Семинар 15
Технология CUDA
Иерархия памяти

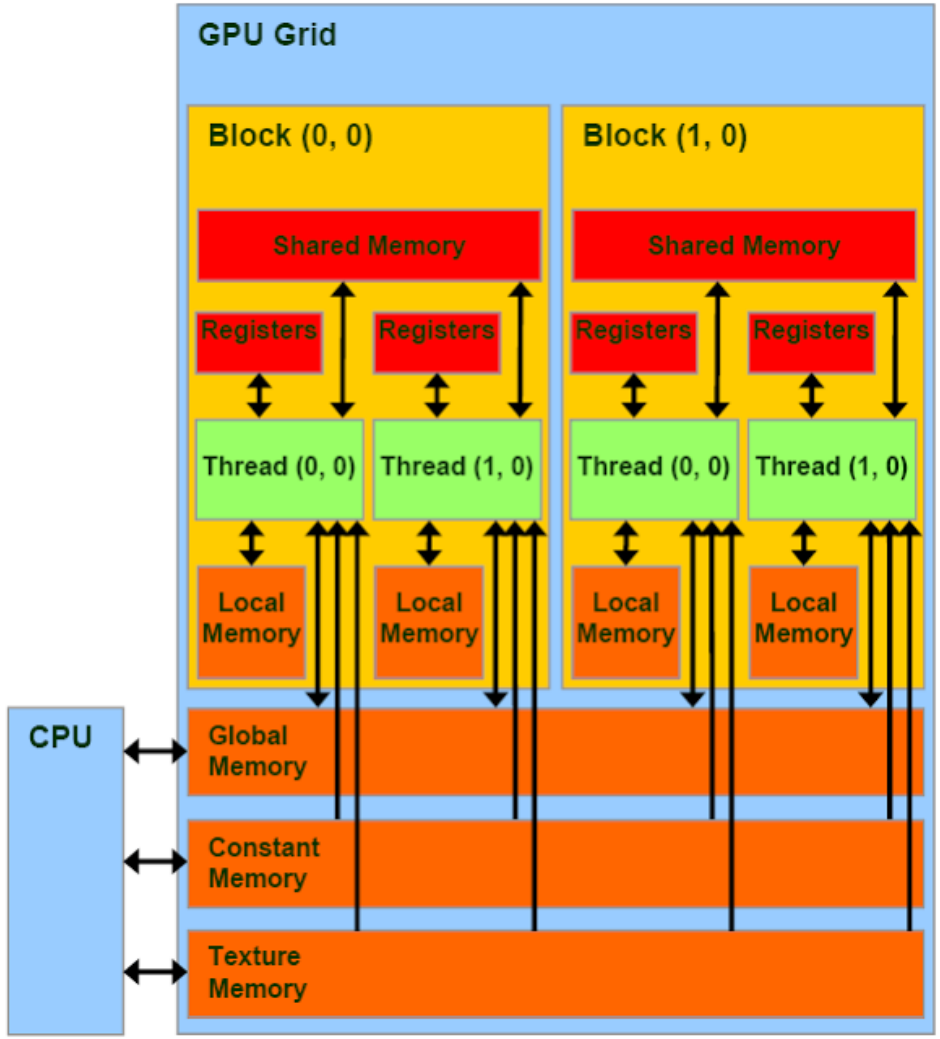
Михаил Курносов

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

Цикл семинаров «Основы параллельного программирования»
Институт физики полупроводников им. А. В. Ржанова СО РАН
Новосибирск, 2015

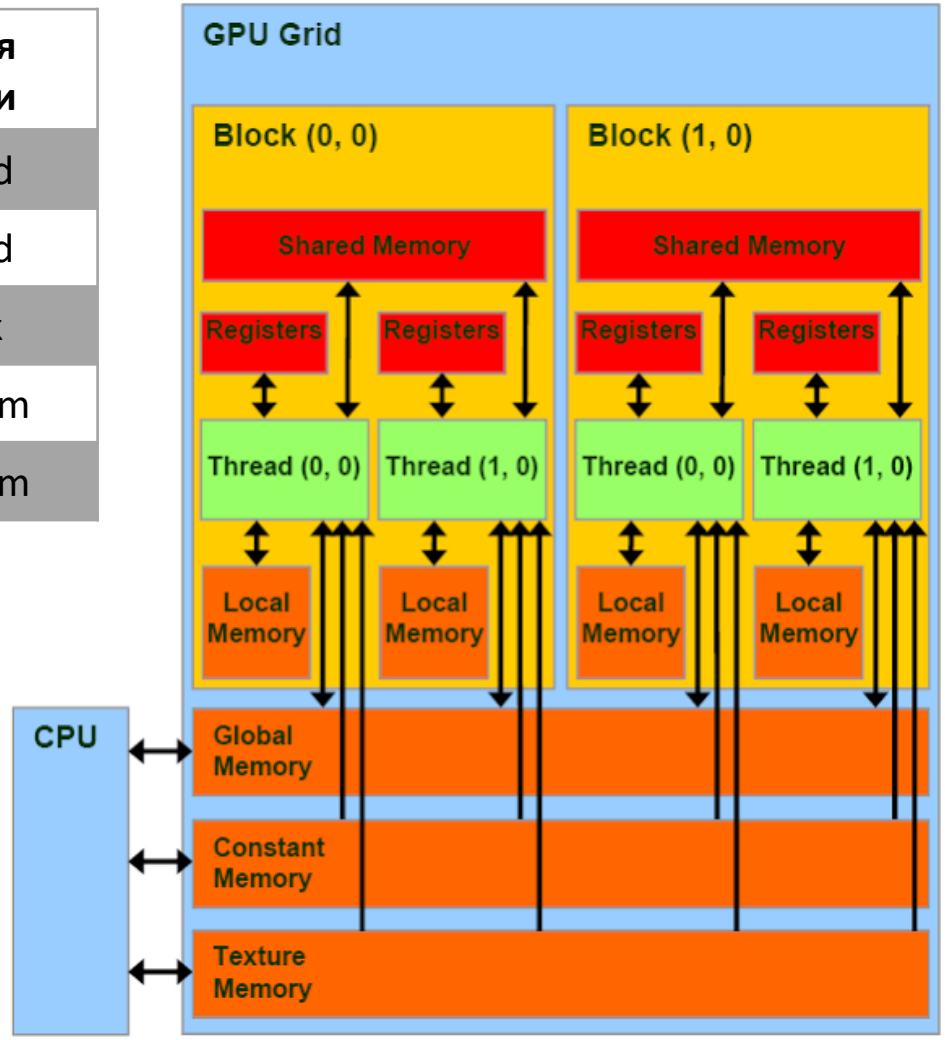
Иерархия памяти (comp. capability >= 3.0)

- **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)**



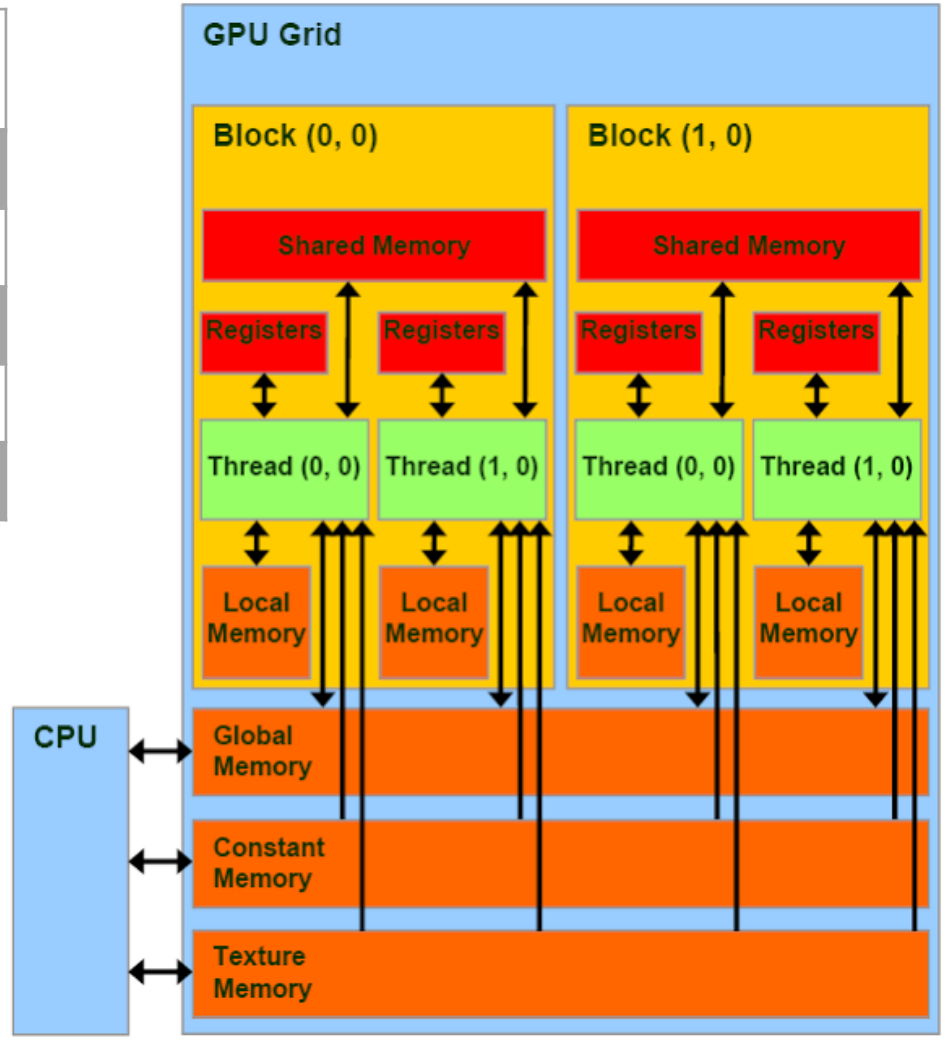
Иерархия памяти (comp. capability >= 3.0)

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)

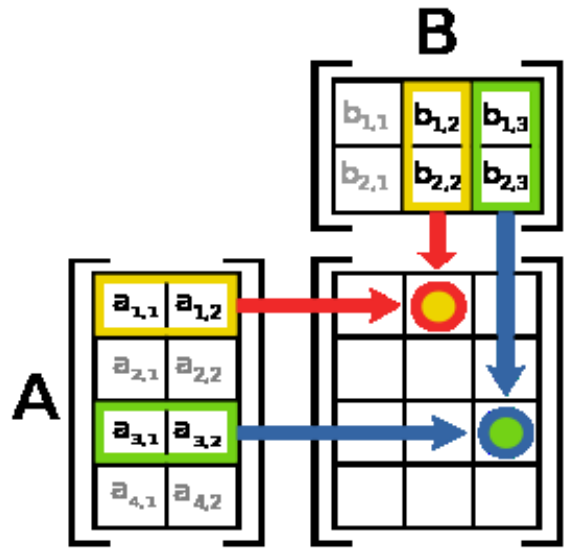
```
// Загружаем данные из глобальной памяти в регистр
float localA = dA[blockIdx.x * blockDim.x + threadIdx.x];

// 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;
        }
    }
}
```

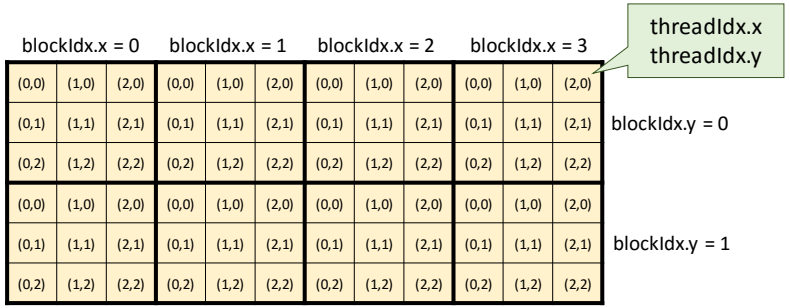


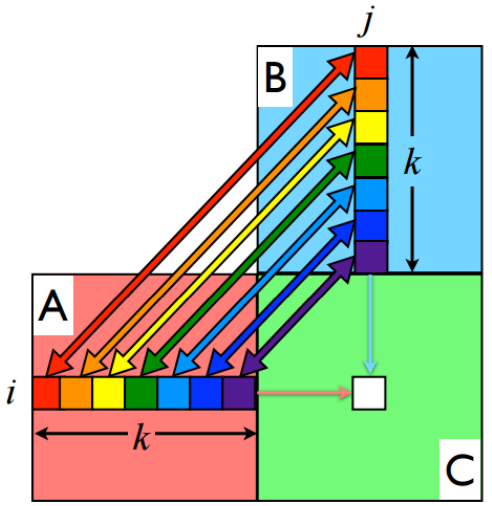
Умножение матриц (CUDA naive)

- **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)

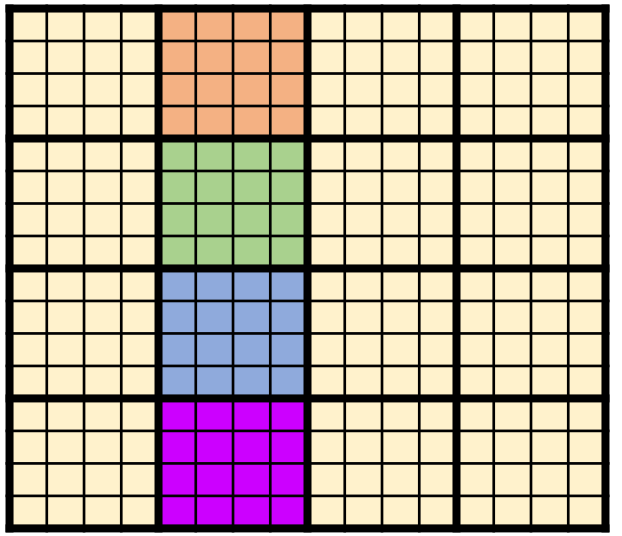




http://users.wfu.edu/choss/CUDA/docs/Lecture%205.pdf

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];

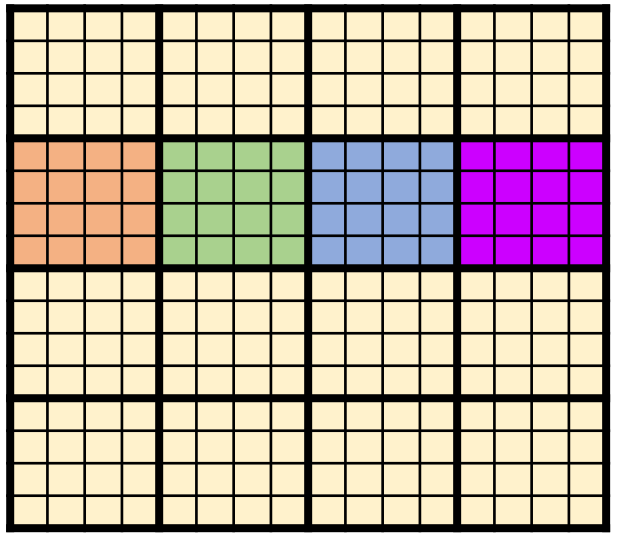


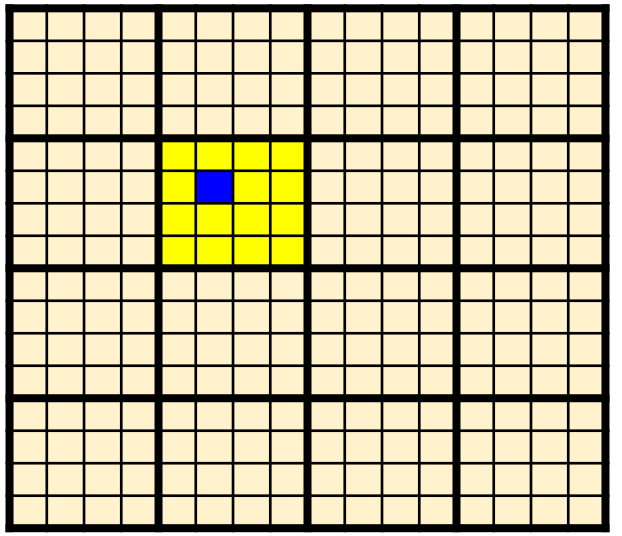
B

- 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

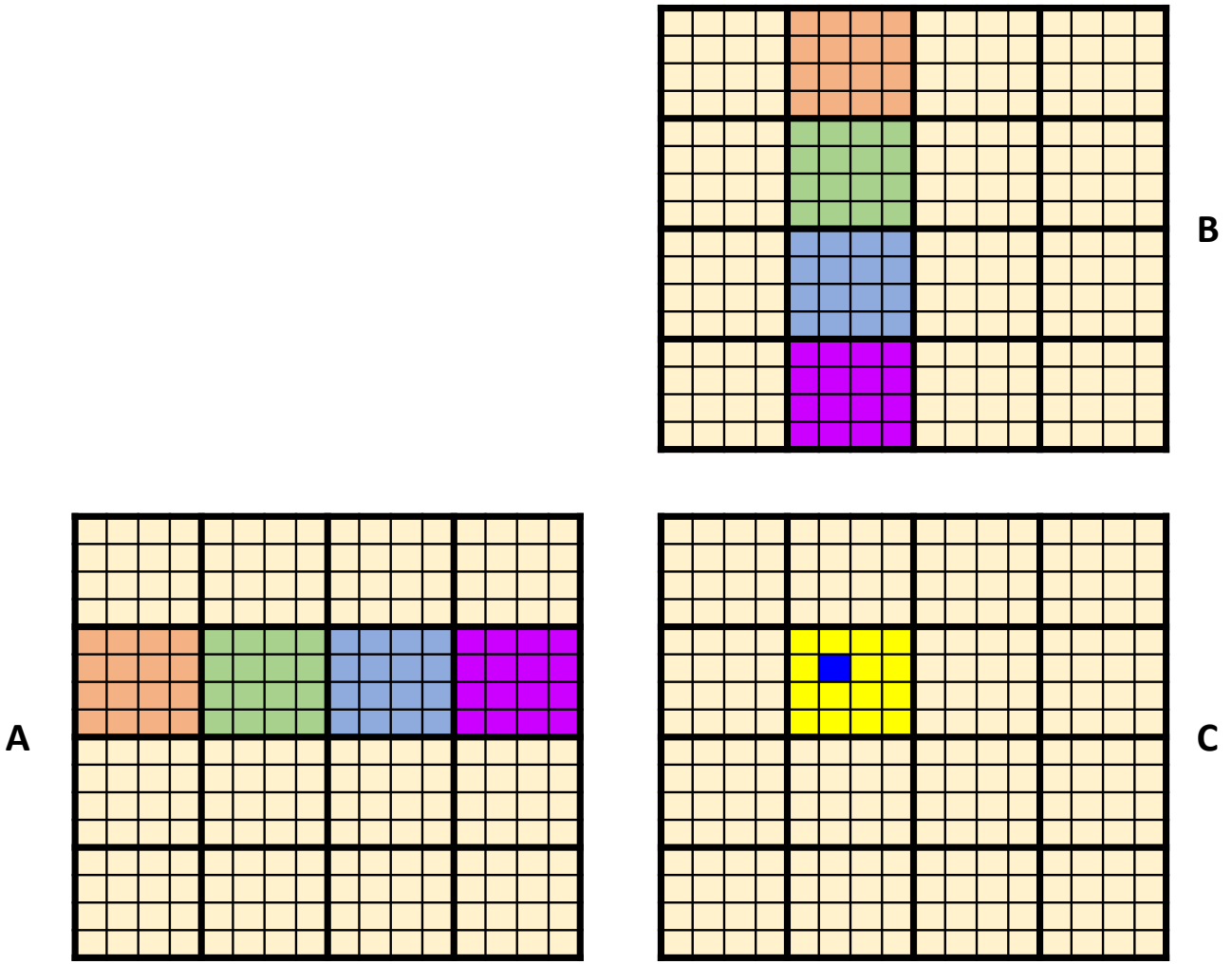




C

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