



Иерархия памяти CUDA. Разделяемая и константная память. Основные алгоритмы и решение СЛАУ на CUDA.

⌘ Лекторы:

☑ Боресков А.В. (ВМиК МГУ)

☑ Харламов А.А. (NVidia)

Типы памяти в CUDA

Тип памяти	Доступ	Уровень выделения	Скорость работы
Регистры	R/W	Per-thread	Высокая(on-chip)
Локальная	R/W	Per-thread	Низкая (DRAM)
Shared	R/W	Per-block	Высокая(on-chip)
Глобальная	R/W	Per-grid	Низкая (DRAM)
Constant	R/O	Per-grid	Высокая(L1 cache)
Texture	R/O	Per-grid	Высокая(L1 cache)

Типы памяти в CUDA



- ⌘ Самая быстрая – *shared* (on-chip)
- ⌘ Самая медленная – глобальная (DRAM)
- ⌘ Для ряда случаев можно использовать кэшируемую константную и текстурную память
- ⌘ Доступ к памяти в CUDA идет отдельно для каждой половины warp'a (*half-warp*)

Работа с shared-памятью



- ⌘ Самая быстрая (*on-chip*)
- ⌘ Сейчас всего 16 Кбайт на один мультипроцессор
- ⌘ Совместно используется всеми нитями блока
- ⌘ Отдельное обращение для каждой половины *warp'a* (*half-warp*)
- ⌘ Как правило, требует явной синхронизации

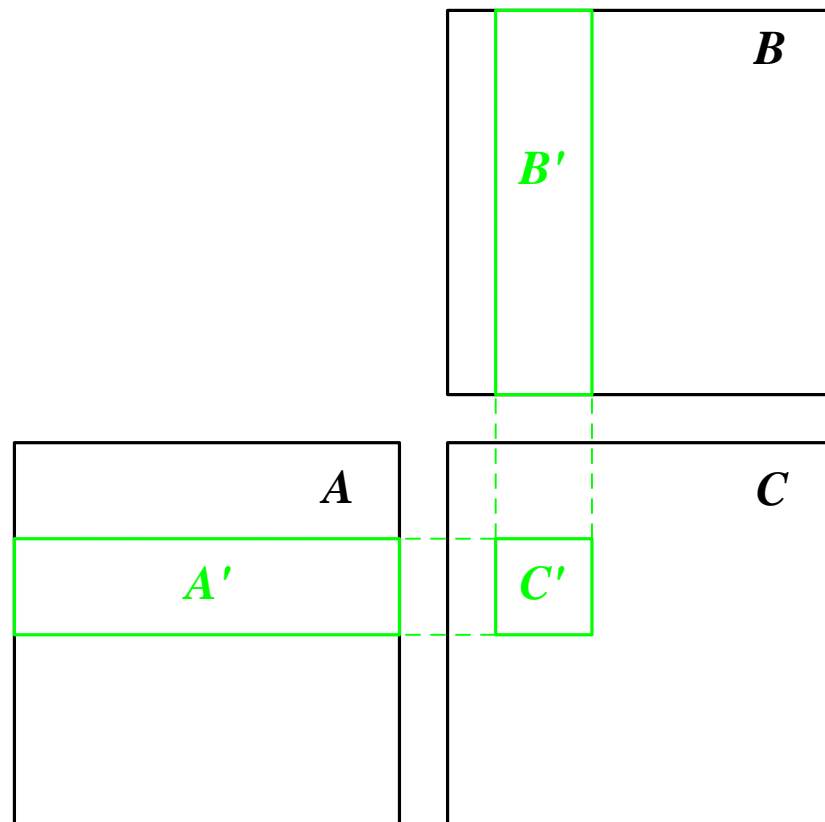
Типичное использование



1. Загрузить необходимые данные в *shared*-память (из глобальной)
2. `__syncthreads ()`
3. Выполнить вычисления над загруженными данными
4. `__syncthreads ()`
5. Записать результат в глобальную память

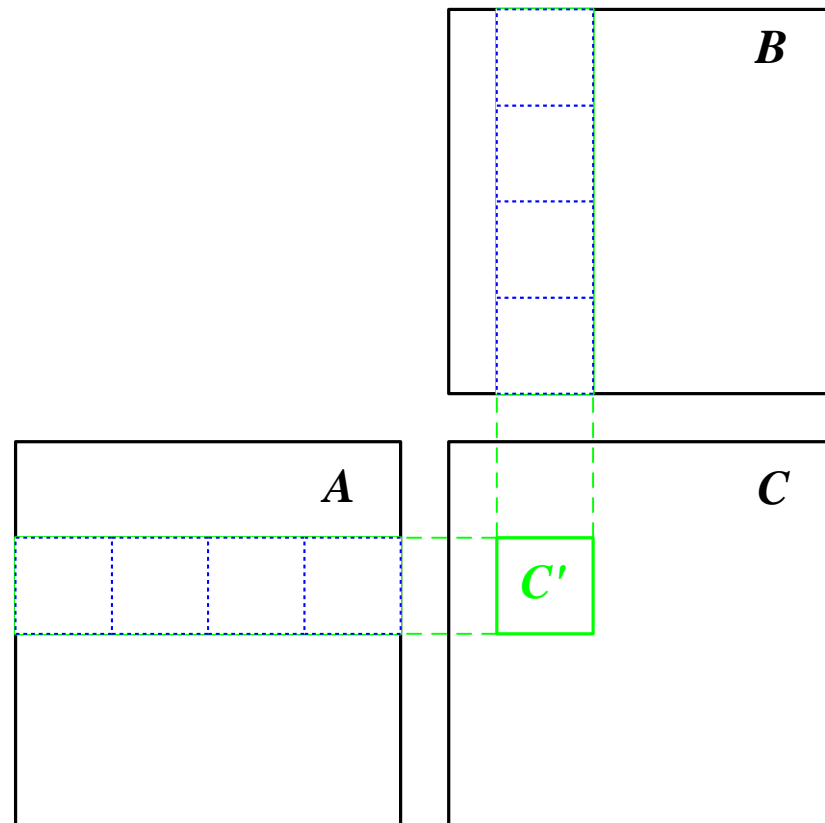
Умножение матриц (2)

- ⌘ При вычислении C' постоянно используются одни и те же элементы из A и B
 - ⌘ По много раз считываются из глобальной памяти
- ⌘ Эти многократно используемые элементы формируют полосы в матрицах A и B
- ⌘ Размер такой полосы $N*16$ и для реальных задач даже одна такая полоса не помещается в *shared*-память



Умножение матриц (2)

- ⌘ «Разделяй и властвуй»
- ⌘ Разбиваем каждую полосу на квадратные матрицы (16×16)
- ⌘ Тогда требуемая подматрица произведения C' может быть представлена как сумма произведений таких матриц 16×16
- ⌘ Для работы нужно только две матрицы 16×16 в *shared*-памяти



$$C' = A'_1 * B'_1 + \dots + A'_{N/16} * B'_{N/16}$$

Эффективная реализация

```
__global__ void matMult ( float * a, float * b, int n, float * c ) {
    int bx      = blockIdx.x,  by = blockIdx.y;
    int tx      = threadIdx.x, ty = threadIdx.y;
    int aBegin  = n * BLOCK_SIZE * by;
    int aEnd    = aBegin + n - 1;
    int bBegin  = BLOCK_SIZE * bx;
    int aStep   = BLOCK_SIZE, bStep  = BLOCK_SIZE * n;
    float sum   = 0.0f;
    for ( int ia = aBegin, ib = bBegin; ia <= aEnd; ia += aStep, ib += bStep ){
        __shared__ float as [BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float bs [BLOCK_SIZE][BLOCK_SIZE];
        as [ty][tx] = a [ia + n * ty + tx];
        bs [ty][tx] = b [ib + n * ty + tx];
        __syncthreads ();          // Synchronize to make sure the matrices are loaded
        for ( int k = 0; k < BLOCK_SIZE; k++ )
            sum += as [ty][k] * bs [k][tx];
        __syncthreads ();          // Synchronize to make sure submatrices not needed
    }
    c [n * BLOCK_SIZE * by + BLOCK_SIZE * bx + n * ty + tx] = sum;
}
```


Эффективная реализация

- ⌘ На каждый элемент

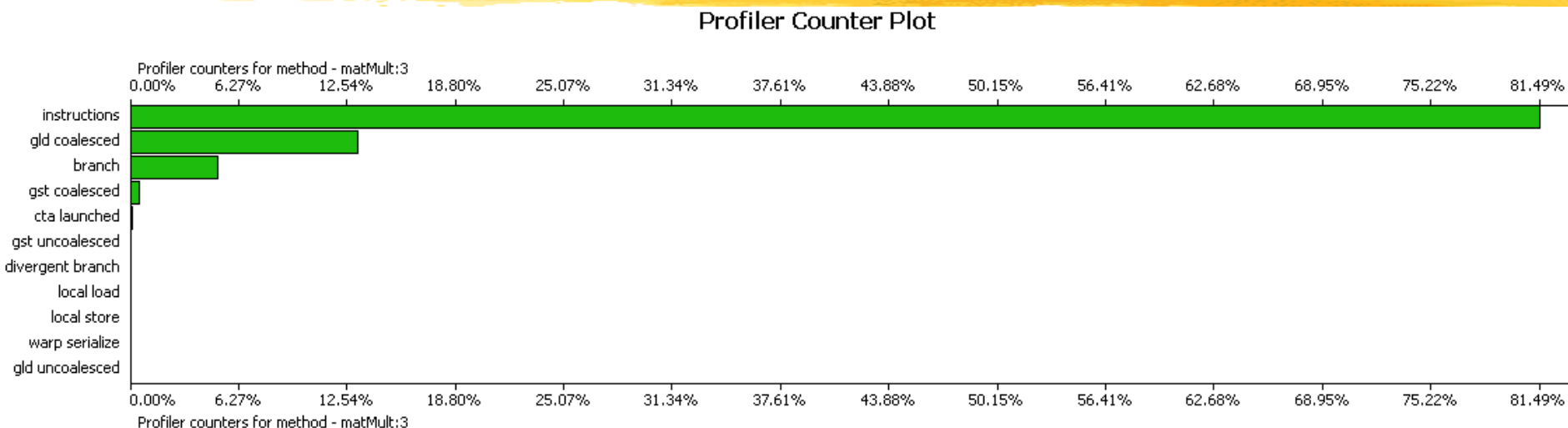
 - ⌘ $2 \cdot N$ арифметических операций

 - ⌘ $2 \cdot N / 16$ обращений к глобальной памяти

- ⌘ Поскольку разные *warp*'ы могут выполнять разные команды нужна явная синхронизация всех нитей блока

- ⌘ Быстродействие выросло более чем на порядок (2578 vs 132 миллисекунд)

Эффективная реализация



⌘ Теперь основное время (81.49%) ушло на вычисления

⌘ Чтение из памяти стало *coalesced* и заняло всего 12.5 %

Эффективная работа с *shared*-памятью



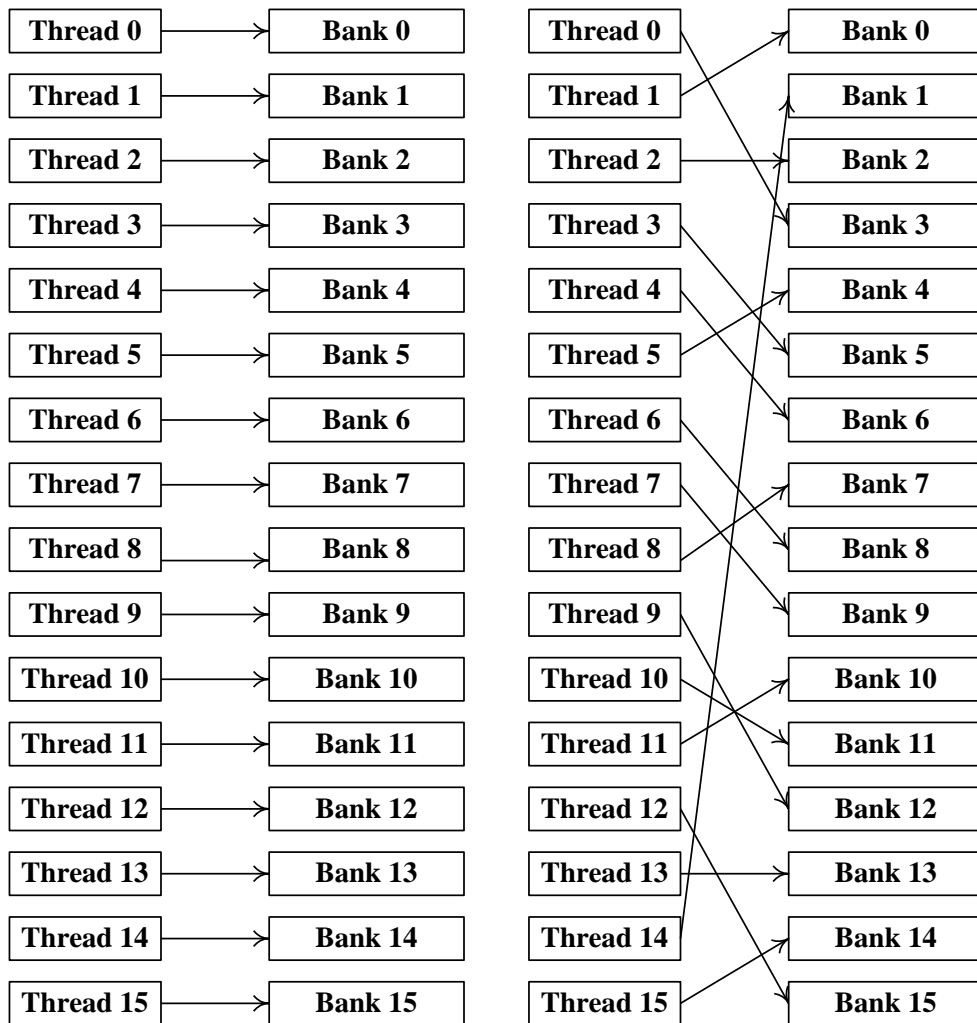
- ⌘ Для повышения пропускной способности вся *shared*-память разбита на 16 банков
- ⌘ Каждый банк работает независимо от других
- ⌘ Можно одновременно выполнить до 16 обращений к *shared*-памяти
- ⌘ Если идет несколько обращений к одному банку, то они выполняются по очереди

Эффективная работа с **shared-памятью**

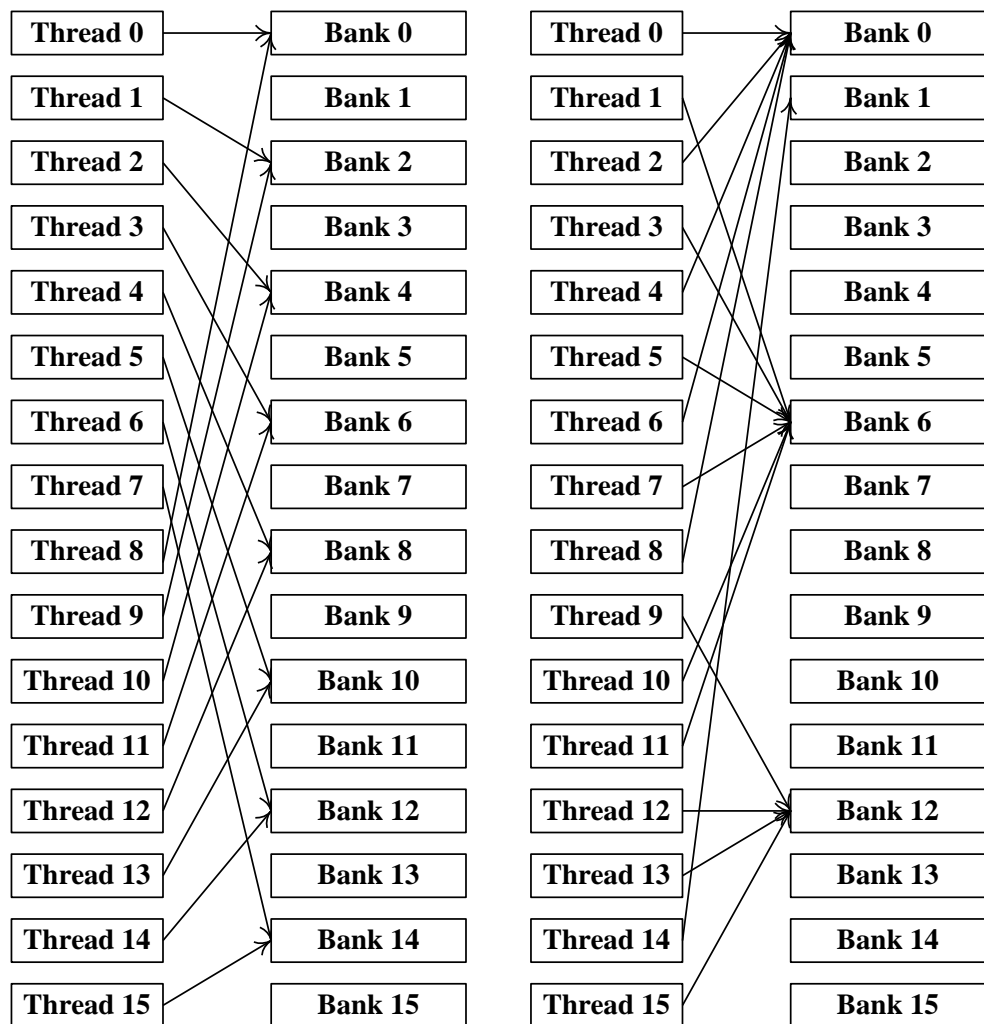


- ⌘ Банки строятся из 32-битовых слов
- ⌘ Подряд идущие 32-битовые слова попадают в подряд идущие банки
- ⌘ ***Bank conflict*** – несколько нитей из одного *half-warp* обращаются к одному и тому же банку
- ⌘ Конфликта не происходит если все 16 нитей обращаются к одному слову (*broadcast*)

Бесконфликтные паттерны доступа



Паттерны с конфликтами банков



⌘ Слева – конфликт второго порядка – вдвое меньшая скорость

⌘ Справа - несколько конфликтов, до 6-го порядка

Доступ к массиву элементов



```
__shared__ float a [N];
```

Нет конфликтов

```
float x = a [base + threadIdx.x];
```

```
__shared__ short a [N];
```

Конфликты 2-го
порядка

```
short x = a [base + threadIdx.x];
```

```
__shared__ char a [N];
```

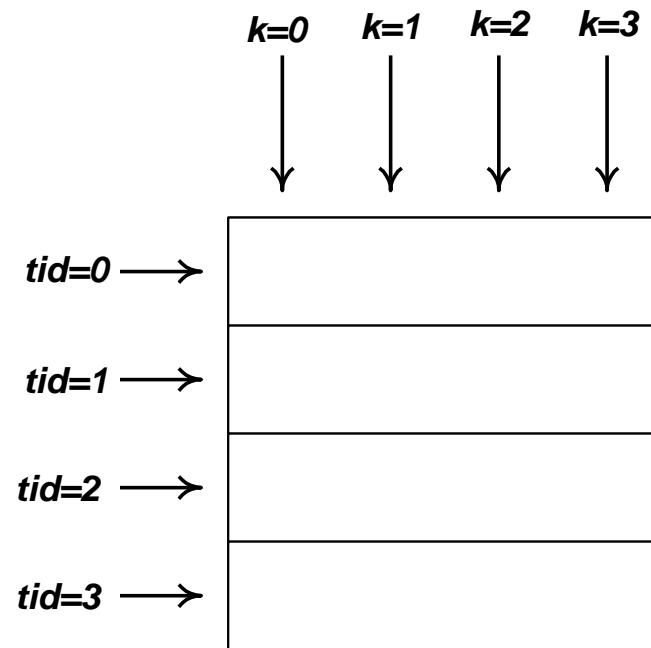
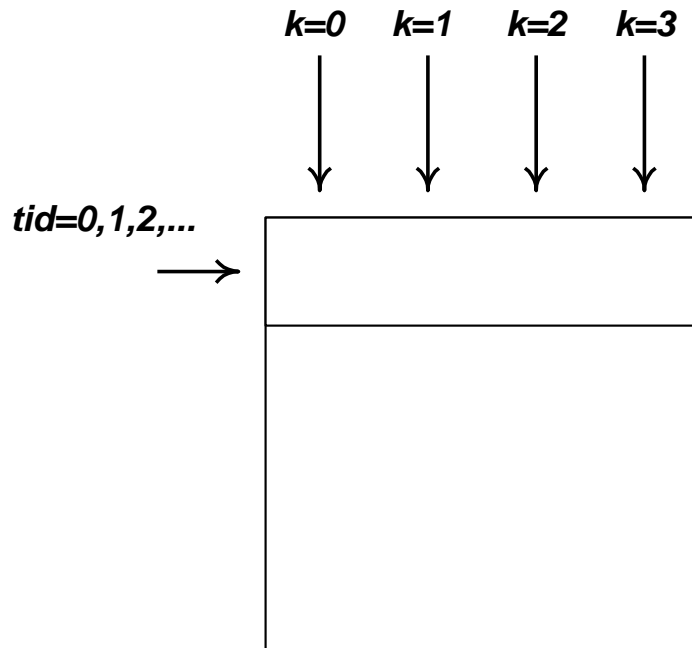
Конфликты 4-го
порядка

```
char x = a [base + threadIdx.x];
```

Пример – матрицы 16×16

- ⌘ Перемножение $A * B^T$ двух матриц 16×16 , расположенных в *shared*-памяти
 - ☐ Доступ к обеим матрицам идет по строкам
 - ☐ Все элементы строки распределены равномерно по 16 банкам
 - ☐ Каждый столбец целиком попадает в один банк
- ⌘ Получаем конфликт 16-го порядка при чтении матрицы B

Пример – матрицы 16×16



Пример – матрицы 16×16

- ⌘ **Дополним каждую строку одним элементом**
- ⌘ Все элементы строки (кроме последнего) лежат в разных банках
- ⌘ **Все элементы столбца также лежат в разных банках**
- ⌘ **Фактически за счет небольшого увеличения объема памяти полностью избавились от конфликтов**

[illegible]

Константная память

```
__constant__ float contsData [256];  
float          hostData  [256];
```

```
cudaMemcpyToSymbol ( constData, hostData, sizeof ( data ), 0,  
                    cudaMemcpyHostToDevice );
```

```
////////////////////////////////////
```

```
template <class T>  
cudaError_t cudaMemcpyToSymbol    ( const T& symbol, const void * src,  
                                   size_t count, size_t offset, enum cudaMemcpyKind kind );
```

```
template <class T>  
cudaError_t cudaMemcpyFromSymbol ( void * dst, const T& symbol,  
                                   size_t count, size_t offset, enum cudaMemcpyKind kind );
```

Базовые алгоритмы на CUDA



Основные алгоритмы:

⌘ Reduce

⌘ Scan (prefix sum)

⌘ Histogram

⌘ Sort

Параллельная редукция (reduce)

⌘ Дано:

☐ Массив элементов a_0, a_1, \dots, a_{n-1}

☐ Бинарная ассоциативная операция '+'

⌘ Необходимо найти

☐ A , $A = a_0 + a_1 + \dots + a_{n-1}$

⌘ Лимитирующий фактор – доступ к памяти

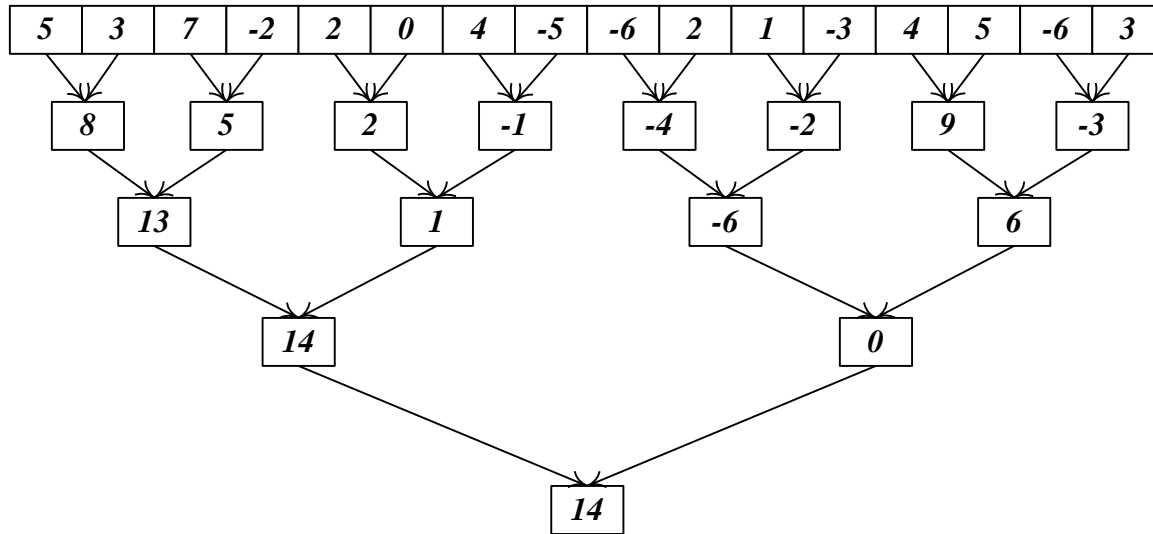
⌘ В качестве операции также может быть *min|max*

Реализация параллельной редукции



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

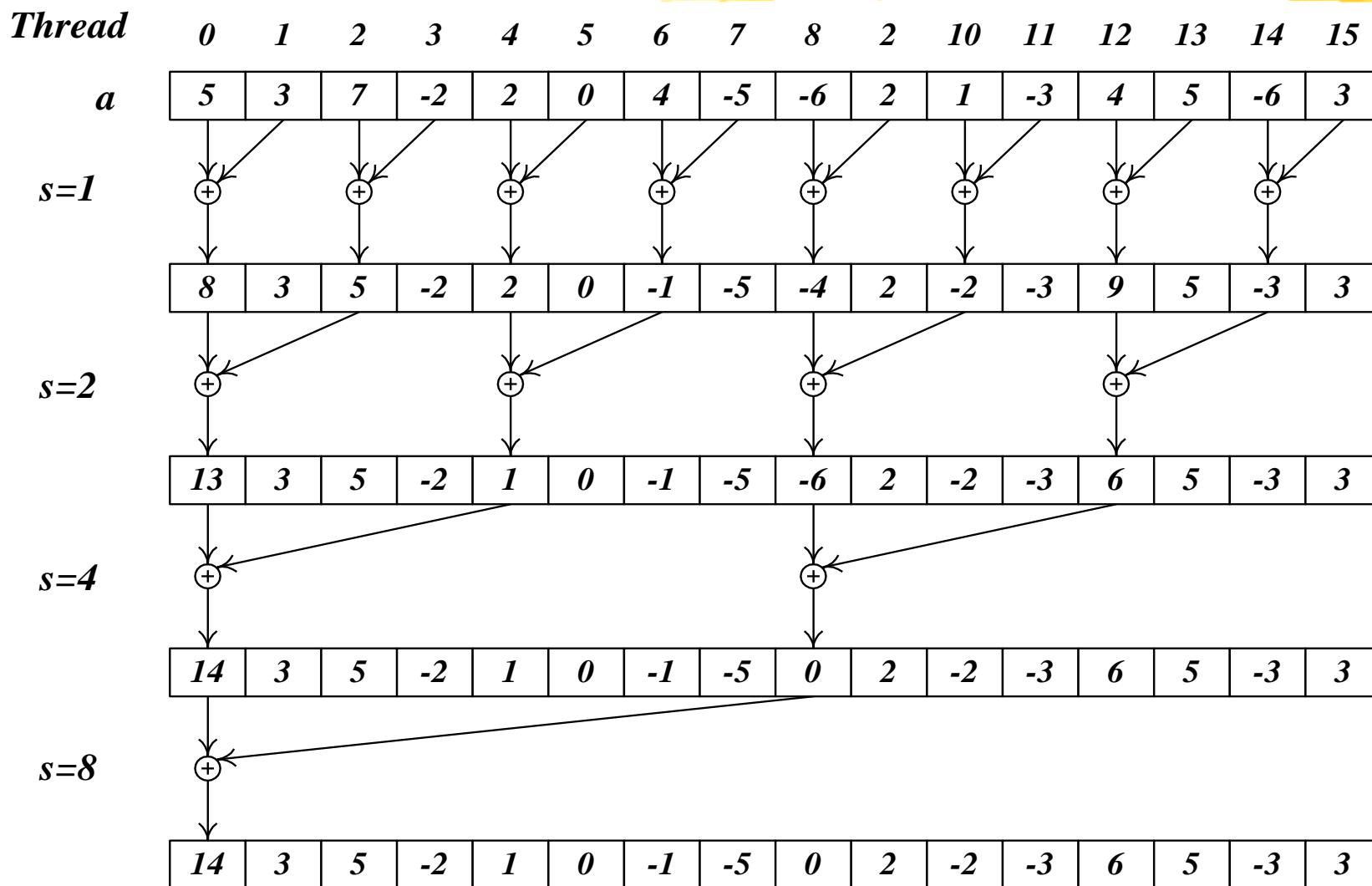
Иерархическое суммирование



⌘ Позволяет проводить суммирование параллельно, используя много нитей

⌘ Требуется $\log(N)$ шагов

Редукция, вариант 1

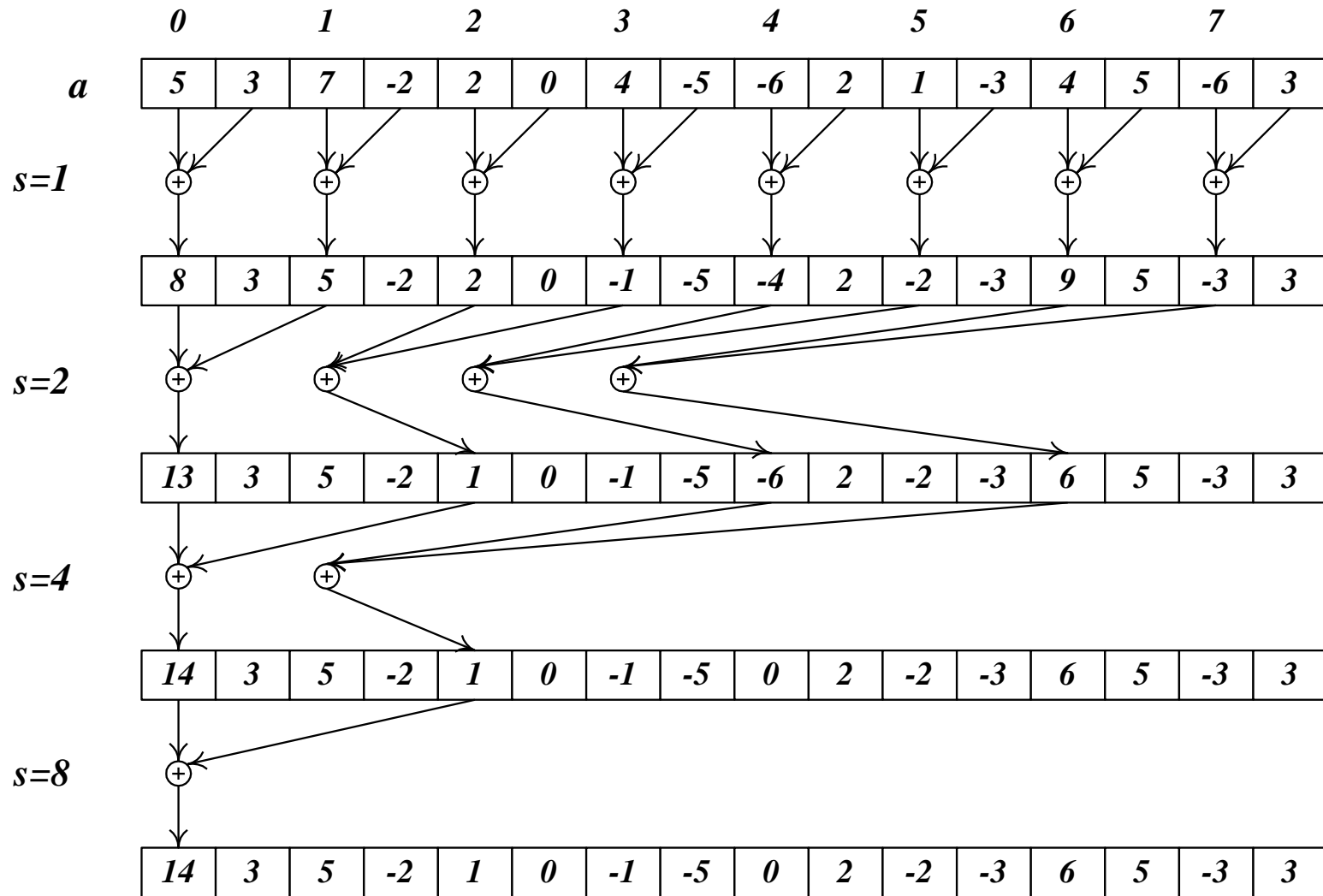


Редукция, вариант 1

```
__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, вариант 2



Редукция, вариант 2

```
__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; // better replace with >>
        if ( index < blockDim.x )
            data [index] += data [index + s];
        __syncthreads ();
    }
    if ( tid == 0 )                // write result of block reduction
        outData [blockIdx.x] = data [0];
}
```

Редукция, вариант 2



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

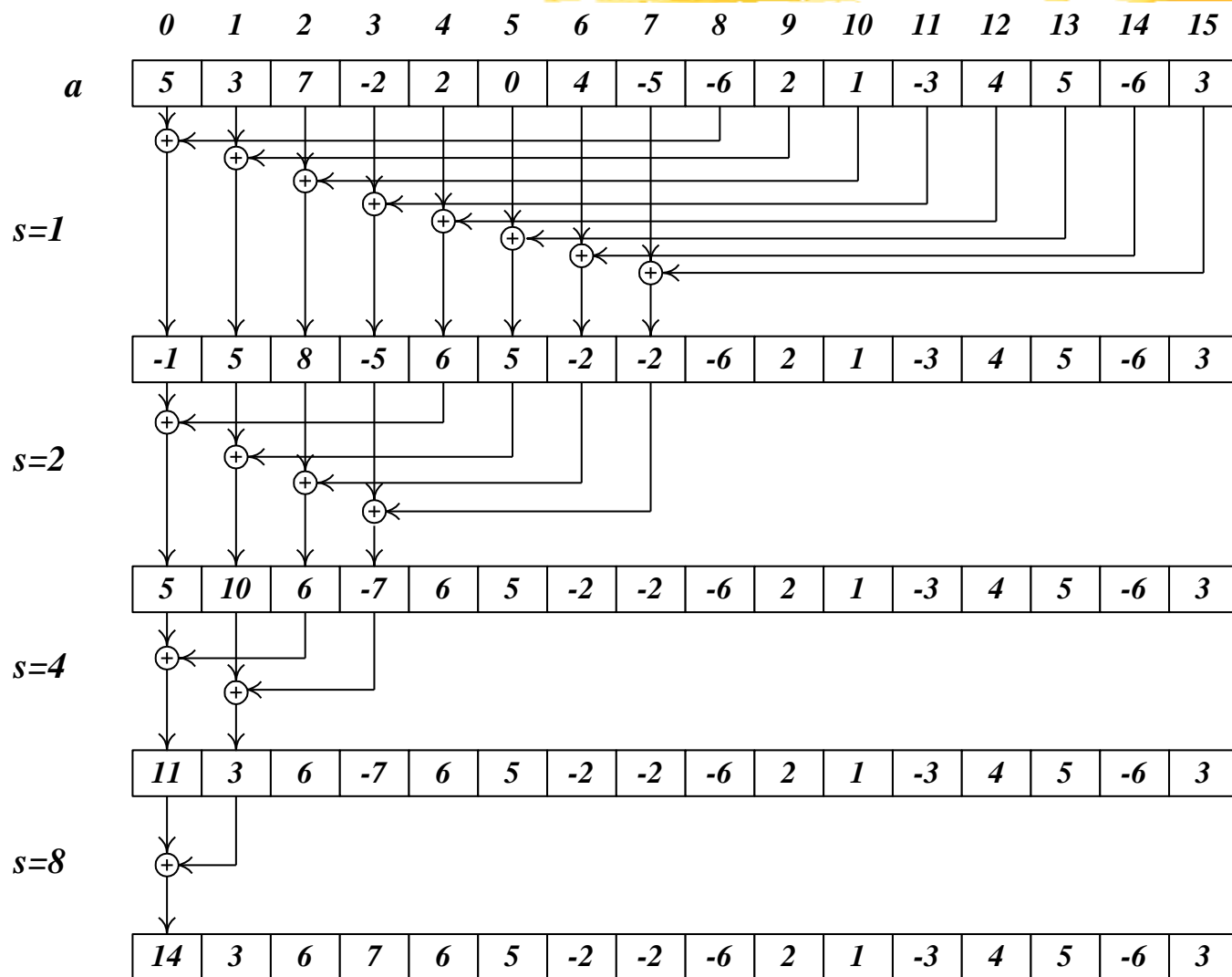
Редукция, вариант 3

⌘ Изменим порядок суммирования

☑ Раньше суммирование начиналось с соседних элементов и расстояние увеличивалось вдвое

☑ Начнем суммирование с наиболее удаленных (на $\dim Block.x/2$) и расстояние будем уменьшать вдвое на каждой итерации цикла

Редукция, вариант 3



Редукция, вариант 3

```
__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, вариант 3



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

Редукция, вариант 4

```
__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, вариант 5

- ⌘ При $s \leq 32$ 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 $s \leq 32$

Редукция, вариант 5

```
. . . . .  
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, быстроедействие

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

Parallel Prefix Sum (Scan)

Имеется входной массив $\{a_0, a_1, \dots, a_{n-1}\}$
и бинарная операция

По нему строится массив
следующего вида

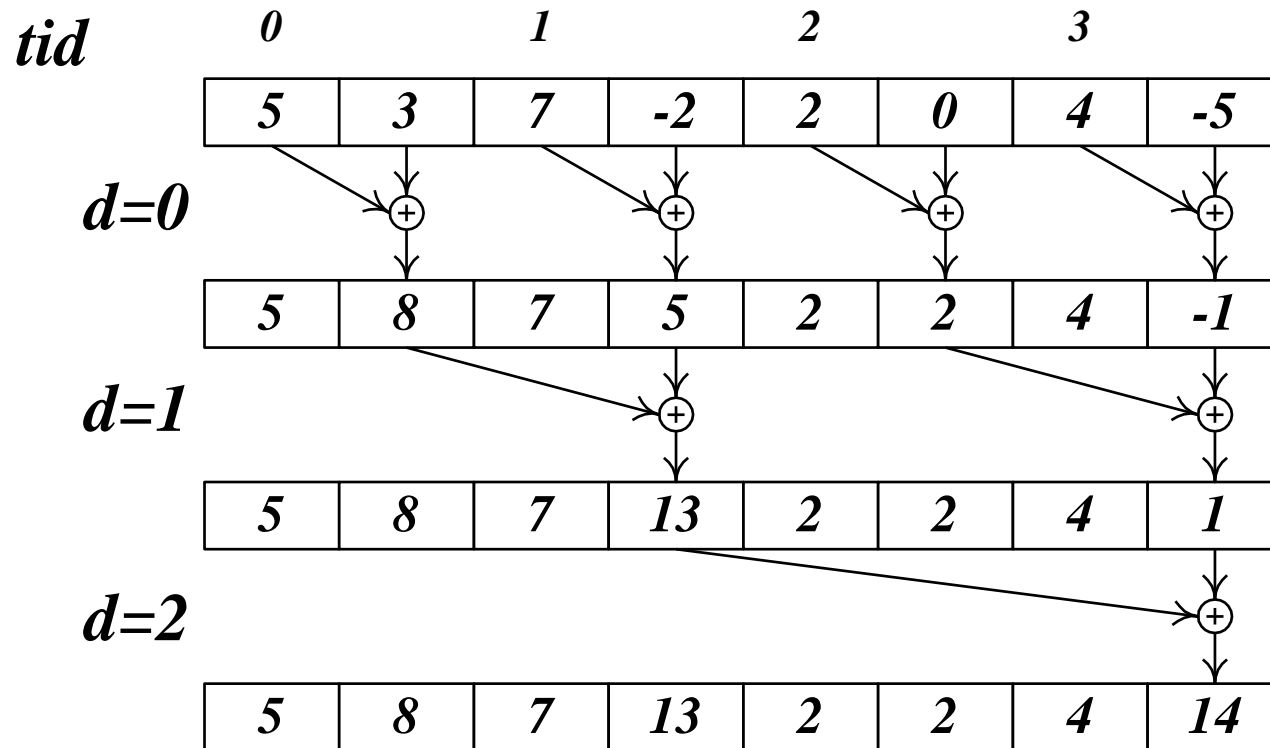
$$\{I, a_0, a_0 \oplus a_1, a_0 \oplus a_1 \oplus a_2, \dots, a_0 \oplus \dots \oplus a_{n-2}\}$$

Parallel Prefix Sum (Scan)



- ⌘ Очень легко делается последовательно
- ⌘ Для распараллеливания используем *sum tree*
- ⌘ Выполняется в два этапа
 - ⌘ Строим *sum tree*
 - ⌘ По *sum tree* получаем результат

Построение sum tree



Построение sum tree



1. Используем одну нить на 2 элемента массива
2. Загружаем данные
3. `__syncthreads ()`
4. 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;

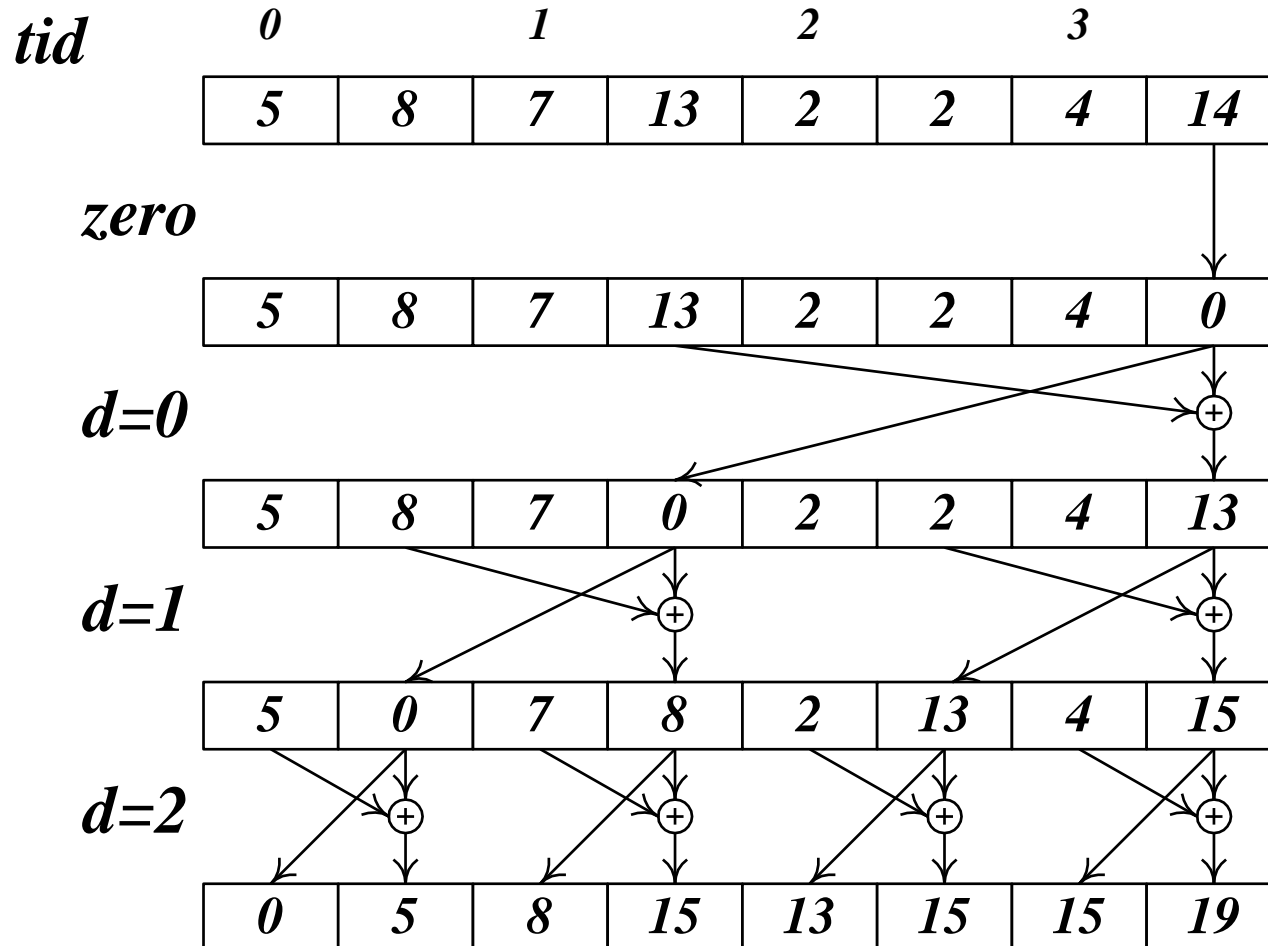
    temp [tid] = inData [tid]; // load into shared memory
    temp [tid+BLOCK_SIZE] = inData [tid+BLOCK_SIZE];

    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



Получение результата по sum tree



1. Одна нить на 2 элемента массива
2. Обнуляем последний элемент
3. Copy and increment
4. 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
- ☒ Branching -> small
- ☒ Конфликты банков -> **конфликты до 16 порядка !**

Scan - 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 CONFLICT_FREE_OFFS(i) ((i) >> LOG_NUM_BANKS)
```

Scan - Оптимизация

```
__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 ){
    int      i = n - 1 + CONFLICT_FREE_OFFS(n-1);
    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

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

Scan больших массивов

```
void scan ( float * inData, float * outData, int n )
{
    int numBlocks = n / (2*BLOCK_SIZE);
    float * sums, * sums2;

    if ( numBlocks < 1 ) numBlocks = 1;

    // allocate sums array
    cudaMalloc ( (void**)&sums, numBlocks * sizeof ( float ) );
    cudaMalloc ( (void**)&sums2, numBlocks * sizeof ( float ) );

    dim3 threads ( BLOCK_SIZE, 1, 1 ), blocks ( numBlocks, 1, 1 );
    scan3<<<blocks, threads>>> ( inData, outData, sums, 2*BLOCK_SIZE );

    if ( n >= 2*BLOCK_SIZE )
        scan ( sums, sums2, numBlocks );
    else
        cudaMemcpy ( sums2, sums, numBlocks*sizeof(float),
                    cudaMemcpyDeviceToDevice );

    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



- ☒ Дан массив элементов и способ классификации элементов: каждому элементу сопоставляется один из k классов.
- ☒ Задача – по массиву получить для каждого класса число элементов, попадающих в него.
- ☒ Полученная таблица частот для классов и является гистограммой

Построение гистограммы

- ⌘ Очень легко реализуется последовательным кодом
- ⌘ Если мы выделяем по одной нити на каждый входной элемент, то нужна операция *atomicIncr*
- ⌘ Очень частые обращения к счетчикам, лучше всего их разместить в *shared*-памяти
- ⌘ Идеальный случай – у каждой нити своя таблица счетчиков в *shared*-памяти

Построение гистограммы для 64 классов (bins)



- ⏏ На каждый счетчик отводим 1 байт
- ⏏ Всего гистограмма – 64 байта (на одну нить)
- ⏏ Всего в разделяемой памяти SM можно разместить 256 таких гистограмм
- ⏏ Размер блока – 64 нити, максимум 4 блока на SM
- ⏏ Каждая нить может обработать не более 255 байт

Построение гистограммы для 64 классов (bins)

Посмотрим на конфликты банков:

⏏ $64 * tid + value$

⏏ $bank = ((64 * tid + value) / 4) \& 0xF = (value \gg 2) \& 0xF$

⏏ Номер банка полностью определяется входными данными, если есть много повторений, то будет много конфликтов по банкам

⏏ $64 * value + tid$

⏏ $bank = ((64 * value + tid) / 4) \& 0xF = (tid \gg 2) \& 0xF$

⏏ Номер банка определяется номером нити

Построение гистограммы для 64 классов (bins)

- ☑ В первом случае все определяется входными данными, очень высока вероятность конфликта банков вплоть до 16-го порядка.
- ☑ Во втором случае номер банка определяется старшими битами номера нити и мы получаем постоянный конфликт четвертого порядка
- ☑ Но зачем в качестве *tid* использовать именно номер нити в блоке – подойдет любое значение, получаемое из номера нити путем фиксированной перестановки битов

Построение гистограммы для 64 классов (bins)

- ☒ Номер банка определяется битами 2..5 величины *tid*.
- ☒ Построим *tid* как следующую перестановку битов номера нити в блоке:
 - ☒ `tid=(threadIdx.x>>4) | ((threadIdx.x & 0xF)<<2)`
- ☒ Легко убедиться, что в этом случае конфликта банков не будет вообще

Построение гистограммы для 64 классов (bins)

```
inline __device__ void addByte ( uchar * base, uint data )
{
    base[64*data]++;
}

inline __device__ void addWord ( uchar * base, uint data )
{
    addByte ( base, (data >> 2) & 0x3FU );
    addByte ( base, (data >> 10) & 0x3FU );
    addByte ( base, (data >> 18) & 0x3FU );
    addByte ( base, (data >> 26) & 0x3FU );
}

__global__ void histogram64Kernel ( uint * partialHist, uint * data, uint dataCount )
{
    __shared__ uchar hist [64*64];
    int tid = (threadIdx.x >> 4) | ((threadIdx.x & 0x0F) << 2);
    uchar * base = hist + tid;

    for ( int i = 0; i < 64 / 4; i++ )
        ((uint *)hist)[threadIdx.x + i * 64] = 0;

    __syncthreads ();

    for ( uint pos = blockIdx.x*blockDim.x + threadIdx.x; pos < dataCount;
          pos += blockDim.x*gridDim.x )
        addWord ( base, data [pos] );
}
```

Построение гистограммы

⌘ Более общий случай—

⌘ Просто не хватит *shared*-памяти давать каждой нити по своей гистограмме

⌘ Давайте выделять по своей таблице счетчиков на определенный набор нитей

⌘ (+) Уменьшаются затраты на *shared*-память

⌘ (-) Появляется проблема синхронизации с записью нитей этого набора

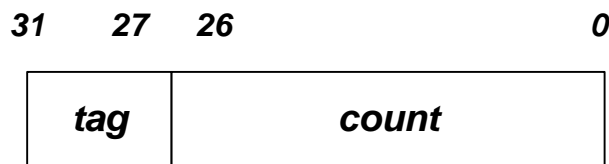
Построение гистограммы

Когда проще всего обеспечивать атомарность записи:

- ⌘ Когда каждый такой набор нитей всегда лежит в пределах одного *warp'a*
- ⌘ По-прежнему сохраняется риск нескольких нитей, одновременно увеличивающих один и тот же элемент гистограммы, но с этим можно бороться
- ⌘ Если несколько нитей одновременно делают запись по одному адресу, то только одна из этих записей проходит

Построение гистограммы

- ⌘ Пусть каждый warp нитей имеет свою таблицу счетчиков
 - ⌘ 192 нити в блоке дают 6 *warp*'ов, т.е. $6 \cdot 256 \cdot 4 = 6\text{K}6$ *shared*-памяти на блок
 - ⌘ 5 старших битов каждого счетчика будут хранить номер нити (внутри *warp*'а), сделавшей последнюю запись



Построение гистограммы

```
__device__ inline void addData256 ( volatile unsigned * warpHist, unsigned data,
                                   unsigned threadTag )
{
    unsigned count;

    do
    {
        count = warpHist [data] & 0x07FFFFFFU;    // mask thread tag bits
        count = threadTag | (count + 1);           // increment count and tag it
        warpHist [data] = count;
    } while ( warpHist [data] != count );           // check whether we've modified value
}
```

⌘ Каждая нить строит новое значение

⌘ Увеличить на единицу

⌘ Выставить старшие биты в номер нити в *warp*

⌘ Как минимум одна запись пройдет и соответствующая нить выйдет из цикла

Построение гистограммы



- ⌘ Каждая нить меняет свой элемент таблицы
 - ⌘ Сразу же выходим, никаких расходов
- ⌘ Две нити пытаются увеличить один и тот же счетчик
 - ⌘ Одной это получится (запишется ее значение)
 - ⌘ Другой нет – ее значение будет отброшено
- ⌘ Та нить, которая записала выходит из цикла и оставшаяся нить со делает запись (со второй попытки)

Построение гистограммы

```
#define WARP_LOG2SIZE 5 // bits to identify warp
#define WARP_N 6 // warps per block
__global__ void histogramKernel ( unsigned * result, unsigned * data, int n )
{
    int      globalTid  = blockIdx.x * blockDim.x + threadIdx.x;
    int      numThreads = blockDim.x * gridDim.x;
    int      warpBase   = (threadIdx.x >> WARP_LOG2SIZE) * BIN_COUNT;
    unsigned threadTag   = threadIdx.x << (32 - WARP_LOG2SIZE);

    volatile __shared__ unsigned hist [BLOCK_MEMORY];

    for ( int i = threadIdx.x; i < BLOCK_MEMORY; i += blockDim.x )
        hist [i] = 0;

    __syncthreads ();
    for ( int i = globalTid; i < n; i += numThreads ) {
        unsigned data4 = data [i];

        addData256 ( hist + warpBase, (data4 >> 0) & 0xFFU, threadTag );
        addData256 ( hist + warpBase, (data4 >> 8) & 0xFFU, threadTag );
        addData256 ( hist + warpBase, (data4 >> 16) & 0xFFU, threadTag );
        addData256 ( hist + warpBase, (data4 >> 24) & 0xFFU, threadTag );
    }
    __syncthreads();

    for ( int i = threadIdx.x; i < BIN_COUNT; i += blockDim.x ){
        unsigned sum = 0;

        for ( int base = 0; base < BLOCK_MEMORY; base += BIN_COUNT )
            sum += hist [base + i] & 0x07FFFFFFU;

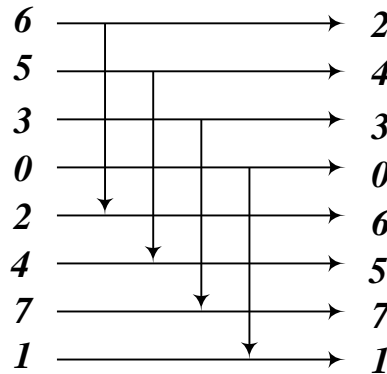
        result[blockIdx.x * BIN_COUNT + i] = sum;
    }
}
```

Сортировка.

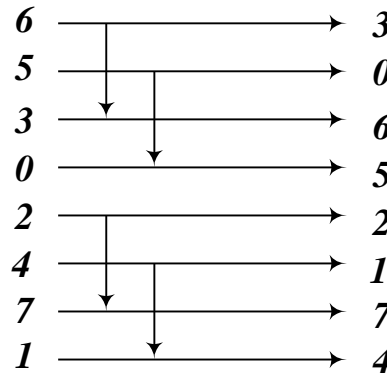
Битоническая сортировка

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

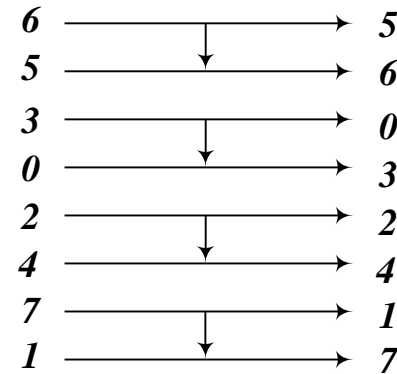
$$B_n : (x_k, x_{k+n/2}) \rightarrow (\min, \max)$$



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 полученной 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 $B_n, B_{n/2}, \dots, B_8, B_4, B_2$ то в результате мы получим отсортированную последовательность (битоническое слияние)!

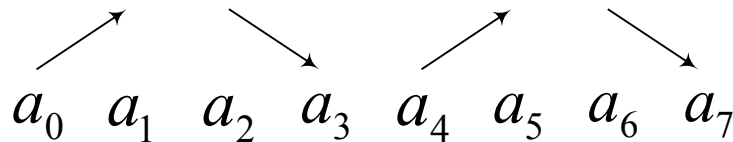
Если у нас произвольная последовательность:

- ⌘ Применим B_2 с чередующимся порядком, в результате каждые 4 подряд идущих элемента будут образовывать битоническую последовательность
- ⌘ При помощи битонического слияния отсортируем каждую пару последовательностей из 4-элементов, но с чередующимся порядком упорядочивания.
- ⌘ При помощи битонического слияния отсортируем каждую пару из 8 элементов

Битоническая сортировка

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

Применим к каждой такой четверке элементов полуочиститель B_4 с чередующимся порядком сортировки.

Тогда каждые восемь элементов будет образовывать битоническую последовательность.

Применим к каждому 8 элементам полуочиститель B_4 с чередующимся порядком сортировки и так далее.

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

Битоническая сортировка



- ⌘ Очень хорошо работает для сортировки через шейдеры (steps3d.narod.ru/tutorials/gpu-sort-tutorial.html)
- ⌘ Плохо использует возможности CUDA, поэтому обычно не используется для сортировки больших массивов

Поразрядная сортировка (radix sort)



Пусть задан массив из 32-
битовых целых чисел:

$$\{a_0, a_1, \dots, a_{n-1}\}$$

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

Поразрядная сортировка



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

Пусть нам надо отсортировать массив по k -му биту. Тогда рассмотрим массив, где из каждого элемента взят данный бит ($b[i] = (a[i] \gg k) \& 1$).

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

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 (Nz).

По этим данным легко посчитать новые позиции для элементов массива:

Поразрядная сортировка

По этим данным легко посчитать новые позиции для элементов массива:

$$a_i \& bit = 0 \Rightarrow a_i \rightarrow i - s_i$$

$$a_i \& bit \neq 0 \Rightarrow a_i \rightarrow N_z + s_i$$

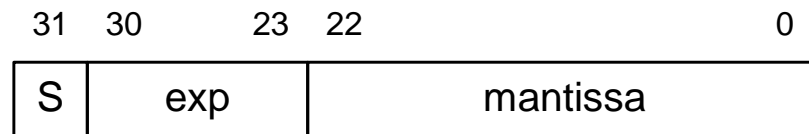
$$bit = 1 \ll k$$

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



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

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;
}
```

Решение системы линейных алгебраических уравнений

$$Ax=f,$$

A – матрица размера $N \times N$,

f – вектор размера N

⌘ Традиционные методы ориентированы на последовательное вычисление элементов и нам не подходят

⌘ Есть еще итеративные методы

Итеративные методы

$$x^{k+1} - x^k = \alpha \cdot (A \cdot x^k - f)$$

- ⌘ Эффективны когда
 - ⌘ Матрица A сильно разрежена
 - ⌘ Параллельные вычисления
- ⌘ В обоих случаях цена (по времени) одной итерации $O(N)$

Сходимость

$$Ax^* = f,$$

$$d^{k+1} = x^{k+1} - x^*,$$

$$d^{k+1} = \alpha \cdot Ad^k,$$

$$\|d^{k+1}\| \leq |\alpha| \cdot \|A\| \cdot \|d^k\|,$$

$$|\alpha| \cdot \|A\| < 1$$

- ⌘ Если есть сходимость, то только к решению системы
- ⌘ Записав уравнения для погрешности получаем достаточное условие сходимости
- ⌘ За счет выбора достаточно малого значения параметра получаем сходимость

Код на CUDA



```
//  
// one iteration  
//  
__global__ void kernel ( float * a, float * f, float alpha,  
                        float * x0, float * x1, int n )  
{  
    int    idx = blockIdx.x * blockDim.x + threadIdx.x;  
    int    ia  = n * idx;  
    float  sum = 0.0f;  
  
    for ( int i = 0; i < n; i++ )  
        sum += a [ia + i] * x0 [i];  
  
    x1 [idx] = x0 [idx] + alpha * (sum - f [idx] );  
}
```

Ресурсы нашего курса



⌘ [CUDA.CS.MSU.SU](https://cuda.cs.msu.su)

- ☑ Место для вопросов и дискуссий
- ☑ Место для материалов нашего курса
- ☑ Место для ваших статей!
 - ☒ Если вы нашли какой-то интересный подход!
 - ☒ Или исследовали производительность разных подходов и знаете, какой из них самый быстрый!
 - ☒ Или знаете способы сделать работу с CUDA проще!

⌘ www.steps3d.narod.ru

⌘ www.nvidia.ru

Вопросы

