Технология CUDA для высокопроизводительных вычислений на кластерах с GPU

Лихогруд Николай
n.lihogrud@gmail.com

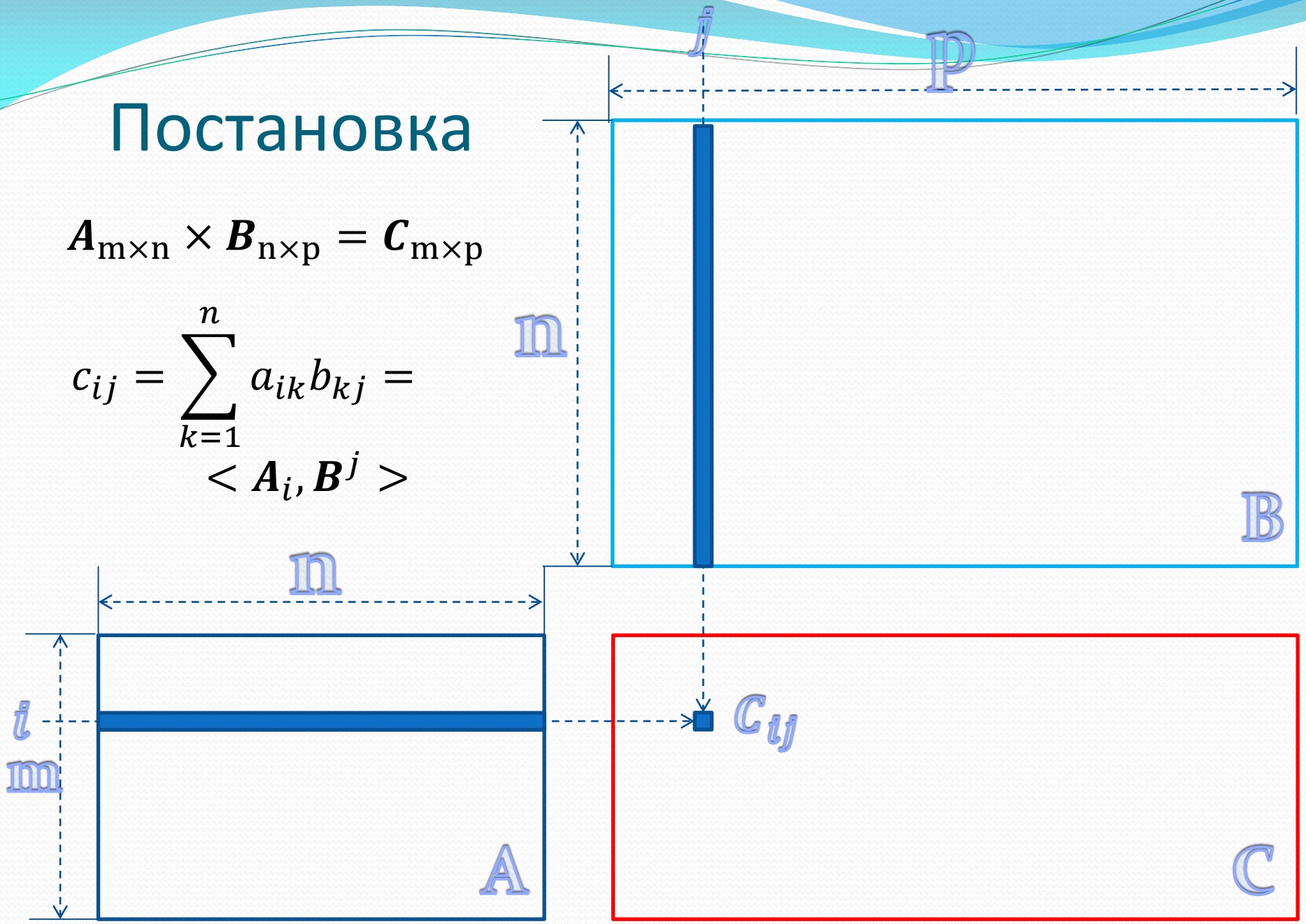
Задание

Часть 1: Простейшее перемножение матриц

Постановка

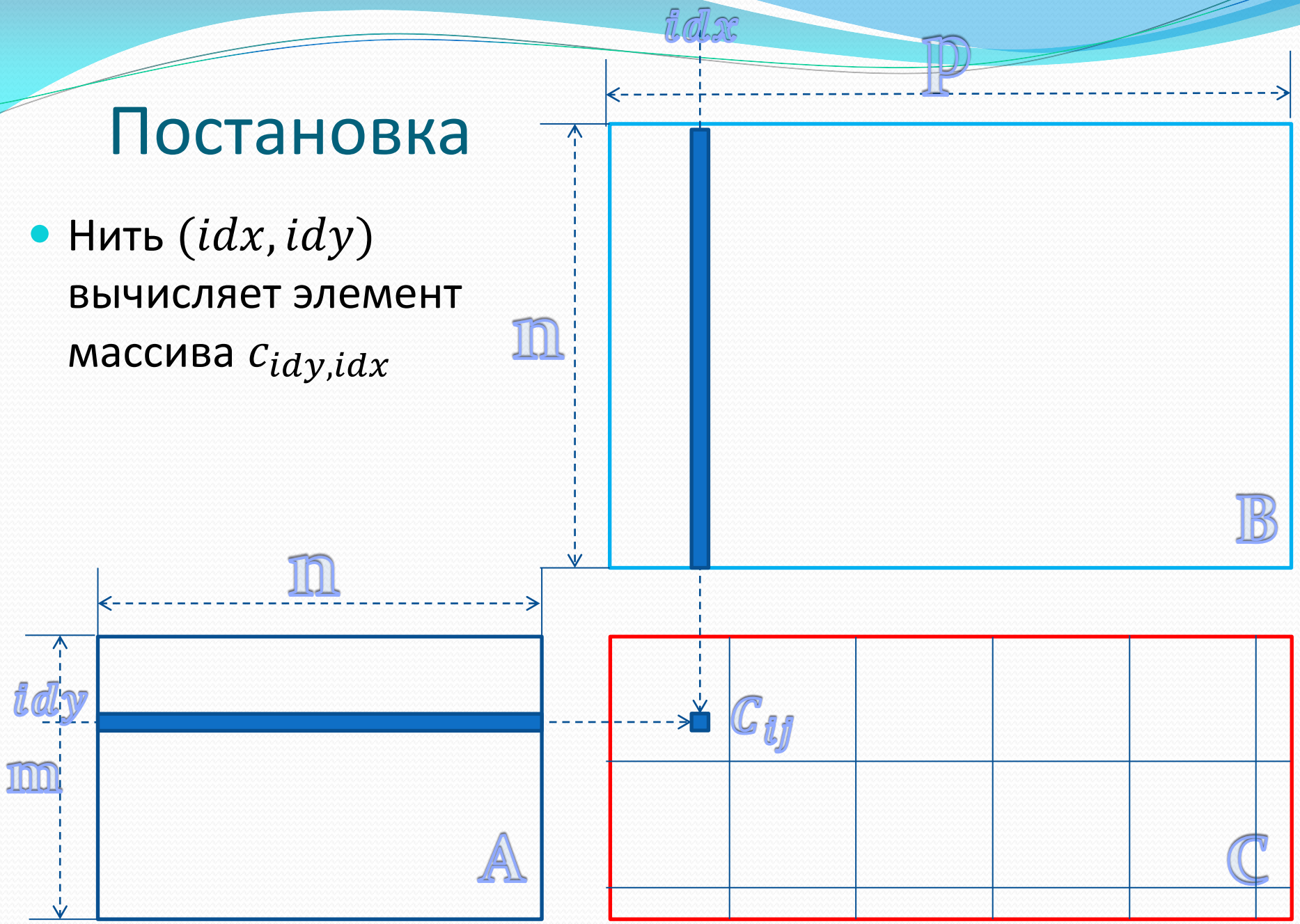
$$A_{m\times n} \times B_{n\times p} = C_{m\times p}$$

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} = < A_i, B^j >$$



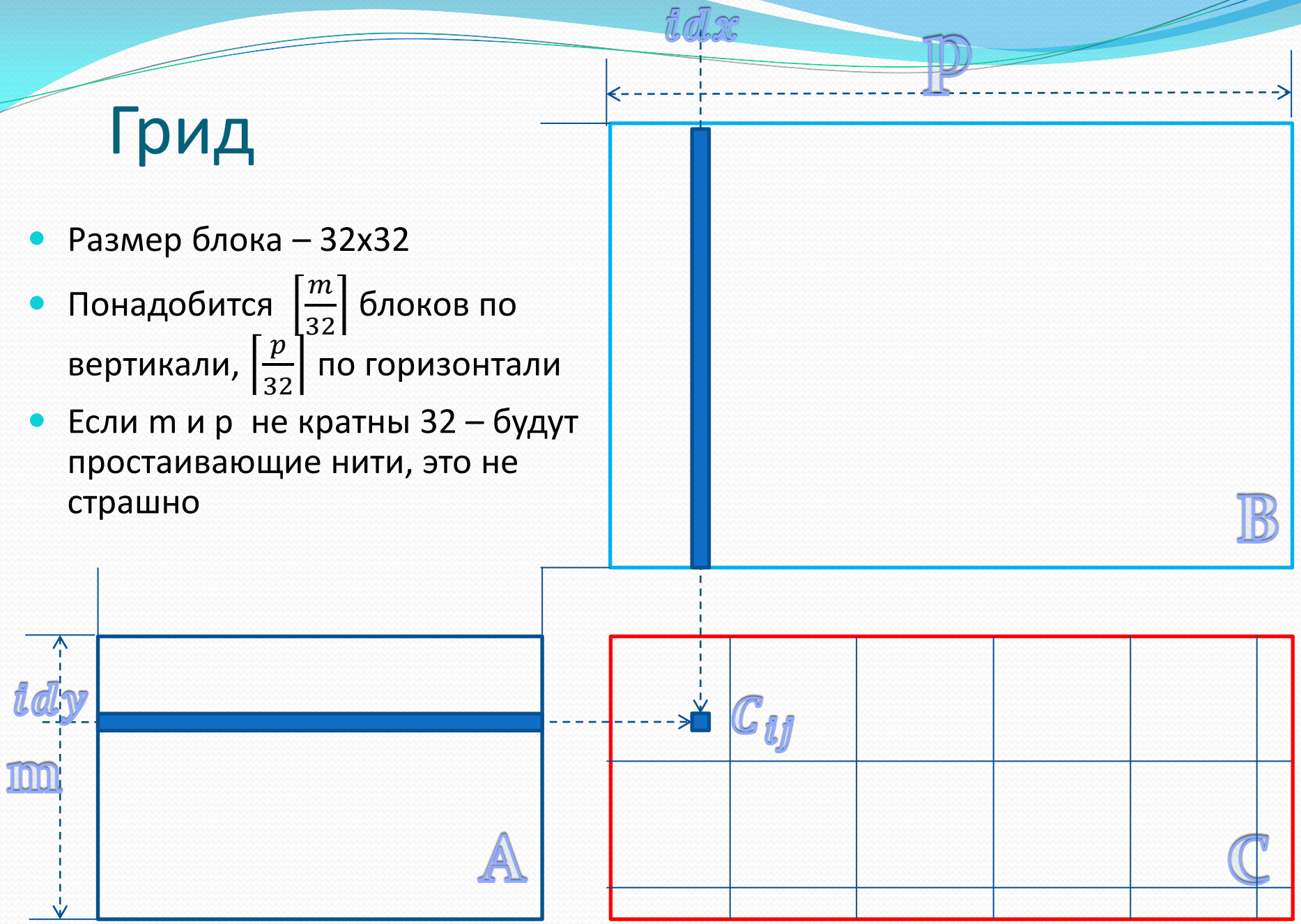
Постановка

- Нить (idx, idy) вычисляет элемент массива $c_{idy,idx}$



Грид

- Размер блока – 32x32
- Понадобится $\left\lceil \frac{m}{32} \right\rceil$ блоков по вертикали, $\left\lceil \frac{p}{32} \right\rceil$ по горизонтали
- Если m и p не кратны 32 – будут простаивающие нити, это не страшно



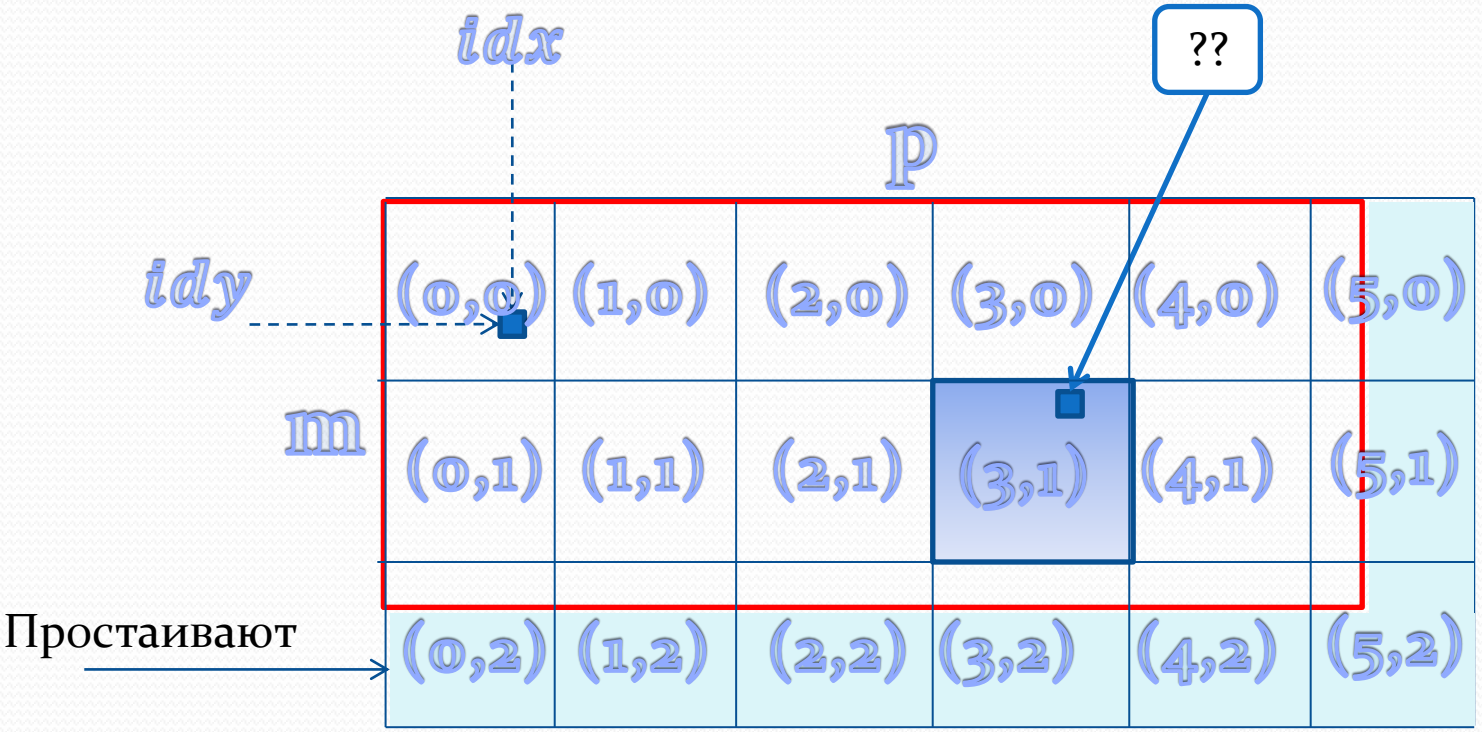
Ядро

```
__global__ void matmul(A,B,C) {

    < Вычислить глобальные индексы нити (idx, idy) в матрице C >;

    if (idx < p && idy < m) {

        < вычислить скалярное произведение строки idy
        матрицы A на столбец idx матрицы B >;

        < записать результат скалярного произведения в элемент
        [idy, idx] матрицы C >;
    }
}
```

Глобальные индексы

- Каковы глобальные индексы нити (`idx, idy`), если
  - индексы нити в блоке - (`threadIdx.x = 22, threadIdx.y = 3`),
  - индексы блока в гриде - (`blockIdx.x = 3, blockIdx.y = 1`),
  - размер каждого блока - (`blockDim.x = blockDim.y = 32`)

?



Индексы CUDA и матричные

- Очень вероятно, что вы будете путаться между `(i,j)` и `(x,y)`:
  - **`i`** – номер строки, соответствует координате по **`y`** в терминологии CUDA
  - **`j`** – номер столбца, соответствует координате по **`x`**

- Путаница будет оттого что элементу **`[i,j]`** в матричной нотации соответсвует элемент **[y, x]** в индексах CUDA (пространственных), но в CUDA первым пишется индекс **`x`**

- Чтобы не путаться, можно глобальные индексы сразу назвать `[i,j]`, а не `[idx, idy]`

Структура cudaPitchedPtr

- Определена в CUDA Toolkit:

```
struct cudaPitchedPtr {
    size_t pitch; // число байтов между началами строк
    void *ptr;    // указатель на память
    size_t xsize;// логическая ширина матрицы в элементах
    size_t ysize;// логическая высота матрицы в элементах
}
```

- `cudaPitchedPtr make_cudaPitchedPtr(void *d, size_t p, size_t xsz, size_t ysz)`
  - создать такую структуру

Обертка матриц матриц

- Вместе с матрицей часто передают её размеры
  - Следует обернуть все матрицы в `cudaPitchedPtr`
  - С обернутыми матрицами легче будет сделать следующую часть

- Матрицы плотные, расположены по строкам
  - Число байтов между началами строк (pitch) = <число столбцов> * <размер элемента>

- Для матрицы с элементами типа `T` из m строк и n столбцов:

```
cudaPitchedPtr C =
    make_cudaPitchedPtr(malloc(n * m * sizeof(T)),
                        n * sizeof(T), n, m);
```

get_elem

- Обращение к элементам матрицы, заданной при помощи cudaPitchedPtr:

```
Type a = ((Type *)((char*)matrix.ptr + Row * matrix.pitch)) [Column];
```

- Удобно определить макрос:

```
#define get_elem(array, Row, Column) \
 (((Type*)((char*)array.ptr + (Row) * array.pitch))[(Column)])

Type a = get_elem(array, 2, 10);
```

Требование к программе

- Матрицы прямоугольные, размеры передаются через параметры запуска
- Тип элементов float, инициализация матриц случайными числами по формуле `rand() / RAND_MAX`
- Обернуть ВСЕ матрицы в `cudaPitchedPtr`, обращаться через `get_elem`
- Реализовать возможность проверки правильности результата (по необязательному последнему флагу в параметрах запуска)
  - Вычисление эталонного произведения матриц на хосте и сравнение с результатом CUDA

Часть 2: оптимизируем работу с памятью

Добавляем padding (набивку)

- На хосте заменяем
  - `cudaMalloc` на `cudaMallocPitch`
  - `cudaMemcpy` на `cudaMemcpy2D`

- На устройстве обращаемся к элементам матриц с учетом их `pitch`
  - Именно для этого обернули матрицы в `cudaPitchedPtr` – чтобы не плодить лишних переменных и хранить `pitch` вместе с указателями

cudaMallocPitch & cudaPitchedPtr

- Выделение памяти под матрицу с элементами типа Type и логическими размерами (в элементах) width x height:

```
cudaPitchedPtr matrix =
      make_cudaPitchedPtr(0,0, width, height);

cudaMallocPitch(&matrix.ptr, &matrix.pitch,
                width * sizeof( Type ), height);
```

  - Если Type==int32_t, а width==1000, то будет выделено 4096* height байтов и в matrix.pitch запишется 4096

cudaMemcpy2D & cudaPitchedPtr

- Пример пересылки с хоста на GPU:

```
cudaPitchedPtr aHost, aDev;
 …// выделение памяти под матрицу на GPU и на хосте
 cudaMemcpy2D( aDev.ptr,  aDev.pitch, aHost.ptr,
               aHost.pitch, aHost.pitch, aHost.ysize,
               cudaMemcpyHostToDevice);
```

  - aHost – матрица на хосте, сплошная (без набивки), значит её фактическая ширина(pitch) равна обычной ширине в байтах

  - aDev – матрица на устройстве, память под неё выделена на при помощи cudaMallocPitch, как было описано выше

Правильная работа с памятью

- Каждая нить рассчитывает один элемент матрицы-результата
- Желательно, чтобы
  - обращения в память нитей **одного варпа** попадали в промежуток памяти размером 128 байт
  - адрес начала этого промежутка делится на 128 (выровнен по 128)
- В этом случае обращения варпа в память будут выполнены за одну транзакцию к кешу L1

Правильная работа с памятью

- При вычислении скалярного произведения нить пробегает столбец матрицы B и строку матрицы A

Нужно ли транспонировать матрицу B?

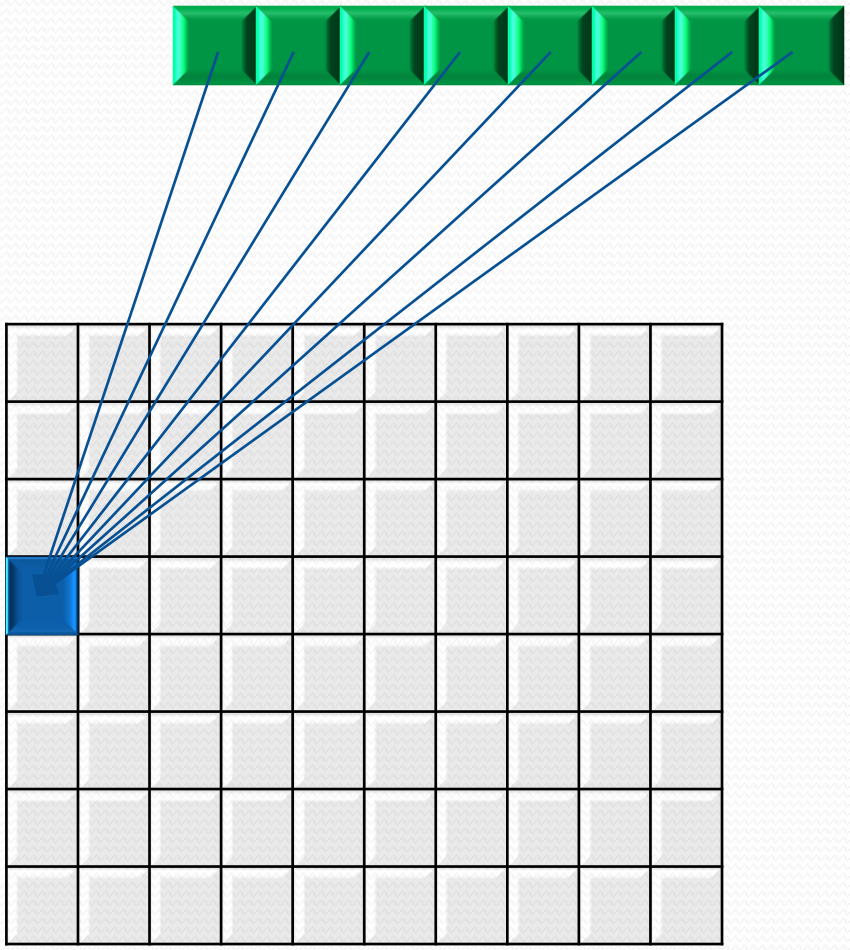
Нужно ли транспонировать матрицу A?

Нужно ли транспонировать матрицу B?
Нужно ли транспонировать матрицу A?

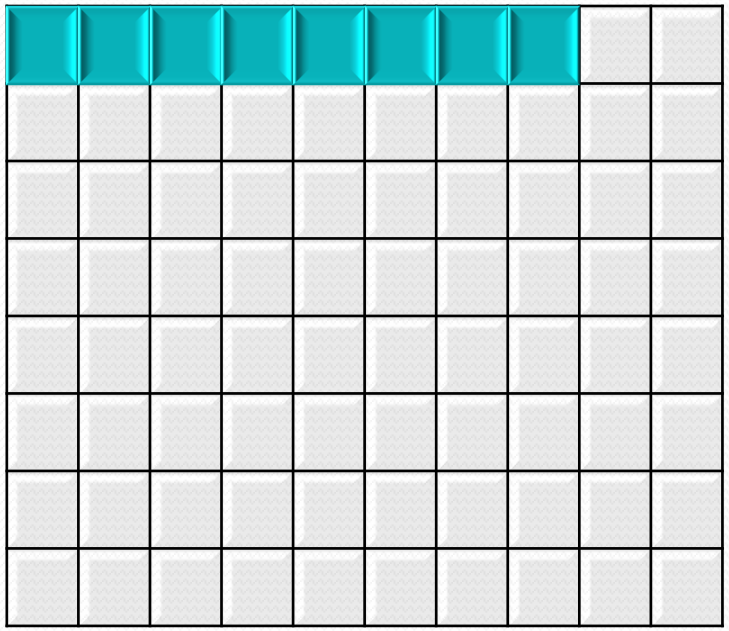
Пусть ширина каждого блока кратна размеру варпа

- т.е. все нити одного варпа лежат в одной строке блока и вычисляют соседние элементы в строке результата

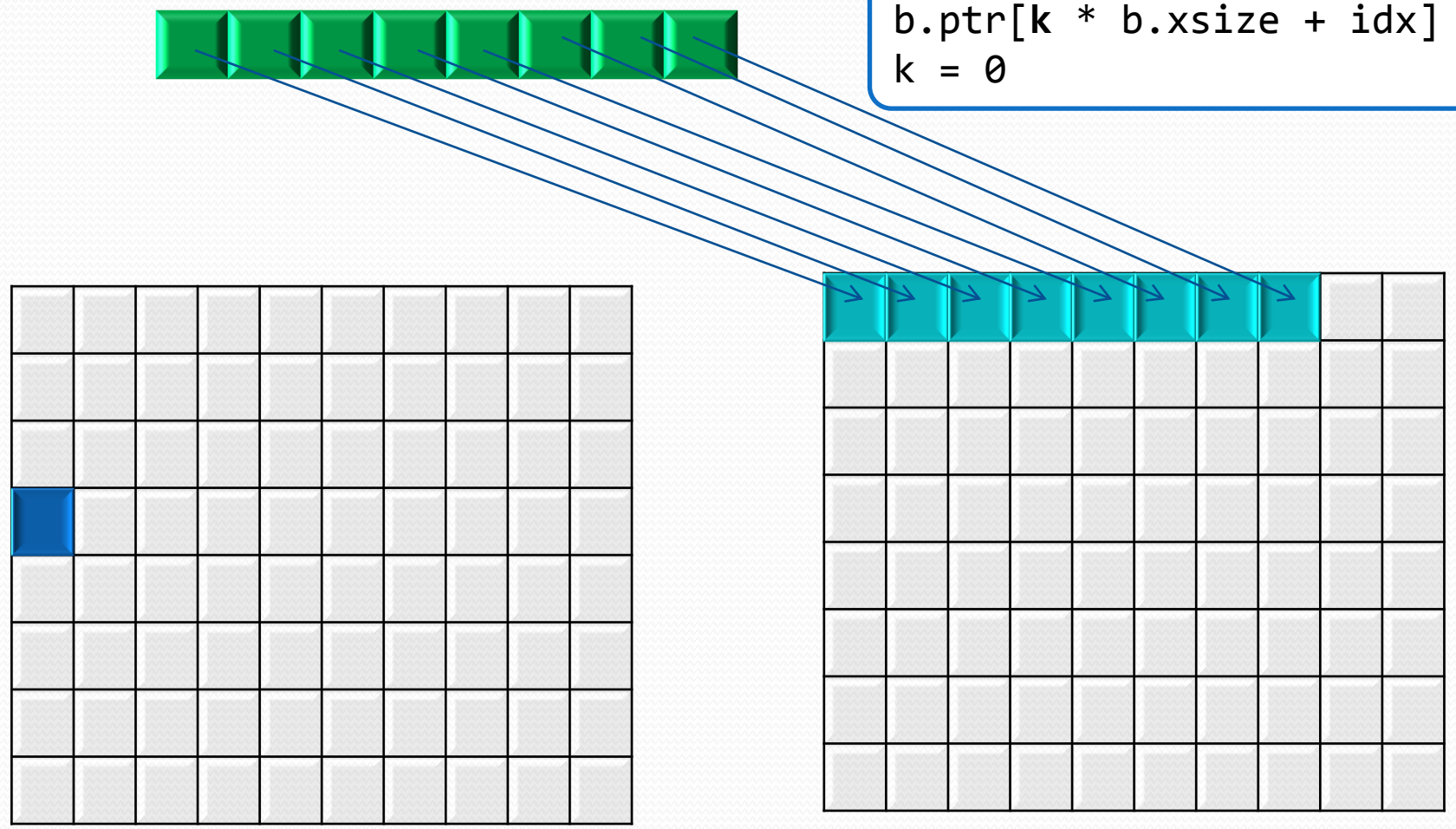
Матрицы 10*10, варп 8 нитей



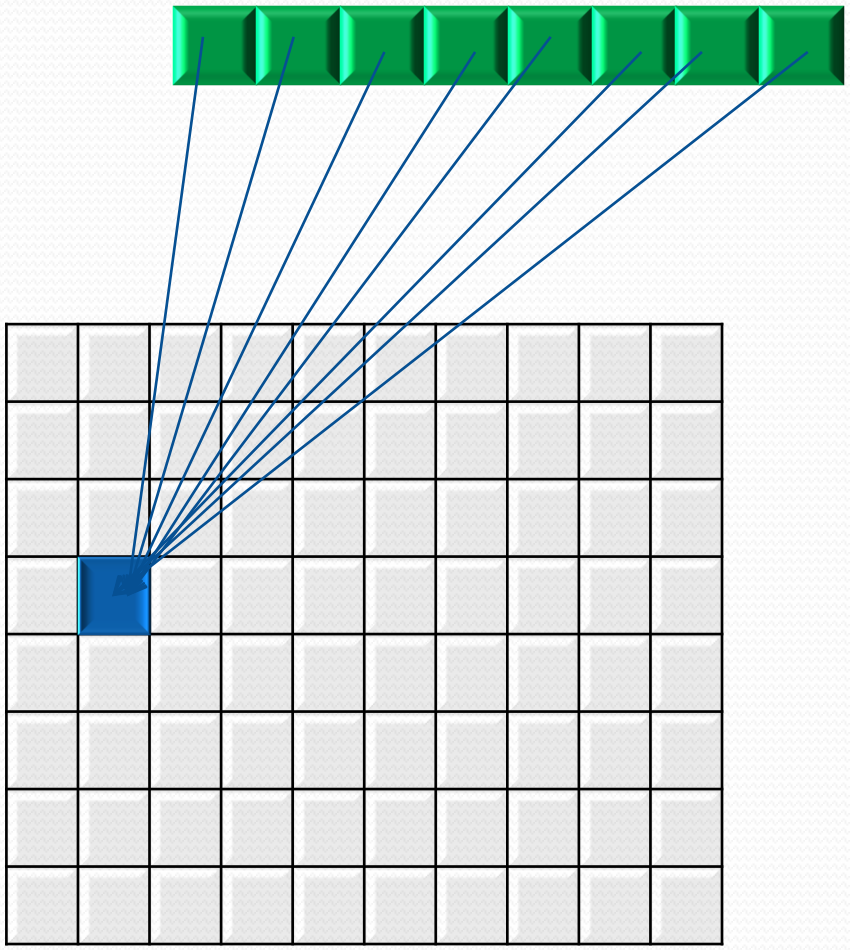
```
a.ptr[idy * a.xsize + k]
k = 0
```



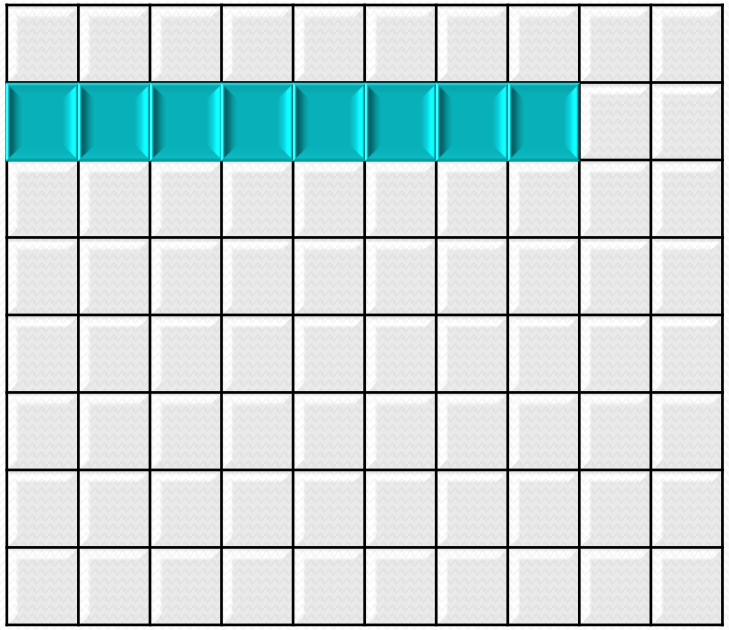
Матрицы 10*10, варп 8 нитей



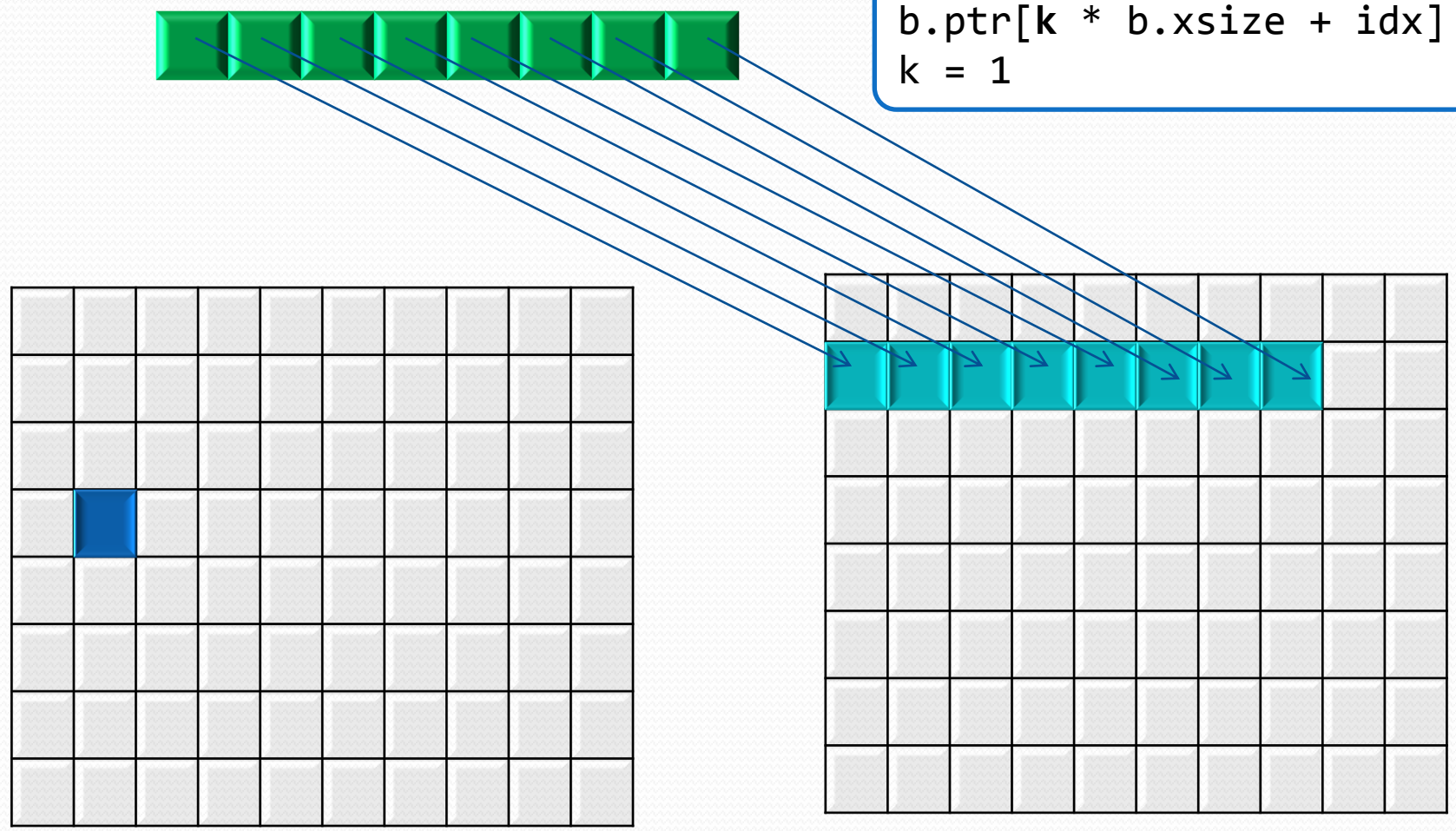
Матрицы 10*10, варп 8 нитей



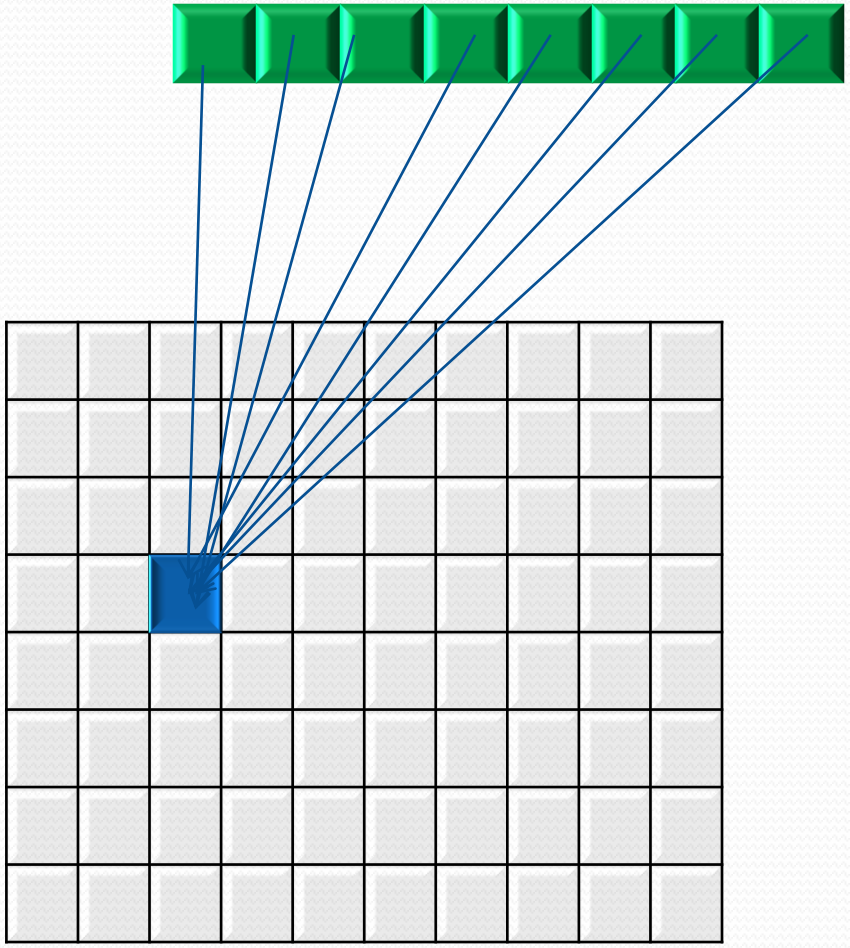
```
a.ptr[idy * a.xsize + k]
k = 1
```



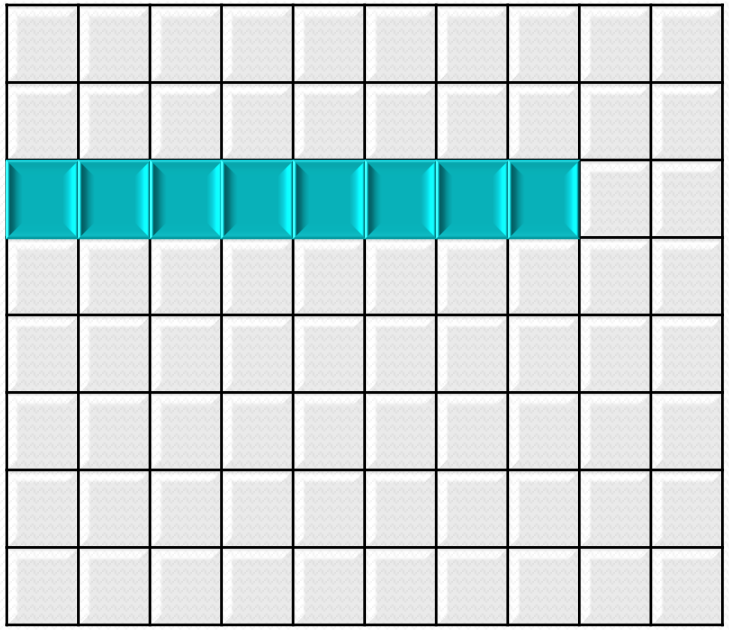
Матрицы 10*10, варп 8 нитей



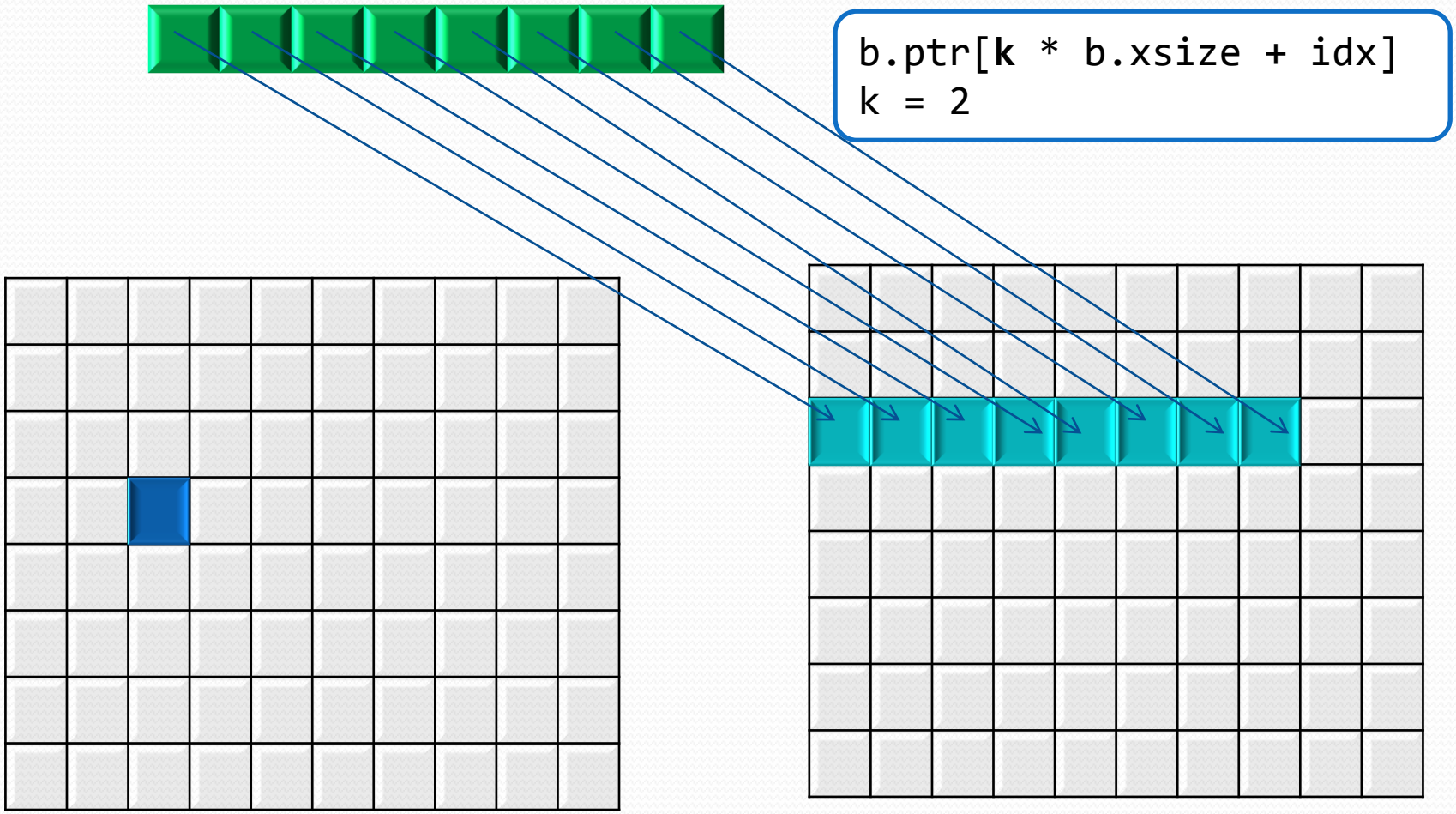
Матрицы 10*10, варп 8 нитей



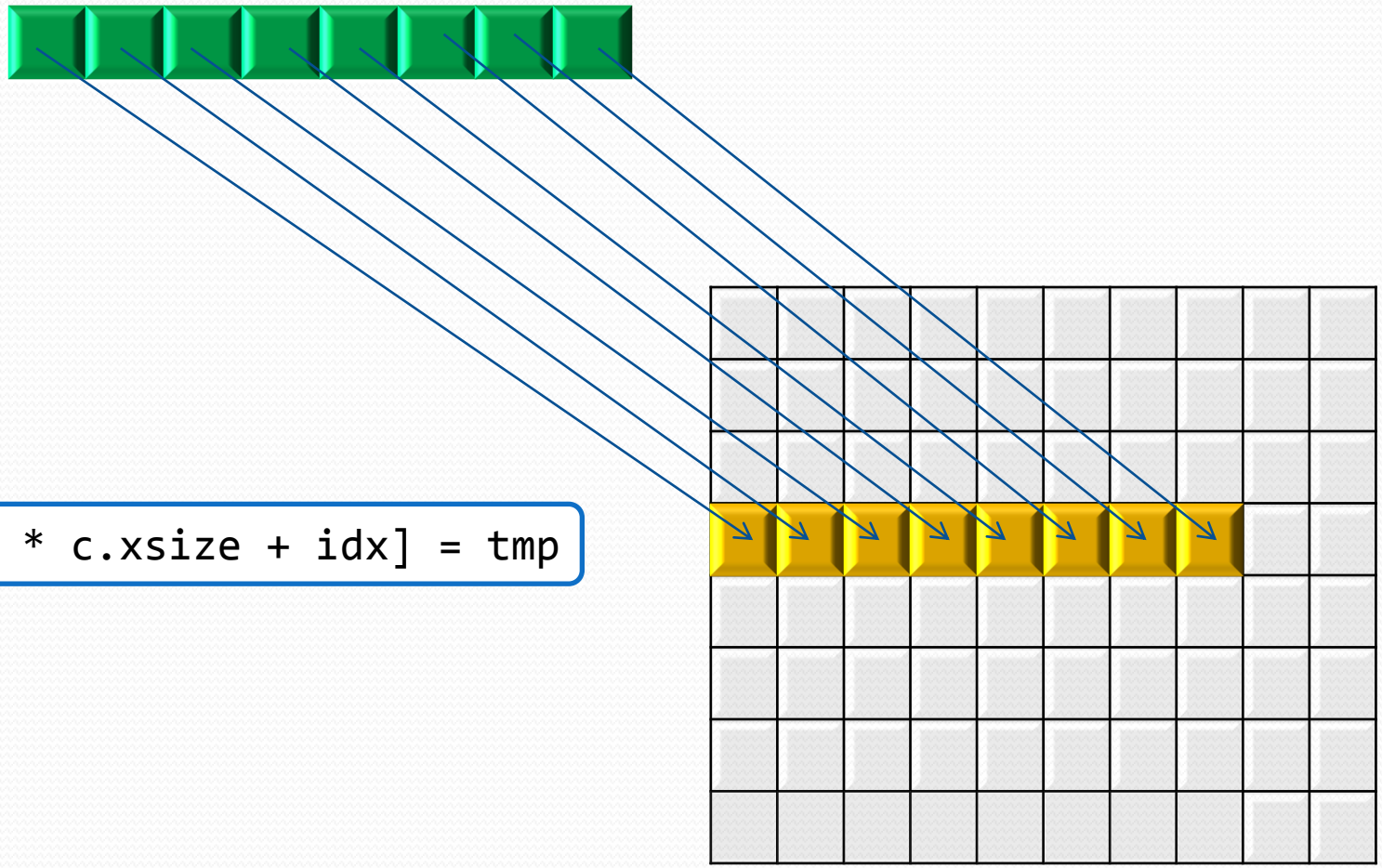
```
a.ptr[idy * a.xsize + k]
k = 2
```



Матрицы 10*10, варп 8 нитей



Матрицы 10*10, варп 8 нитей



```
c.ptr[idy * c.xsize + idx] = tmp
```

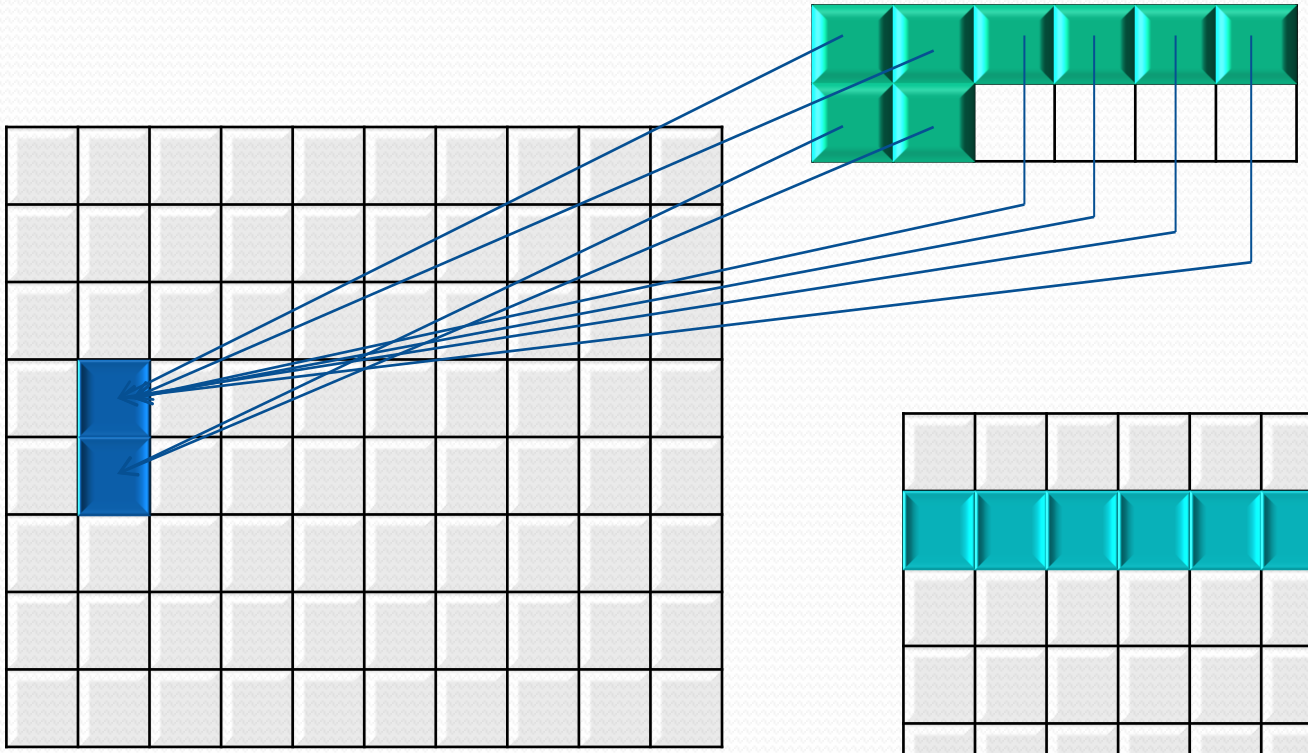
Нужно ли транспонировать матрицу B?
Нужно ли транспонировать матрицу A?

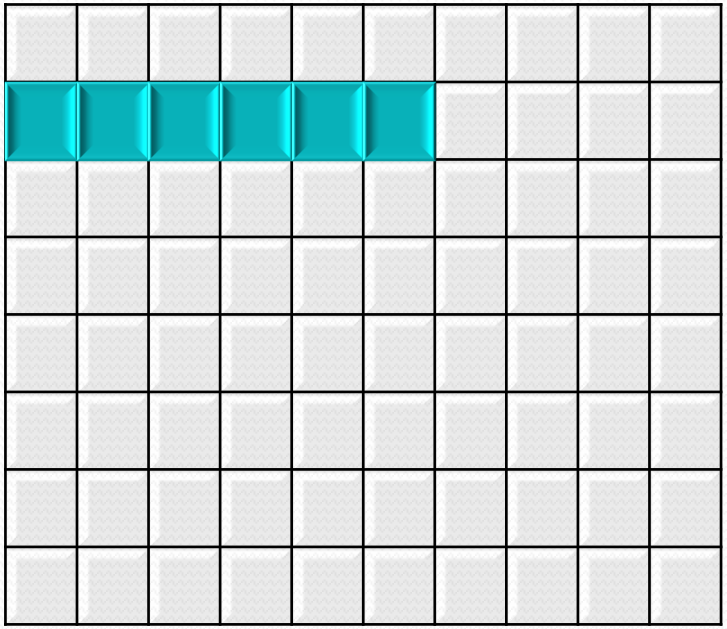
- Если ширина блока кратна размеру варпа, то все нити варпа обращаются к одному элементу матрицы A и соседним элементам в строке матрицы B
  - Транспонировать не нужно

Нужно ли транспонировать матрицу B?
Нужно ли транспонировать матрицу A?

- Что если ширина блока не кратна размеру варпа?
  - т.е. нити одного варпа лежат не в одной строке блока

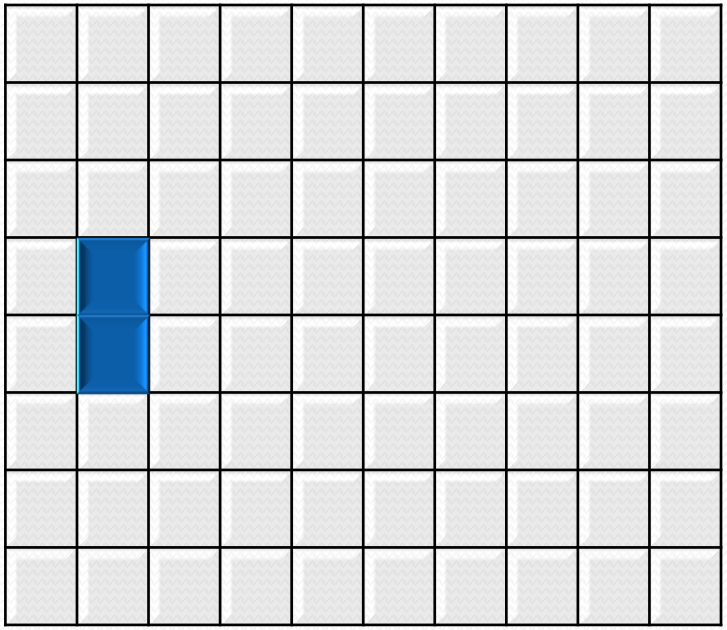
Матрицы 10*10, варп 8 нитей, блок 6x2

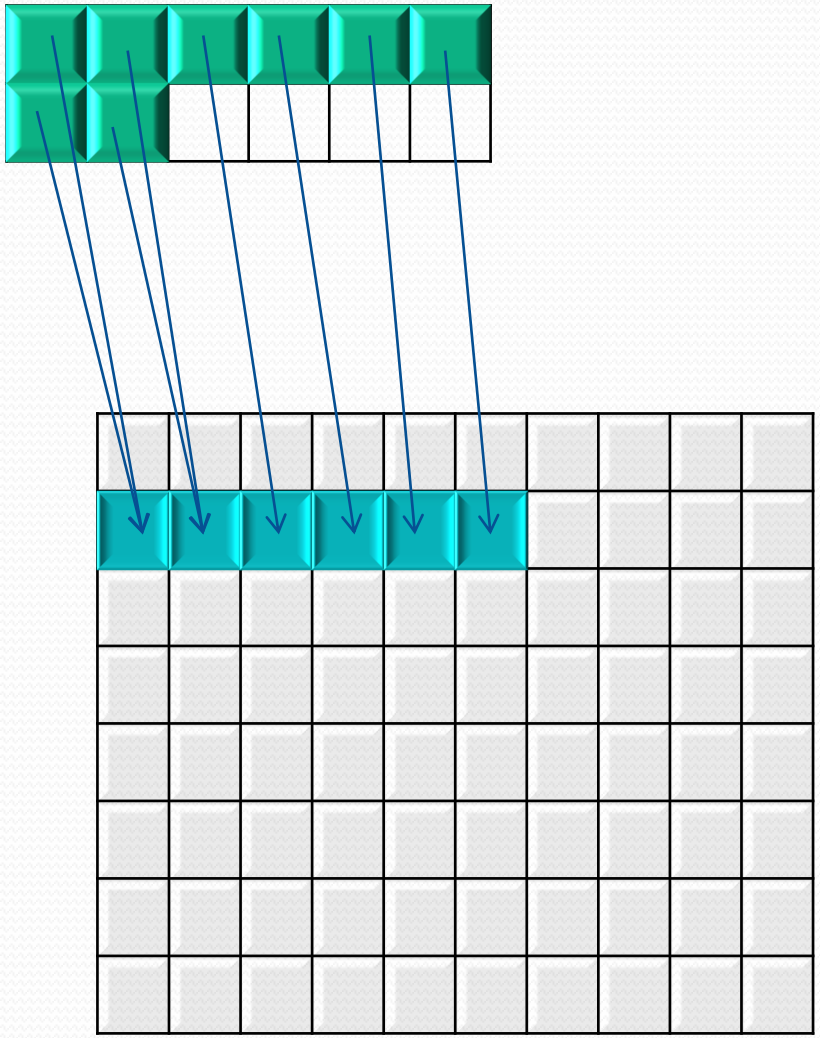




```
a.ptr[idy * a.xsize + k]
k = 0
```

Матрицы 10*10, варп 8 нитей, блок 6x2





```
b.ptr[k * b.xsize + idx]
k = 0
```

Нужно ли транспонировать матрицу B?
Нужно ли транспонировать матрицу A?

- Если ширина блока не кратна размеру варпа, то все нити варпа обращаются к соседним элементам столбца матрицы A и соседним элементам в строке матрицы B
  - Нужно транспонировать матрицу A

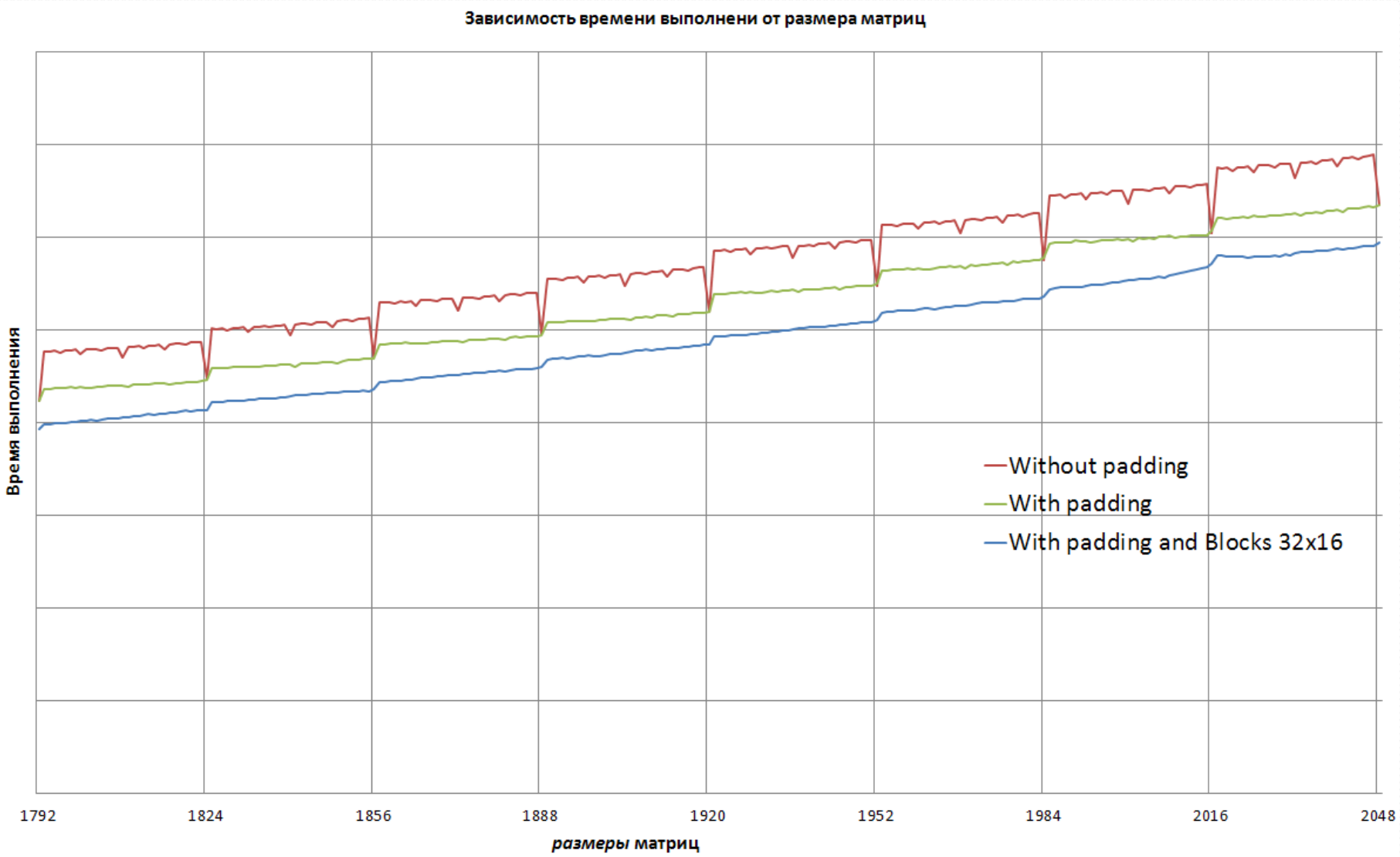
Требования к программе

- Программа должна запускать ядра на устройствах с `cudaDeviceProp::major == 2`
- Память под все матрицы выделять через `cudaMallocPitch`
- Выводить время работы ядер
  - Время работы считать через события
- Реализовать возможность задания высоты блока грида

Требования к программе

- Таким образом, параметры программы:
  - M, N, K - размеры матриц
  - Высота блоков 16 или 32?
  - Проверять результат?
- На выходе:
  - Время работы ядра без набивки
  - Время работы ядра с набивкой

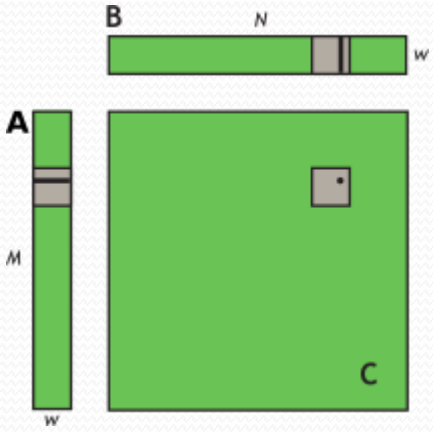
Результаты тестов



Часть 3: общая память

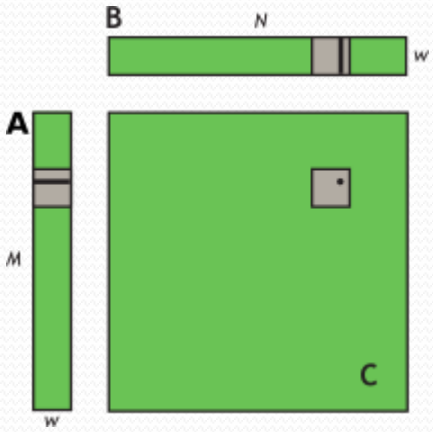
Идея

- Пусть нужно перемножить две «полоски» шириной 32 элемента
- Пусть размер блока 32x32
  - Делаем два массива в общей памяти с размерами 32x32 (наш управляемый кеш)
  - В первый копируем блок матрицы A, во второй – B (закешировали)
  - Вычисляем блок матрицы C используя данные из общей памяти



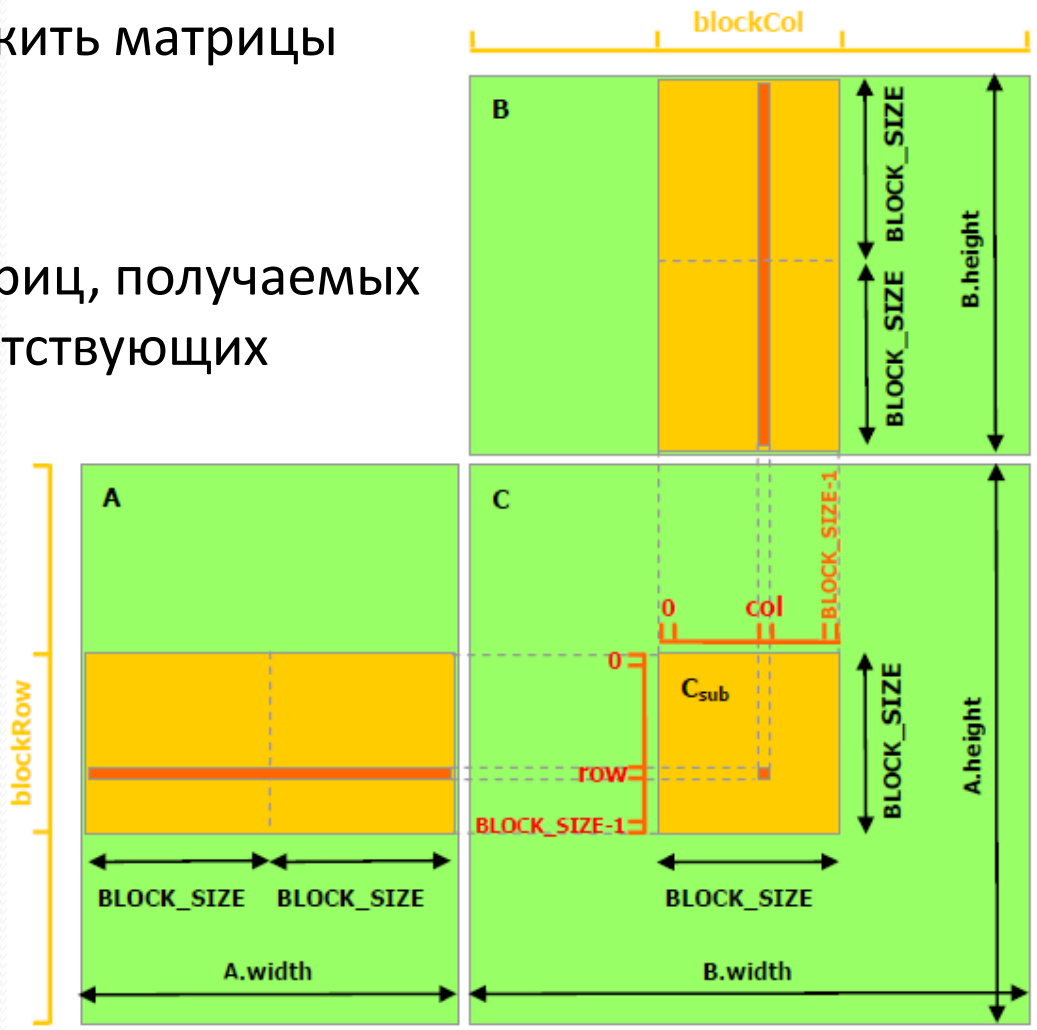
Идея

- Вычисляем блок матрицы C используя данные из общей памяти
  - В итоге за время работы блока к каждому элементу в глобальной памяти мы обратимся всего один раз
  - Без общей памяти – 32 раза



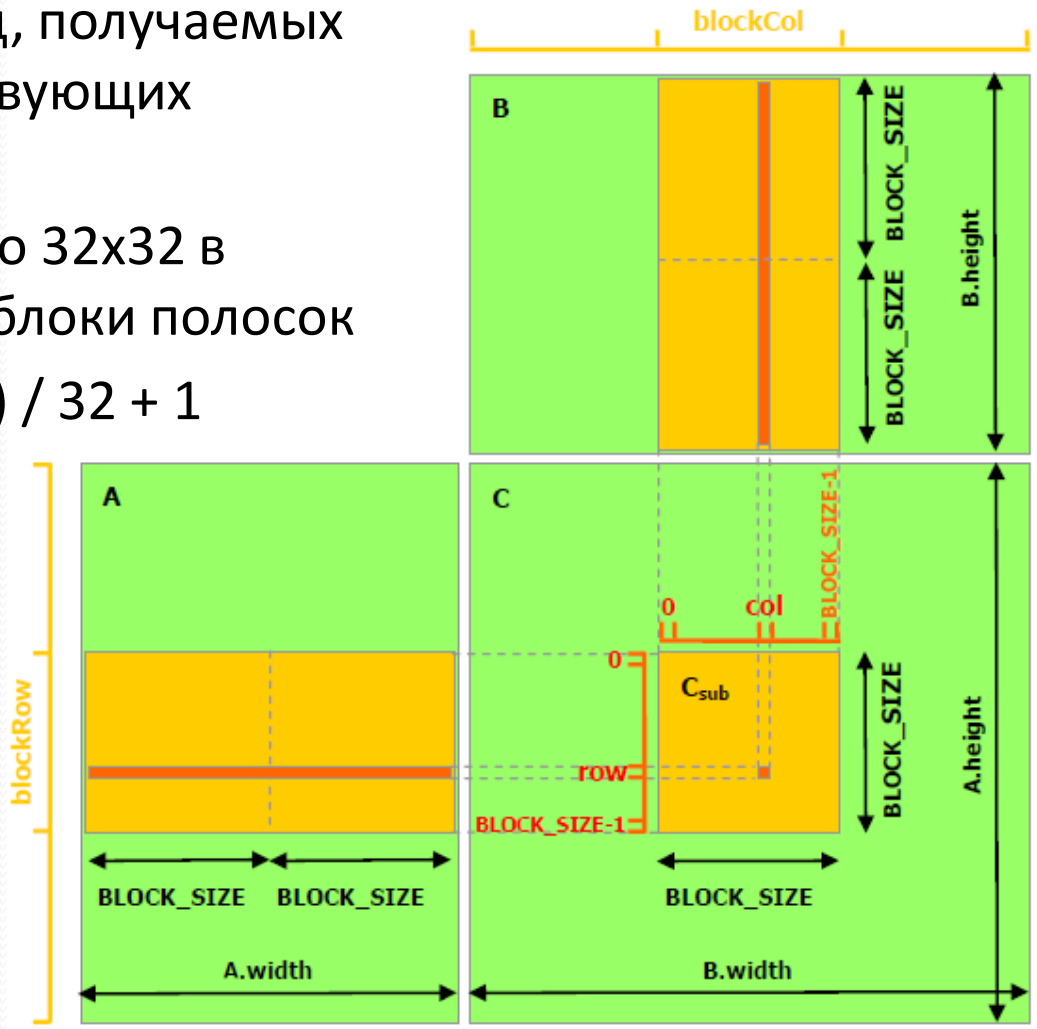
Идея

- Теперь пусть нужно перемножить матрицы произвольного размера
  - Разрежем их на «полоски»
  - Матрица C есть сумма матриц, получаемых при перемножении соответствующих полосок



Идея

- Матрица C есть сумма матриц, получаемых при перемножении соответствующих полосок
  - Так же делаем 2 массива по 32x32 в общей памяти, кешируем блоки полосок
  - Число этапов – (A.width - 1) / 32 + 1
  - Накапливаем результат



Матрицы произвольного размера

- Чтобы не возиться с выходом за границы матриц, следует явно дополнить матрицы нулями до размеров, кратных 32
  - `new_width = ((width - 1)/32 + 1)`
  - `new_height = ((height - 1)/32 + 1)`

Оптимизация

- В интернете легко можно найти примеры реализации перемножения матриц с общей памятью, поэтому проведем некоторые оптимизации:
  - Высота блока – 16 нитей
  - Каждая нить вычисляет два элемента: `a[i,j]` и `a[i+16, j]`
    - Получается ускорение 1.5x, по сравнению с обычным вариантом
- По аналогии можно сделать ядра, где каждая нить считает > 2 элементов, которые будут еще эффективнее

Требования к программе

- Общую память выделять динамически
- Дополнить матрицы нулями до размеров, кратных 32
- Для наглядности ускорения добавить в графики из прошлой части:
  - График скорости работы варианта с общей памятью при блоке 32x32, нить вычисляет один элемент
  - График скорости работы варианта с общей памятью при блоке 32x16, нить вычисляет два элемента

end