# Параллельные вычисления на основе технологии OpenCL

Лекция 2

Силаков Роман Дмитриевич

# Содержание

- Префиксная сумма (scan)
  - Определения
  - Применения
  - Наивный алгоритм
  - Hillis and Steele scan
  - Blelloch scan
  - Общие замечания
  - Compact
- Сортировка на GPU
  - Параллельная сортировка слиянием
  - Битоническая сортировка, сортирующие сети
  - Radix sort

# Scan. Определения

- Пусть $\oplus$ – бинарный ассоциативный оператор с единицей (I)
- Массив $[a_0, a_1, a_2, \ldots, a_{N-1}]$

- Операция *all-prefix-sum* или ***exclusive scan***:

  $[I, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{N-2})]$
- Операция ***inclusive scan***:

  $[a_0, (a_0 \oplus a_1), (a_0 \oplus a_1 \oplus a_2), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{N-1})]$

# Scan. Применение

- Параллельные версии Radix sort, Quicksort
- Распределение ресурсов/заданий
- Сжатие потоков (stream compaction)
- Графика: Summed-Area Tables (SAT)
- Решение рекуррентных соотношений
- Гистограммы
- … и многое другое

# Naïve parallel scan

- Сначала последовательный код

```
out[0] = in[0]
for (int i = 1; i < N; ++i)
   out[i] = in[i] + out[i - 1];
```

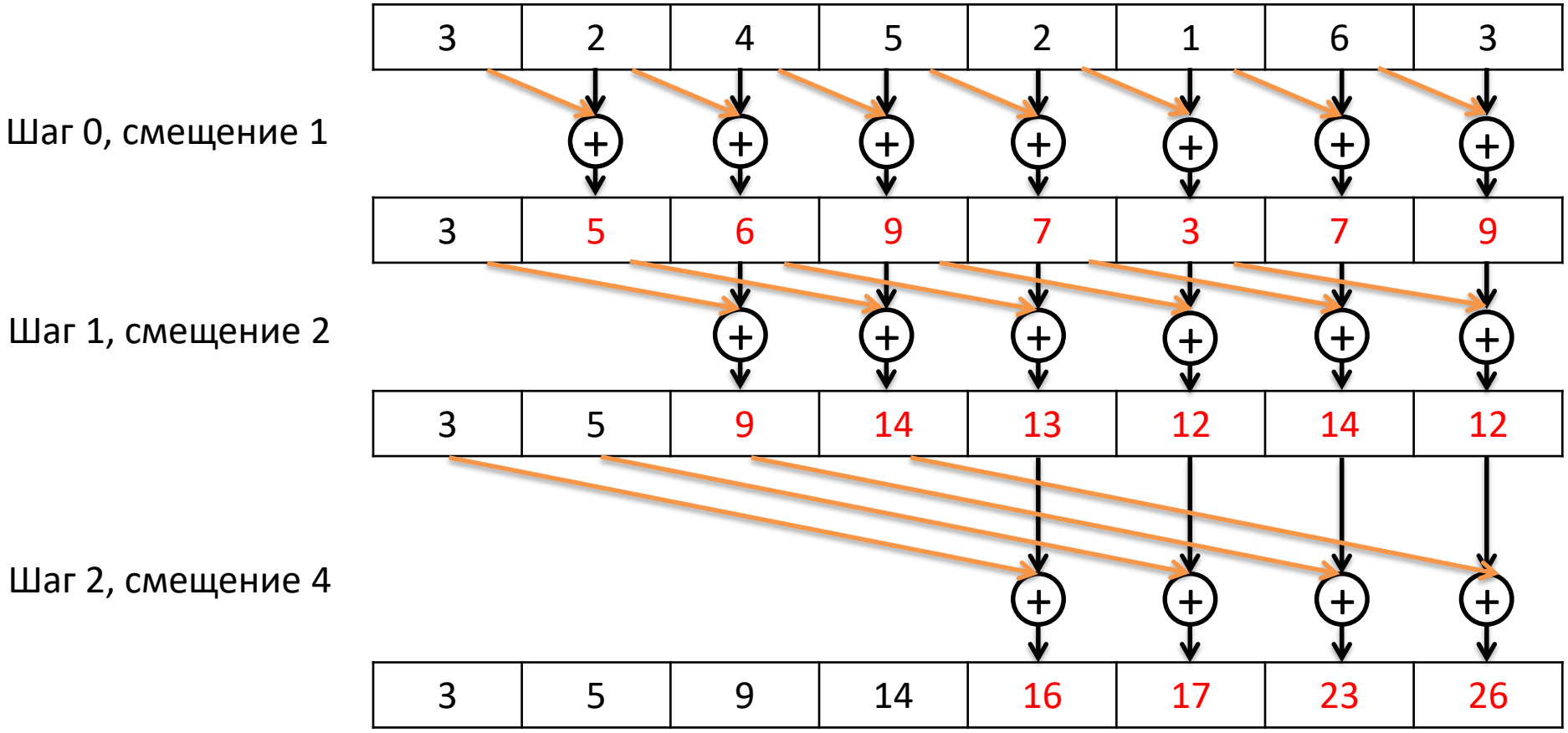
Сложность – O(N)

- «Наивный» параллельный алгоритм

```
Thread0: out[0] = in[0]
Thread1: out[1] = in[0] + in[1]
Thread2: out[2] = in[0] + in[1] + in[2]
                ...
```

Work complexity – $O(N^2)$, асимптотически хуже сложности последовательного алгоритма, следовательно алгоритм **work-inefficient**

# Hillis-Steele scan

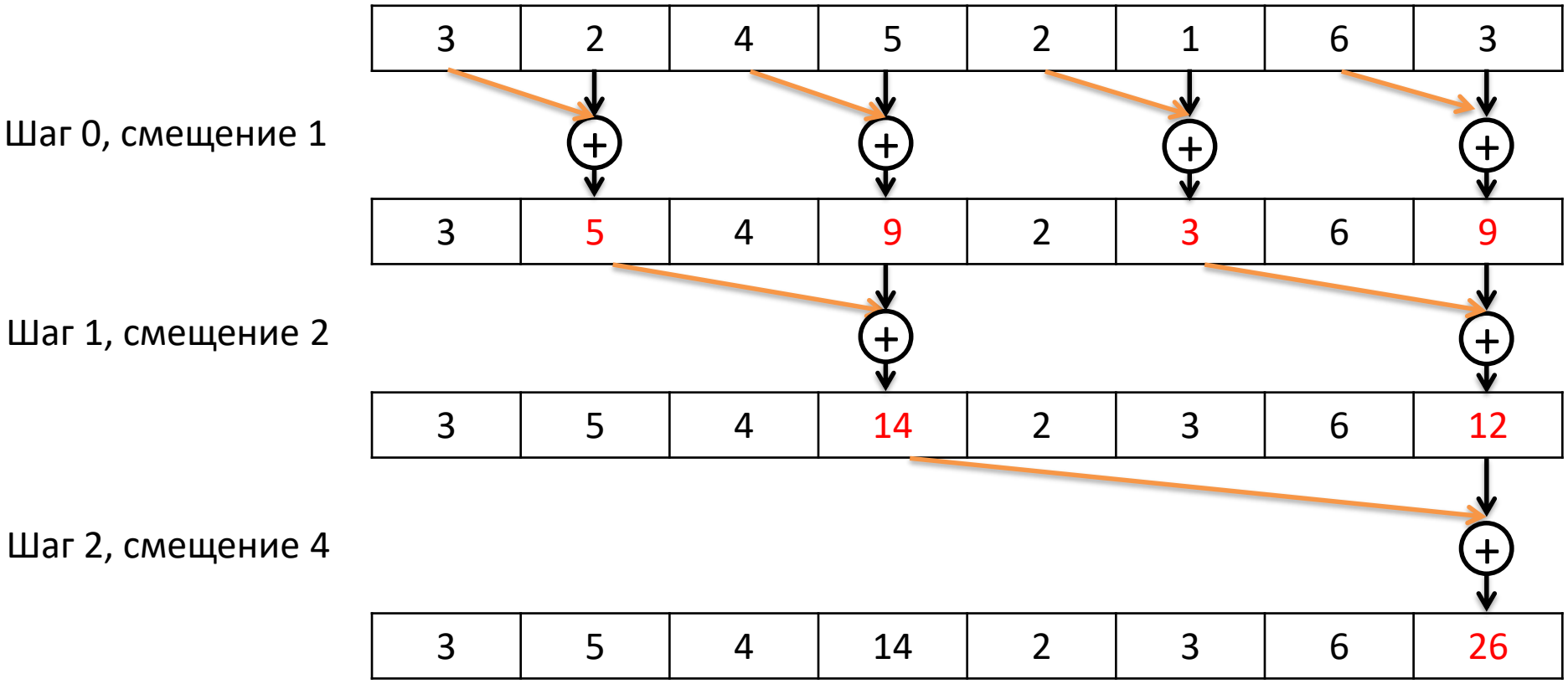



**Алгоритм**:

```
for (i = 0..n)
  for (shift = 0..log(n)-1)
    if (i >= 2^shift):
      x[i] = x[i] + x[i – 2^shift]
```

Work complexity: O(N log(N))
Step complexity: O(log(N))

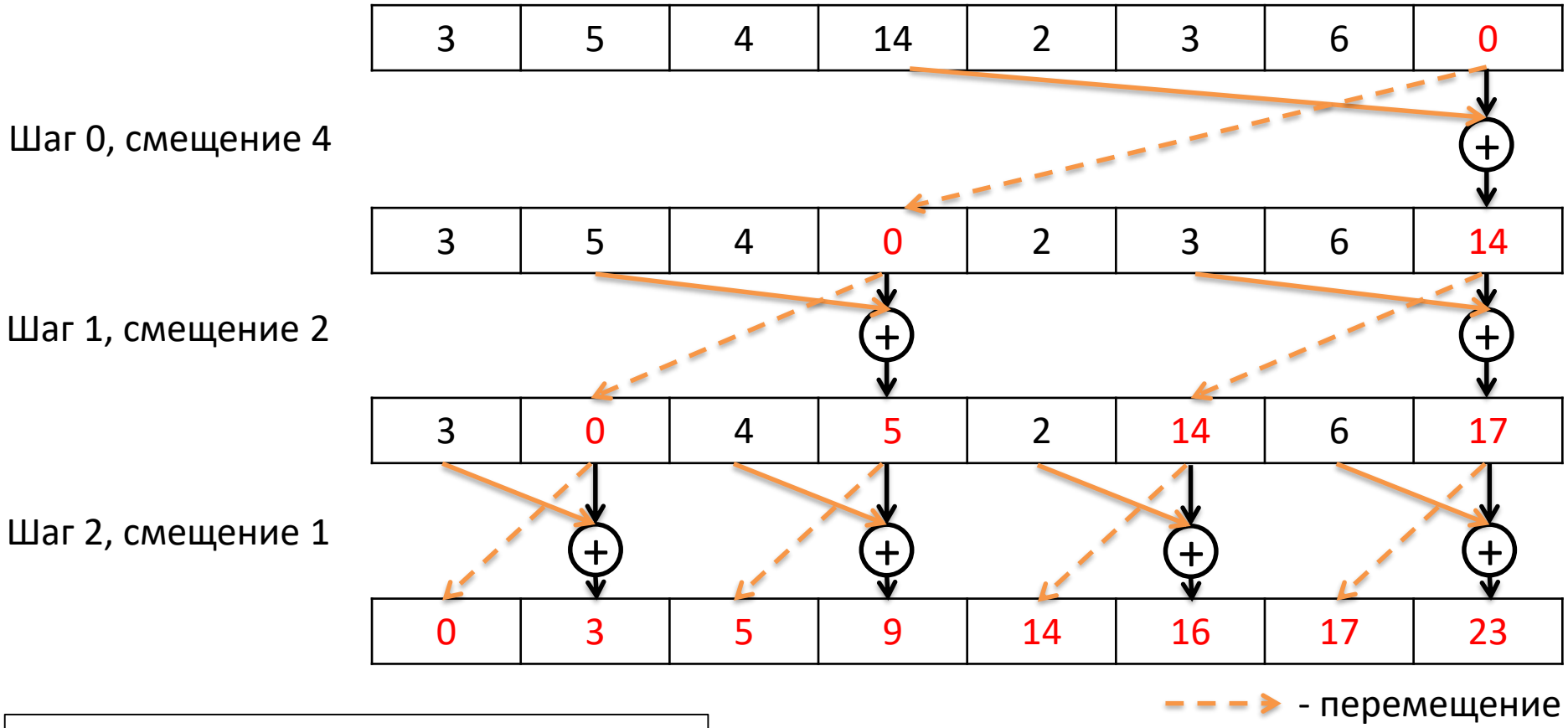
# Exclusive Blelloch scan. Часть 1: Up-sweep




**Алгоритм**:

```
for (i = 0..n)
  for (shift = 0..log(n)-1)
    if ((i+1) % 2^(shift+1) == 0):
      x[i] = x[i] + x[i – 2^shift]
```

# Exclusive Blelloch scan. Down-sweep




**Алгоритм**:

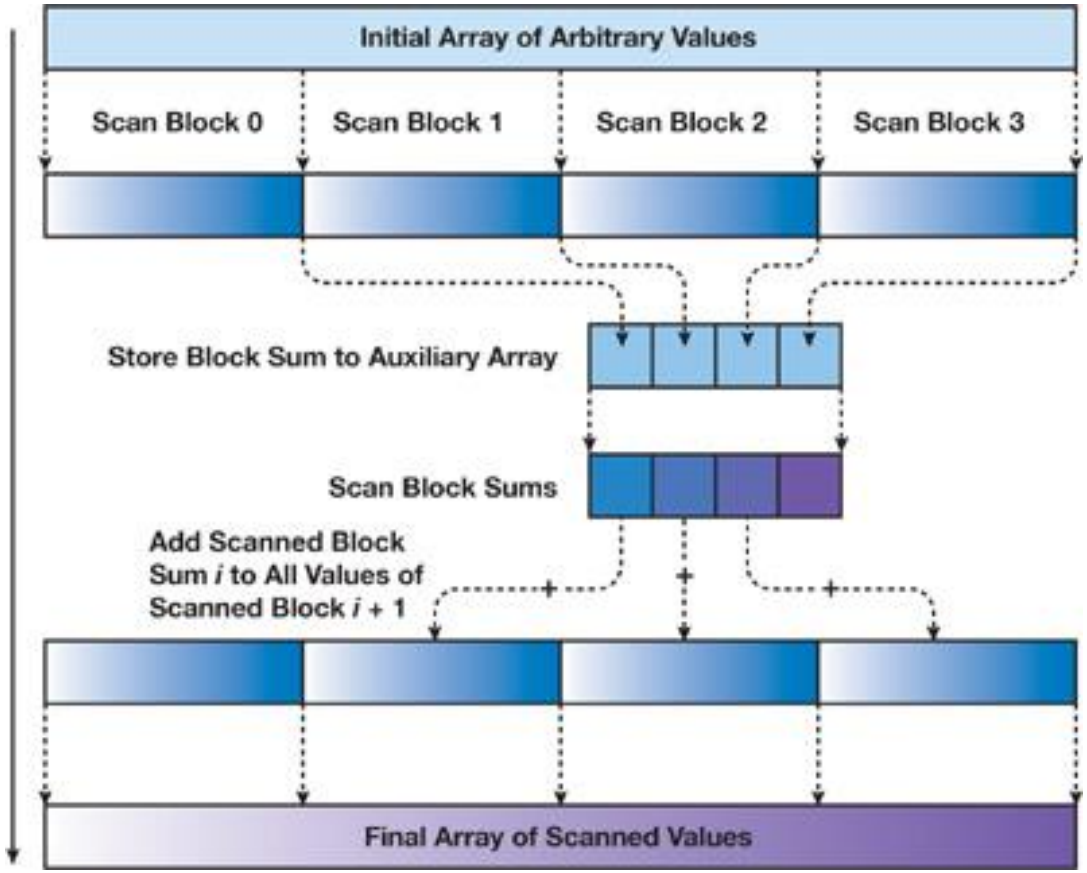
```
for (i = 0..n)
  for (shift = log(n)-1..0)
    if ((i+1) % 2^(shift+1) == 0):
      tmp = x[i]
      x[i] = x[i] + x[i – 2^shift]
      x[i – 2^shift] = tmp
```

Work complexity: O(N)
Step complexity: O(log(N))

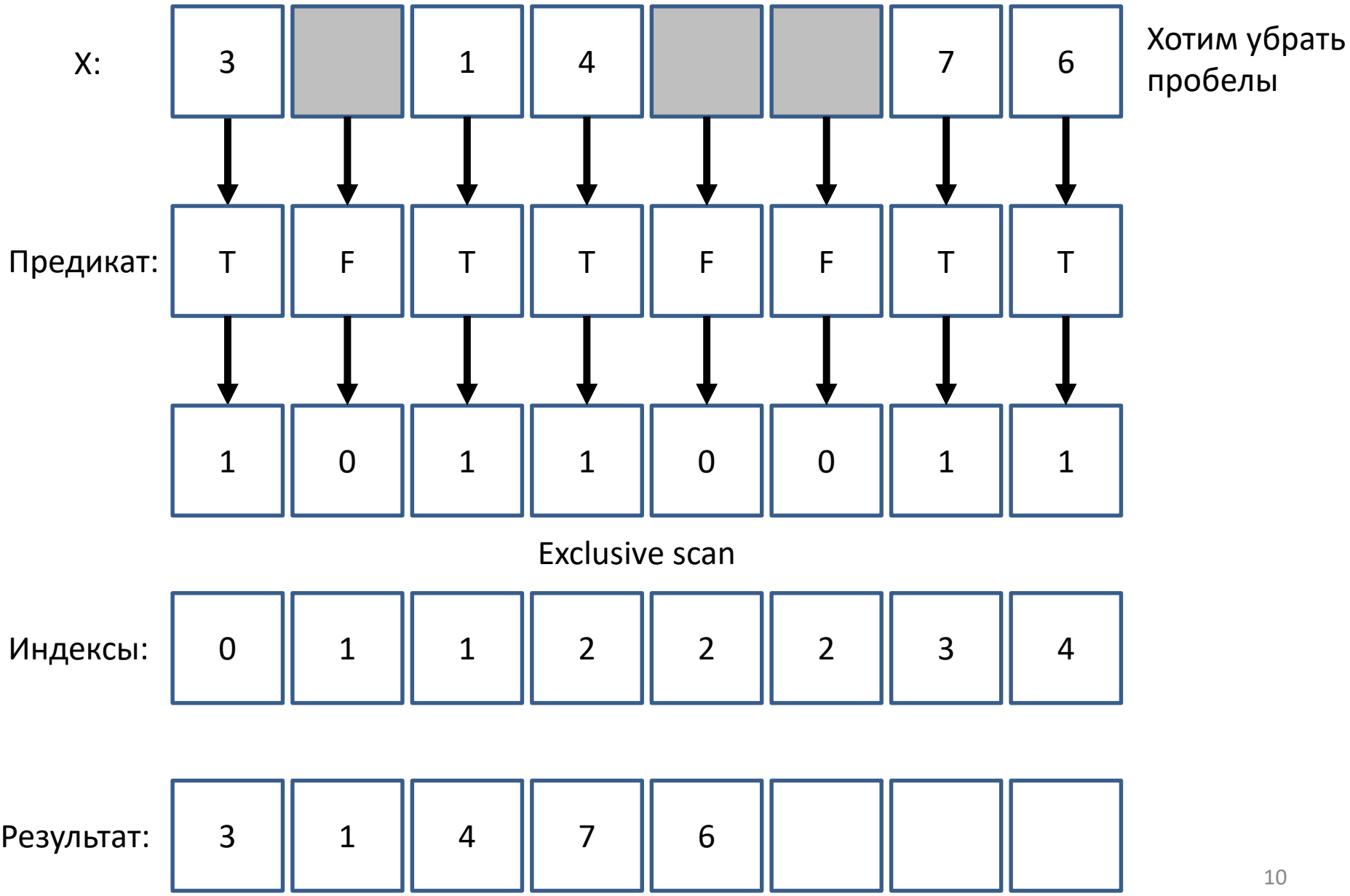
Доказательство:
https://www.cs.cmu.edu/~guyb/papers/Ble93.pdf

# Scan. Общие замечания

- Использование разделяемой памяти
- Рассмотрели алгоритм для блоков размера $2^k$
- Задача произвольного размера:



# Применение scan. Compact



# Radix sort

RADIX_SORT($A, d$)

1 **for** $i \leftarrow 1$ **to** $d$

2 **do** Устойчивая сортировка массива $A$ по $i$-ой цифре

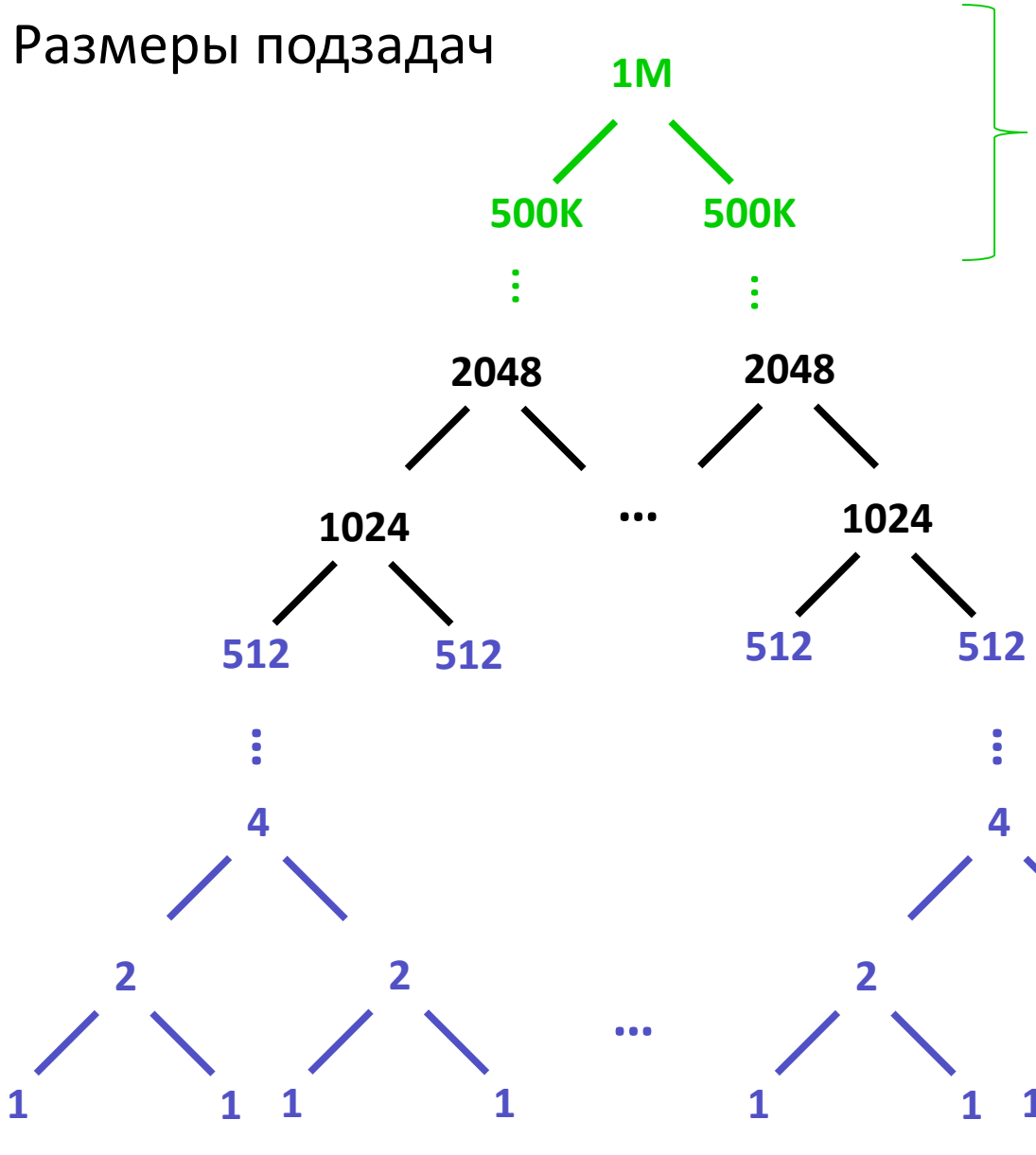
- Устойчивая сортировка – например, сортировка подсчётом
- Получается, надо отсортировать массив из 0 или 1.
  X = [0 0 1 0 1 1 0 1]

1. Запустим exclusive scan для инвертированной последовательности. Получили позиции для 0.
   [1 1 0 1 0 0 1 0] => [0 1 x 2 x x 3 3], x – значение не важно

2. Запустим exclusive scan для исходной последовательности.
   [0 0 1 0 1 1 0 1] => [x x 0 x 1 2 x 3]
   Прибавим к элементам 1 + последние значение массива из шага 1. Получили позиции для 1.
3. Перестановка

# Параллельная сортировка слиянием

Размеры подзадач



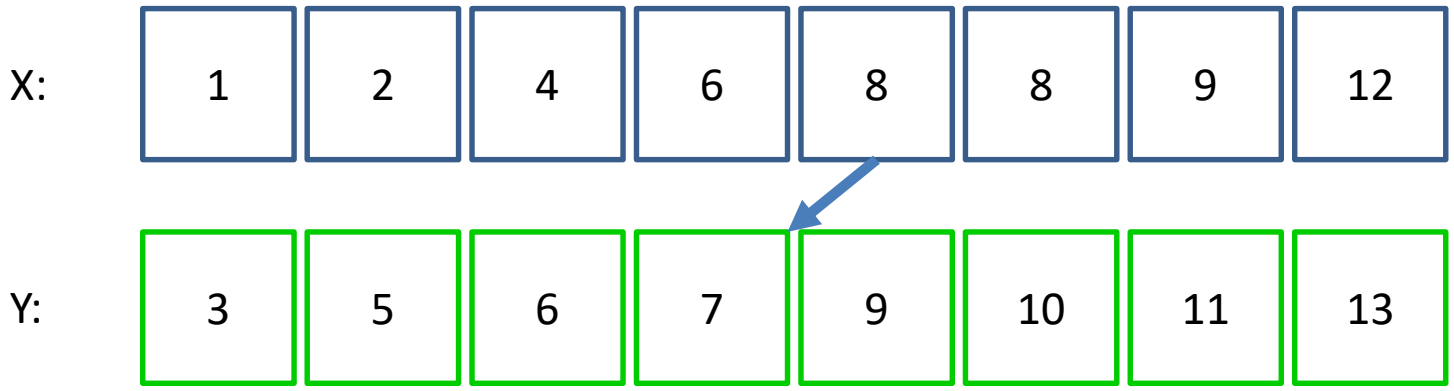

Необходимо разбить процедуру слияния сортированных последовательностей, т.к. подзадача велика и все, кроме одного SM будут простаивать

Размер подзадачи ~= размер блока

Много маленьких подзадач
Лучше использовать другую параллельную сортировку

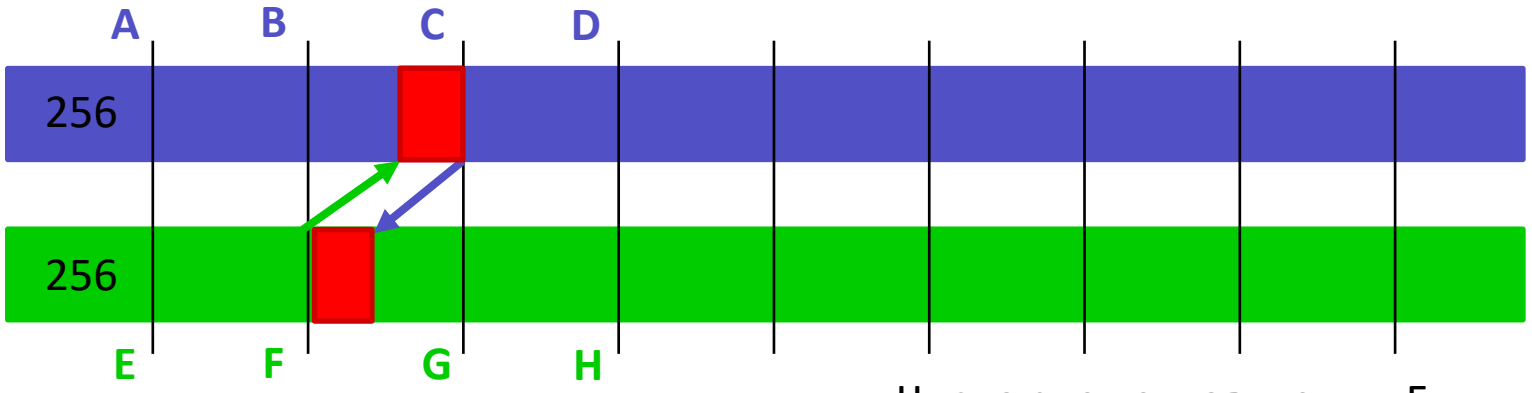
# Слияние отсортированных последовательностей

**Для блока:**




- Назначаем каждому потоку элемент из X или Y
- Результирующая позиция элемента = «свой» индекс + результат бинарного поиска в другом массиве

**Для «большой» подзадачи:** разобьём на блоки равного размера




Слияние ABCDEFGH...: E--A--B--F--C--G--D--H...

Число элементов между F и C не больше 512

# Сортирующие сети.

**Сравнивающее устройство, comparator** — устройство, подключенное к двум проводам, которое упорядочивает текущие значения на проводах.

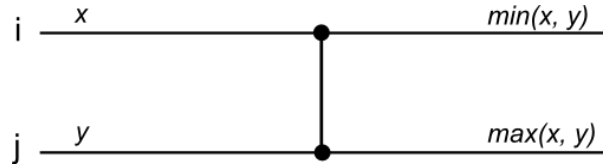



Рис. 1. Сравнивающее устройство

**Сортирующая сеть (Sorting network)** — метод сортировки, основанный только на сравнениях данных. Схематически изображается в виде параллельных прямых (проводов, wires), соединенных вертикальными линиями (сравнивающими устройствами, comparators). Особенность сети сортировки в том, что сравнения выполняются независимо от предыдущих (data independent sorting).

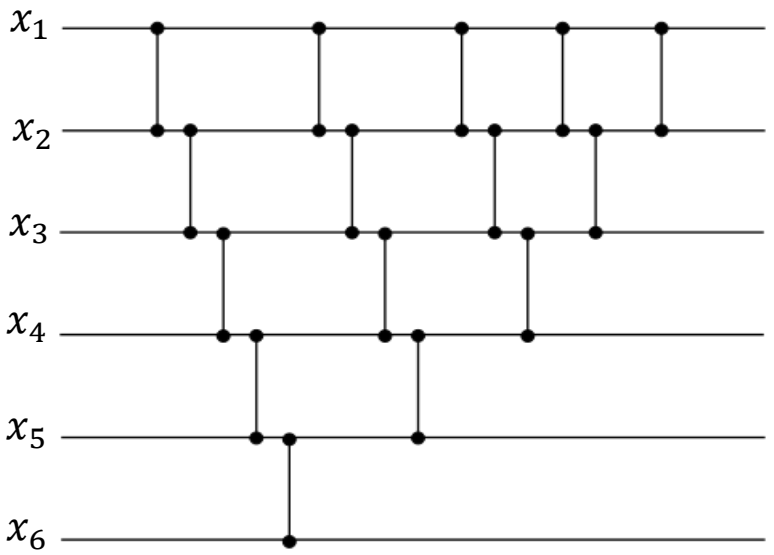



Рис.2 Сортирующая сеть для сортировки пузырьком

**Теорема («0-1 принцип»):** Если сеть компараторов сортирует все последовательности из нулей и единиц, то она сортирующая

# Битоническая сортировка (1)

**Битоническая последовательность** – последовательность чисел, которая сначала возрастает, потом убывает или приводится к такой циклическим сдвигом.

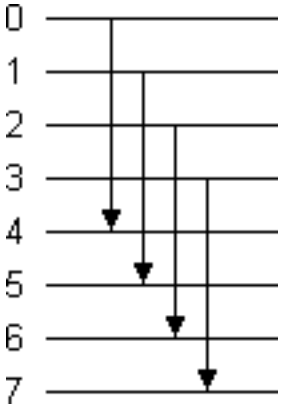



Рис.1 Сеть $B_8$

Рассмотрим последовательность из n чисел $X = \{x_0, x_1, \ldots, x_{n-1}\}$

**Определение:** Сортирующая сеть $B_n$ - сеть, из компараторов $[x_0, x_{n/2}]$ , $[x_1, x_{\frac{n}{2}+1}]$ ,…, $[x_{\frac{n}{2}-1}, x_{n-1}]$

**Утверждение:** Результат применения сети $B_n$ к битонической последовательности $X$ – последовательность чисел $\{a_0, a_1, \ldots, a_{\frac{n}{2}-1}, b_0, b_1, \ldots, b_{\frac{n}{2}-1}\}$: такая, что

1. $a_i \leq b_j, \forall i, j$
2. $\{a_0, a_1, \ldots, a_{\frac{n}{2}-1}\}$ – битоническая
3. $\{b_0, b_1, \ldots, b_{\frac{n}{2}-1}\}$ – битоническая

# Битоническая сортировка (2)

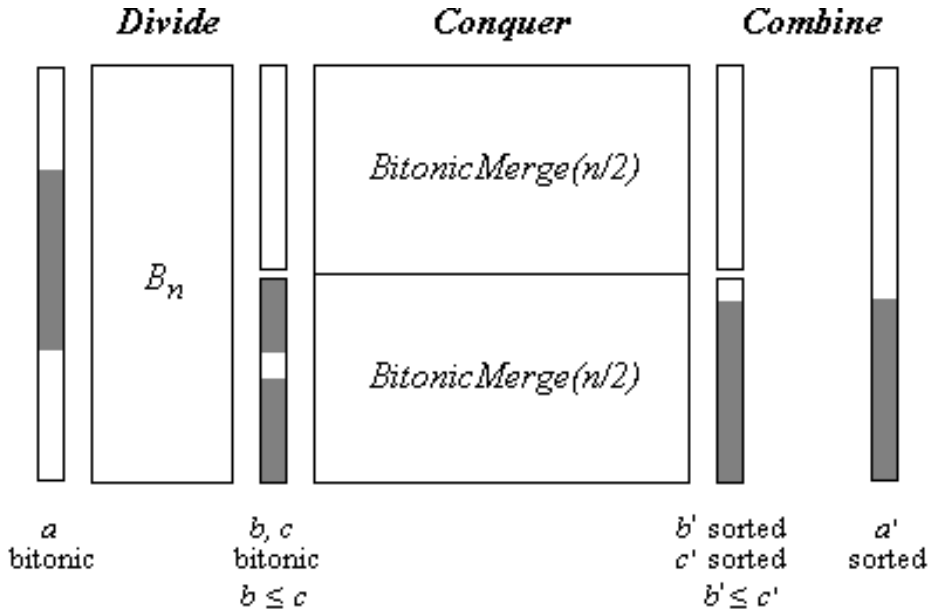



Рис.1. BitonicMerge(n)

**Утверждение:** BitonicMerge(n) – сортирует битоническую последовательность

# Битоническая сортировка (3)

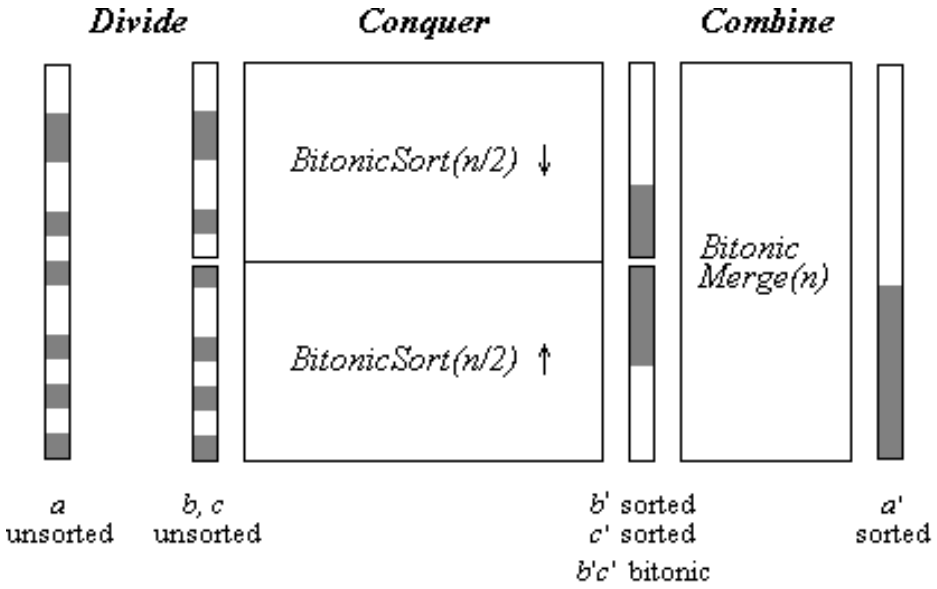



Рис.1. BitonicSort(n)

**Утверждение:** BitonicSort(n) – сортирует  произвольную последовательность

# Далее

Nested parallelism:

- https://software.intel.com/en-us/articles/sierpinski-carpet-in-opencl-20
- https://software.intel.com/en-us/articles/gpu-quicksort-in-opencl-20-using-nested-parallelism-and-work-group-scan-functions

# Часть 2. Оптимизации

# Содержание

- Пример 1: параллельная редукция
  - Расходящиеся ветвления
  - Конфликт банков памяти (banks conflict)
  - Развертка циклов
    - Отказ от barrier(CLK_LOCAL_MEM_FENCE);
    - Способ реализации
- Параллелизм на уровне инструкций (Instruction level parallelism)
- Асинхронная передача данных
- Другое
  - Ключевое слово restrict
  - Типы данных: float vs half, bool vs bit, uint vs int

# Параллельная редукция. Interleaved addressing (1)

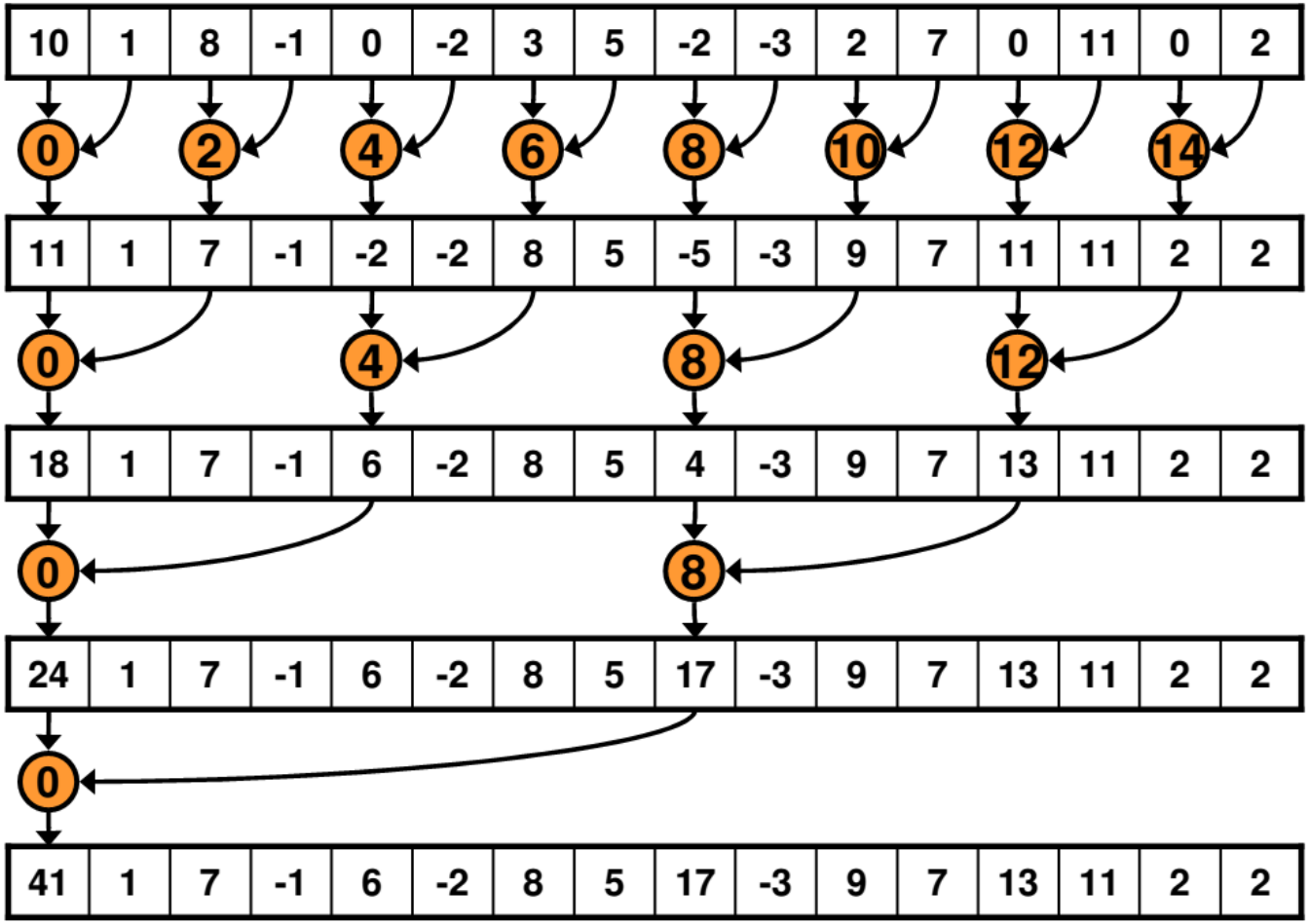
```
__kernel void gpu_reduce_lmem(__global int * g_in, __global int * g_out,
                              __local int * sdata)
{
   size_t idx        = get_global_id (0);
   size_t thread_idx = get_local_id  (0);
   size_t block_size = get_local_size(0);

   //Каждый поток загружает один элемент из глобальной памяти в разделяемую
   sdata[thread_idx] = g_in[idx];
   barrier(CLK_LOCAL_MEM_FENCE);

   for(size_t s = 1; s < block_size; s *= 2)
   {
      if (thread_idx % (2 * s) == 0)
         sdata[thread_idx] += sdata[thread_idx + s];
      barrier(CLK_LOCAL_MEM_FENCE);
   }
   if(thread_idx == 0) g_out[get_group_id(0)] = sdata[0];
}
```

**Примечание:** Код работает только для $\mathrm{BlockDim.x} = 2^k; N \,\%\, BlockDim.x = 0$, где $N$ - размер входного массива.

# Параллельная редукция. Interleaved addressing (2)

Массив в разделяемой памяти:

Номер потока:



# Расходящиеся ветвления

**Напоминание:** Потоки запускаются группами

- Nvidia: warp из 32 потоков.
- AMD: wavefront из 64 потоков

**Расходящиеся ветвления:** В случае ветвления соответствующие инструкции внутри wapr-а будут выполнены последовательно.

**Пример:**

```
if (threadIdx.x % 3 == 0)
   some_instruction1;
else
   some_instruction2;
```

Отдельно выполнится *some_instruction1* для потоков 0, 3, 6, 9, …

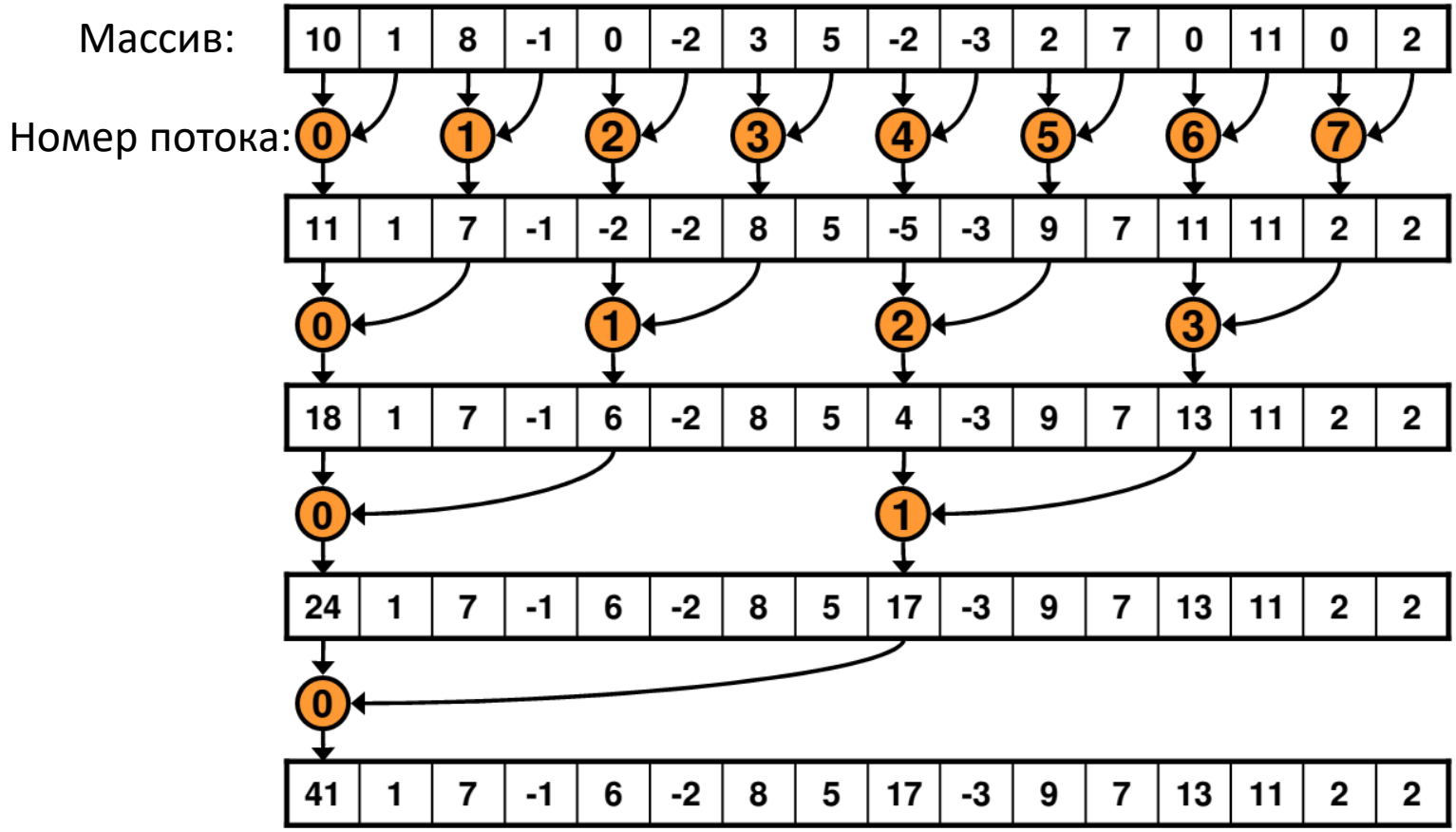
Отдельно выполнится *some_instruction2* для остальных потоков.

Скорость выполнения ниже ожидаемой.

**Что делать:**

Постараться сделать так, чтобы внутри warp-а(wavefront-а) условие ветвления было одинаковым для всех потоков.

# Параллельная редукция. Interleaved addressing (3)




```
for(size_t s = 1; s < block_size; s *= 2)
   {
      if (tid % (2 * s) == 0)
         sdata[tid] += sdata[tid + s];
      barrier(CLK_LOCAL_MEM_FENCE);
   }
```

```
for(size_t s=1; s<block_size; s*=2)
{
  int index = 2 * s * tid;
  if(index < block_size)
     sdata[index] += sdata[index + s];
  barrier(CLK_LOCAL_MEM_FENCE);
}
```

# Банки разделяемой памяти (Nvidia)

Разделяемая память разбивается на банки (banks).

*Compute capability 1.x:* 16 банков памяти, каждому банку ставится в соответствие 32 последовательных бита разделяемой памяти.

| | Банк 0 | Банк 1 | Банк 2 | Банк 3 | Банк 4 | … | Банк 15 |
|---|---|---|---|---|---|---|---|
| Соответствующие адреса памяти (в битах) | 0-31<br>512-543<br>1024-1055<br>… | 32-63<br>544-575<br>… | 64-95<br>576-607<br>… | 96-127<br>608-639<br>… | 128-159<br>640-671<br>… | … | 480-511<br>992-1023<br>… |

*Compute capability 2.x:* 32 банка памяти.

*Compute capability 3.x:* 32 банка памяти. Но в соответствие банку можно ставить по 32 или по 64 последовательных бита с помощью команды `cudaDeviceSetSharedMemConfig()`.

# Конфликт доступа к банкам разделяемой памяти

**Конфликт доступа:**

*Compute capability 1.x:* два потока из одной половины warp-а одновременно обращаются к одному банку памяти.

*Compute capability 2.x* и *3. x:* два потока из одного warp-а одновременно обращаются к одному банку памяти.

В случае конфликта между потоками инструкции доступа будут выполнены последовательно

**Исключение:** «вещательный» доступ (Broadcast Access)

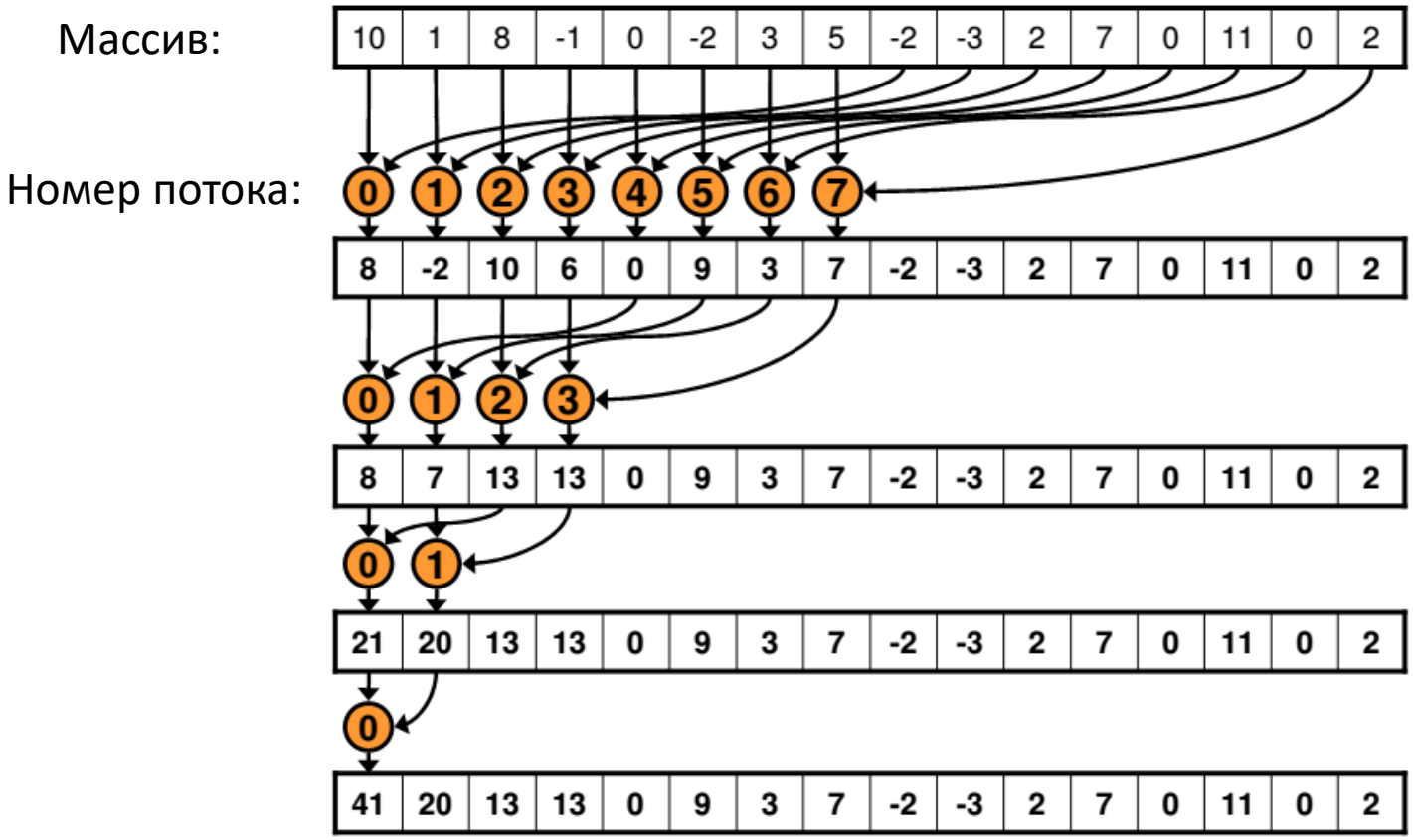
*Compute capability 1.x:* Если несколько потоков читают из одного банка, но обращаются к одному 32 битному слову, то они могут выполнить доступ одновременно. Но одновременно будет выполнено не более одного «вещательного» доступа.

*Compute capability 2.x* и *3.x:* Может быть более одного «вещательного» доступа одновременно.

*Compute capability 3.x:* «вещательный доступ» может работать при обращение к одному 64 битному слову.

Подробнее в cuda_c_programming_guide. Разделы G.3.3, G.4.3, G.5.3.

# Параллельная редукция. Sequential addressing (1)




```
for(size_t s=1; s<block_size; s*=2)
{
  int index = 2 * s * tid;
  if(index < block_size)
     sdata[index] += sdata[index + s];
  barrier(CLK_LOCAL_MEM_FENCE);
}
```

```
for(size_t s=block_size/2; s>0; s/=2)
{
  if(tid < s)
    sdata[tid] += sdata[tid + s];
  barrier(CLK_LOCAL_MEM_FENCE);
}
```

# Развертка циклов (1)

```
for(size_t s=block_size/2; s>0; s/=2)
{
  if(tid < s)
    sdata[tid] += sdata[tid + s];
  barrier(CLK_LOCAL_MEM_FENCE);
}
```

```
for(size_t s=block_size/2; s>32; s/=2)
{
  if (tid < s)
    sdata[tid] += sdata[tid + s];
  barrier(CLK_LOCAL_MEM_FENCE);
}
volatile float * v_sdata = sdata; //volatile!!!
if (tid < 32)
{ //Можно убрать barrier благодаря SIMD архитектуре
  v_sdata[tid] += v_sdata[tid + 32];
  v_sdata[tid] += v_sdata[tid + 16];
  v_sdata[tid] += v_sdata[tid + 8];
  v_sdata[tid] += v_sdata[tid + 4];
  v_sdata[tid] += v_sdata[tid + 2];
  v_sdata[tid] += v_sdata[tid + 1];
}
```

# Развертка циклов(2)

**Идея:** Развернуть весь цикл.

**Проблема:** Как сделать для произвольного размера блока? **Решение:** Макросы!

```
#if BLOCK_SIZE >= 512 //Считаем, что размер блока не более 512
  if (tid < 256) {sdata[tid] += sdata[tid + 256];} barrier(CLK_LOCAL_MEM_FENCE);
#endif
#if BLOCK_SIZE >= 256
  if (tid < 128) {sdata[tid] += sdata[tid + 128];} barrier(CLK_LOCAL_MEM_FENCE);
#endif
#if BLOCK_SIZE >= 128
  if (tid < 64) {sdata[tid] += sdata[tid + 64];} barrier(CLK_LOCAL_MEM_FENCE);
#endif
volatile float * v_sdata = sdata; //volatile!!!
if (tid < 32) {
  #if (BLOCK_SIZE >= 64) v_sdata[tid] += v_sdata[tid + 32]; #endif
  #if (BLOCK_SIZE >= 32) v_sdata[tid] += v_sdata[tid + 16]; #endif
  #if (BLOCK_SIZE >= 16) v_sdata[tid] += v_sdata[tid + 8];  #endif
  #if (BLOCK_SIZE >= 8)  v_sdata[tid] += v_sdata[tid + 4];  #endif
  #if (BLOCK_SIZE >= 4)  v_sdata[tid] += v_sdata[tid + 2];  #endif
  #if (BLOCK_SIZE >= 2)  v_sdata[tid] += v_sdata[tid + 1];  #endif
}
```

**Сборка программы:** program.build(devices, “-D BLOCK_SIZE=some_value”);

# Развертка циклов (3)

В OpenCL 2.0 можно использовать шаблоны C++, но только в функциях

```
template<size_t BLOCK_SIZE>
void gpu_reduce_lmem(…)
{
  if (BLOCK_SIZE >= 512) //Считаем, что размер блока не более 512
    if (tid < 256) {sdata[tid] += sdata[tid + 256];} barrier(CLK_LOCAL_MEM_FENCE);
  if (BLOCK_SIZE >= 256)
    if (tid < 128) {sdata[tid] += sdata[tid + 128];} barrier(CLK_LOCAL_MEM_FENCE);
  if (BLOCK_SIZE >= 128)
    if (tid < 64) {sdata[tid] += sdata[tid + 64];} barrier(CLK_LOCAL_MEM_FENCE);
  volatile float * v_sdata = sdata; //volatile!!!
  if (tid < 32) {
    if (BLOCK_SIZE >= 64) v_sdata[tid] += v_sdata[tid + 32];
    if (BLOCK_SIZE >= 32) v_sdata[tid] += v_sdata[tid + 16];
    if (BLOCK_SIZE >= 16) v_sdata[tid] += v_sdata[tid + 8];
    if (BLOCK_SIZE >= 8)  v_sdata[tid] += v_sdata[tid + 4];
    if (BLOCK_SIZE >= 4)  v_sdata[tid] += v_sdata[tid + 2];
    if (BLOCK_SIZE >= 2)  v_sdata[tid] += v_sdata[tid + 1];
  }
}
```

# Instruction level parallelism (1)

Instruction level parallelism (ILP) – используем то, что разные инструкции могут выполнятся параллельно.

```
__global void ilp_example()
{
  a = a + foo;
  b = b + foo;
  c = b + foo;
}
```

Эти инструкции независимы, то есть могут быть выполнены параллельно!

**Факт 1:** До четырех инструкций из одного потока могут выполнятся параллельно. Проверено экспериментально - volkov10-GTC.

**Факт 2:** Уменьшение числа потоков не обязательно уменьшает скорость передачи данных. Можно скрывать задержки чтения данных с помощью считывания большего числа данных из одного потока (см. volkov10-GTC).

# Instruction level parallelism (2)

**Идея:** Уменьшение числа потоков на потоковый мультипроцессор увеличивает число регистров доступное одному потоку.

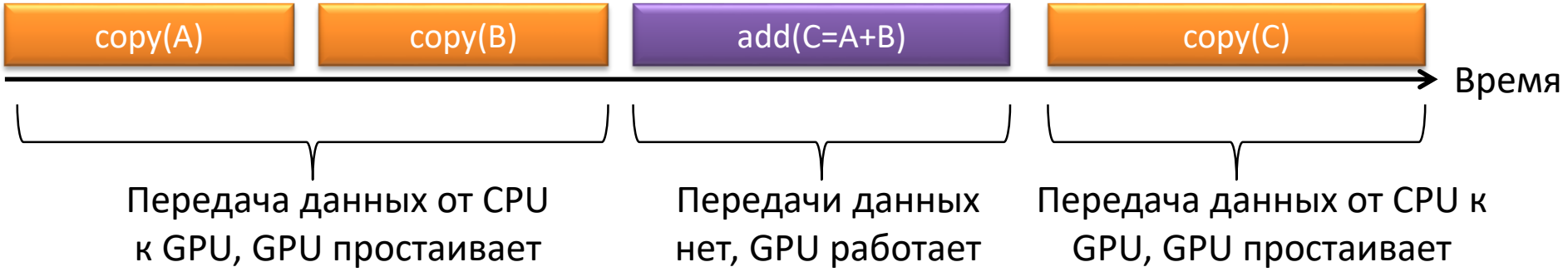
**Идея:** Чтобы меньше обращаться к разделяемой памяти, а больше к регистрам будем записывать в память больше результирующих данных из одного потока.

**Вывод:** Уменьшив число потоков на потоковый мультипроцессор (occupancy < 100 %) можно ускорить программу благодаря эффективному использованию регистров.

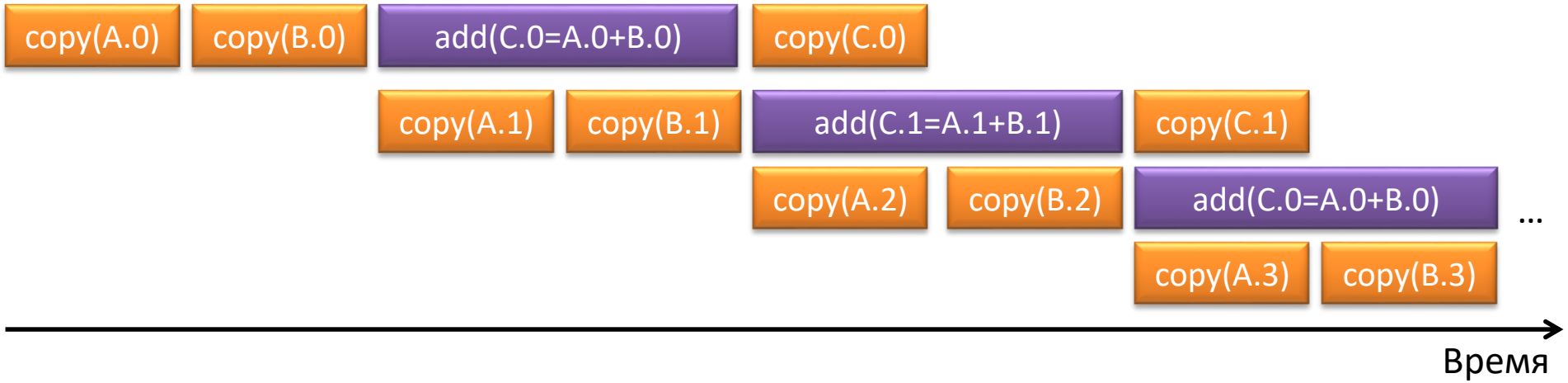
Примеры: volkov10-GTC

# Асинхронная передача данных (1)

Рассмотрим как работает программа сложения двух массивов:




**Идея:** Передать часть данных, запустить вычисления, передать еще данных, запустить вычисления для них и т.д.



# Ключевое слово restrict

Ключевое слово **restrict** означает, что данные, на которые указывают такие указатели не указывают на пересекающиеся объекты.

**Пример использования:**

```
__kernel void foo(__global const int * a, __global const int * b,
                  __global int * c )
{
   c[0] = a[0] + b[0];
   c[1] = a[0] + b[0];
   //c[0] может указывать на a[0] или b[0], поэтому
   //a[0] иb[0] будут два раза загружены из глобальной памяти
}
```

```
__kernel void foo(__global const int * restrict a,
                  __global const int * restrict b,
                  __global int * restrict c )
{
   c[0] = a[0] + b[0];
   c[1] = a[0] + b[0];
   //a[0] и b[0] могут быть загружены один раз из
   //глобальной памяти и помещены в регистры
}
```

# Выбор типов данных

- Если не требуется высокая точность, то можно использовать half вместо float
  - Объем передаваемых данных уменьшится
  - При вычислениях данных все равно будут приводится к типу float, но такая операция эффективна (появляются видеокарты с аппаратной поддержкой типа half!)
- Размер типа bool может быть больше одного бита. При вычисления тип bool будет приводится к int. Можно паковать 32 bool-а в один int
  - Битовые операции для int эффективны
- Компилятор обязан проверять переполнение для uint, но не обязан делать это для int. Поэтому int немного быстрее

# Ссылки

Изображения на слайдах в данной презентации взяты из следующих источников:

- http://neerc.ifmo.ru/wiki/index.php?title=%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D1%83%D1%8E%D1%89%D0%B8%D0%B5_%D1%81%D0%B5%D1%82%D0%B8 (слайд 14)
- http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonicen.htm (слайды 15-18)
- https://en.wikipedia.org/wiki/Bitonic_sorter
- developer.download.nvidia.com/assets/cuda/files/reduction.pdf (слайды 5, 7, 10)