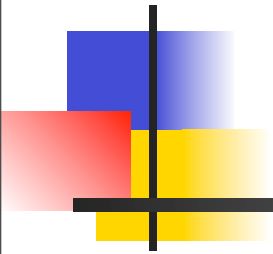


intro



Оптимизация программ. Примеры



Лекция 3

Лектор: **Селиверстов
Евгений Юрьевич**

Направления оптимизации

- Оптимизация алгоритма
- Оптимизация вычислительных ресурсов
- Оптимизация передачи данных
- Оптимизация доступа к памяти

Параллелизм алгоритма

Закон Амдала

$$S_{max} = \frac{1}{\alpha + \frac{1-\alpha}{p}}$$

Применимые типы параллелизма:

- по данным
- геометрический

Параллелизм алгоритма

- Повышение степени параллелизма
- Уменьшение доли последовательного кода
- Повышение арифметической интенсивности
- Вычисление вместо хранения
- Декомпозиция на уровне задач, данных
- Мелкозернистый параллелизм
- Параметризация алгоритма.

Эффективное отображение параллельного алгоритма на архитектуру ГП.

Оптимизация вычислительных ресурсов

- Повышение загрузки мультипроцессора
- Учет пропускной способности операций

Повышение загрузки мультипроцессора

Резидентные блоки (до 8)

Резидентные варпы (до 24/32/48)

Загрузка (оссирансу) - заполненность мультипроцессора резидентными варпами. Скрывает латентность памяти.

Определяется статически:

- потреблением потоками регистров
- конфигурацией сети
- потреблением общей памяти

Рекомендации по конфигурации сети

MP мультипроцессоров, M блоков, N потоков.

- $M \geq 2 * MP$
- $N \bmod 32 = 0$
- $N \geq 64$, если $M > MP$
- рекомендуется $N = 128 \dots 256$
- $f(N, M, MP, CC, U) > 0$ – ограничение ресурсов ГП

Метрики производительности

Аппаратные характеристики

Теоретические пределы

Пропускная способность памяти

B (Гб / с)

$$B = \frac{clock \cdot 10^6 \cdot \frac{bus}{8} \cdot 2}{10^9}$$

Производительность инструкции

TI (операций / такт) – по архитектуре

Метрики производительности

Производительность алгоритма и программы

Эффективная пропускная способность памяти

BE (Гб / с)

$$BE = \frac{data_read + data_written}{T * 10^9}$$

Производительность программы

TP (GFLOPS)

$$TP = \frac{floating_op_count}{T * 10^9}$$

Производительность инструкций

Производительность инструкции:

- время чтения операндов
- время выполнения инструкции
- время записи результата

Время выполнения инструкции определяется:

- архитектурой устройства и операцией (типы и размеры операндов).
- внутренней разрядностью – 24, 32, 64 бита.
- точностью – \approx IEEE 754

Производительность инструкций

Операция \ CC	1.x	2.0	2.1
float	8	32	48
double	1	16	4
24-битный int	8	---	---
32-битный int (*)	---	16	16
32-битный int (+,bool)	8	32	48
функции	2	4	8
синхронизация	8	16	16

Производительность инструкций

Ветвление в потоках в пределах варпа приводит к дивергенции потоков.

При дивергенции на выполнение последовательно назначаются различные ветви условия.

Перенос различных ветвей условия в различные варпы.

Оптимизация передачи данных

- минимизация копирований хост-устройство
- использование зафиксированных страниц памяти
- асинхронное копирование данных (потoki)
- перекрывающееся выполнение ядер
- перекрывающееся копирование и выполнение

Оптимизация доступа к памяти

Три направления:

- оптимизация работы с глобальной памятью
- оптимизация работы с общей памятью
- оптимизация работы с текстурами

Оптимизация доступа к памяти

Доступ к глобальной памяти - $T = 400-800$ тактов.

Одновременное выполнение $N=16$ потоков.

Доступ всех потоков в рамках полуварпа:

- пессимистичное - $T*N$
- оптимистичное - T

Планировщик извлекает сегмент данных для мультипроцессора (всех потоков).

Объединенный доступ

Выровненный (объединенный) доступ.

Описывает требования:

- выравнивание стартового адреса
- размер сегмента выборки
- правила доступа потоками в полуварпе

Объединенный доступ

Требования к выравниванию адреса

Используются стандартные CUDA-типы или спецификаторы `__align__` для структур.

Адреса в выборке выравниваются по 1...16 байт в зависимости от типа.

Для двумерных массивов автоматически выровненный адрес - выделение памяти через `cudaMallocPitch`.

Объединенный доступ СС 1.0/1.1

Размер и число сегментов зависит от размера слов

- 4 байта - один сегмент, 64 байта
- 8 байт - один сегмент, 128 байт
- 16 байт - два сегмента, 128 байт

Протокол:

Доступ k -го потока полуварпа к k -му слову сегмента.

Строгая последовательность

Объединенный доступ СС 1.2/1.3

Размер сегмента 128 байт.

Размер сегмента 32 или 64 байта для типов <4б

Возможно автоматическое уменьшение размера сегмента.

Протокол:

Последовательность и кратность доступа к словам в сегментах неважны.

Объединенный доступ СС 2.0

Размер сегмента зависит от размера слов:

- 4 байта - один сегмент, 128 байт
- 8 байт - два сегмента, 128 байт
- 16 байт - четыре сегмента, 128 байт

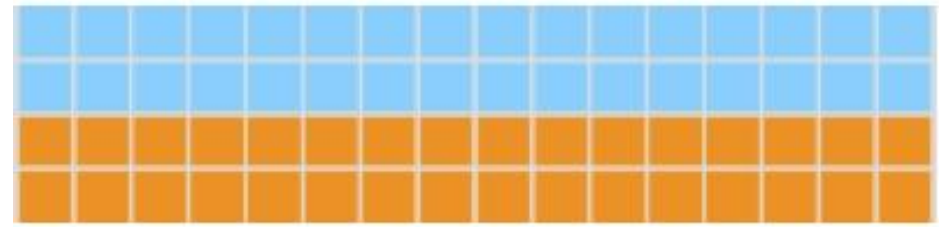
Сегменты кэшируются в кэше L1.

Линия кэша – 128 байт.

Выборки 1-4 сегментов параллельные.

Примеры объединенного доступа

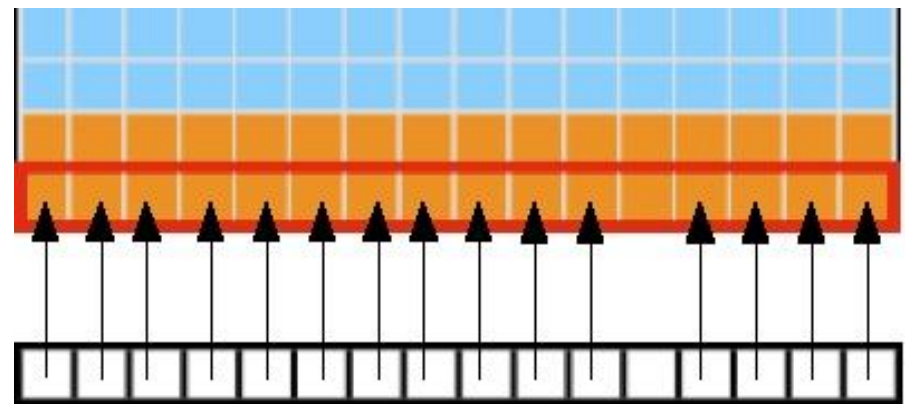
Сегменты: 1
 2



Полуварп (16 потоков):

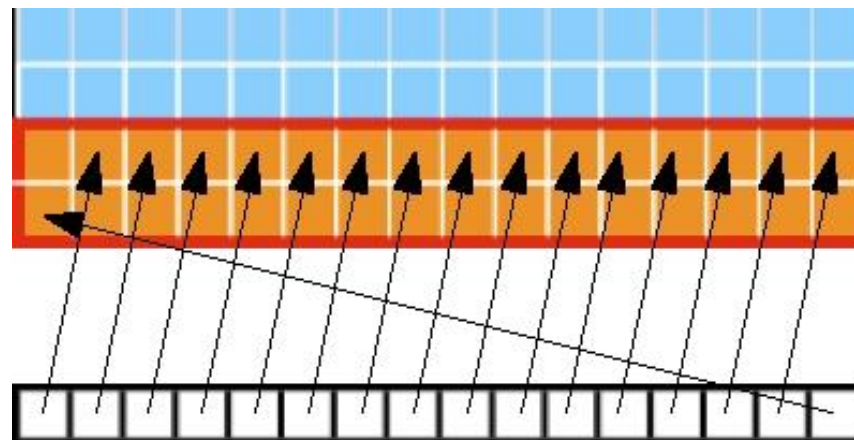


1. Последовательный:
1 выборка

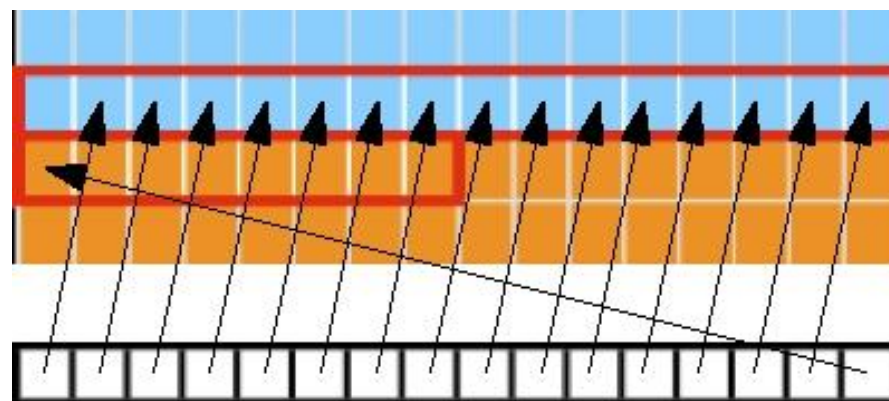


Примеры объединенного доступа

2. Со смещением,
в пределах сегмента:
СС 1.0 – 16 выборок
СС 1.2 – 1 выборка



3. Со смещением,
в двух сегментах:
СС 1.0 – 16 выборок
СС 1.2 – 2 выборки



Разделяемая память

Тип: на мультипроцессоре, SRAM

Разрядность: 32-битная.

Организация: послойная (банки)

СС 1.х: 16 кб, 16 банков

СС 2.х: 48 кб, 32 банка

Выборки: 1 адрес за 1 такт на 1 банк

Применение:

- для обмена данными внутри блока
- для локальных данных

Разделяемая память

Конфликты банков

При доступе нескольких потоков к одному банку (не обязательно к одному слову из банка).

Конфликт *k*-way – доступ *k* потоков к банку. Тогда запросы сериализуются в *k* отдельных.

Бесконфликтный доступ (СС 1)

Конфликт при доступе потоков к одному банку.

1. Доступ i -го потока в полуварпе к j -му слову

Строгая последовательность не требуется.

```
__shared__ float shared[32];  
float data = shared[BaseIndex + s * tid];
```

2. Доступ к k -го слову n потоками (бroadкаст)

$$i \in [1 \dots n], j \in [1 \dots n] \quad n = 32, k = const$$

Бесконфликтный доступ (СС 2)

Конфликт при доступе потоков к разным словам в одном банке.

1. Доступ i -го потока в варпе к j -му слову
2. Доступ $n^* < n$ потоков к k -му слову
3. Доступ к k -го слову n потоками (несколько бродкастов за выборку)

$$i \in [1 \dots n], j \in [1 \dots n] \quad n = 32, k = const$$

Бесконфликтный доступ

Размеры слов:

- 32-битные слова - обычный доступ
- 8/16-битные слова - выборка 32-битного слова,
далее выбор 8/16 бит
- 64-битные слова - выборка двух 32-битных слов,
далее склеивание

Текстурная и константная память

Данные считываются из глобальной памяти в текстурный кэш на мультипроцессоре.

Организация потока данных в виде текстур.

Локальность доступа - 1d/2d/3d в пределах размера текстуры.

Примеры

1. Сложение векторов
2. Умножение матриц
3. Эффективное умножение матриц

Пример 1, CPU

```
    // Host code
1. int main(int argc, char** argv)
2. {
3.     const int N = 50000;
4.     const size_t size = N * sizeof(float);
5.     int i;
6.     float *d_A, *d_B, *d_C;
7.
8.     // Allocate input vectors h_A and h_B in host memory
9.
10.    float* h_A = (float*)malloc(size);
11.    float* h_B = (float*)malloc(size);
12.    float* h_C = (float*)malloc(size);
13.
14.    // Initialize input vectors
15.    for (int i = 0; i < N; ++i)
16.    {
17.        h_A[i] = rand() / (float)RAND_MAX;
18.        h_B[i] = rand() / (float)RAND_MAX;
19.    }
    .....
```

Пример 1, CPU

```
.....  
20. // Allocate vectors in device memory  
21. cutilSafeCall( cudaMalloc((void**) &d_A, size) );  
22. cutilSafeCall( cudaMalloc((void**) &d_B, size) );  
23. cutilSafeCall( cudaMalloc((void**) &d_C, size) );  
24.  
25. cutilSafeCall( cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice) );  
26. cutilSafeCall( cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice) );  
27.  
28. const int threadsPerBlock = 256;  
29. const int blocksPerGrid = (N+threadsPerBlock-1) / threadsPerBlock;  
30.  
31. VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);  
32. cutilCheckMsg("kernel launch failure");  
33.  
34. // Copy result from device memory to host memory  
35. cutilSafeCall( cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost) );  
  
.....
```


Пример 1, CPU

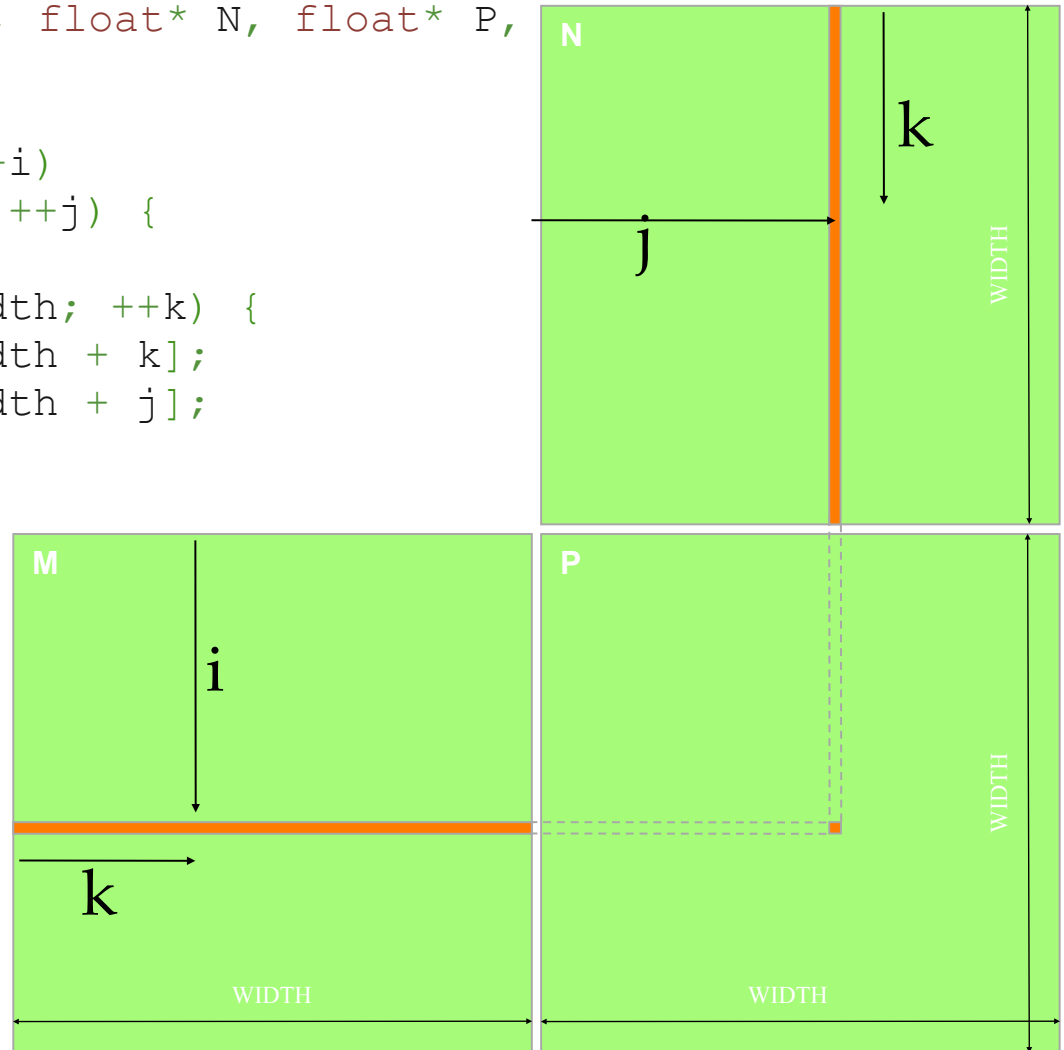
```
.....
26. // Verify result
27. for (i = 0; i < N; ++i)
28.     if (fabs(h_C[i] - h_A[i] - h_B[i]) > 1e-5)
29.         break;
30. printf("%s \n", (i == N) ? "PASSED" : "FAILED");
31.
32. cudaFree(d_A);
33. cudaFree(d_B);
34. cudaFree(d_C);
35.
36. // Free host memory
37. free(h_A);
38. free(h_B);
39. free(h_C);
40.
41. cutilSafeCall( cudaThreadExit() );
42.
43. exit(0);
44. }
```

Пример 1, GPU

```
// Device code
1. __global__ void VecAdd(const float*A, const float*B, float*C, int N)
2. {
3.     int i = blockDim.x * blockIdx.x + threadIdx.x;
4.     if (i < N)
5.         C[i] = A[i] + B[i];
6. }
```

Пример 2, CPU

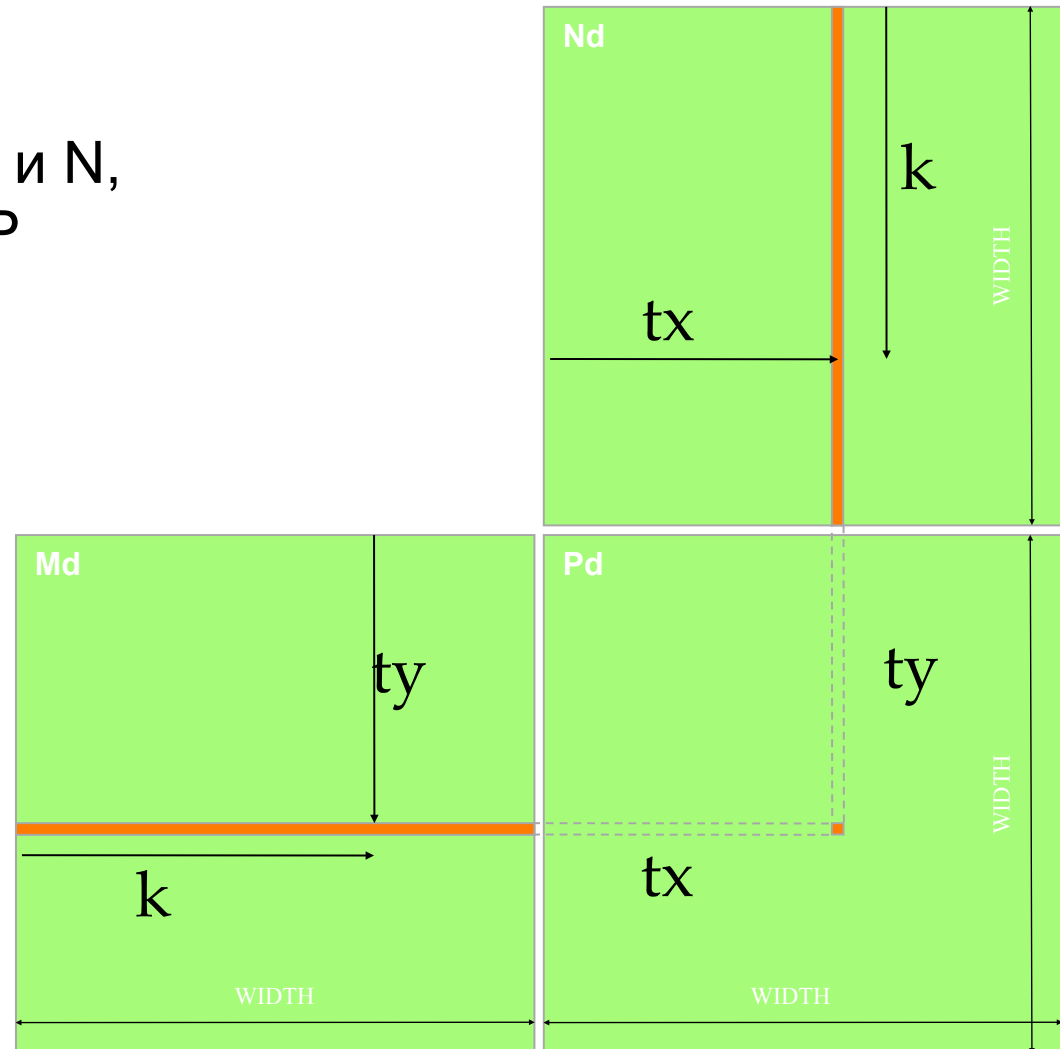
```
void MatrixMulOnHost(float* M, float* N, float* P,
1.     int Width)
2. {
3.     for (int i = 0; i < Width; ++i)
4.         for (int j = 0; j < Width; ++j) {
5.             float sum = 0;
6.             for (int k = 0; k < Width; ++k) {
7.                 float a = M[i * width + k];
8.                 float b = N[k * width + j];
9.                 sum += a * b;
10.            }
11.            P[i * Width + j] =
12.                sum;
13.        }
14. }
```



Пример 3

Схема распараллеливания

Глобальная память:
 W^4 чтений из матриц M и N ,
 W^2 записей в матрицу P



Пример 3, CPU

```
void MatrixMulOnDevice(float* M, float* N, float* P, int W)
1. {
2.     int size = W * W * sizeof(float);
3.     float* Md, Nd, Pd;
4.     cudaMalloc(&Md, size);
5.     cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
6.
7.     cudaMalloc(&Nd, size);
8.     cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);
9.
10.    cudaMalloc(&Pd, size);
11.
12.    dim3 dGrid(1, 1);
13.    dim3 dBlock(W, W);
14.    MatrixMulKernel<<<dGrid, dBlock>>>(Md, Nd, Pd, W);
15.
16.    cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
17.
18.    cudaFree(Md); cudaFree(Nd); cudaFree(Pd);
19. }
```

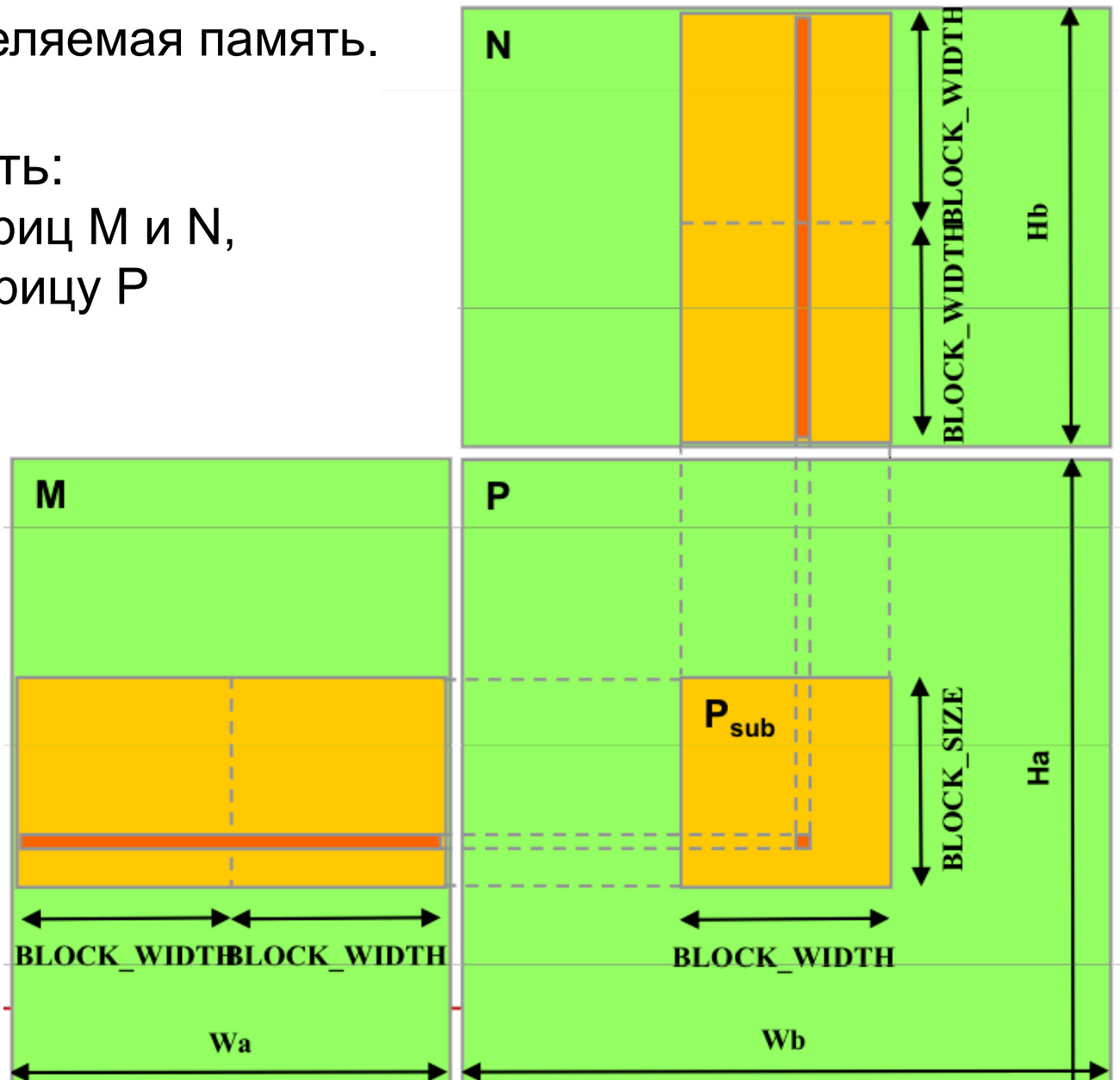
Пример 3, GPU

```
1.  __global__ void MatrixMulKernel(float* Md, float* Nd,  
2.  float* Pd, int Width)  
3.  {  
4.  float Pvalue = 0;  
5.  for (int k = 0; k < Width; ++k)  
6.  {  
7.      float Melement = Md[threadIdx.y*Width+k];  
8.      float Nelement = Nd[k*Width+threadIdx.x];  
9.      Pvalue += Melement * Nelement;  
10. }  
11. Pd[threadIdx.y*Width+threadIdx.x] = Pvalue;  
12. }
```

Пример 4

Применяется разделяемая память.

Глобальная память:
 W^2 чтений из матриц M и N ,
 W^2 записей в матрицу P



Пример 4, CPU

```
#define BLOCK_SIZE 16
1. void Mul(float*A, float*B, int hA, int wA, int wB, float* C)
2. {
3.     int size = hA * wA * sizeof(float);
4.     float *Ad, *Bd, *Cd;
5.     cudaMalloc((void**)&Ad, size);
6.     cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);
7.     size = wA * wB * sizeof(float);
8.     cudaMalloc((void**)&Bd, size);
9.     cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);
10.    size = hA * wB * sizeof(float);
11.    cudaMalloc((void**)&Cd, size);
12.
13.    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
14.    dim3 dimGrid(wB / dimBlock.x, hA / dimBlock.y);
15.    // Launch the device computation
16.    Mul<<<dimGrid, dimBlock>>>(Ad, Bd, wA, wB, Cd);
17.    // Read C from the device
18.    cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);
19.    cudaFree(Ad); cudaFree(Bd); cudaFree(Cd);
20. }
```


Пример 4, GPU

```
__global__ void Mul(float*A, float*B, int wA, int wB, float* C)
{
1.  int bx = blockIdx.x, by = blockIdx.y;
2.  int tx = threadIdx.x, ty = blockIdx.y;
3.
4.  // first and last sub-matrix of A processed by the block
5.  int aBegin = wA * BLOCK_SIZE * by;
6.  int aEnd = aBegin + wA - 1;
7.  // Step size used to iterate through the sub-matrices of A
8.  int aStep = BLOCK_SIZE;
9.
10.  int bBegin = BLOCK_SIZE * bx;
11.  int bStep = BLOCK_SIZE * wB;
12.
13.  // Step size used to iterate through the sub-matrices of B
14.  float Csub = 0;
15.  for (int a= aBegin, b= bBegin; a <= aEnd; a+=aStep, b+=bStep)
16.  {
17.      // Shared memory for the sub-matrix of A, B
18.      __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
19.      __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
```

Пример 4, GPU

```
    // each thread loads one element of each matrix
21.   As[ty][tx] = A[a + wA * ty + tx];
22.   Bs[ty][tx] = B[b + wB * ty + tx];
23.
24.   __syncthreads();
25.   // Make sure the matrices are loaded
26.   // Multiply the two matrices together;
27.   // each thread computes one element
28.   for (int k = 0; k < BLOCK_SIZE; ++k)
29.       Csub += As[ty][k] * Bs[k][tx];
30.
31.   // Make sure that the preceding computation is done
32.   __syncthreads();
33. }
34. // Write the block sub-matrix to global memory;
35. // each thread writes one element
36. int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
37. C[c + wB * ty + tx] = Csub;
38. }
```

Ресурсы этого курса

<http://omniverse.ru/bmstu/cuda-rk6>



Наш FTP: <ftp://cad.bmstu.ru/manichev/ZHUK-CUDA-6-kurs/>

NVidia: <http://nvidia.com/cuda>

CUDA C Programming Guide

CUDA C Best Practices Guide

CUDA Toolkit Reference Manual

Programming Massively Parallel Processors: <http://amzn.to/99ulEZ>

CUDA By Example: <http://amzn.to/9lbtEw>

Курс МГУ: <http://groups.google.com/group/cudacsmsusu>