Оптимизация высокопроизводительной обработки данных с использованием GPU

Колганов А.С.
ВМК, МГУ им. М.В. Ломоносова / ИМП им. М.В. Келдыша РАН

GPU vs CPU

- GPU
  - быстрая HBM память
  - быстрые вычислительные ядра
  - новые тензорные ядра в поколении Volta

- CPU
  - многоядерность, многопоточность
  - поддержка расширенных векторных инструкций

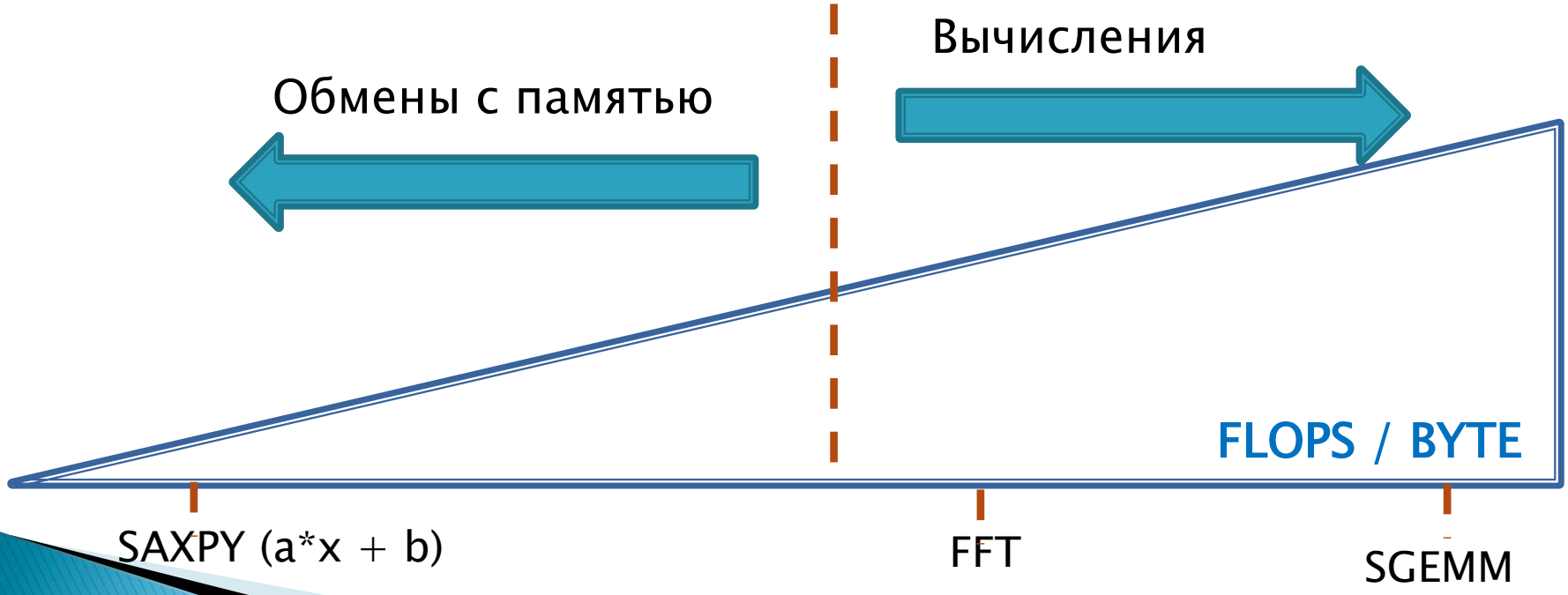
GPU vs CPU

- GPU
  - быстрая HBM память
  - быстрые вычислительные ядра
  - новые тензорные ядра в поколении Volta
  - **CUDA ToolKit + инфраструктура отладки!**

- CPU
  - многоядерность, многопоточность
  - поддержка расширенных векторных инструкций
  - **оптимизации трудоемкие и низкоуровневые!**

Причины задержек на GPU

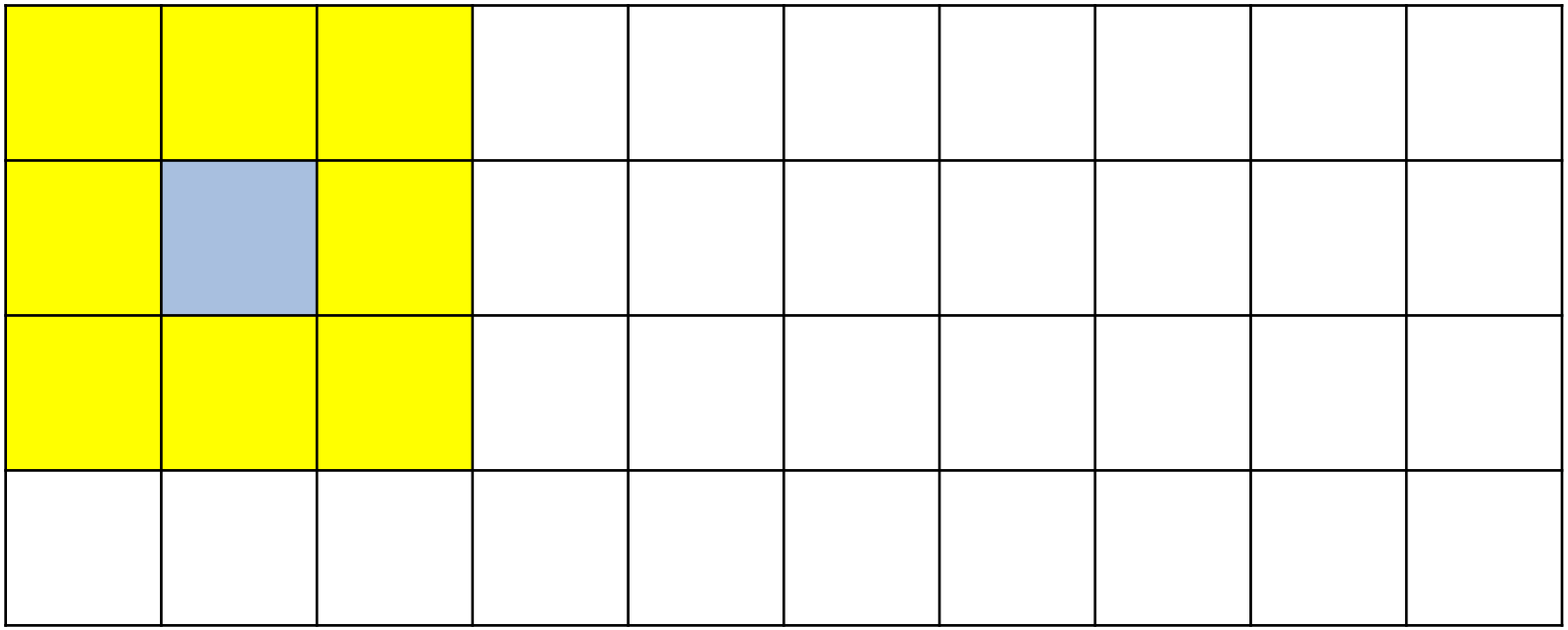
- Многоуровневая память и задержки
  - L1 ~ 2 ТБ/с, L2 ~ 0.8 ТБ/с, GDDR5 ~ 0.4 ТБ/с
- Большое количество «маленьких» ядер
- Вычислительная интенсивность:



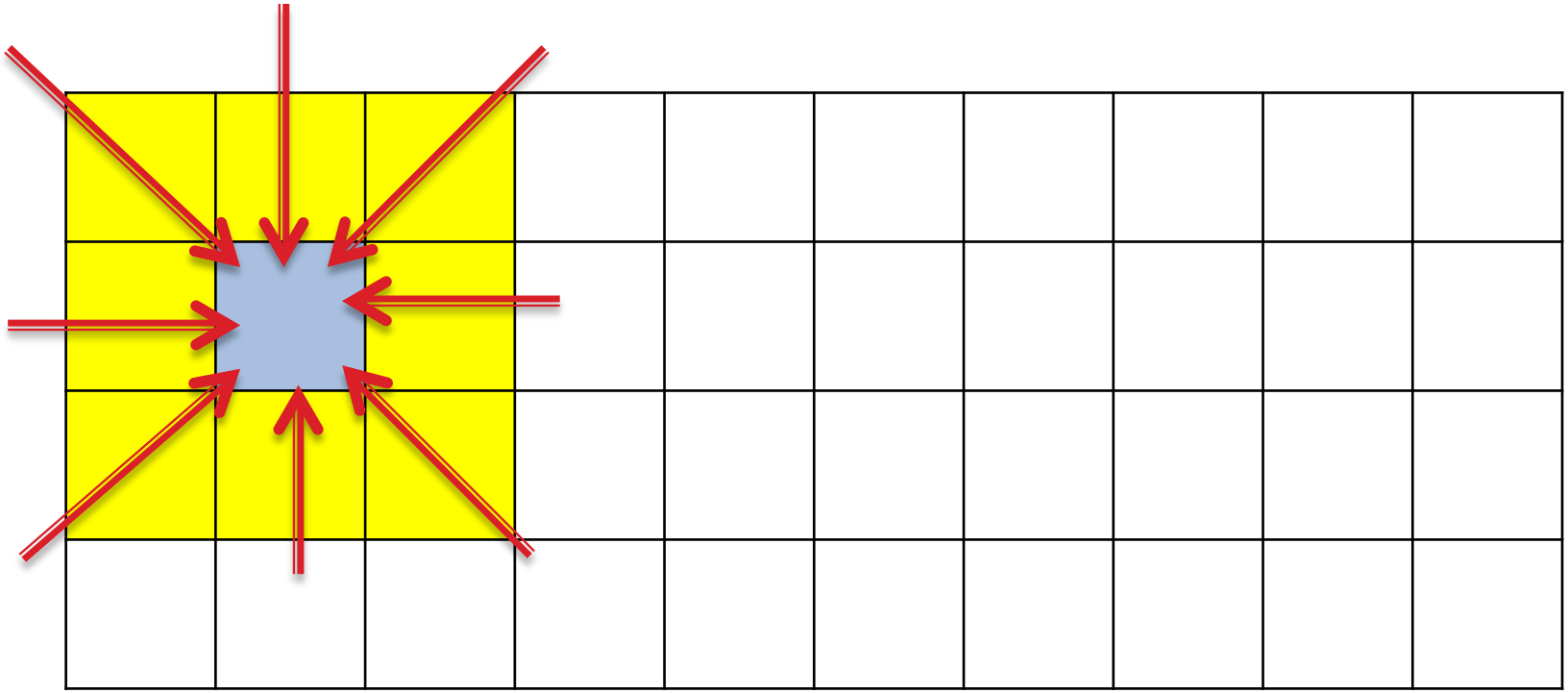
Алгоритм: медианный фильтр

- Для фильтра 3x3 необходимо загрузить из памяти 9 точек и выполнить 50 операций;
- Простой алгоритм в реализации, но сложный в оптимизации;
- Используется как одно из звеньев в конвейерной обработке изображений.

Алгоритм: медианный фильтр



Алгоритм: медианный фильтр



Средняя точка по яркости

Алгоритм: медианный фильтр

```
inline void Load(const uchar *src, int a[3])
{
    a[0] = src[-1];
    a[1] = src[ 0];
    a[2] = src[ 1];
}
inline void Load(const uchar *src, int stride, int a[9])
{
    Load(src - stride, a + 0);
    Load(src          , a + 3);
    Load(src + stride, a + 6);
}
```

Загрузка 9ти элеметов

Простая сортировка

```
void MedianFilter(const uchar * src, int srcStride,
                  int width, int height, uchar *dst, int dstStride)
{
    int a[9];
    for(size_t y = 0; y < height; ++y) {
        for(size_t x = 0; x < width; ++x) {
            Load(src + x, srcStride, a);
            std::sort(a, a + 9)
            dst[x] = (uint8_t)a[4];
        }
        src += srcStride; dst += dstStride;
    }
}
```

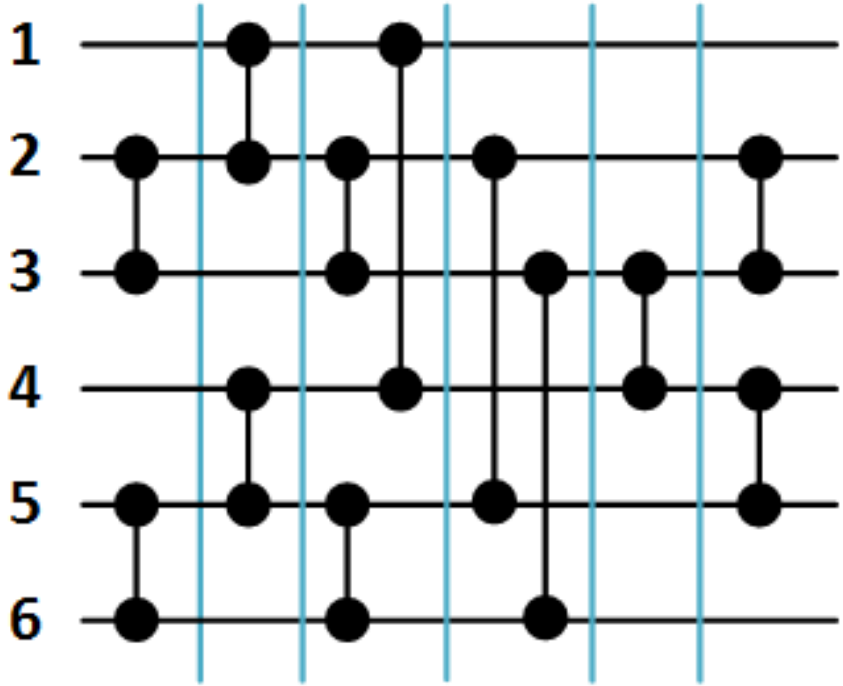
Оптимизация вычислений

- Стандартная функция сортировки
- Обработка нескольких точек «одновременно»

Простая сортировка

```
void MedianFilter(const uchar * src, int srcStride,
                  int width, int height, uchar *dst, int dstStride)
{
    int a[9];
    for(size_t y = 0; y < height; ++y) {
        for(size_t x = 0; x < width; ++x) {
            Load(src + x, srcStride, a);
            std::sort(a, a + 9)
            dst[x] = (uint8_t)a[4];
        }
        src += srcStride; dst += dstStride;
    }
}
```

Битоническая сортировка

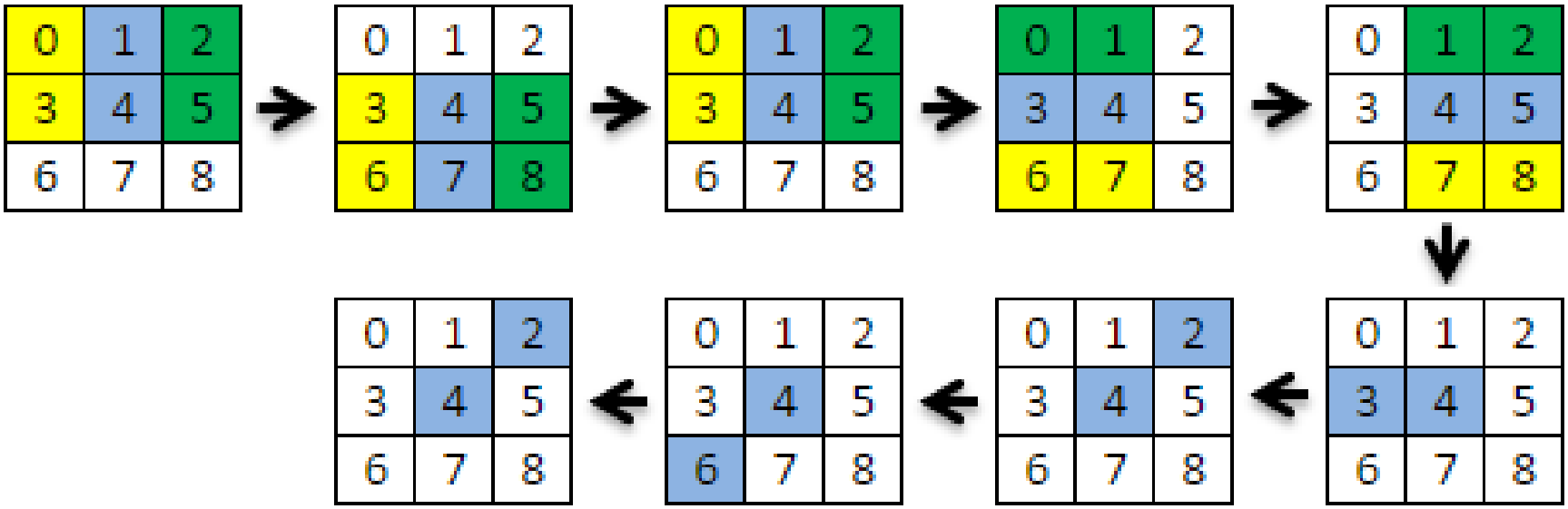


- «прямолинейна» с фиксированным количеством операций

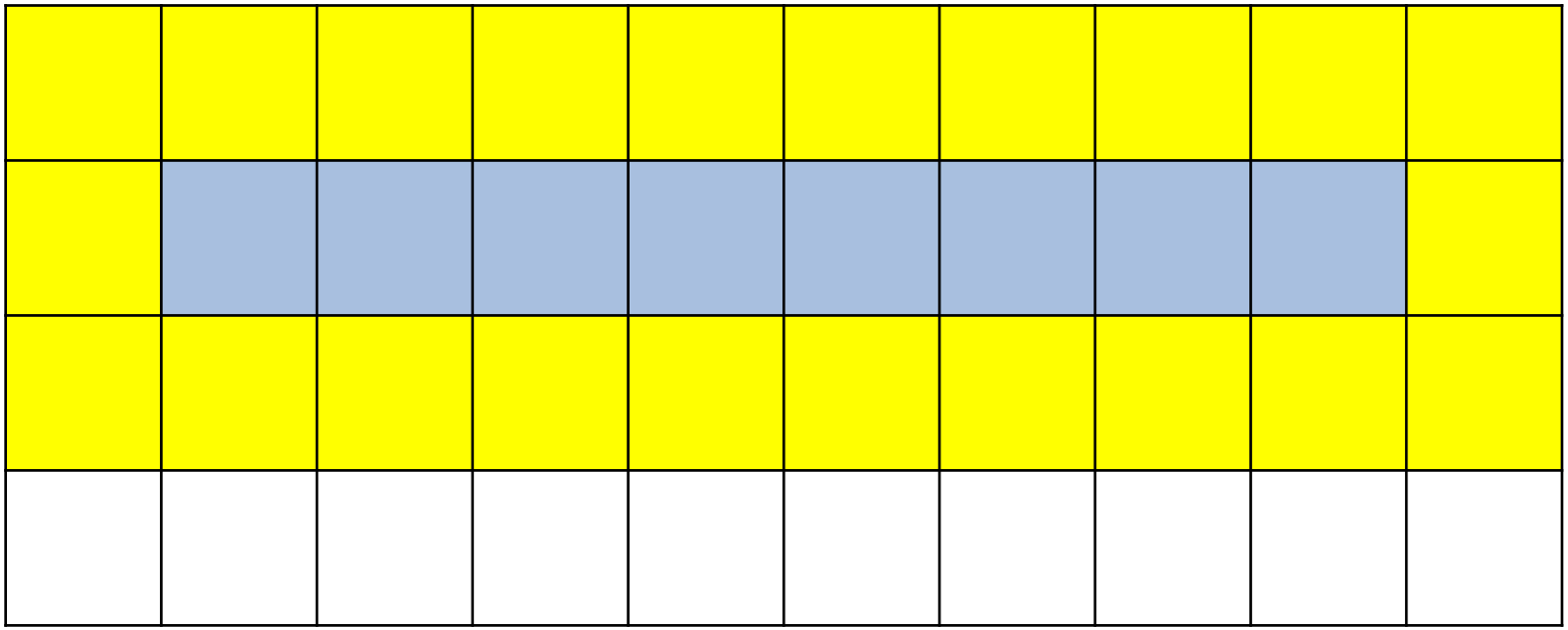
- набор независимых команд, выполняющих обмены.

Битоническая сортировка

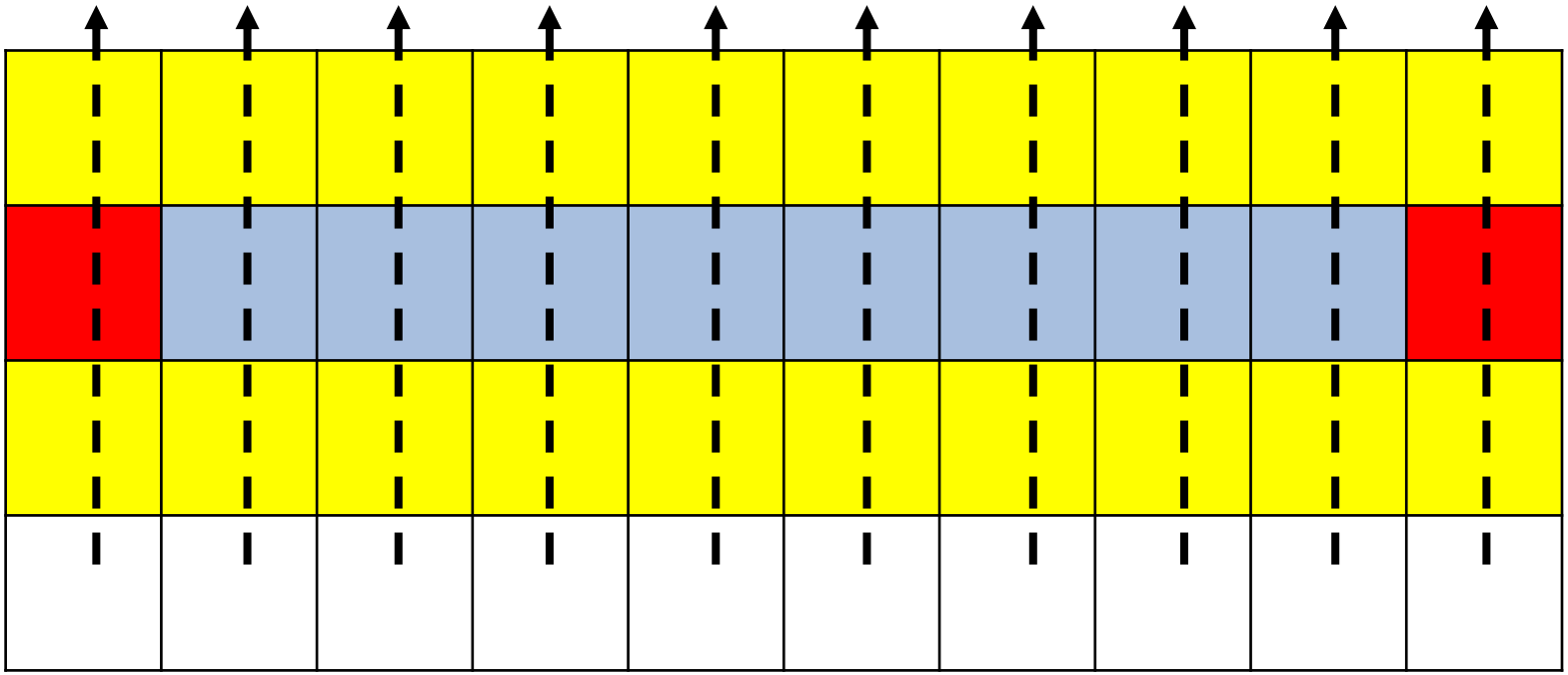
- «прямолинейна» с фиксированным количеством операций



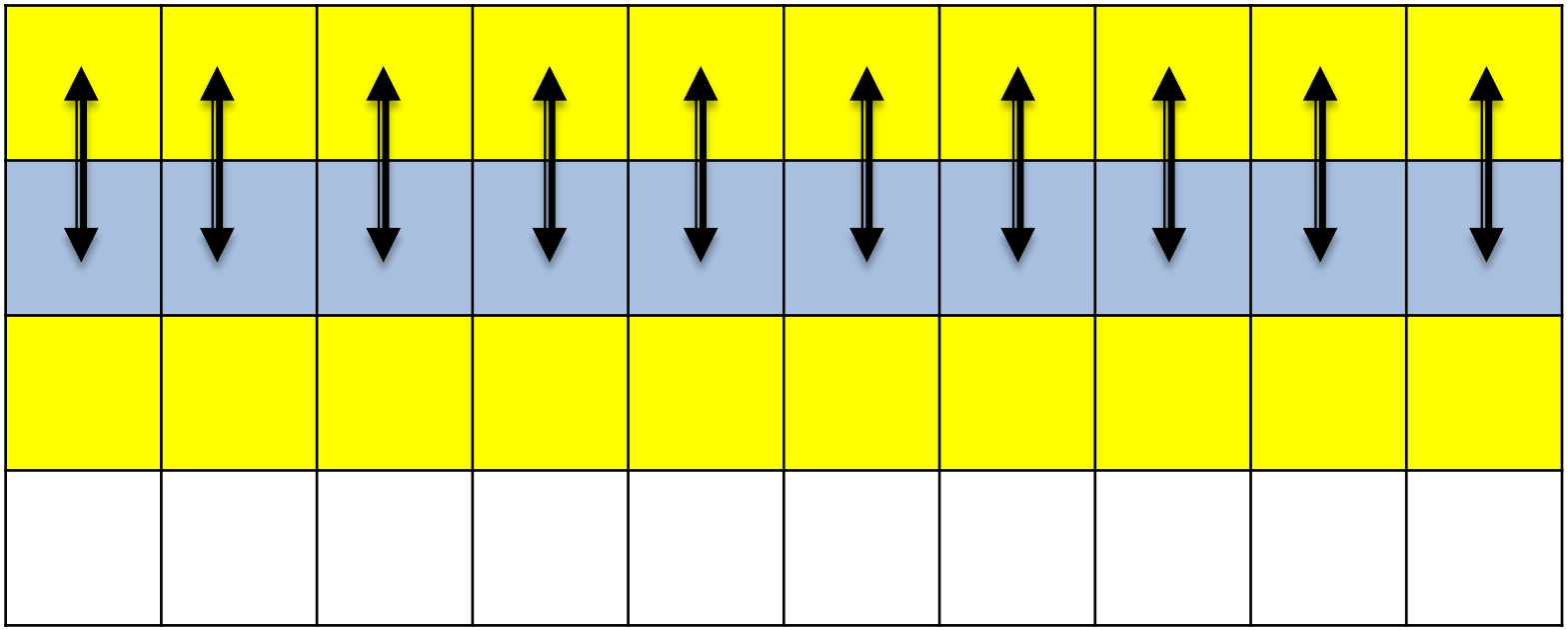
Оптимизация работы с памятью



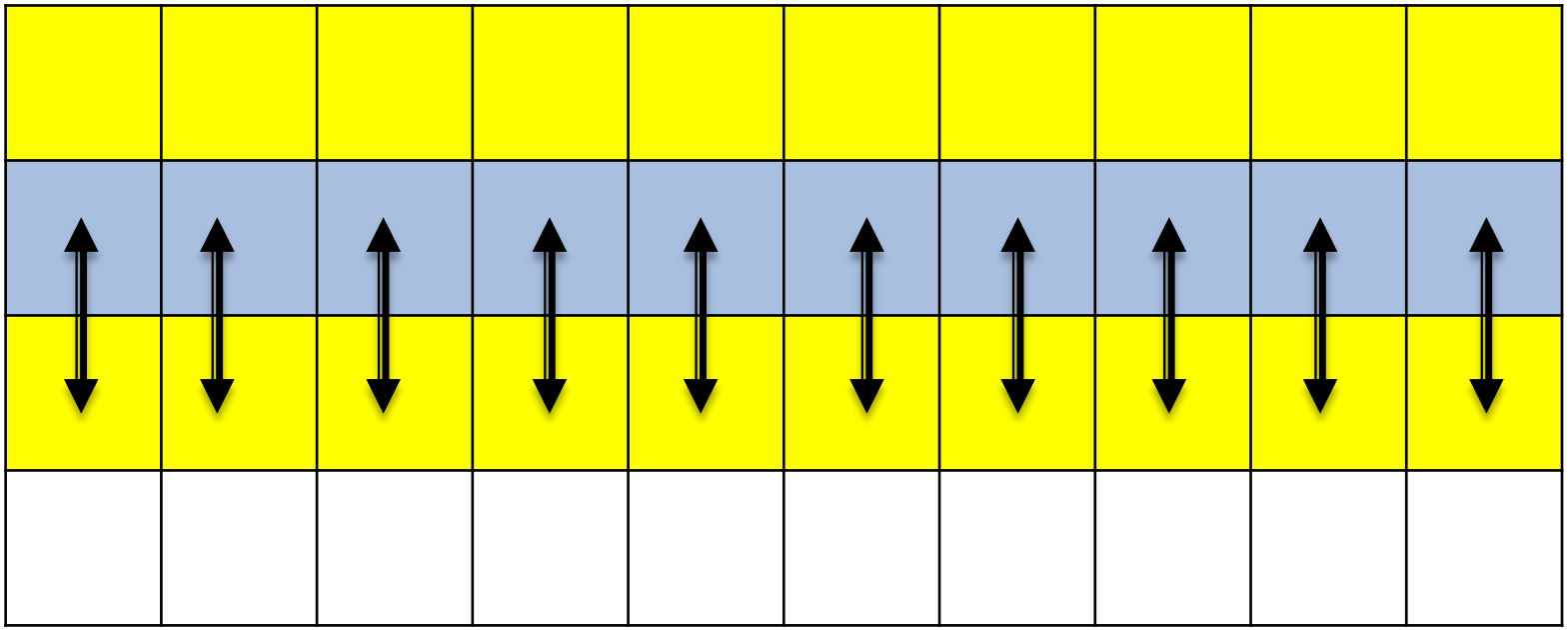
Оптимизация работы с памятью



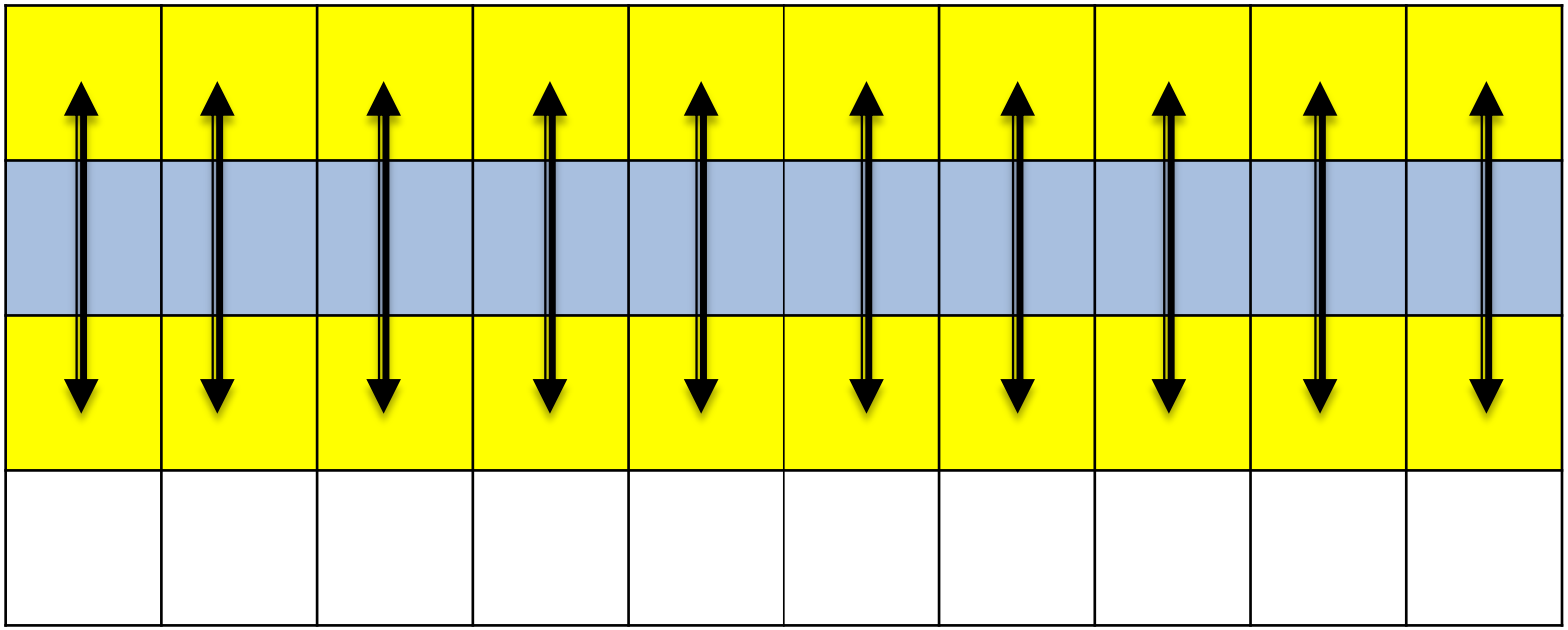
Оптимизация работы с памятью



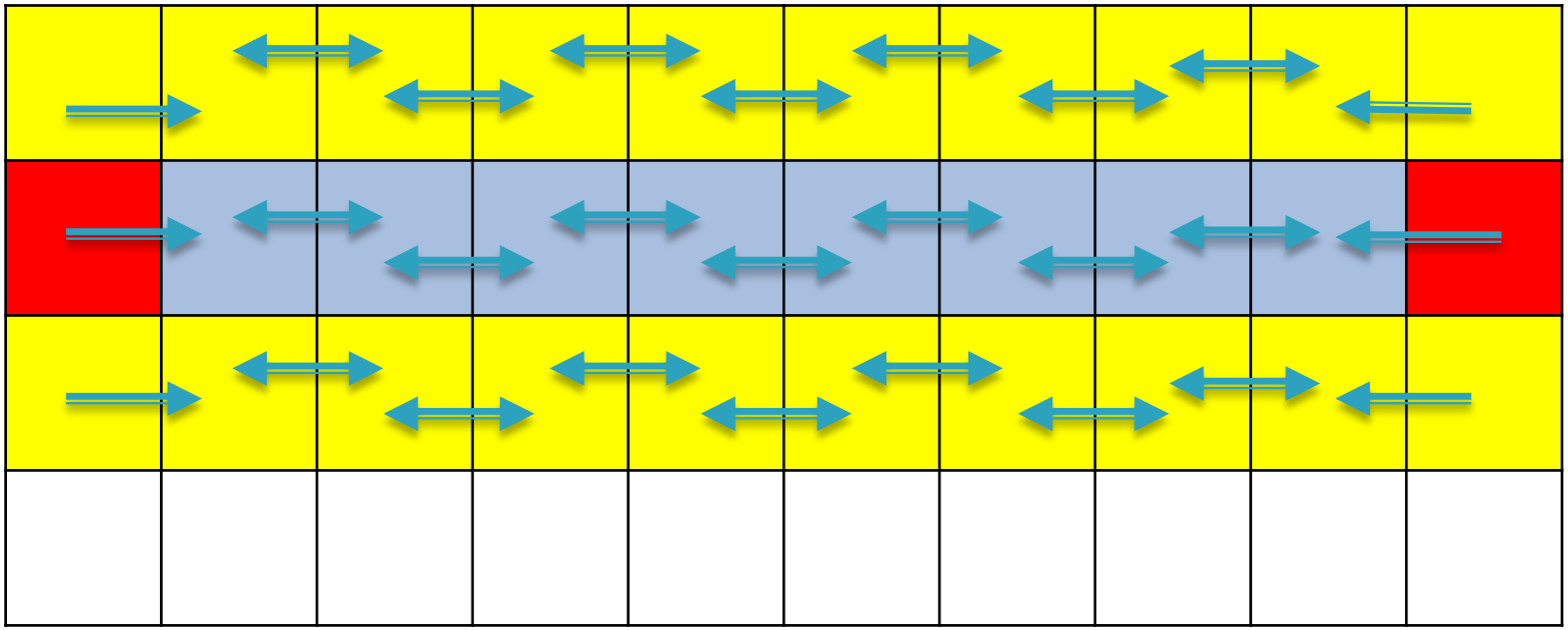
Оптимизация работы с памятью



Оптимизация работы с памятью



Оптимизация работы с памятью

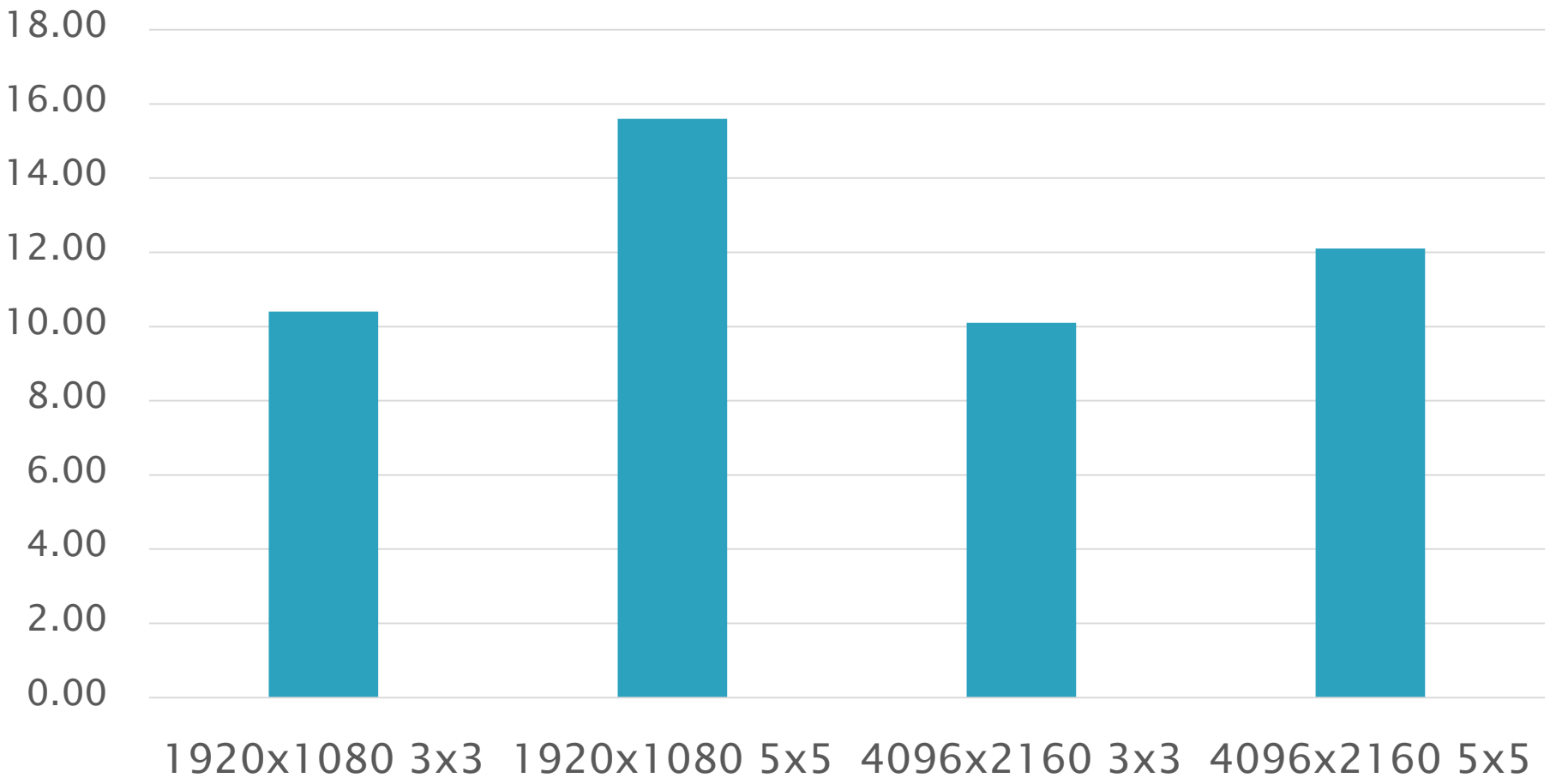


Производительность

- CPU Intel Core i7 Skylake 4ГГц
  - Большая частота отдельного ядра
  - Поддержка векторных расширений AVX2
  - Использовалась низкоуровневая оптимизация

- GPU Titan X Pascal
  - Большое количество маленьких ядер
  - Вдвое низкая частота (~2 ГГц)
  - Использовались «стандартные» средства CUDA

Ускорение CUDA / AVX2



Спасибо за внимание!

Александр Колганов, МГУ ВМК / ИПМ РАН,
alexander.k.s@mail.ru