:                            1058 MHz (1.06 GHz)
  Memory Clock rate:                             3004 Mhz
  Memory Bus Width:                              256-bit
  L2 Cache Size:                                 524288 bytes
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  ...
```

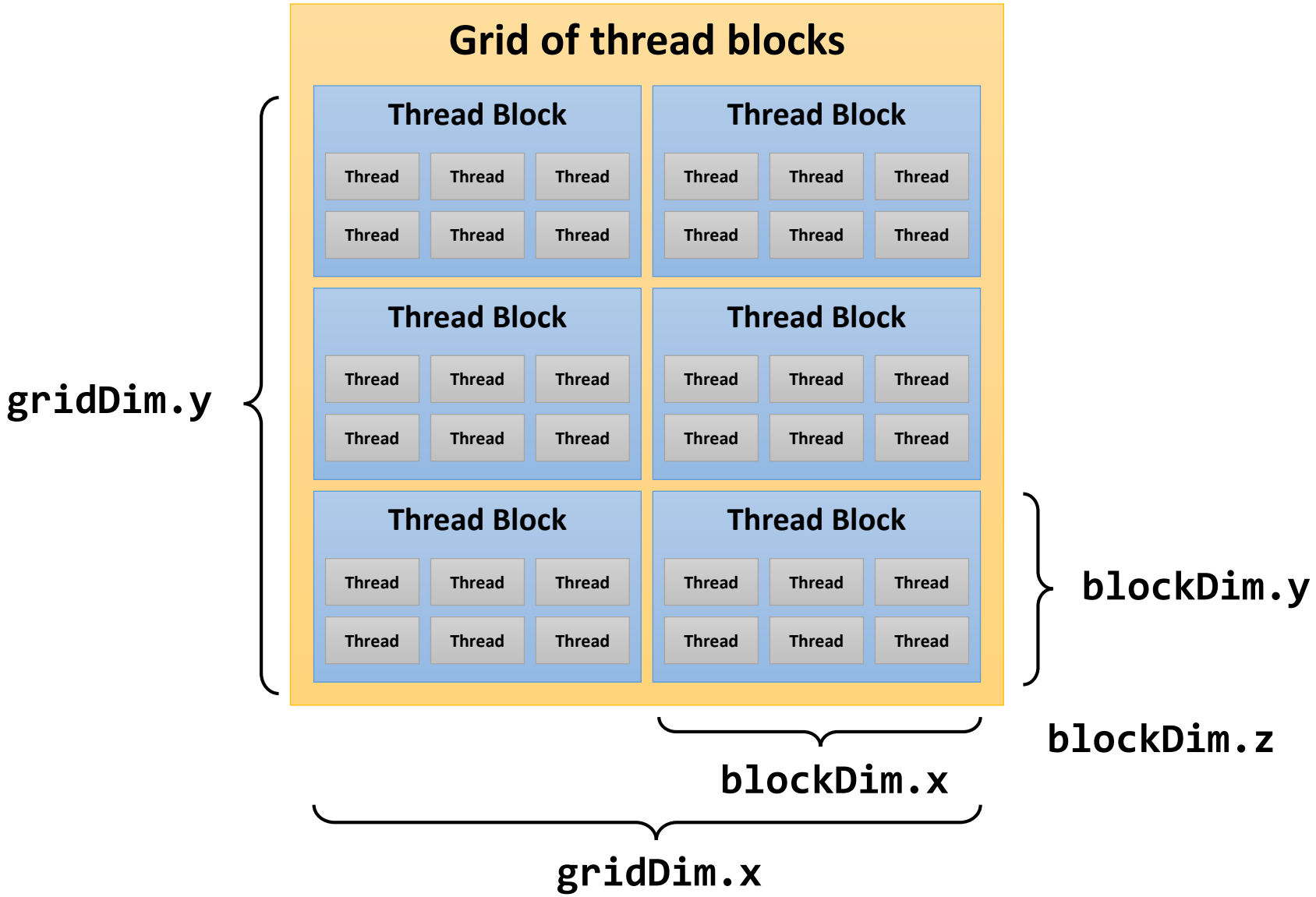
/opt/NVIDIA_CUDA-7.5_Samples

Вычислительные потоки CUDA

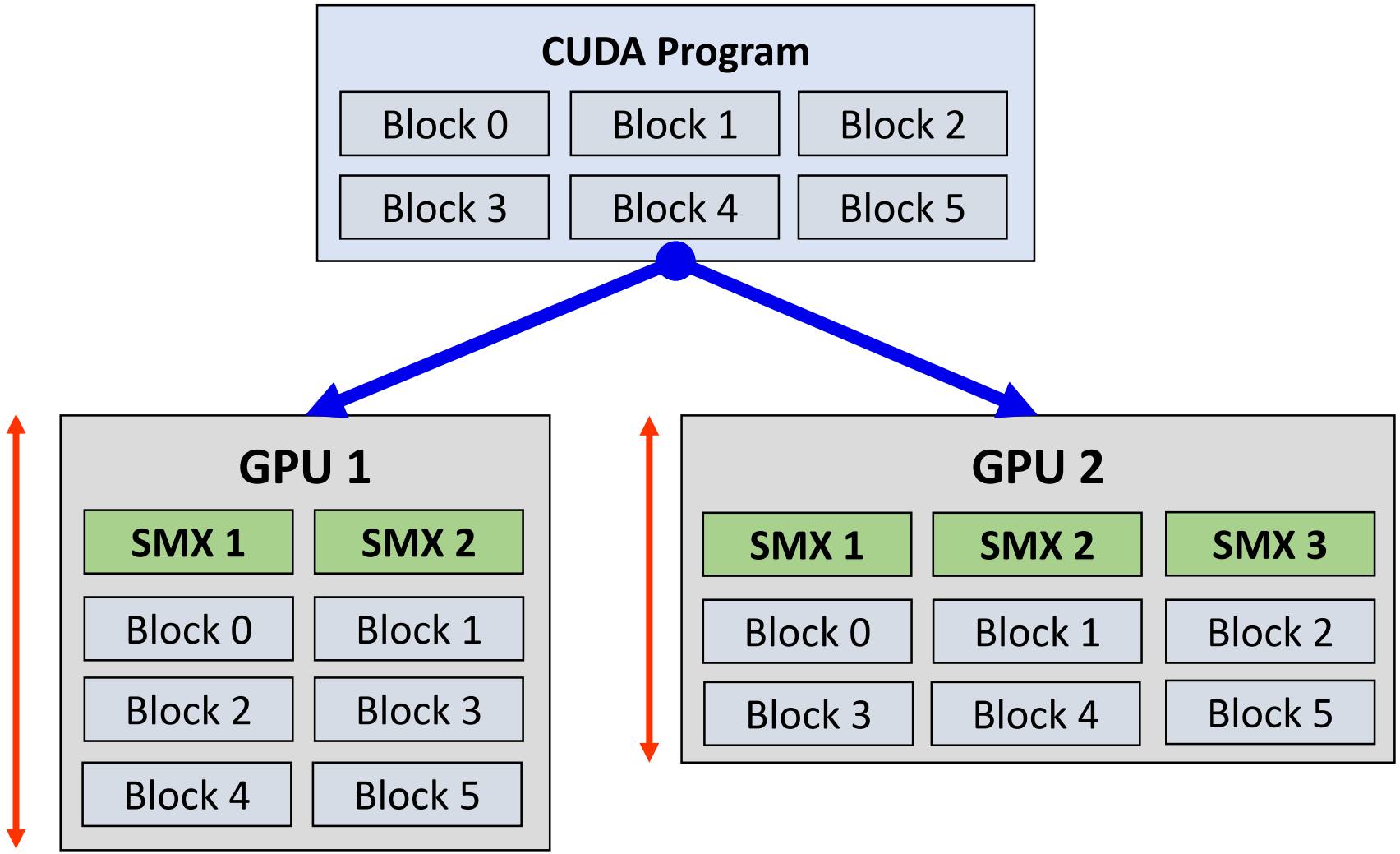
- **Номер потока (thread index)** – это трехкомпонентный вектор (координаты потока)
- Потоки логически сгруппированы в одномерный, двухмерный или трёхмерный *блок* (thread block)
- Количество потоков в блоке ограничено (в Kepler 1024)
- Блоки распределяются по потоковым мультипроцессорам SMX
- Предопределенные переменные
  - **`threadIdx.{x, y, z}`** – номер потока
  - **`blockDim.{x, y, z}`** – размерность блока
  - **`blockIdx.{x, y, z}`** – номер блока



Вычислительные потоки CUDA



Выполнение CUDA-программы



Сложение векторов (CPU version)

```
/*
 * Host code (CPU version)
 */
void vadd(float *a, float *b, float *c, int n)
{
    for (int i = 0; i < n; i++)
        c[i] = a[i] + b[i];
}
```

Сложение векторов (GPU version)

```
#include <cuda_runtime.h>

enum { NELEMS = 1024 * 1024 };

int main()
{
    /* Allocate vectors on host */
    size_t size = sizeof(float) * NELEMS;
    float *h_A = malloc(size);
    float *h_B = malloc(size);
    float *h_C = malloc(size);
    if (h_A == NULL || h_B == NULL || h_C == NULL) {
        fprintf(stderr, "Allocation error.\n");
        exit(EXIT_FAILURE);
    }

    for (int i = 0; i < NELEMS; ++i) {
        h_A[i] = rand() / (float)RAND_MAX;
        h_B[i] = rand() / (float)RAND_MAX;
    }
```

Сложение векторов (GPU version)

```
/* Allocate vectors on device */
float *d_A = NULL, *d_B = NULL, *d_C = NULL;
if (cudaMalloc((void **)&d_A, size) != cudaSuccess) {
    fprintf(stderr, "Allocation error\n");
    exit(EXIT_FAILURE);
}
if (cudaMalloc((void **)&d_B, size) != cudaSuccess) {
    fprintf(stderr, "Allocation error\n");
    exit(EXIT_FAILURE);
}
if (cudaMalloc((void **)&d_C, size) != cudaSuccess) {
    fprintf(stderr, "Allocation error\n");
    exit(EXIT_FAILURE);
}
```

Сложение векторов (GPU version)

```
/* Copy the host vectors to device */
if (cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice) != cudaSuccess) {
    fprintf(stderr, "Host to device copying failed\n");
    exit(EXIT_FAILURE);
}
if (cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice) != cudaSuccess) {
    fprintf(stderr, "Host to device copying failed\n");
    exit(EXIT_FAILURE);
}

/* Launch the kernel */
int threadsPerBlock = 1024;
int blocksPerGrid =(NELEMS + threadsPerBlock - 1) / threadsPerBlock;
vadd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, NELEMS);
if (cudaGetLastError() != cudaSuccess) {
    fprintf(stderr, "Failed to launch kernel!\n");
    exit(EXIT_FAILURE);
}
```

$$blocks = \left\lceil \frac{N}{threads} \right\rceil = \left\lfloor \frac{N + threads - 1}{threads} \right\rfloor$$

Сложение векторов (GPU version)

```
    /* Copy the device vectors to host */
    if (cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost) != cudaSuccess) {
        fprintf(stderr, "Device to host copying failed\n");
        exit(EXIT_FAILURE);
    }

    for (int i = 0; i < NELEMS; ++i) {
        if (fabs(h_A[i] + h_B[i] - h_C[i]) > 1e-5) {
            fprintf(stderr, "Result verification failed at element %d!\n", i);
            exit(EXIT_FAILURE);
        }
    }

    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    free(h_A);
    free(h_B);
    free(h_C);
    cudaDeviceReset();
    return 0;
}
```

Сложение векторов (GPU version)

```
__global__ void vadd(const float *a, const float *b, float *c, int n)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < n)
        c[i] = a[i] + b[i];
}
```

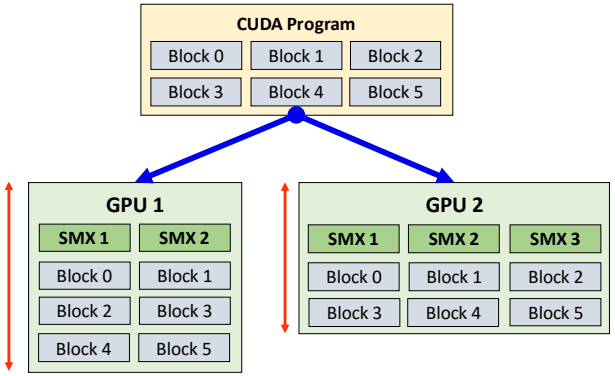
Сложение векторов (scalability)

NELEMS	cngpu1 GeForce GTX 680	cngpu2 GeForce GT 630	cngpu3 GeForce GTS 250 (threadsPerBlock = 512)
2^20	CPU version (sec.): 0.001658 GPU version (sec.): 0.000161 Memory ops. (sec.): 0.002341 Memory bw. (MiB/sec.): 5125.94 CPU perf (MFLOPS): 603.15 GPU perf (MFLOPS): 6213.78 **Speedup: 10.30** Speedup (with mem ops.): 0.66	CPU version (sec.): 0.002311 GPU version (sec.): 0.001054 Memory ops. (sec.): 0.002514 Memory bw. (MiB/sec.): 4773.49 CPU perf (MFLOPS): 432.71 GPU perf (MFLOPS): 948.72 **Speedup: 2.19** Speedup (with mem ops.): 0.65	CPU version (sec.): 0.002195 GPU version (sec.): 0.001993 Memory ops. (sec.): 0.009481 Memory bw. (MiB/sec.): 1265.70 CPU perf (MFLOPS): 455.61 GPU perf (MFLOPS): 501.77 **Speedup: 1.10** Speedup (with mem ops.): 0.19
10 * 2^20	CPU version (sec.): 0.015133 GPU version (sec.): 0.000892 Memory ops. (sec.): 0.021551 Memory bw. (MiB/sec.): 5568.15 CPU perf (MFLOPS): 660.81 GPU perf (MFLOPS): 11211.72 **Speedup: 16.97** Speedup (with mem ops.): 0.67	CPU version (sec.): 0.014987 GPU version (sec.): 0.009118 Memory ops. (sec.): 0.021882 Memory bw. (MiB/sec.): 5484.00 CPU perf (MFLOPS): 667.25 GPU perf (MFLOPS): 1096.72 **Speedup: 1.64** Speedup (with mem ops.): 0.48	CPU version (sec.): 0.017394 GPU version (sec.): 0.006778 Memory ops. (sec.): 0.101824 Memory bw. (MiB/sec.): 1178.51 CPU perf (MFLOPS): 574.91 GPU perf (MFLOPS): 1475.36 **Speedup: 2.57** Speedup (with mem ops.): 0.16

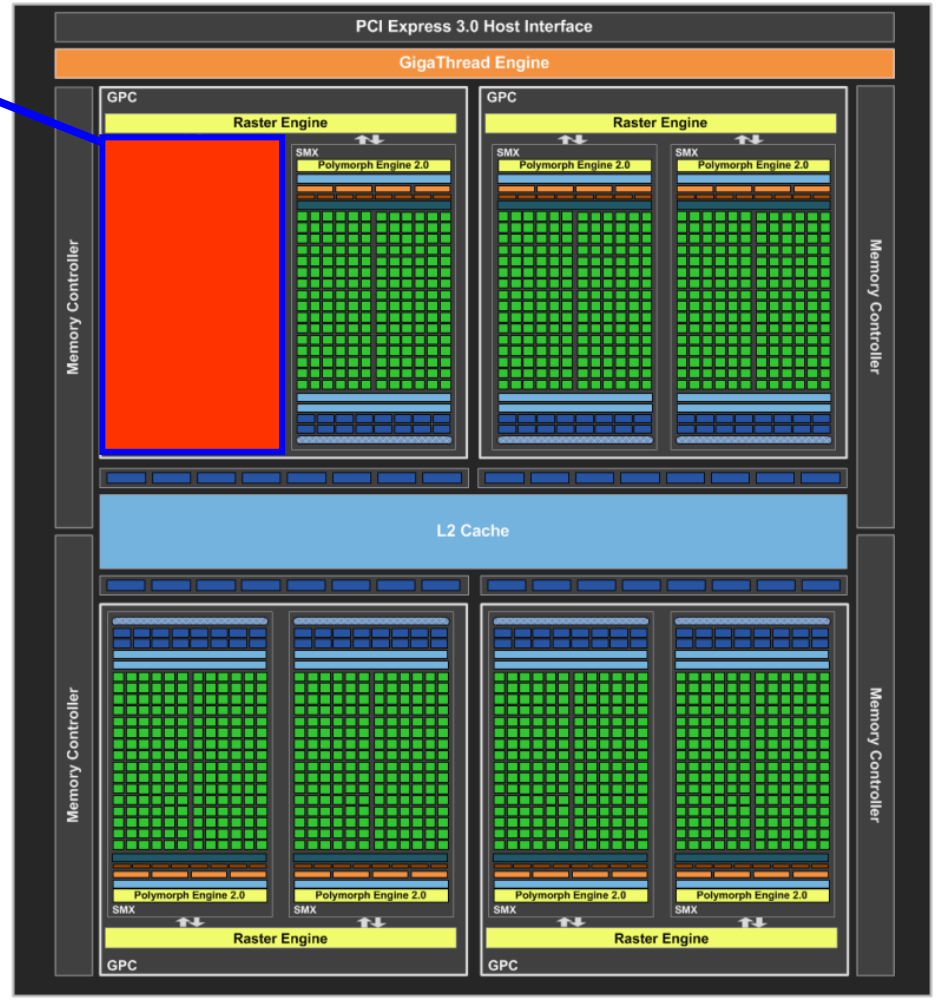
Архитектура SIMT (Single-Instruction, Multiple-Thread)

Streaming Multiprocessor (SM)
192 cores, 32 special function units,
32 LD/ST units, 4 warp schedulers

- Блоки потоков распределяются по SM
- Если число блоков меньше количества SM, то часть процессоров GPU простаивает
- Порядок выполнения блоков заранее неизвестен – алгоритмы для GPU не должны быть чувствительны к порядку выполнения блоков



NVIDIA GeForce 680 (GK104, Kepler)

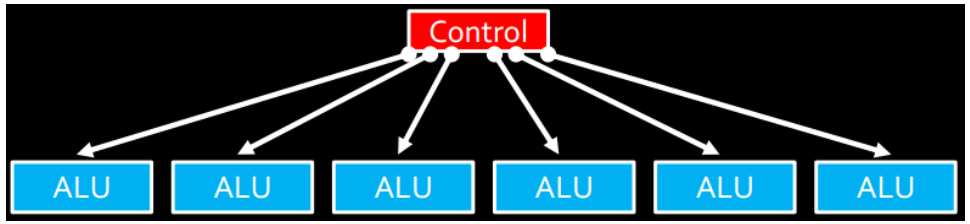


http://www.nvidia.com/content/PDF/product-specifications/GeForce_GTX_680_Whitepaper_FINAL.pdf

Архитектура SIMT (Single-Instruction, Multiple-Thread)

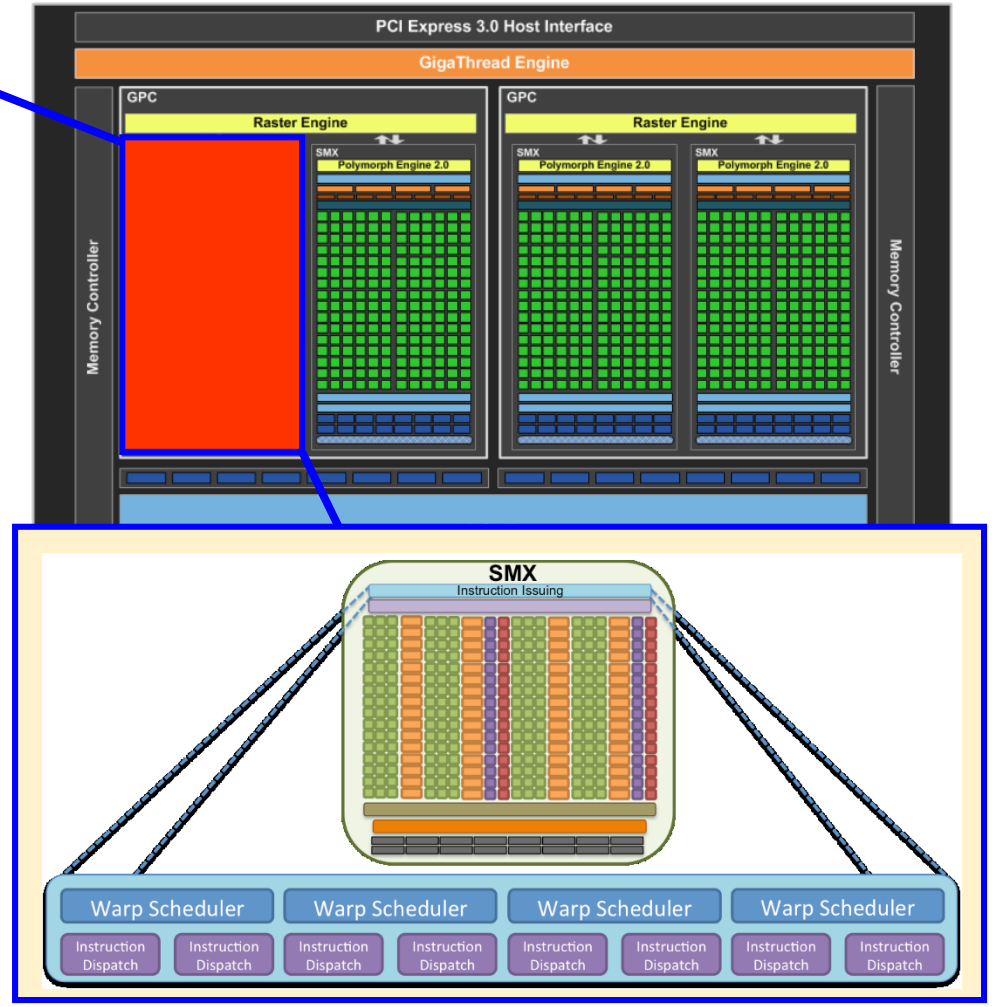
Streaming Multiprocessor (SM)
192 cores, 32 special function units,
32 LD/ST units, **4 warp schedulers**

- Блоки назначенные на SM логически разбиваются на **группы (warps)** по 32 потока
- Потоки всегда одинаково распределяются по warp-ам (детерминировано, на базе номера потока)
- Если размер блока не кратен размеру группы, то последняя группа (warp) потоков блокируется



http://mc.stanford.edu/cgi-bin/images/3/34/Darve_cme343_cuda_3.pdf

NVIDIA GeForce 680 (GK104, Kepler)

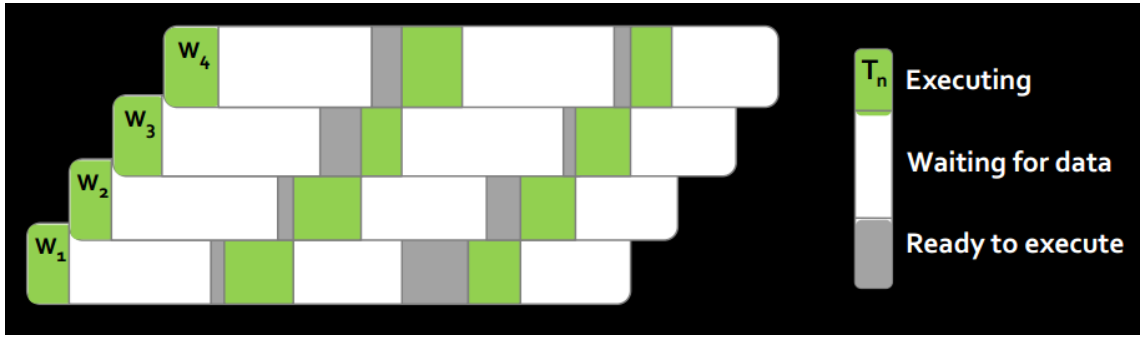


http://www.nvidia.com/content/PDF/product-specifications/GeForce_GTX_680_Whitepaper_FINAL.pdf

Архитектура SIMT (Single-Instruction, Multiple-Thread)

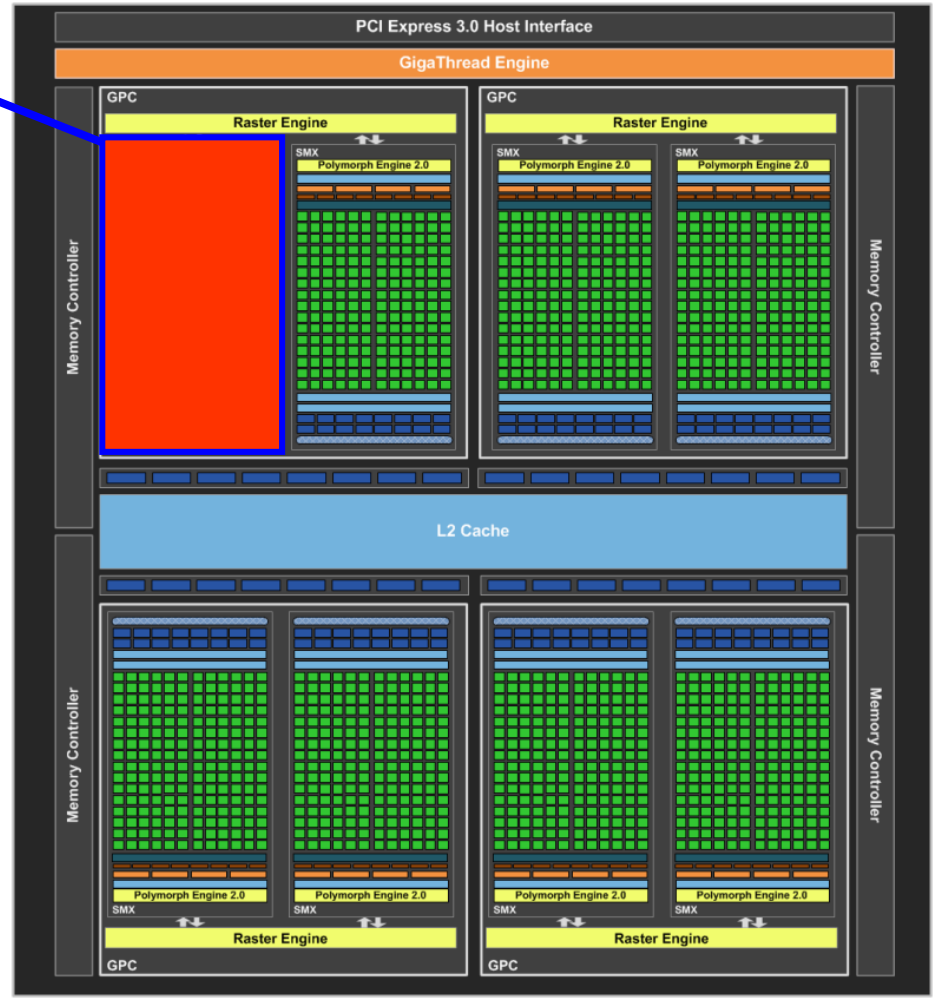
Streaming Multiprocessor (SM)
192 cores, 32 special function units,
32 LD/ST units, **4 warp schedulers**

- Планировщик (warp scheduler) запускает на выполнение группу потоков, у которых «готовы» входные данные
- Все активные потоки группы в каждый момент времени выполняют одну и ту же инструкцию
- Планировщик выполняет переключение контекста между группами потоков



http://mc.stanford.edu/cgi-bin/images/3/34/Darve_cme343_cuda_3.pdf

NVIDIA GeForce 680 (GK104, Kepler)

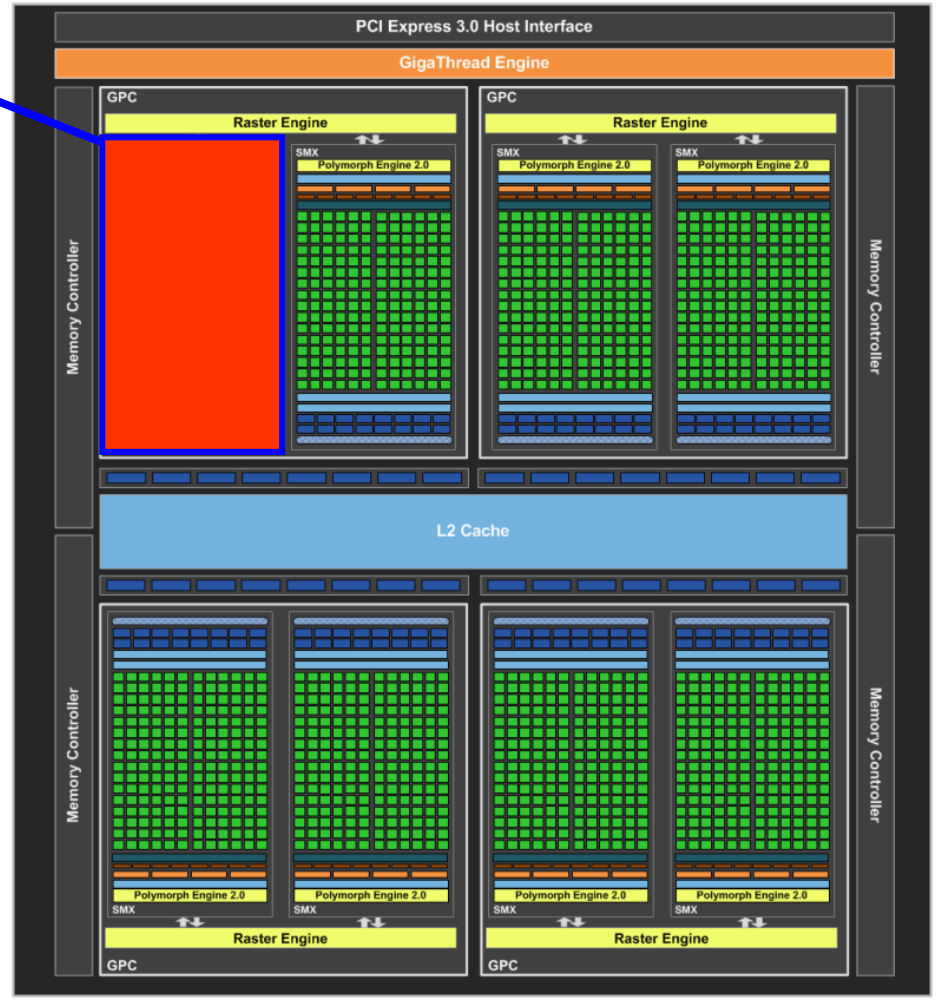


http://www.nvidia.com/content/PDF/product-specifications/GeForce_GTX_680_Whitepaper_FINAL.pdf

Архитектура SIMT (Single-Instruction, Multiple-Thread)

Streaming Multiprocessor (SM)
192 cores, 32 special function units,
32 LD/ST units, **4 warp schedulers**

NVIDIA GeForce 680 (GK104, Kepler)



Control Flow Divergence

- Все потоки группы в каждый момент времени выполняют одну и ту же инструкцию
- Что если потоки управления расходятся?

```
__global__ void kernel()
{
    if (val < 50) {
        // branch 1
    } else {
        // branch 2
    }
}
```

Потоки группы выполняют обе ветви

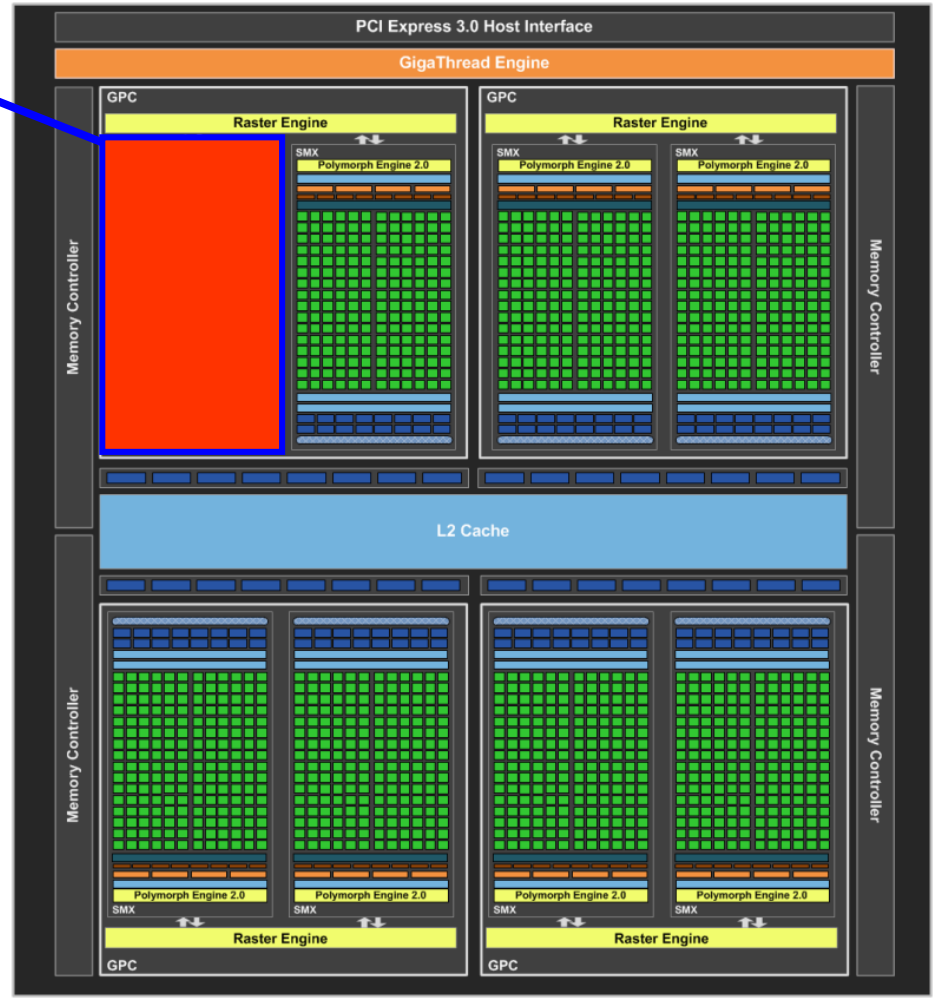
Потоки, которые не должны выполнять ветвь, блокируются

http://www.nvidia.com/content/PDF/product-specifications/GeForce_GTX_680_Whitepaper_FINAL.pdf

Архитектура SIMT (Single-Instruction, Multiple-Thread)

Streaming Multiprocessor (SM)
192 cores, 32 special function units,
32 LD/ST units, **4 warp schedulers**

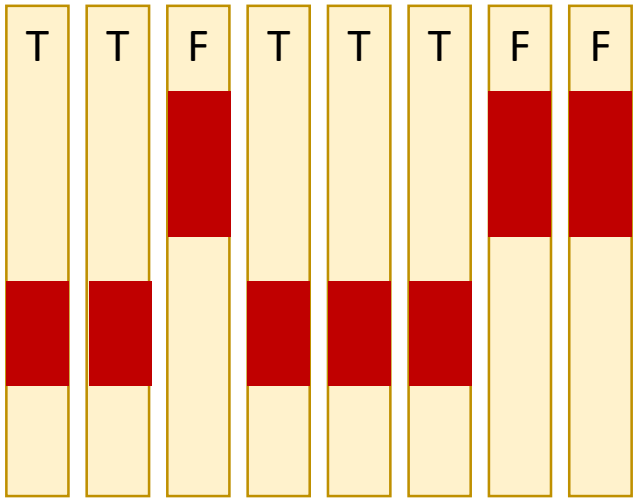
NVIDIA GeForce 680 (GK104, Kepler)



http://www.nvidia.com/content/PDF/product-specifications/GeForce_GTX_680_Whitepaper_FINAL.pdf

Control Flow Divergence

```
__global__ void kernel()
{
    if (val < 50) {
        // branch 1
        // branch 1
        // branch 1
    } else {
        // branch 2
        // branch 2
    }
}
```

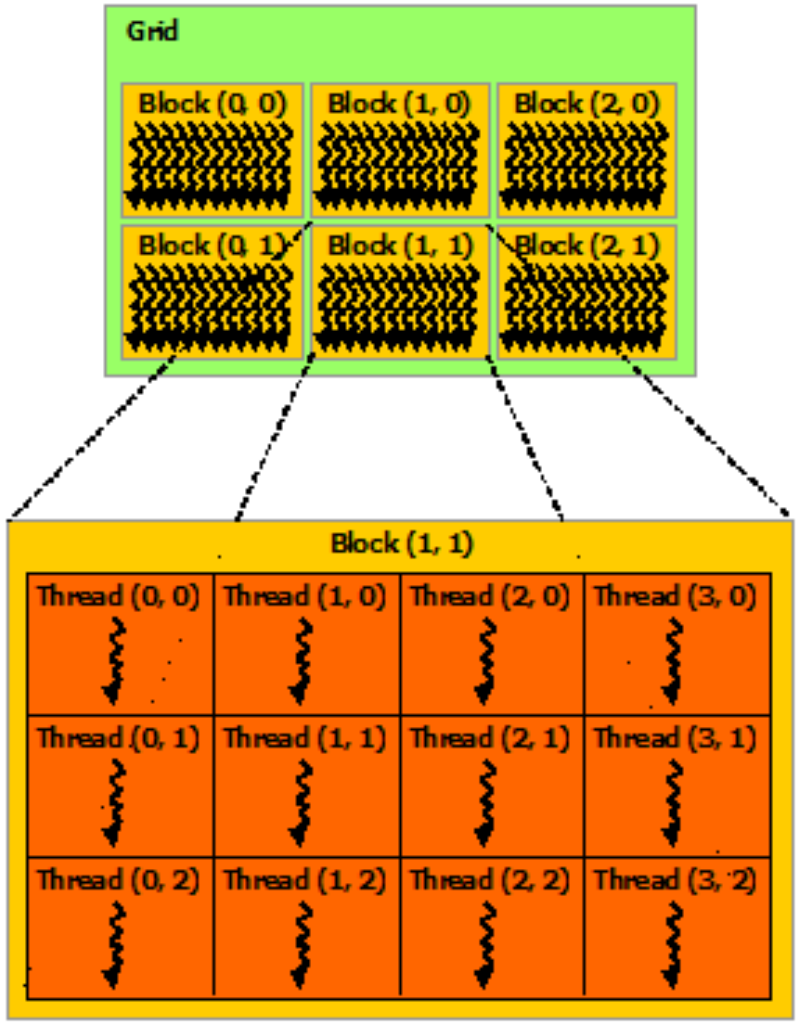


Информация об устройстве (cngpu1)

```
./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce GTX 680"
  CUDA Driver Version / Runtime Version          7.5 / 7.5
  CUDA Capability Major/Minor version number:    3.0
  Total amount of global memory:                 2048 MBytes (2147287040 bytes)
  ( 8) Multiprocessors, (192) CUDA Cores/MP:     1536 CUDA Cores
  GPU Max Clock rate:                            1058 MHz (1.06 GHz)
  Memory Clock rate:                             3004 Mhz
  Memory Bus Width:                              256-bit
  L2 Cache Size:                                 524288 bytes
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  ...
```

/opt/NVIDIA_CUDA-7.5_Samples

Иерархия потоков



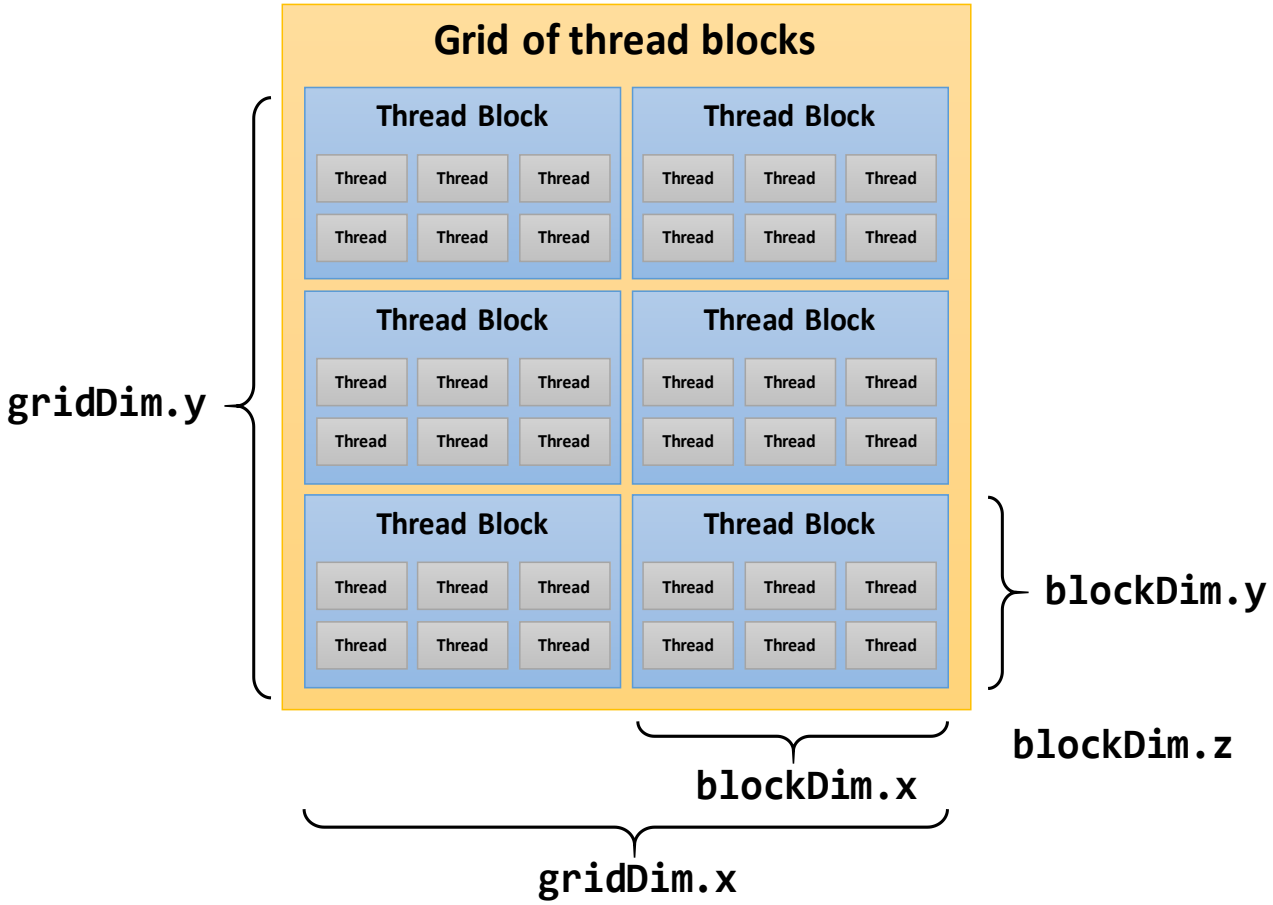
- **Сетка потоков** (1D, 2D, 3D) – совокупность блоков потоков
- Сетка потоков, CUDA-программа, выполняется GPU

- **Блок потоков** (1D, 2D, 3D) – совокупность потоков
- Блоки распределяются по потоковым мультипроцессорам

```
Device 0: "GeForce GTX 680"
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  ...
```

Иерархия потоков

- **Предопределенные переменные**
  - **`threadIdx.{x, y, z}`** – номер потока в блоке
  - **`blockDim.{x, y, z}`** – размерность блока
  - **`blockIdx.{x, y, z}`** – номер блока в сетке

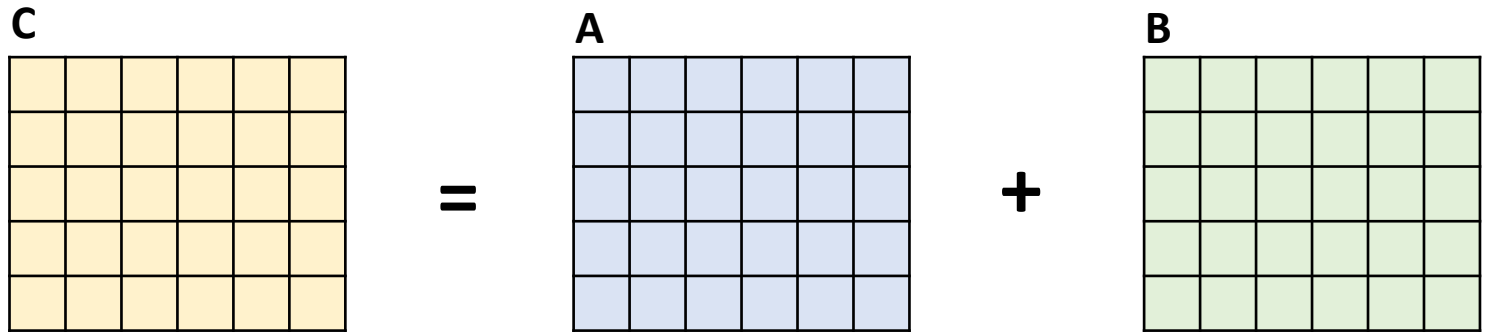


- **Здание структуры сетки**

```
mykernel<<<B,T>>>(arg1, … );
```

  - B – структура сетки
  - T – структура блока

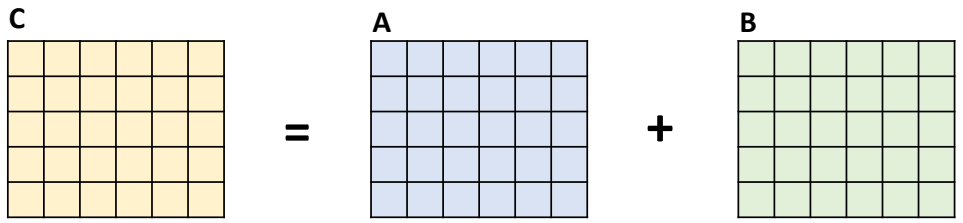
Сложение матриц: CPU



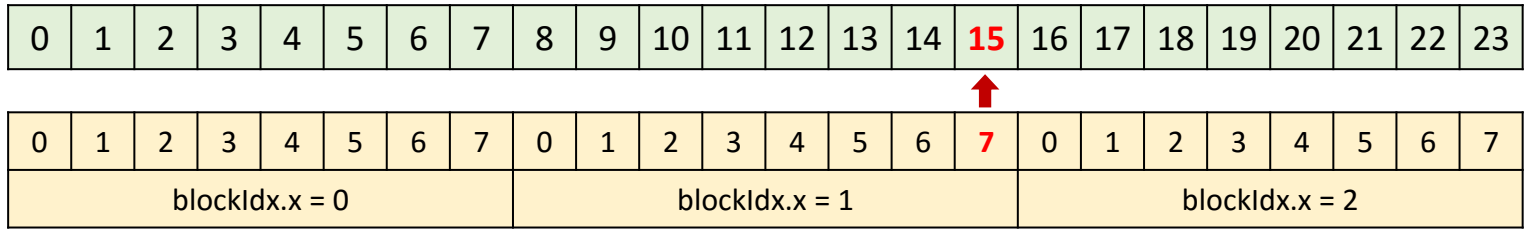

```
void matadd_host(float *a, float *b, float *c, int m, int n)
{
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            int idx = i * n + j;
            c[idx] = a[idx] + b[idx];
        }
    }
}
```

Сложение матриц: CUDA 1D

- Каждый поток вычисляет один элемент c[i * n + j] матрицы
- Одномерные сетка блоков потоков и одномерные блоки потоков



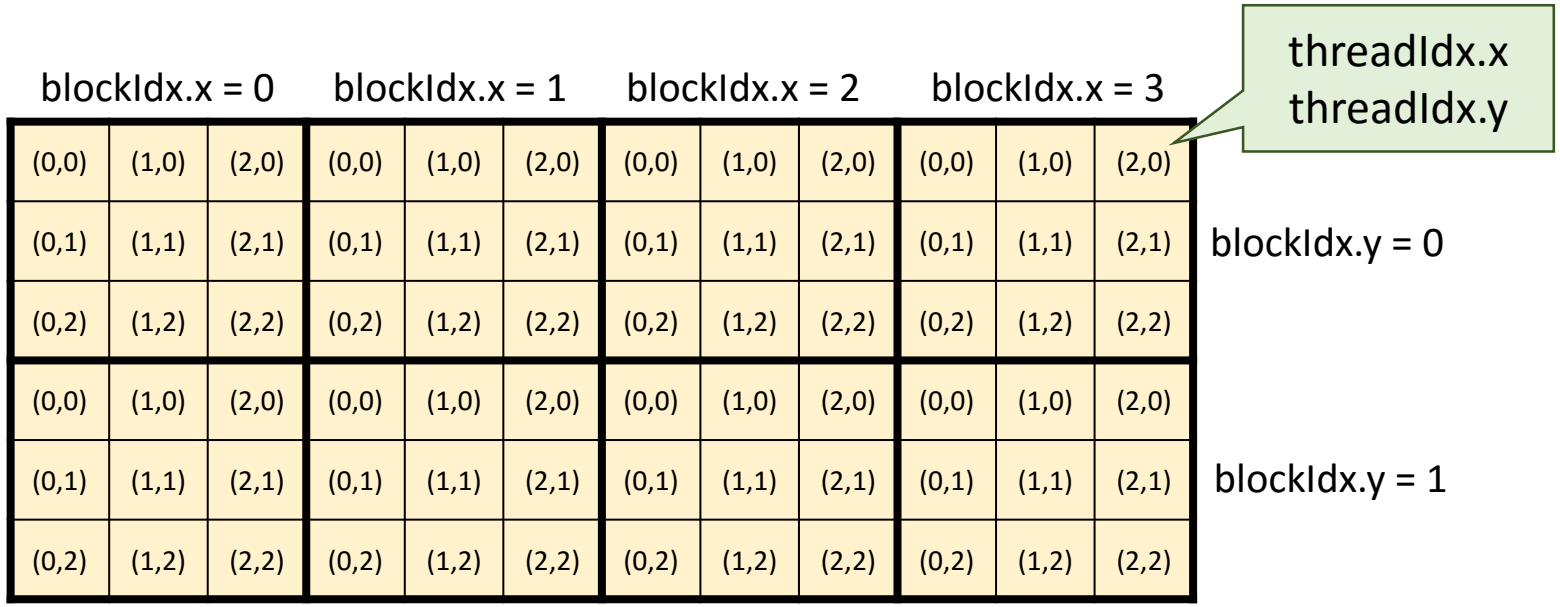
1D-сетка из
3-х 1D-блоков потоков




```
__global__ void matadd(const float *a, const float *b, float *c, int m, int n)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < m * n)
        c[idx] = a[idx] + b[idx];
}

{
    int threadsPerBlock = 1024;
    int blocksPerGrid =(ROWS * COLS + threadsPerBlock - 1) / threadsPerBlock;
    matadd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, ROWS, COLS);
}
```

Сложение матриц: CUDA 2D

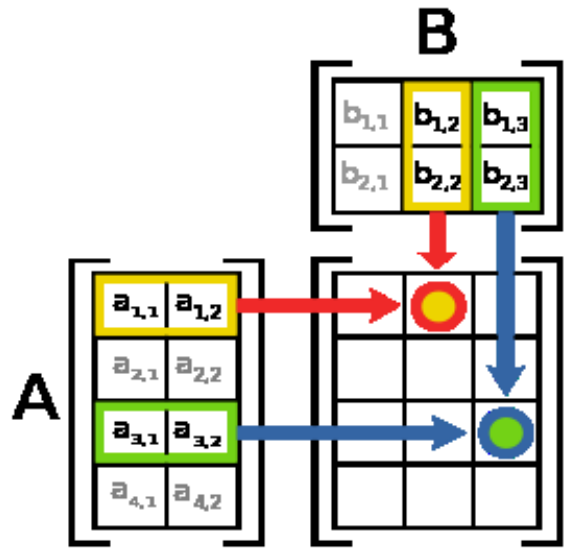
- Каждый поток вычисляет один элемент c[i * n + j] матрицы
- 2D-сетка блоков потоков и 2D-блоки потоков




```
__global__ void matadd(const float *a, const float *b, float *c, int m, int n)
{
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    int idx = y * n + x;
    if (idx < m * n) c[idx] = a[idx] + b[idx];
}

{
    int threadsPerBlockDim = 32;
    dim3 blockDim(threadsPerBlockDim, threadsPerBlockDim, 1);            // Grid: X x Y x Z=1
    int blocksPerGridDimX = ceilf(COLS / (float)threadsPerBlockDim);
    int blocksPerGridDimY = ceilf(ROWS / (float)threadsPerBlockDim);
    dim3 gridDim(blocksPerGridDimX, blocksPerGridDimY, 1);               // Blocks: X x Y x Z=1
    matadd<<<gridDim, blockDim>>>(d_A, d_B, d_C, ROWS, COLS);
}
```

Умножение матриц (CPU)

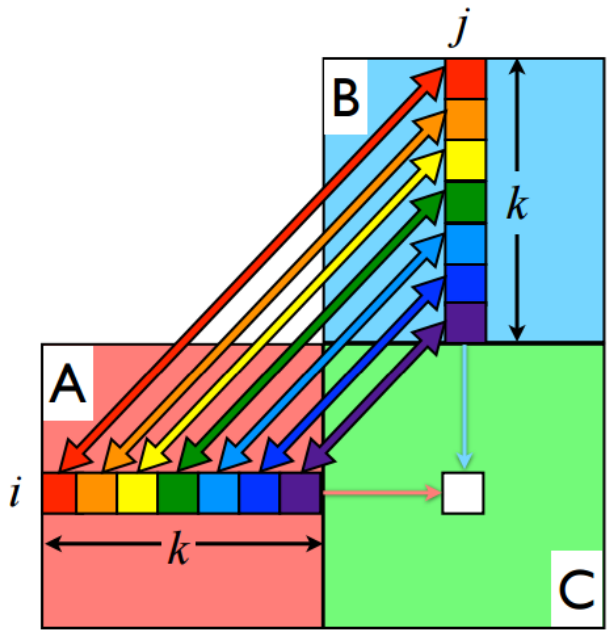
```
void sgemm_host(float *a, float *b, float *c, int n)
{
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            float s = 0.0;
            for (int k = 0; k < n; k++)
                s += a[i * n + k] * b[k * n + j];
            c[i * n + j] = s;
        }
    }
}
```



Умножение матриц (CUDA)

- **Каждый поток вычисляет один элемент результирующей матрицы C**
- Общее число потоков N * N

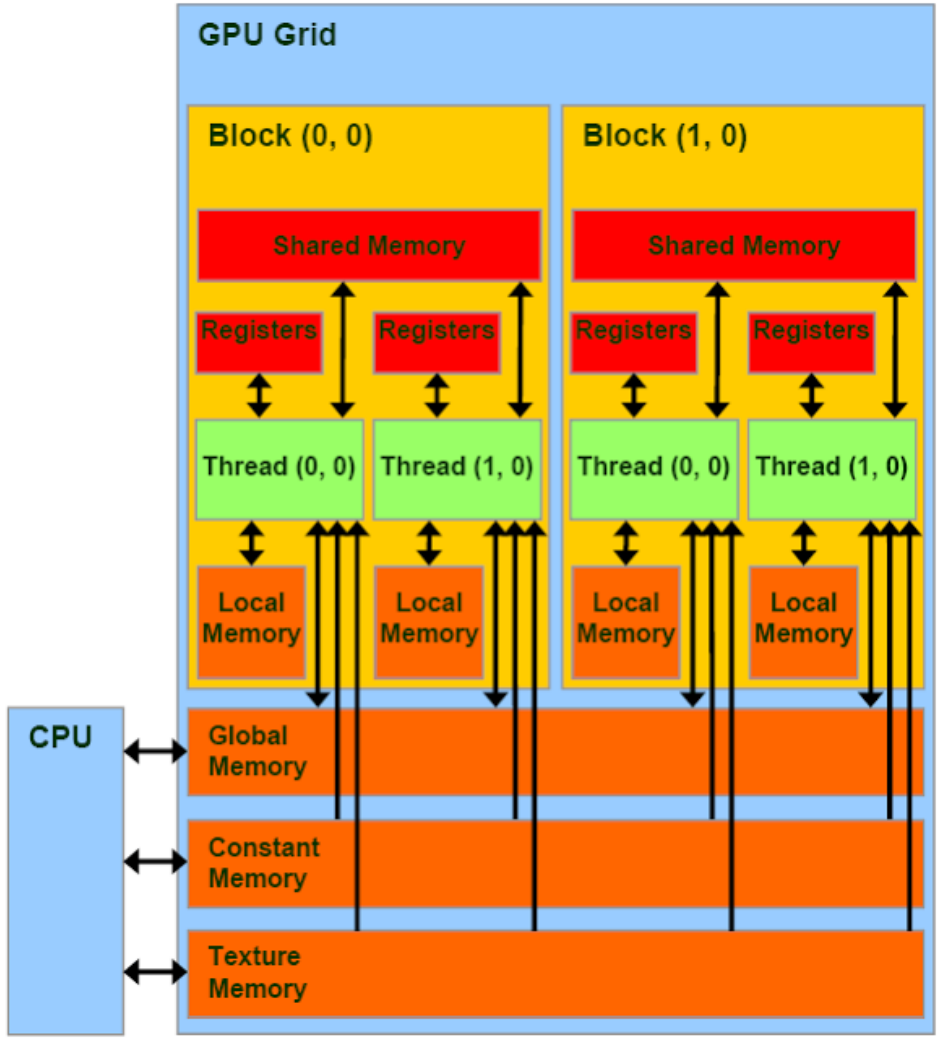
```
__global__ void sgemm(const float *a, const float *b, float *c, int n)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < n && col < n) {
        float s = 0.0;
        for (int k = 0; k < n; k++)
            s += a[row * n + k] * b[k * n + col];
        c[row * n + col] = s;
    }
}
{
    int threadsPerBlockDim = 32;
    dim3 blockDim(threadsPerBlockDim, threadsPerBlockDim, 1);
    int blocksPerGridDimX = ceilf(N / (float)threadsPerBlockDim);
    int blocksPerGridDimY = ceilf(N / (float)threadsPerBlockDim);
    dim3 gridDim(blocksPerGridDimX, blocksPerGridDimY, 1);
    sgemm<<<gridDim, blockDim>>>(d_A, d_B, d_C, N);
}
```



http://users.wfu.edu/choss/CUDA/docs/Lecture%205.pdf

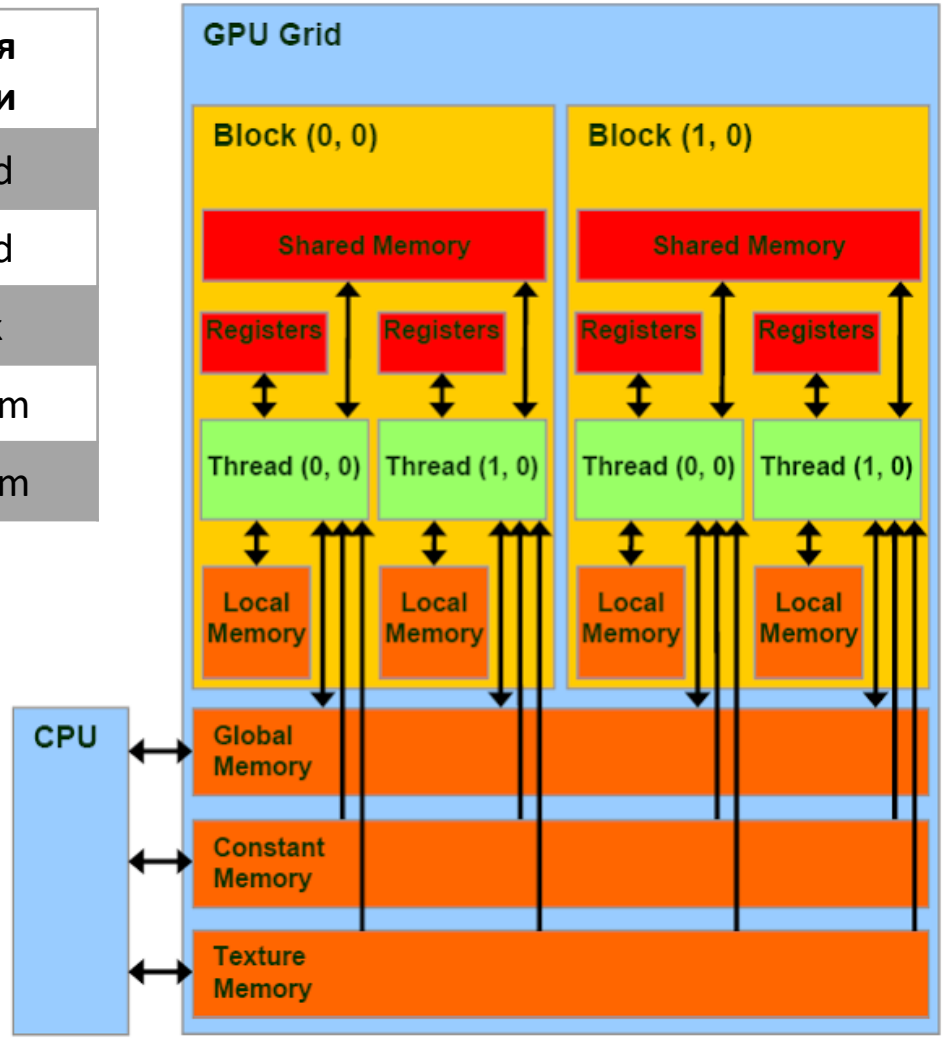
Иерархия памяти (comp. capability >= 3.0)

- **Глобальная память (global memory)**
  - Относительно медленная
  - Кешируется
  - Содержимое сохраняется между запусками ядер
- **Память констант (constant memory)**
  - Содержит константы и аргументы ядер
  - Кешируется
  - 64 KiB / GPU (8 KiB const. cache per SM)
- **Разделяемая память (shared memory)**
  - Быстрая
  - Некешируется
- **Локальная память (local memory)**
  - Часть глобальной памяти
  - Кешируется
- **Регистры (32 bit, 65536 / block)**



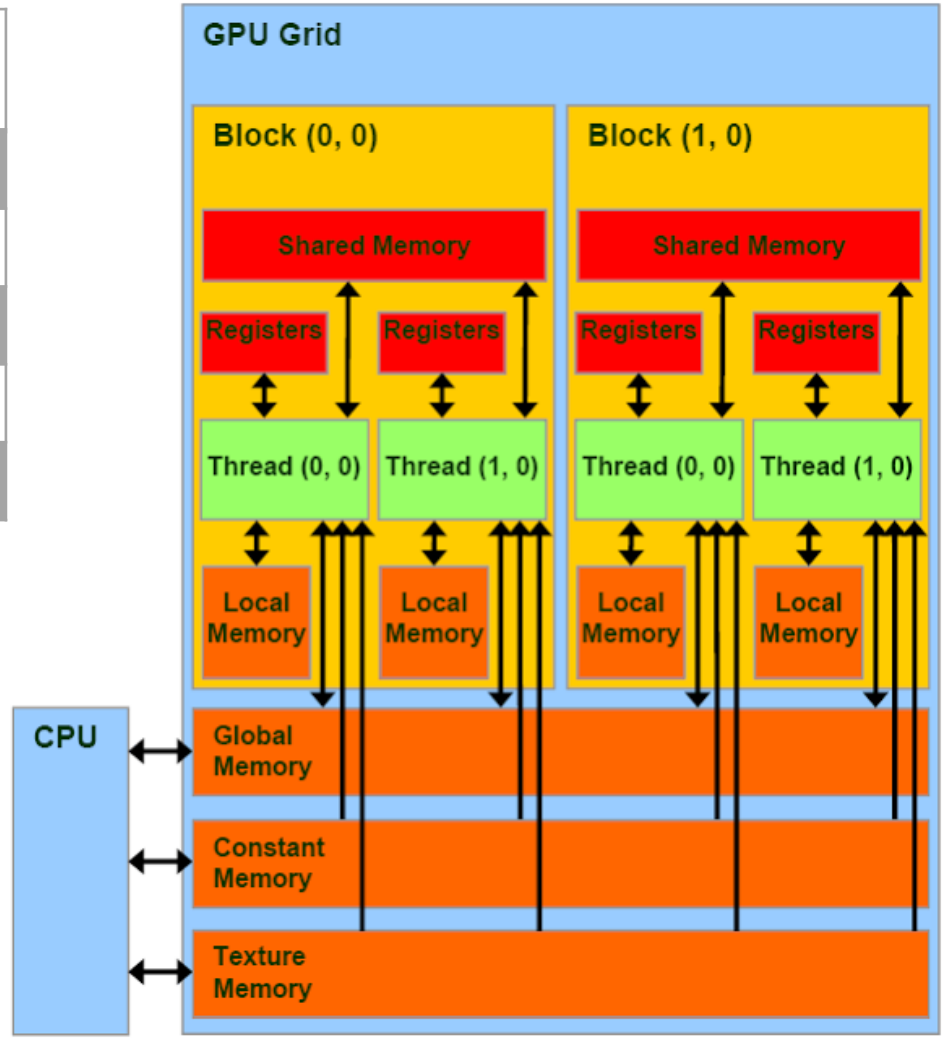
Иерархия памяти (comp. capability >= 3.0)

Объявление	Память	Область видимости	Время жизни
`int localVar;`	register	thread	thread
`int localArray[10];`	local	thread	thread
`__shared__ int sharedVar;`	shared	block	block
`__device__ int globalVar;`	global	grid	program
`__constant__ int constVar;`	constant	grid	program



Иерархия памяти (comp. capability >= 3.0)

Объявление	Память	Накладные расходы
`int localVar;`	register	1x
`int localArray[10];`	local	100x
`__shared__ int sharedVar;`	shared	1x
`__device__ int globalVar;`	global	100x
`__constant__ int constVar;`	constant	1x



Иерархия памяти

	FERMI GF100	FERMI GF104	KEPLER GK104	KEPLER GK110	KEPLER GK210
Compute Capability	2.0	2.1	3.0	3.5	3.7
Threads / Warp	32				
Max Threads / Thread Block	1024				
Max Warps / Multiprocessor	48		64		
Max Threads / Multiprocessor	1536		2048		
Max Thread Blocks / Multiprocessor	8		16		
32-bit Registers / Multiprocessor	32768		65536		131072
Max Registers / Thread Block	32768		65536		65536
Max Registers / Thread	63			255	
Max Shared Memory / Multiprocessor	48K				112K
Max Shared Memory / Thread Block	48K				
Max X Grid Dimension	2^16-1		2^32-1		
Hyper-Q	No			Yes	
Dynamic Parallelism	No			Yes	

Использование иерархии памяти (thread local)

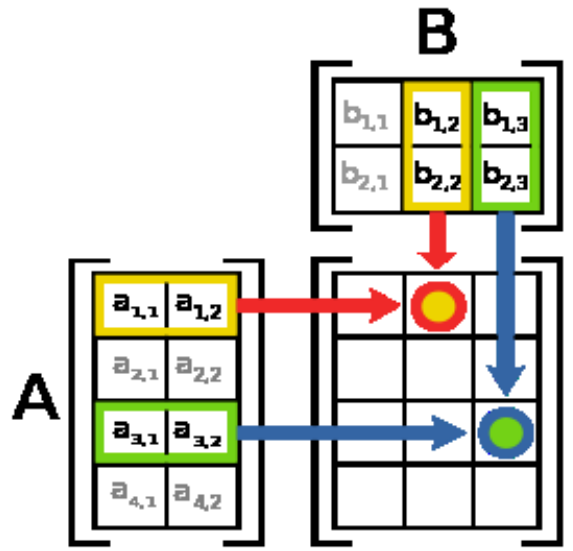
```
// Загружаем данные из глобальной памяти в регистр
float localA = dA[blockIdx.x * blockDim.x + threadIdx.x];

// Вычисления над данными в регистрах
float res = f(localA);

// Записываем результат в глобальную память
dA[blockIdx.x * blockDim.x + threadIdx.x] = res;
```

Использование иерархии памяти (block local)

```
// Загружаем данные из глобальной памяти в разделяемую
__shared__ float sharedA[BLOCK_SIZE];
int idx = blockIdx.x * blockDim.x + threadIdx.x;
sharedA[threadIdx.x] = dA[idx];

__syncthreads();   // барьерная синхронизация потоков блока

// Вычисления над данными в разделяемой памяти
float res = f(shared[threadIdx.x]);

// Записываем результат в глобальную память
dA[idx] = res;
```

Умножение матриц (CPU)

```
void sgemm_host(float *a, float *b, float *c, int n)
{
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            float s = 0.0;
            for (int k = 0; k < n; k++)
                s += a[i * n + k] * b[k * n + j];
            c[i * n + j] = s;
        }
    }
}
```

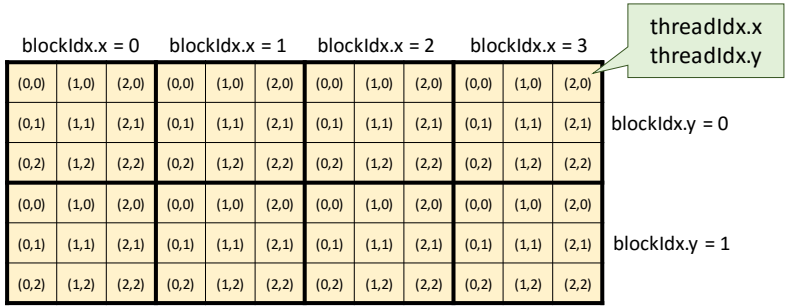


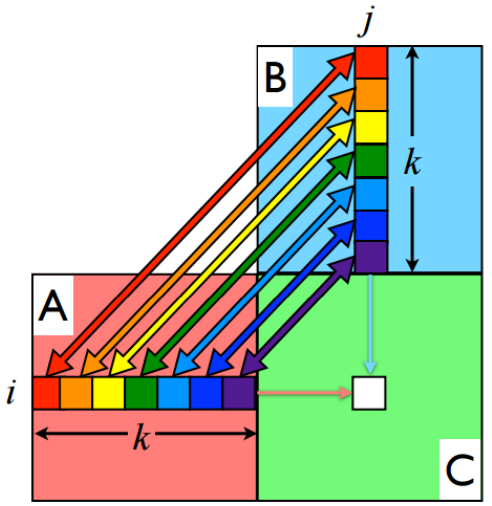
Умножение матриц (CUDA naive)

- **Каждый поток вычисляет один элемент результирующей матрицы C**
- Общее число потоков N * N

```
__global__ void sgemm(const float *a, const float *b, float *c, int n)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < n && col < n) {
        float s = 0.0;
        for (int k = 0; k < n; k++)
            s += a[row * n + k] * b[k * n + col];
        c[row * n + col] = s;
    }
}
{
    int threadsPerBlockDim = 32;
    dim3 blockDim(threadsPerBlockDim, threadsPerBlockDim, 1);
    int blocksPerGridDimX = ceilf(N / (float)threadsPerBlockDim);
    int blocksPerGridDimY = ceilf(N / (float)threadsPerBlockDim);
    dim3 gridDim(blocksPerGridDimX, blocksPerGridDimY, 1);
    sgemm<<<gridDim, blockDim>>>(d_A, d_B, d_C, N);
}
```

Каждый поток обращается к глобальной памяти (*n* loads + 1 store)

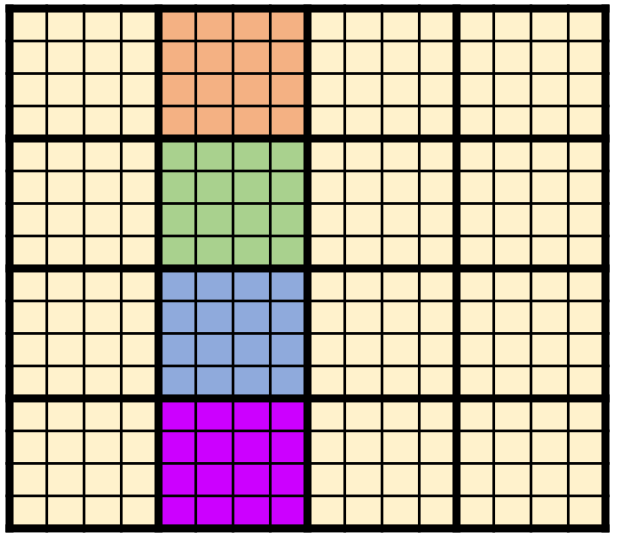




http://users.wfu.edu/choss/CUDA/docs/Lecture%205.pdf

Умножение матриц с разделяемой памятью (tiled)

- Каждый элемент матрицы C вычисляется одним потоком
- Вычисления разбиты на несколько стадий – по числу подматриц
- c[i,j] = a[i,0]*b[0,j] + a[i,1]*b[1,j] + a[i,2]*b[2,j] + a[i,3]*b[3,j] +
  a[i,4]*b[4,j] + a[i,5]*b[5,j] + a[i,6]*b[6,j] + a[i,7]*b[7,j] +
  a[i,8]*b[8,j] + a[i,9]*b[9,j] + a[i,10]*b[10,j] + a[i,11]*b[11,j] +
  a[i,12]*b[12,j] + a[i,13]*b[13,j] + a[i,14]*b[14,j] + a[i,15]*b[15,j];

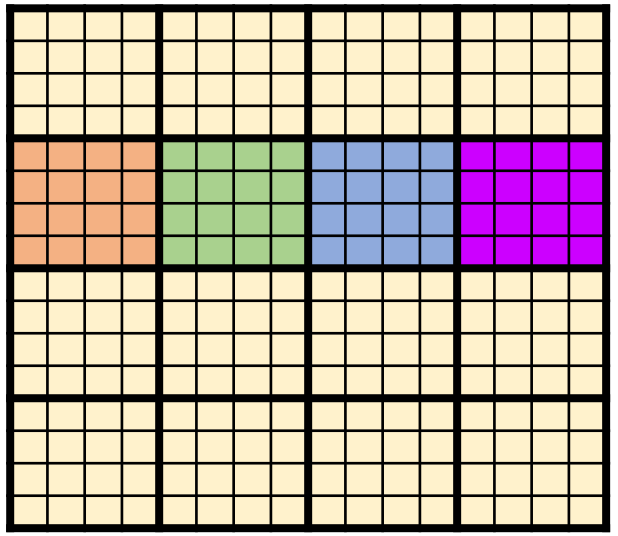


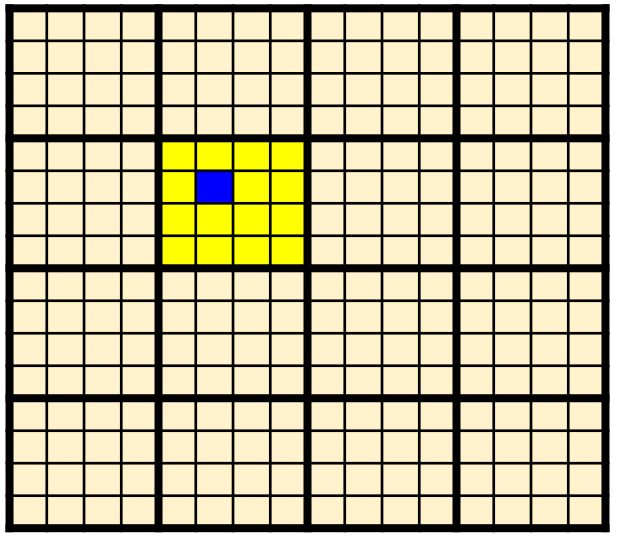
B

- На каждой стадии загружаем подматрицы в разделяемую память и вычисляем часть результатов всеми потоками блока

```
Astart = blockIdx.y * blockDim.y * N
Astep = blockDim.x

Bstart = blockIdx.x * blockDim.x
Bstep = N * blockDim.x
```

A

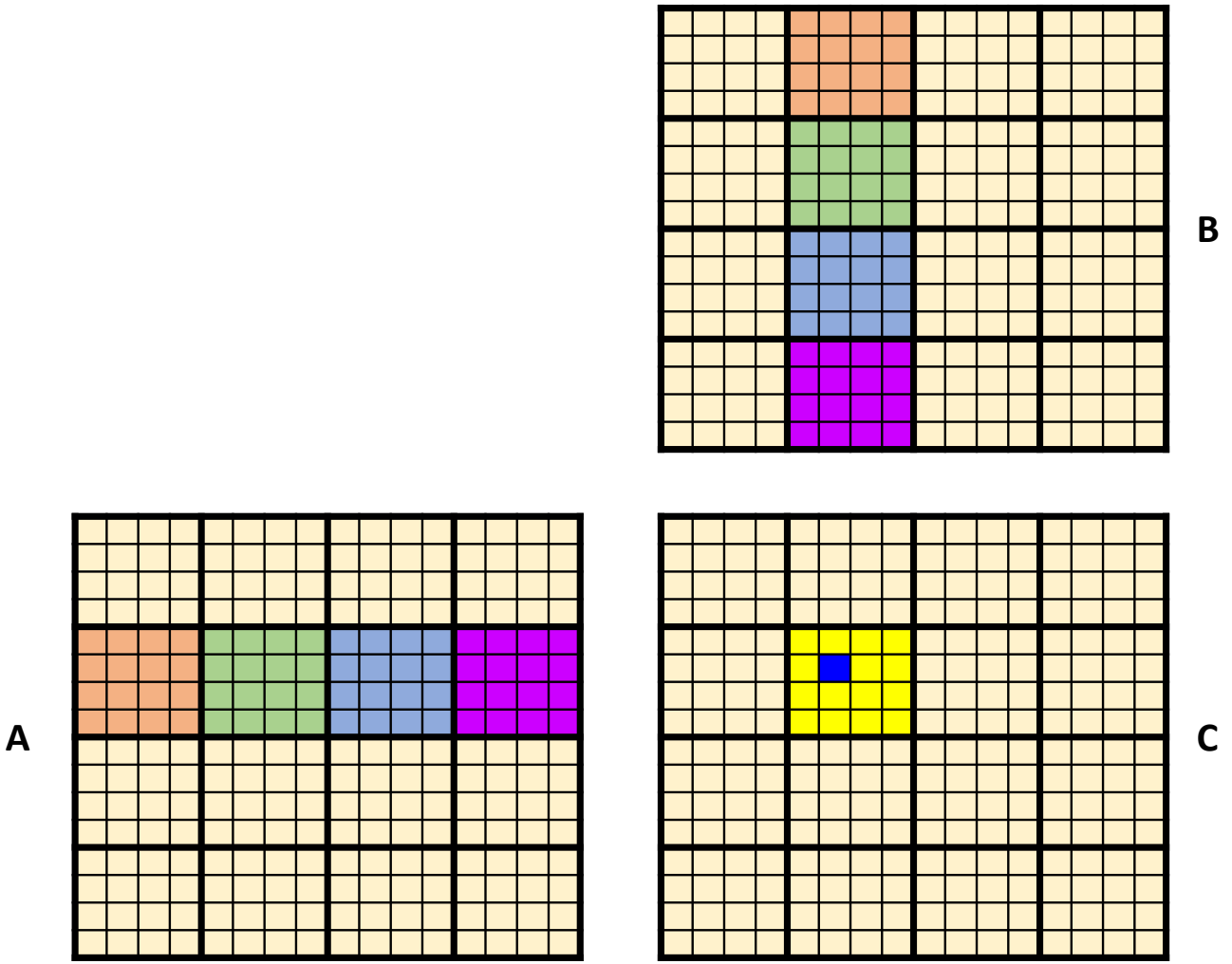




C

Умножение матриц с разделяемой памятью (tiled)

```
__global__ void sgemm_tailed(const float *a, const float *b, float *c, int n)
{
    int tail_size = blockDim.x;
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int bx = blockIdx.x;
    int by = blockIdx.y;

    // Result for c[i, j]
    float sum = 0.0;

    // Index of first tail (sub-matrix) in A
    int Astart = by * n * tail_size;
    int Aend = Astart + n - 1;
    int Astep = tail_size;

    // Index of first tail (sub-matrix) in B
    int Bstart = bx * tail_size;
    int Bstep = n * tail_size;
```



Умножение матриц с разделяемой памятью (tiled)

```
    __shared__ float as[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float bs[BLOCK_SIZE][BLOCK_SIZE];

    int ai = Astart;
    int bi = Bstart;
    while (ai <= Aend) {
        // Load tail to shared memory - each thread load one item
        as[ty][tx] = a[ai + ty * n + tx];
        bs[ty][tx] = b[bi + ty * n + tx];

        // Wait all threads
        __syncthreads();

        // Compute partial result
        for (int k = 0; k < tail_size; k++)
            sum += as[ty][k] * bs[k][tx];

        // Wait for all threads before overwriting of as and bs
        __syncthreads();

        ai += Astep;
        bi += Bstep;
    }

    int Cstart = by * n * tail_size + bx * tail_size;
    c[Cstart + ty * n + tx] = sum;
}
```

Умножение матриц

GeForce GTX 680

Tailed version

```
CUDA kernel launch with 1024 (32 32)
blocks of 1024 (32 32) threads
CPU version (sec.): 3.517567
GPU version (sec.): 0.009149
Memory ops. (sec.): 0.002338
Memory bw. (MiB/sec.): 5133.26
CPU GFLOPS: 0.61
GPU GFLOPS: 234.72
Speedup: 384.47
Speedup (with mem ops.): 306.23
```

Naive version

```
CUDA kernel launch with 1024 (32 32)
blocks of 1024 (32 32) threads
CPU version (sec.): 2.824214
GPU version (sec.): 0.023105
Memory ops. (sec.): 0.002345
Memory bw. (MiB/sec.): 5117.08
CPU GFLOPS: 0.76
GPU GFLOPS: 92.94
Speedup: 122.23
Speedup (with mem ops.): 110.97
```

x2.5

GeForce GT 630

Tailed version

```
CUDA kernel launch with 1024 (32 32)
blocks of 1024 (32 32) threads
CPU version (sec.): 3.009662
GPU version (sec.): 0.089633
Memory ops. (sec.): 0.002516
Memory bw. (MiB/sec.): 4769.42
CPU GFLOPS: 0.71
GPU GFLOPS: 23.96
Speedup: 33.58
Speedup (with mem ops.): 32.66
```

Naive version

```
CUDA kernel launch with 1024 (32 32)
blocks of 1024 (32 32) threads
CPU version (sec.): 3.101753
GPU version (sec.): 0.254254
Memory ops. (sec.): 0.002534
Memory bw. (MiB/sec.): 4735.76
CPU GFLOPS: 0.69
GPU GFLOPS: 8.45
Speedup: 12.20
Speedup (with mem ops.): 12.08
```

x2.8