Базовые операции над массивами – reduce, построение гистограмм и сортировки

Лекция 3

Редукция

- $A=a_0+a_1+a_2+\ldots a_{n-1}$
- Последовательная реализация:

$A=(\ldots((a_0+a_1)+a_2)+\ldots+a_{n-1})$

```
for (int i = 0; i < n; i++)
sum += a[i];
```

Редукция

- Параллельная реализация:

$A=(a_0+\ldots+a_k)+(a_{k+1}+\ldots+a_m)+\ldots$

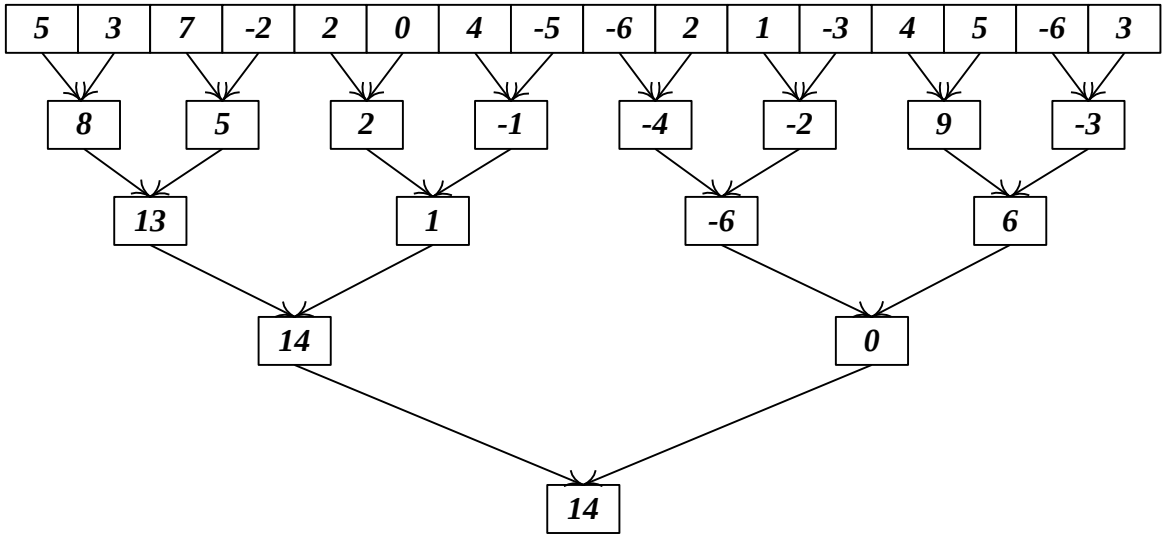
поток_1 поток_2

- CUDA реализация:

$A=(a_0+\ldots+a_{n-1})+(a_n+\ldots+a_{2n-1})+\ldots$

блок_1 блок_2

Иерархическое суммирование

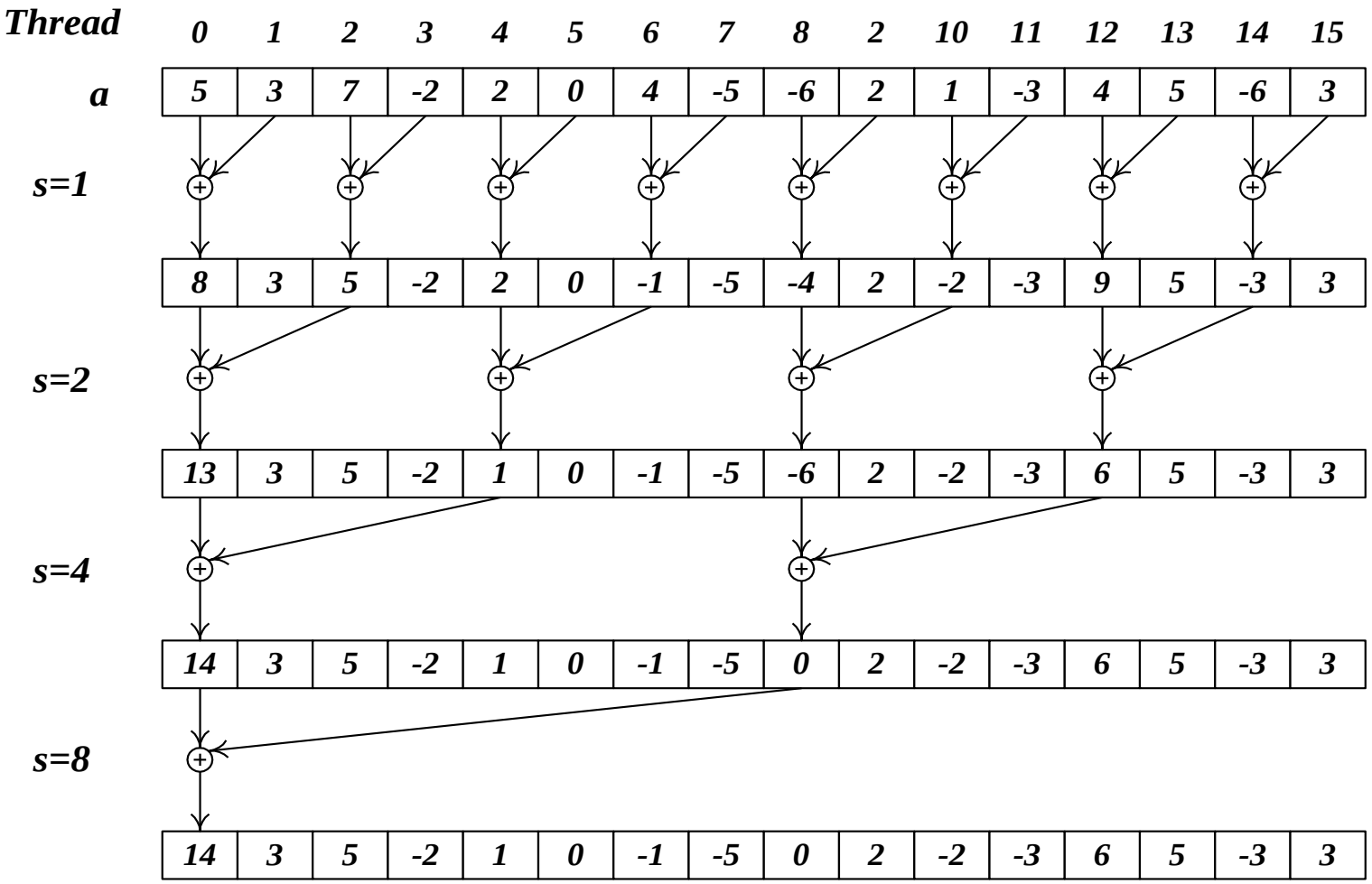


- Позволяет проводить суммирование параллельно, используя много нитей
- Требует $\log_2 N$ шагов

Реализация параллельной редукции

- Каждому блоку сопоставляем часть массива
- Блок
  - Копирует данные в shared-память
  - Иерархически суммирует данные в shared-памяти
  - Сохраняет результат

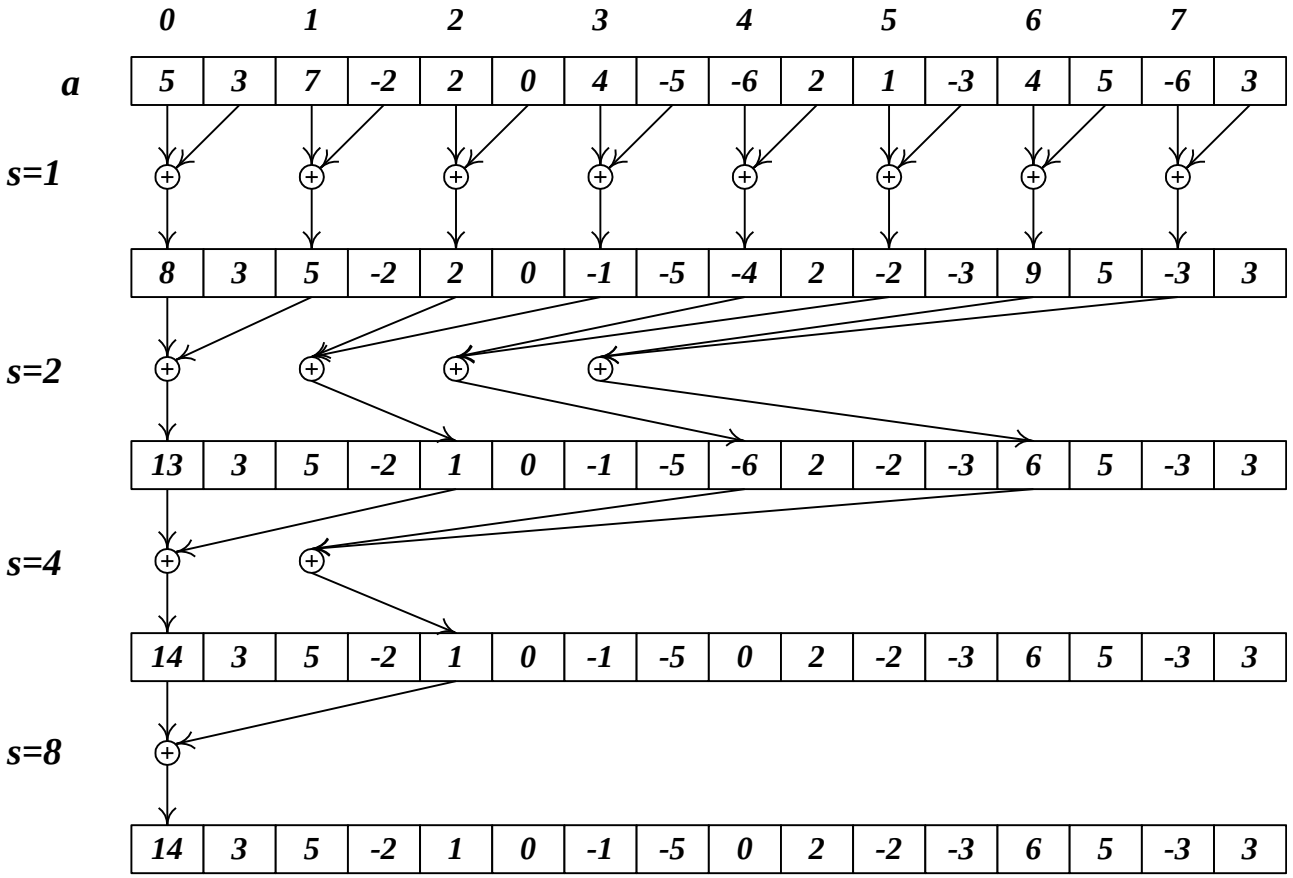
Редукция 1. Простейший вариант



Редукция 1. Простейший вариант

```
__global__ void reduce1 ( int * inData, int * outData )
{
  __shared__ int data [BLOCK_SIZE];
  int tid = threadIdx.x;
  int i   = blockIdx.x * blockDim.x + threadIdx.x;

  data [tid] = inData [i];        // load into shared memory
  __syncthreads ();
  for ( int s = 1; s < blockDim.x; s *= 2 ) {
    if ( tid % (2*s) == 0 )       // heavy branching !!!
      data [tid] += data [tid + s];
    __syncthreads ();
  }
  if ( tid == 0 )                 // write result of block reduction
    outData[blockIdx.x] = data [0];
}
```

Редукция 2. Перераспределение данных и операций по нитям



Редукция 2. Перераспределение данных и операций по нитям

```
__global__ void reduce2 ( int * inData, int * outData )
{
  __shared__ int data [BLOCK_SIZE];
  int tid = threadIdx.x;
  int i   = blockIdx.x * blockDim.x + threadIdx.x;
  data [tid] = inData [i];        // load into shared memory
  __syncthreads ();
  for ( int s = 1; s < blockDim.x; s <<= 1 )
  {
    int index = 2 * s * tid;
    if ( index < blockDim.x )
      data [index] += data [index + s];
    __syncthreads ();
  }
  if ( tid == 0 )                 // write result of block reduction
    outData [blockIdx.x] = data [0];
}
```

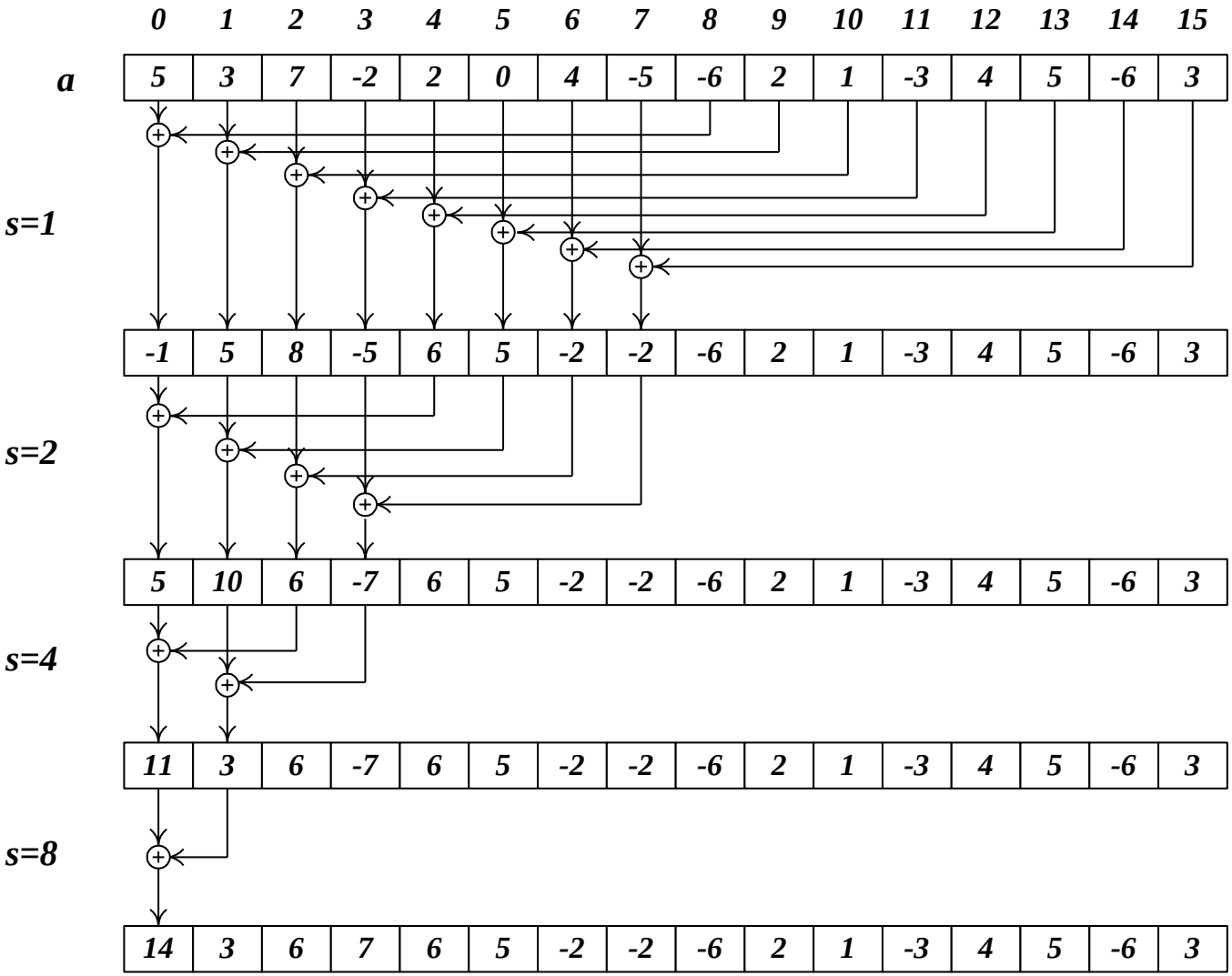
Редукция 2. Перераспределение данных и операций по нитям

- Практически полностью избавились от ветвления
- Однако получили много конфликтов по банкам
  - Для каждого следующего шага цикла степень конфликта удваивается

Редукция 3. Изменение порядка перебора пар

- Изменим порядок суммирования
  - Раньше суммирование начиналось с соседних элементов и расстояние увеличивалось вдвое
  - Начнем суммирование с наиболее удаленных (на dimBlock.x/2) и расстояние будем уменьшать вдвое

Редукция 3. Изменение порядка перебора пар



Редукция 3. Изменение порядка перебора пар

```
__global__ void reduce3 ( int * inData, int * outData )
{
  __shared__ int data [BLOCK_SIZE];
  int tid = threadIdx.x;
  int i   = blockIdx.x * blockDim.x + threadIdx.x;

  data [tid] = inData [i];
  __syncthreads ();
  for ( int s = blockDim.x / 2; s > 0; s >>= 1 )
  {
    if ( tid < s )
      data [tid] += data [tid + s];
    __syncthreads ();
  }
  if ( tid == 0 )
    outData [blockIdx.x] = data [0];
}
```

Редукция 3-4. Изменение порядка перебора пар

- Избавились от конфликтов по банкам
- Избавились от ветвления
- Но, на первой итерации половина нитей простаивает
  - Просто сделаем первое суммирование при загрузке

Редукция 4. Уменьшение числа блоков

```
__global__ void reduce4 ( int * inData, int * outData )
{
  __shared__ int data [BLOCK_SIZE];
  int tid = threadIdx.x;
  int i   = 2 * blockIdx.x * blockDim.x + threadIdx.x;

  data [tid] = inData [i] + inData [i+blockDim.x]; // sum
  __syncthreads ();
  for ( int s = blockDim.x / 2; s > 0; s >>= 1 )
  {
    if ( tid < s )
      data [tid] += data [tid + s];
    __syncthreads ();
  }
  if ( tid == 0 )
    outData [blockIdx.x] = data [0];
}
```

Редукция 5. Оптимизация для s<=32

- При *s<=32* в каждом блоке останется всего по одному *warp*'у, поэтому
  - синхронизация уже не нужна
  - проверка *tid<s* не нужна (она все равно ничего в этом случае не делает).
  - развернем цикл для *s<=32*

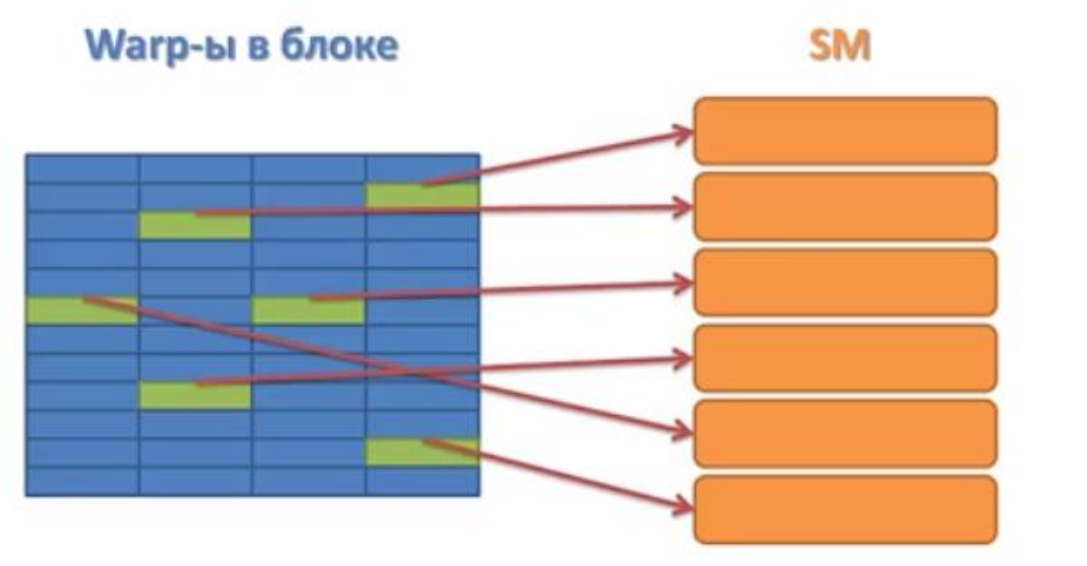
Редукция 5. Оптимизация для s<=32

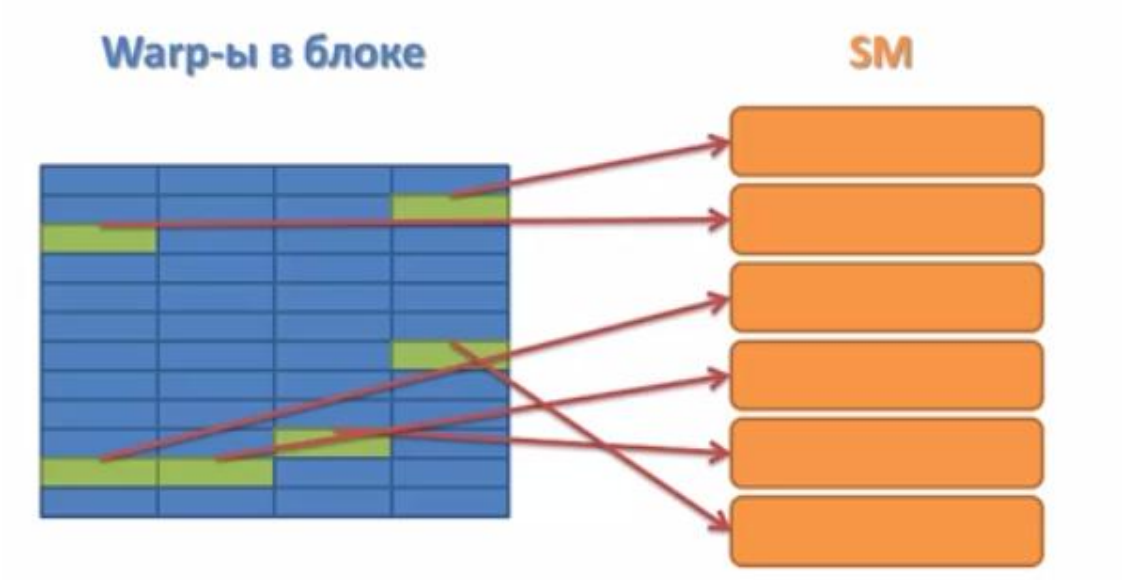
```
for ( int s = blockDim.x / 2; s > 32; s >>= 1 )
{
  if ( tid < s )
    data [tid] += data [tid + s];
  __syncthreads ();
}

if ( tid < 32 ) // unroll last iterations
{
  data [tid] += data [tid + 32];
  data [tid] += data [tid + 16];
  data [tid] += data [tid + 8];
  data [tid] += data [tid + 4];
  data [tid] += data [tid + 2];
  data [tid] += data [tid + 1];
}
```

Редукция, быстродействие

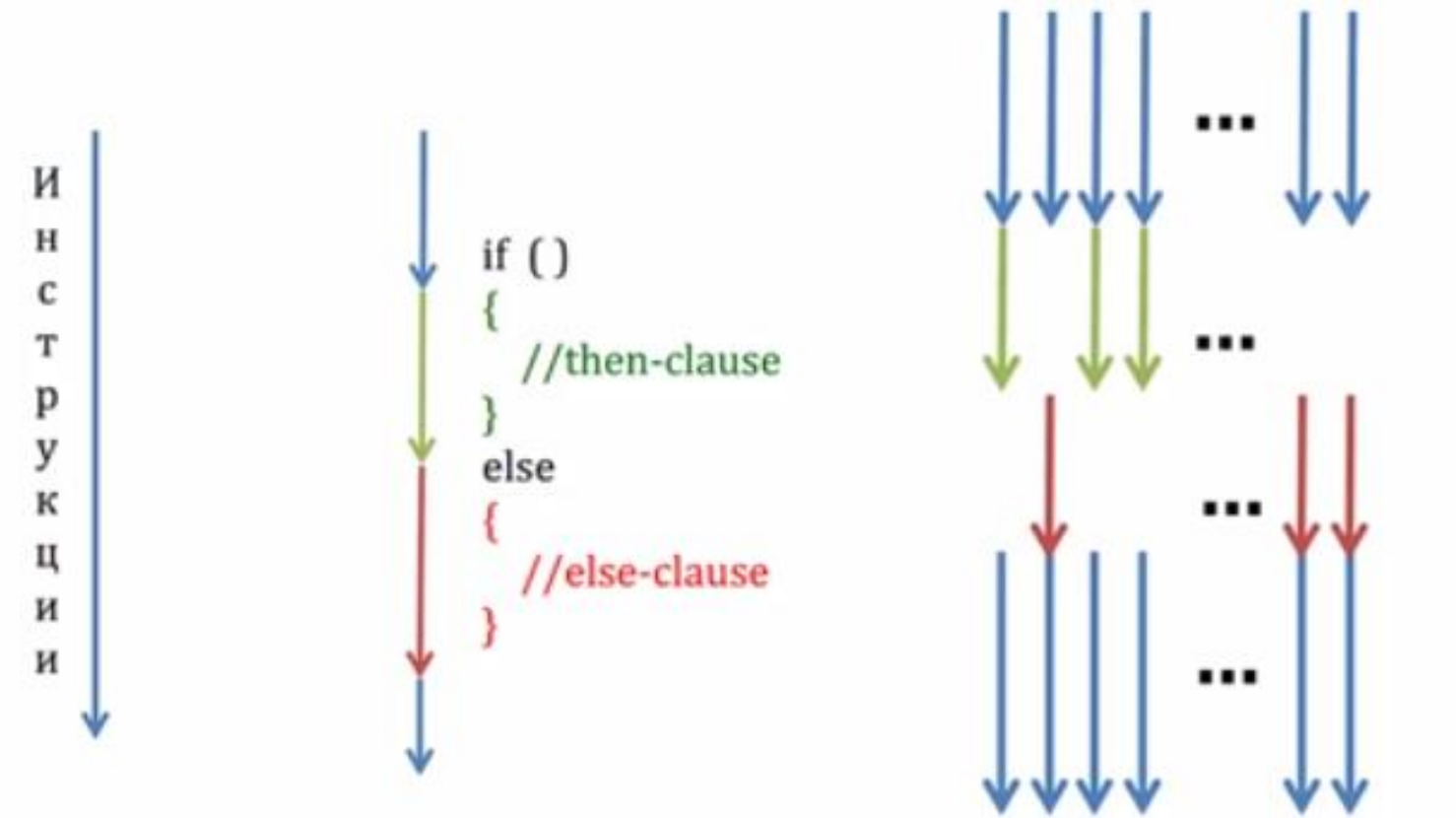
Вариант алгоритма	Время выполнения (миллисекунды)
reduction1	19.09
reduction2	11.91
reduction3	10.62
reduction4	9.10
reduction5	8.67

Warp-ы в блоке

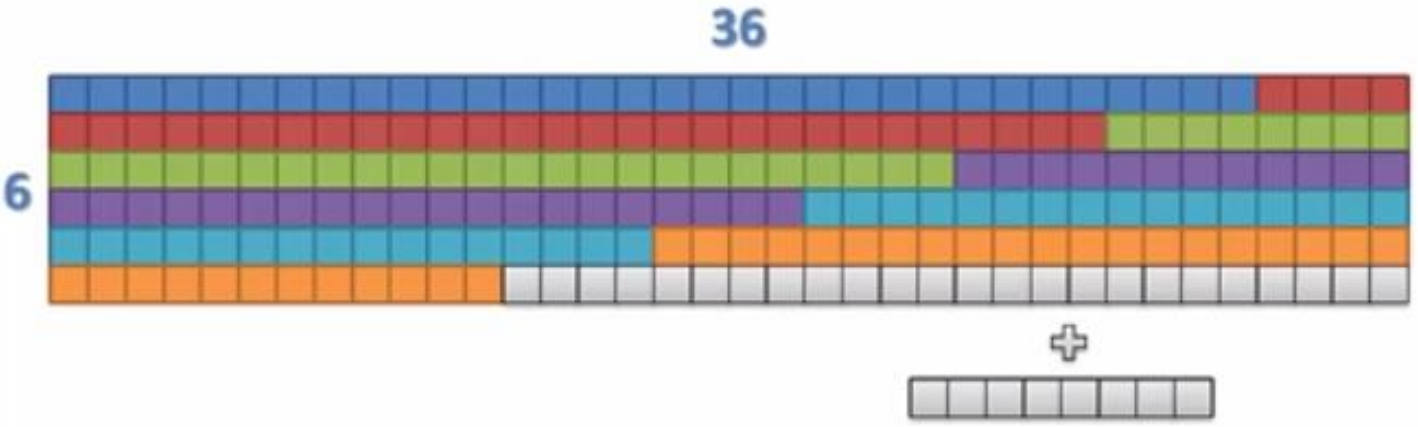




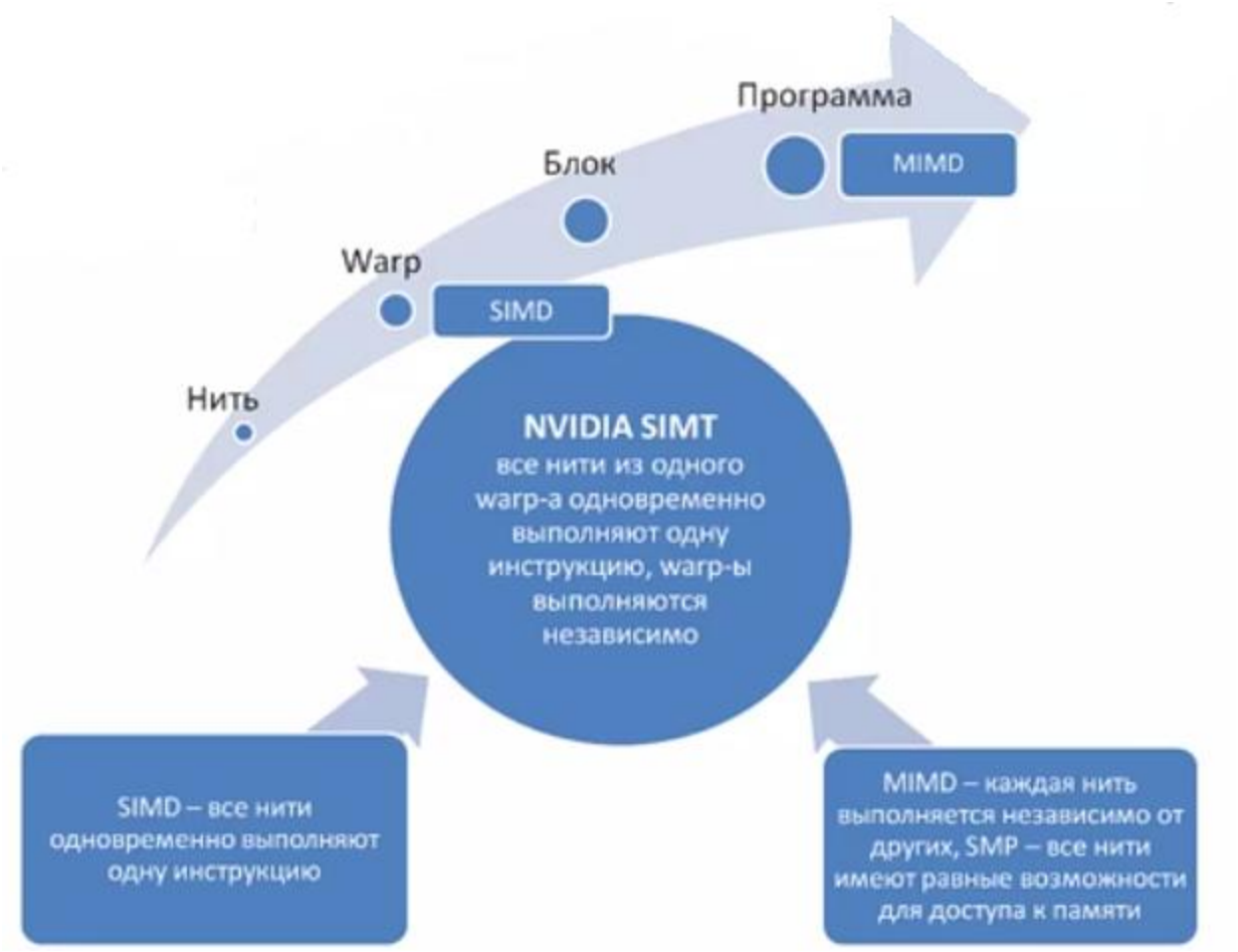
Ветвление в Warp-пах



Распределение нитей в блоке по warp-ам



Программная модель CUDA



Глобальная память. Цифры

Размер глобальной памяти

<=6 Гб

Скорость передачи данных через PCI-E 3.0:

6-8 ГБайт/с

Скорость чтения из глобальной памяти:

150-200 ГБайт/с

Ширина шины памяти DRAM GPU

до 384 бит

Контрольная работа 1

Семь заданий:

1) один балл;
2)-6) два балла;
7) три балла.

Parallel Prefix Sum (Scan)

Имеется входной массив и бинарная операция

$\{a_0, a_1, \ldots.., a_{n-1}\}$

По нему строится массив следующего вида:

$\{I, a_0, a_0 \oplus a_1, a_0 \oplus a_1 \oplus a_2, a_0 \oplus \ldots. \oplus a_{n-2}\}$

Очень легко делается последовательно

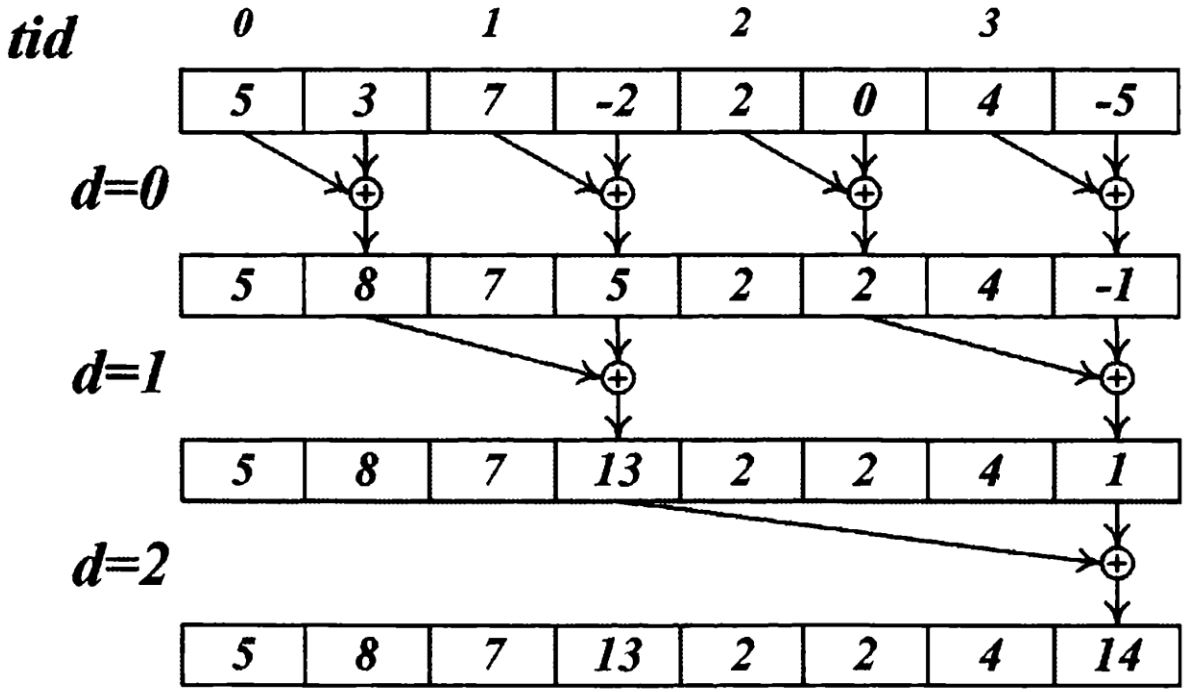
Для распараллеливания используем sum tree

Scan рассчитывается в два этапа:

1) построение sum tree;
2) по sum tree получаем результат.

```
sum [0] = 0
for ( i = 1; i < n; i++ )
      sum [i] = sum [i-1] + a [i-1];
```

Построение sum tree

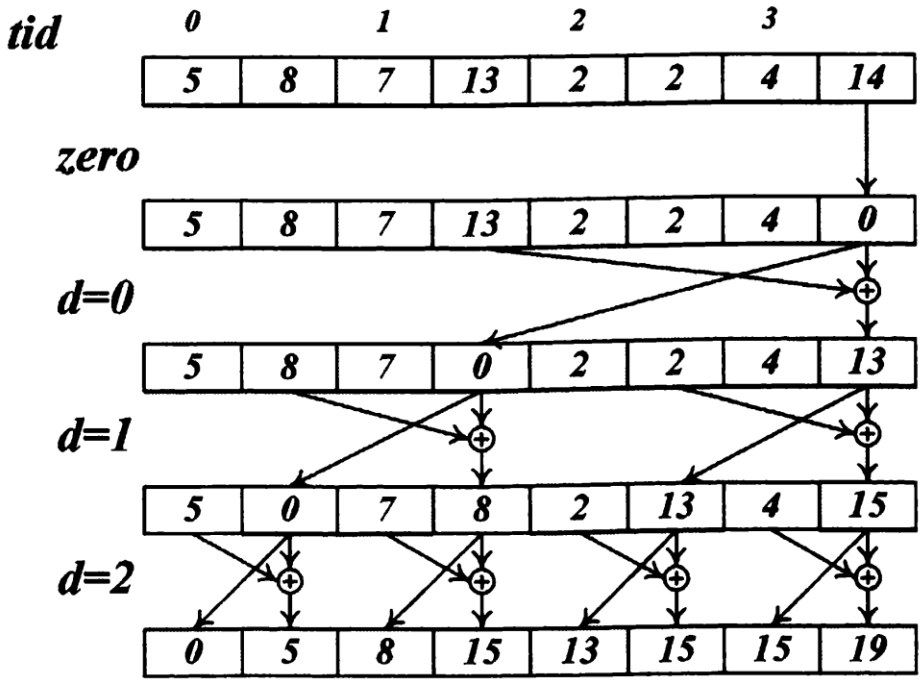


- Используем одну нить на 2 элемента массива
- Загружаем данные
- __syncthreads ()
- Выполняем log(n) проходов для построения дерева

Построение sum tree

```
#define BLOCK_SIZE 256
__global__ void scan1 ( float * inData, float * outData, int n )
{
        __shared__ float temp [2*BLOCK_SIZE];
        int tid = threadIdx.x;
        int offset = 1;

        temp [tid] = inData [tid]; // load into shared memory
        temp [tid+BLOCK_SIZE] = inData [tid+BLOCK_SIZE];

        for ( int d = n >> 1; d > 0; d >>= 1 ){
                __syncthreads ();
                if ( tid < d )
                {
                        int ai = offset * (2 * tid + 1) - 1;
                        int bi = offset * (2 * tid + 2) - 1;
                        temp [bi] += temp [ai];
                }
        offset <<= 1;
        }
```

Получение результата по sum tree



- Одна нить на 2 элемента массива
- Обнуляем последний элемент
- Копирование и суммирование
- Выполняем log(n) проходов для получения результата

Получение результата по sum tree

```
if ( tid == 0 ) temp [n-1] = 0; // clear the last element
for ( int d = 1; d < n; d <<= 1 )
{
        offset >>= 1;
        __syncthreads ();
        if ( tid < d )
        {
                int ai = offset * (2 * tid + 1) - 1;
                int bi = offset * (2 * tid + 2) - 1;
                float t = temp [ai];
                temp [ai] = temp [bi];
                temp [bi] += t;
        }
}
__syncthreads ();
outData [2*tid] = temp [2*tid]; // write results
outData [2*tid+1] = temp [2*tid+1];

}
```

Scan - Оптимизация

Возможные проблемы:
-Доступ к глобальной памяти -> coalesced
-Конфликты банков -> конфликты до 16 порядка !

Оптимизация:
•добавим по одному «выравнивающему» элементу на каждые 16 элементов в shared-памяти;
•к каждому индексу добавим соответствующее смещение.

```
#define LOG_NUM_BANKS 4
#define CONLICT_FREE_OFFS(i) ((i) >> LOG_NUM_BANKS)
```

Scan - Оптимизация

```
__global__ void scan2 ( float * inData, float * outData, int n )
{
__shared__ float temp [2*BLOCK_SIZE+CONFLICT_FREE_OFFS(2*BLOCK_SIZE)];

int tid = threadIdx.x;
int offset = 1;
int ai = tid
int bi = tid + (n / 2);
int offsA = CONFLICT_FREE_OFFS(ai);
int offsB = CONFLICT_FREE_OFFS(bi);
temp [ai + offsA] = inData [ai + 2*BLOCK_SIZE*blockIdx.x];
temp [bi + offsB] = inData [bi + 2*BLOCK_SIZE*blockIdx.x];

for ( int d = n>>1; d > 0; d >>= 1, offset <<= 1 )
{
        __syncthreads ();
        if ( tid < d )
        {
                int ai = offset * (2 * tid + 1) - 1;
                int bi = offset * (2 * tid + 2) - 1;
                ai += CONFLICT_FREE_OFFS(ai);
                bi += CONFLICT_FREE_OFFS(bi);
                temp [bi] += temp [ai];
        }
}
```

Scan - Оптимизация

```
if ( tid == 0 )
{
    temp [i] = 0; // clear the last element
}
for ( int d = 1; d < n; d <<= 1 )
{
        offset >>= 1;
        __syncthreads ();
        if ( tid < d )
        {
                int ai = offset * (2 * tid + 1) - 1;
                int bi = offset * (2 * tid + 2) - 1;
                float t;
                ai += CONFLICT_FREE_OFFS(ai);
                bi += CONFLICT_FREE_OFFS(bi);
                t = temp [ai];
                temp [ai] = temp [bi];
                temp [bi] += t;
        }
}
__syncthreads ();
outData [ai + 2*BLOCK_SIZE*blockIdx.x] = temp [ai + offsA];
outData [bi + 2*BLOCK_SIZE*blockIdx.x] = temp [bi + offsB];
}
```

Scan больших массивов

Рассмотренный код хорошо работает для небольших массивов, целиком, помещающихся в shared-память

В общем случае:

- Выполняем отдельный scan для каждого блока;
- Для каждого блока запоминаем сумму элементов (перед обнулением);
- Применяем scan к массиву сумм;
- К каждому элементу, кроме элементов 1-го блока добавляем значение, соответствующее данному блоку.

Scan3 - Оптимизация

```
__global__ void scan3 ( float * inData, float * outData, int n )
{
__shared__ float temp [2*BLOCK_SIZE+CONFLICT_FREE_OFFS(2*BLOCK_SIZE)];

int tid = threadIdx.x;
int offset = 1;
int ai = tid
int bi = tid + (n / 2);
int offsA = CONFLICT_FREE_OFFS(ai);
int offsB = CONFLICT_FREE_OFFS(bi);
temp [ai + offsA] = inData [ai + 2*BLOCK_SIZE*blockIdx.x];
temp [bi + offsB] = inData [bi + 2*BLOCK_SIZE*blockIdx.x];

for ( int d = n>>1; d > 0; d >>= 1, offset <<= 1 )
{
        __syncthreads ();
        if ( tid < d )
        {
                int ai = offset * (2 * tid + 1) - 1;
                int bi = offset * (2 * tid + 2) - 1;
                ai += CONFLICT_FREE_OFFS(ai);
                bi += CONFLICT_FREE_OFFS(bi);
                temp [bi] += temp [ai];
        }
}
```

Scan3 - Оптимизация

```
if ( tid == 0 )
{
    int i = n - 1 + CONFLICT_FREE_OFFS(n-1);  // отличие от scan 2
    sums [blockIdx.x] = temp [i];             // save the sum
    temp [i] = 0;                             // clear the last element
}
for ( int d = 1; d < n; d <<= 1 )
{
        offset >>= 1;
        __syncthreads ();
        if ( tid < d )
        {
                int ai = offset * (2 * tid + 1) - 1;
                int bi = offset * (2 * tid + 2) - 1;
                float t;
                ai += CONFLICT_FREE_OFFS(ai);
                bi += CONFLICT_FREE_OFFS(bi);
                t = temp [ai];
                temp [ai] = temp [bi];
                temp [bi] += t;
        }
}
__syncthreads ();
outData [ai + 2*BLOCK_SIZE*blockIdx.x] = temp [ai + offsA];
outData [bi + 2*BLOCK_SIZE*blockIdx.x] = temp [bi + offsB];
}
```

Scan больших массивов

```
__global__ void scanDistribute(float * data, float * sums)
{
        data[threadIdx.x + blockIdx.x*2*BLOCK_SIZE] += sums[blockIdx.x];
}
void scan ( float * inData, float * outData, int n )
{
        int numBlocks = n / (2*BLOCK_SIZE);
        float * sums; // суммы элементов для каждого блока
        float * sums2; // результаты scan этих сумм
        if ( numBlocks < 1 ) numBlocks = 1;
                                    // выделяем память под массивы
        cudaMalloc ( (void**)&sums, numBlocks * sizeof ( float ) );
        cudaMalloc ( (void**)&sums2, numBlocks * sizeof ( float ) );
                                    // поблочный scan
        dim3 threads ( BLOCK_SIZE, 1, 1 ), blocks ( numBlocks, 1, 1 );
        scan2<<<blocks, threads>>> ( inData, outData, sums, 2*BLOCK_SIZE );
                                    // выполняем scan для сумм
        if ( n >= 2*BLOCK_SIZE )
                scan ( sums, sums2, numBlocks );
        else cudaMemcpy ( sums2, sums, numBlocks*sizeof(float),
                        cudaMemcpyDeviceToDevice );
                                    // корректируем результат
        threads = dim3 ( 2*BLOCK_SIZE, 1, 1 );
        blocks = dim3 ( numBlocks - 1, 1, 1 );
        scanDistribute<<<blocks,threads>>> ( outData + 2*BLOCK_SIZE, sums2 + 1 );

        cudaFree ( sums );
        cudaFree ( sums2 );
}
```

Построение гистограммы

Пусть задан массив элементов: $a_0,\ldots..,\ a_{n-1}$.

Пусть есть некоторый критерий, позволяющий разделить все элементы на k групп (от 0 до k-1).

Гистограммой для исходного массива элементов будет называться массив $c_0,\ldots..,\ c_{k-1}$, каждый элемент которого равен числу элементов исходного массива, принадлежащий каждой группе.

```
for ( int i = 0; i < k; i++ )
        c [i] = 0;

for ( int i = 0; i < n; i++ )
        c [a[i]]++;
```

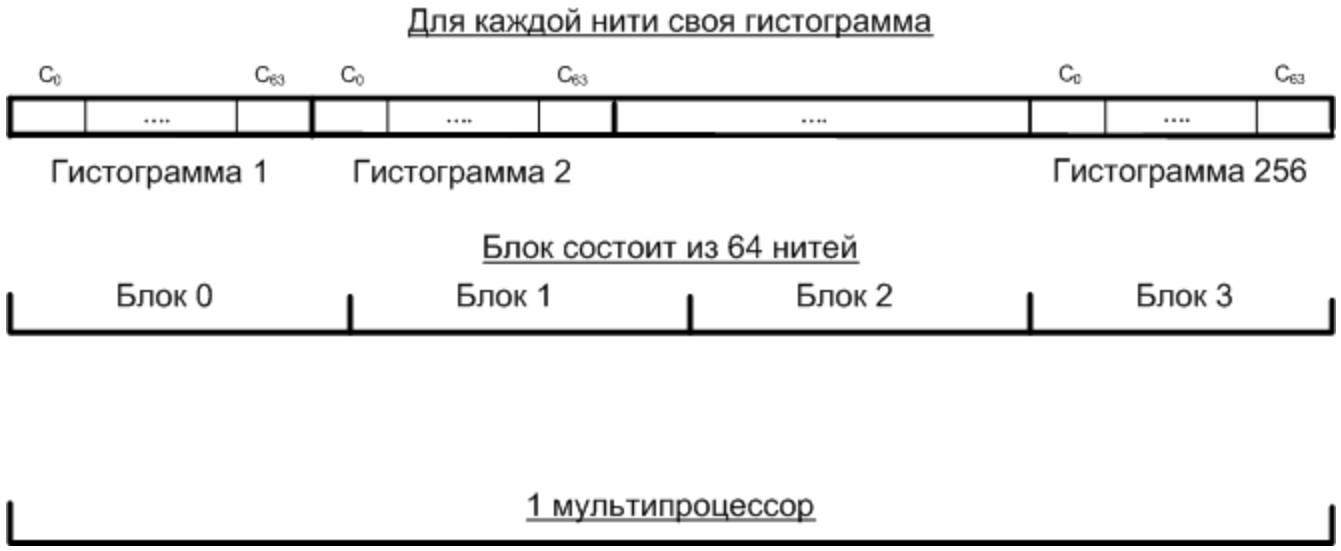
Построение гистограммы. Способы решения задачи с помощью CUDA

1) решение с помощью atomicAdd ;
2) решение без обеспечения атомарности доступа, число групп меньше 64;
3) Решение с обеспечением атомарности в пределах одного warp’а.

Построение гистограммы. Решение без обеспечения атомарности доступа, число групп меньше 64

Вариант 1:

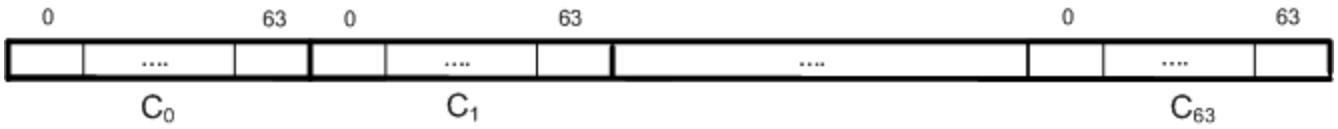
конфликты определяются входными данными




```
tid*64+data
```

Вариант 2:

конфликты не зависят от входных данных

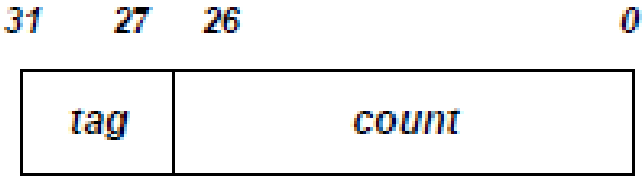



```
tid+64*data
```

```
tid = (threadIdx.x >> 4) | ((threadIdx.x & 0x0F) << 2);
```

Построение гистограммы. Решение с обеспечением атомарности в пределах одного warp’а

- Пусть каждый warp нитей имеет свою таблицу счетчиков
  - 192 нити в блоке дают 6 warp’ов, т.е. 6*256*4=6Кб shared-памяти на блок;
  - 5 старших битов каждого счетчика будут хранить номер нити (внутри warp’а), сделавшей последнюю запись.



Построение гистограммы. Решение с обеспечением атомарности в пределах одного warp’а

```
#define N  (6*1024*1024)

#define NUM_BINS 256        // число счетчиков в гистограмме
__device__ inline void addByte( volatile unsigned * warp_hist, unsigned data, unsigned ttag)
{
   unsigned count;
   do { // прочесть текущее значение счетчика и снять идентификатор нити
   count = warp_hist[data] & 0x07FFFFFU;
   // увеличить его на единицу и поставить свой идентификатор
   count = ttag | (count + 1);
   warp_hist[data] = count;  //осуществить запись
   } while (warp_hist[data] != count); // проверить, прошла ли запись
}
```

–Каждая нить строит новое значение
  –Увеличить на единицу
  –Выставить старшие биты в номер нити в warp’е
–Как минимум одна запись пройдет и соответствующая нить выйдет из цикла

Построение гистограммы. Решение с обеспечением атомарности в пределах одного warp’а

```
#define LOG2_WARP_SIZE  5 // логарифм размера warp's по основанию 2
#define WARP_SIZE       32 // Размер warp'a
#define WARP_N          6  // Число warp'ов в блоке
__global__ void histogramKernel ( unsigned * result, unsigned * data, int n )
{
  __shared__ unsigned hist[NUM_BINS*WARP_N]; //1536 элементов
    // очистить счетчики гистограмм
  for (int i = 0; i < NUM_BINS / WARP_SIZE; i++)
              hist[threadIdx.x + i*WARP_N*WARP_SIZE/*число нитей в блоке=192*/] = 0;

  int warp_base = (threadIdx.x >> LOG2_WARP_SIZE) * NUM_BINS;
  unsigned ttag = threadIdx.x << (32 - LOG2_WARP_SIZE); // получить id для данной нити

   __syncthreads ();
   int global_tid = blockIdx.x * blockDim.x + threadIdx.x;
   int numThreads = blockDim.x * gridDim.x;
   for (int i = global_tid; i < n; i += numThreads)
   {
            unsigned data4 = data [i];
            addByte(hist + warp_base, (data4 >> 0) & 0xFFU, ttag);
            addByte(hist + warp_base, (data4 >> 8) & 0xFFU, ttag);
            addByte(hist + warp_base, (data4 >> 16) & 0xFFU, ttag);
            addByte(hist + warp_base, (data4 >> 24) & 0xFFU, ttag);
   }
   __syncthreads();
  // объединить гистограммы данного блока и записать результат в глобальную память
  // 192 нити суммирую данные до 256 элементов гистограмм
  for (int bin = threadIdx.x; bin < NUM_BINS; bin += (WARP_N*WARP_SIZE))
  { unsigned sum = 0;
            for (int i = 0; i < WARP_N; i++)
                 sum += hist [bin + i*NUM_BINS] & 0x07FFFFFU;

    result[blockIdx.x * NUM_BINS + bin] = sum;
   }
}
```

Построение гистограммы. Решение с обеспечением атомарности в пределах одного warp’а

```
// объединить гистограммы, один блок на каждый NUM_BINS элементов
__global__ void mergeHistogramKernel (unsigned * out_histogram, unsigned * partial_histograms, int histogram_count)
{
     unsigned  sum = 0;
     for (int i = threadIdx.x; i < histogram_count; i += 256)
         sum += partial_histograms[blockIdx.x + i * NUM_BINS];

     __shared__ unsigned data[NUM_BINS];
    data[threadIdx.x] = sum;

    for (unsigned stride = NUM_BINS / 2; stride > 0; stride >>= 1)
    {
         __syncthreads ();
         if (threadIdx.x < stride) data[threadIdx.x] +=  data[threadIdx.x + stride];
     }
     if (threadIdx.x == 0 ) out_histogram[blockIdx.x] = data[0];
}
void histogram( uint * histogram, void * data, unsigned byte_count)
{
  int n = byte_count / 4;
  int num_blocks = n / (WARP_N*WARP_SIZE);
  int num_partials = 240;
  unsigned *partial_histograms = nullptr;
  unsigned int h[NUM_BINS] = {0};
  int *pdata = (int*)data;

  //выделить память под гистограммы блока
  cudaMalloc((void**)&partial_histograms, num_partials*NUM_BINS* sizeof(unsigned));
    // построить гистограмму для каждого блока
  histogramKernel <<<dim3(num_partials), dim3(WARP_N*WARP_SIZE) >>> (partial_histograms, (unsigned*)data, n);


    //объдинить гистограммы отдельных блоков вместе
  mergeHistogramKernel<<<dim3(NUM_BINS), dim3(256)>>> (histogram, partial_histograms, num_partials);
    // освободить выделенную память
  cudaFree(partial_histograms);
}
```

Сортировка. Сеть компараторов (*comparator networks*)

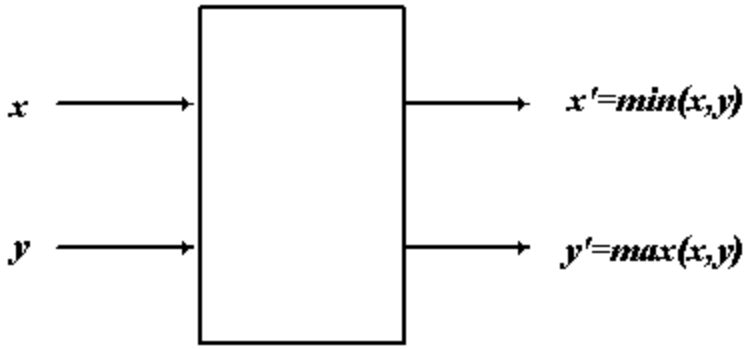


Рисунок - Компаратор

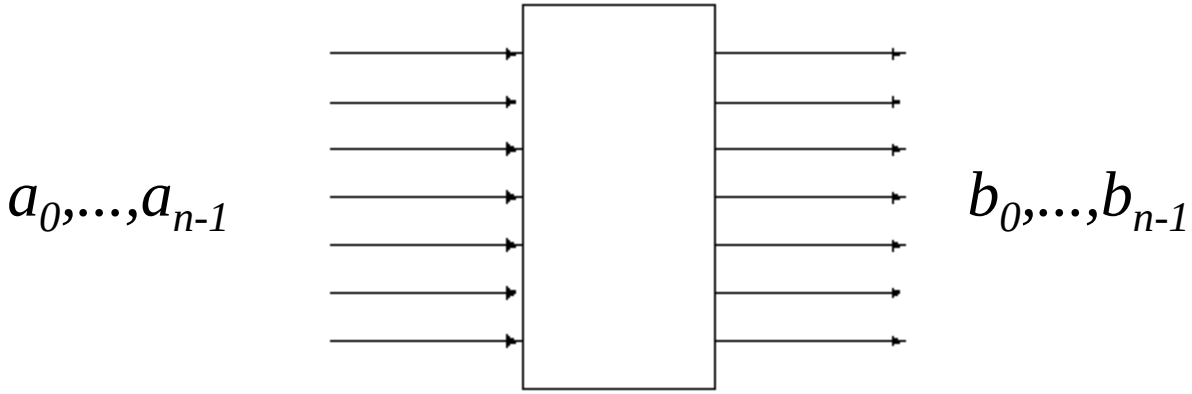
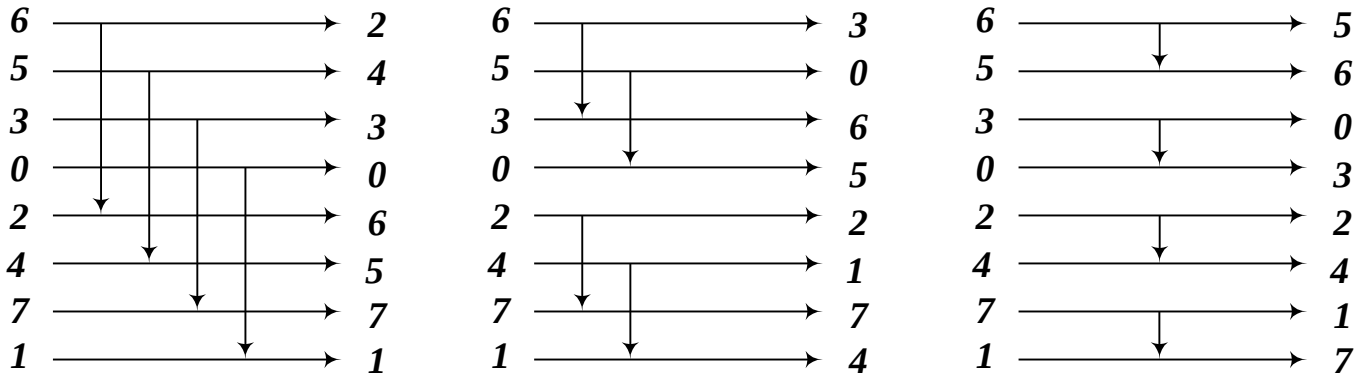


Рисунок – Пример компаратора

Сортировка. Битоническая сортировка

Базовая операция – полуочиститель, упорядочивающий пары элементов на заданном расстоянии:

$$B_n : (x_k, x_{k+n/2}) \to (\min, \max)$$



B_8 B_4 B_2

Битоническая сортировка

- Последовательность называется битонической, если она
  - Состоит из двух монотонных частей
  - Получается из двух монотонных частей циклическим сдвигом
- Примеры:
  - 1,3,4,7,6,5,2
  - 5,7,6,4,2,1,3 (получена сдвигом 1,3,5,7,6,4,2)

Битоническая сортировка

- Если к битонической последовательности из n элементов применить полуочиститель Bn, то в результате у полеченной последовательности
  - Обе половины будут битоническими
  - Любой элемент первой половины будет не больше любого элемента второй половины
  - Хотя бы одна из половин будет монотонной

Битоническая сортировка

Если к битонической последовательности длины n применить получистители $B_n, B_{n/2}, \ldots, B_8, B_4, B_2$

то в результате мы получим отсортированную последовательность (битоническое слияние)!

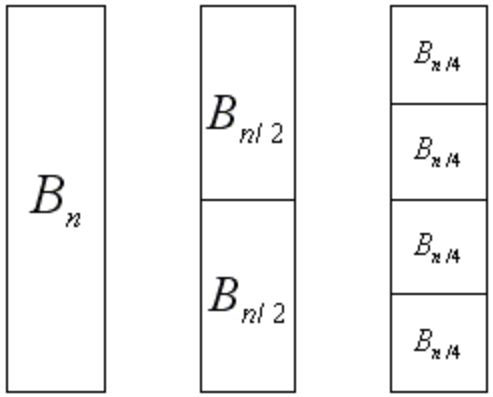
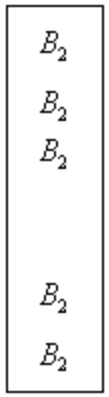


Рисунок - Битоническое слияние



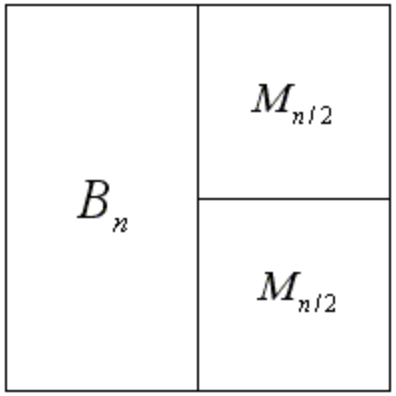
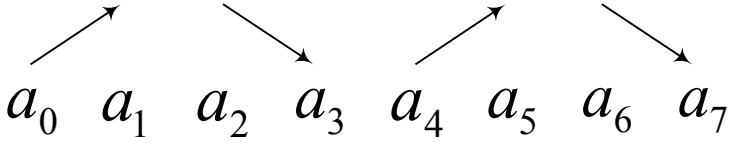


Рисунок - Рекурсивное представление битонического слияния

Битоническая сортировка

1) Пусть есть произвольная последовательность длины n . Применим к каждой паре элементов полуочиститель B_2 с чередующимся порядком сортировки.

Тогда каждая четверка элементов будет образовывать битоническую последовательность.



Битоническая сортировка

2) Применим к каждой такой четверке элементов операцию битонического слияния M_4 с чередующимся порядком сортировки (сначала по возрастанию, потом по убыванию). Получим битоническую последовательность.

3) Применим операцию битонического слияния M_8. Получим отсортированную последовательность.

Всего потребуется *log(n)*log(n)* проходов для полной сортировки массива.

Битоническая сортировка

Очень хорошо работает для сортировки через шейдеры, но плохо использует возможности CUDA, поэтому обычно не используется для сортировки больших массивов

Битоническая сортировка

```
#define N  1024
#define BLOCK_SIZE (N/2) //размер блока

__device__ void Comparator(unsigned int& keyA, unsigned int& valA, unsigned int& keyB, unsigned int& valB, unsigned int dir)
{
   unsigned int t;
  if ((valA > valB) == dir) //поменять местами (keyA, valA) и (keyB, valB)
  {
    t = keyA; keyA = keyB; keyB = t;
    t = valA; valA = valB; valB = t;
  }
}
__global__ void bitonicSortShared(unsigned int* dstKey, unsigned int * dstVal, unsigned int* srcKey, unsigned int* srcVal, unsigned
int arrayLength, unsigned int dir)
{
    __shared__ unsigned int sk[BLOCK_SIZE*2];
    __shared__ unsigned int sv[BLOCK_SIZE*2];
    int index = blockIdx.x * BLOCK_SIZE*2 + threadIdx.x;

    sk[threadIdx.x] = srcKey[index]; sv[threadIdx.x] = srcVal[index];
    sk[threadIdx.x + BLOCK_SIZE] = srcKey[index + BLOCK_SIZE];  sv[threadIdx.x + BLOCK_SIZE] = srcVal[index + BLOCK_SIZE];

     for (unsigned int size = 2; size < arrayLength; size <<=1 )
     {//битоническое слияние
        unsigned int ddd =  dir ^ ((threadIdx.x&(size / 2)) != 0);
        for(unsigned int stride = size >> 1; stride > 0; stride >>=1)
        {
            __syncthreads ();
            unsigned int pos = 2 * threadIdx.x - (threadIdx.x&(stride -1));
            Comparator(sk[pos], sv[pos], sk[pos+stride], sv[pos+stride], ddd);
        }
      }
      //последний шаг - битоническое слияние
      for (unsigned int stride = arrayLength >> 1; stride > 0; stride >>= 1)
      {
         __syncthreads ();
         unsigned int pos = 2 * threadIdx.x - (threadIdx.x&(stride -1));
         Comparator(sk[pos], sv[pos], sk[pos+stride], sv[pos+stride], dir);
      }
       __syncthreads ();

       dstKey[index] = sk[threadIdx.x]; dstVal[index] =  sv[threadIdx.x];
       dstKey[index + BLOCK_SIZE] = sk[threadIdx.x + BLOCK_SIZE]; dstVal[index + BLOCK_SIZE] =  sv[threadIdx.x + BLOCK_SIZE];
}
```

Поразрядная сортировка (radix sort)

Пусть задан массив из 32-битовых целых чисел: $\{a_0, \ldots, a_{n-1}\}$
Отсортируем этот массив по старшему (31-му) биту, затем по 30-му биту и т.д.

После того, как мы дойдем до 0-го бита и отсортируем по нему, последовательность будет отсортирована

Поскольку бит может принимать только два значения, то сортировка по одному биту заключается в разделении всех элементов на два набора, где:

- соответствующий бит равен нулю;
- соответствующий бит равен единице.

Поразрядная сортировка (radix sort)

Пусть нам надо отсортировать массив по k-му биту.

Тогда рассмотрим массив, где из каждого элемента взят данный бит (b[i]=(a[i] >> k) & 1).

Каждый элемент этого массива равен или нулю или единице. Применим к нему операцию scan, сохранив при этом общую сумму элементов

```
b:0 1 1 0 1 0 0 1 1 0 1
s:0 0 1 2 2 3 3 3 4 5 5, 6
```

Поразрядная сортировка (radix sort)

В результате мы получим сумму всех выбранных бит (т.е.число элементов исходного массива, где в рассматриваемой позиции стоит единичный бит) и массив частичных сумм битов s_n

•Отсюда легко находится количество элементов исходного массива, где в рассматриваемой позиции стоит ноль ($N_z=n-s_n$).

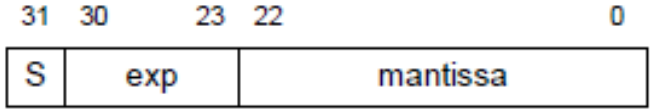
•По этим данным легко посчитать новые позиции для элементов массива:
a_i& bit = 0, тогда a_i -> $i-s_i$
a_i& bit != 0, тогда a_i -> N_z-s_i

Поразрядная сортировка - float

Поразрядная сортировка легко адаптируется для floating point-величин.

Положительные значения можно непосредственно сортировать

Отрицательные значения при поразрядной сортировке будут отсортированы в обратном порядке



$$f = (-1)^S \cdot 2^{\text{exp}-127} \cdot 1.mantissa$$

Поразрядная сортировка - float

Чтобы сортировать значения разных знаков достаточно произвести небольшое преобразование их тип uint, приводимое ниже

```
uint flipFloat ( uint f )
{
	uint mask = -int(f >> 31) | 0x80000000;
	return f ^ mask;
}
uint unflipFloat ( uint f )
{
	uint mask = ((f >> 31) - 1) | 0x80000000;
	return f ^ mask;
}
```

Лабораторная работа № 3

Цель лабораторной работы: исследование параллельной реализации базовых операций над массивами с использованием CUDA.
Общее задание: изучить параллельную реализацию алгоритма, нарисовать схему алгоритма, запустить программу с реализацией базовых операций на GPU и CPU (по вариантам), измерить время выполнения.

Вариант 1: параллельная редукция, построение гистограммы.
Вариант 2: нахождение префиксной суммы, битоническая сортировка.