

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

## Лекторы:

[Боресков А.В. \(ВМиК МГУ\)](#)

[Харламов А.А. \(NVidia\)](#)

# План


- Разделяемая память
- Константная память
- Базовые алгоритмы

# План

- **Разделяемая память**
- Константная память
- Базовые алгоритмы

# Типы памяти в CUDA

Тип памяти	Доступ	Уровень выделения	Скорость работы	Расположение
Регистры	R/W	Per-thread	Высокая	SM
Локальная	R/W	Per-thread	Низкая	DRAM
<b>Shared</b>	<b>R/W</b>	<b>Per-block</b>	<b>Высокая</b>	<b>SM</b>
Глобальная	R/W	Per-grid	Низкая	DRAM
Constant	R/O	Per-grid	Высокая	DRAM
Texture	R/O	Per-grid	Высокая	DRAM

Легенда:  
-интерфейсы  
доступа 

# Типы памяти в CUDA

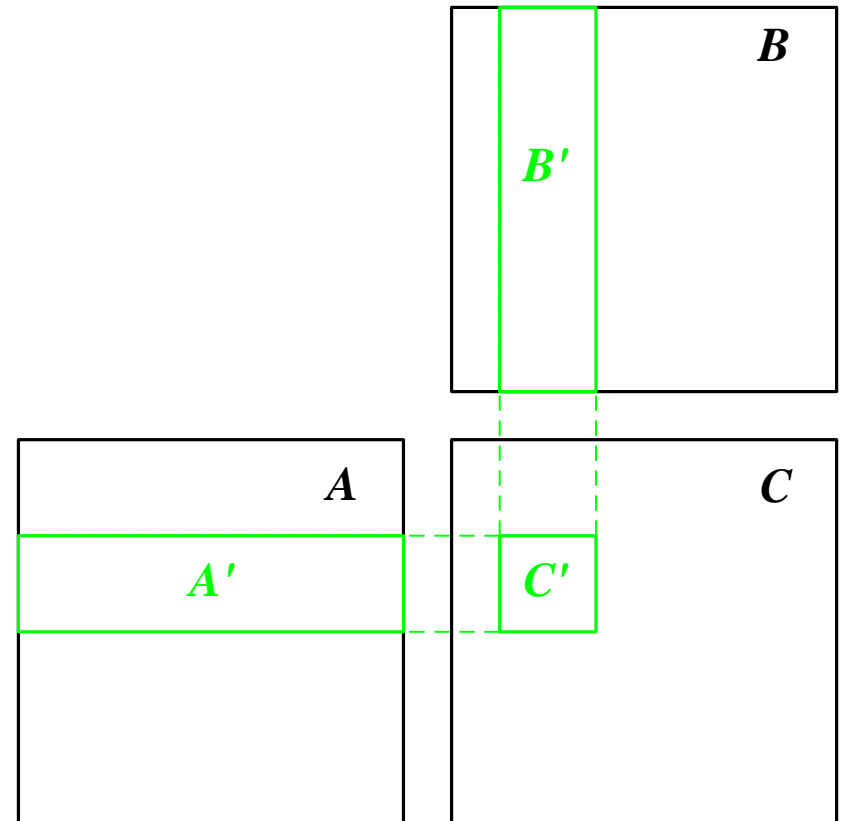
- Самая быстрая - *shared* (on-chip)
- Сейчас всего 16 Кбайт на один мультипроцессор Tesla 10
- Shared memory и кэш объединены на Tesla 20
  - Можно контролировать соотношение 16 | 48 Кб
- Совместно используется всеми нитями блока
- Как правило, требует явной синхронизации

# Типичное использование

- Загрузить необходимые данные в *shared*-память (из глобальной)
- `__syncthreads ()`
- Выполнить вычисления над загруженными данными
- `__syncthreads ()`
- Записать результат в глобальную память

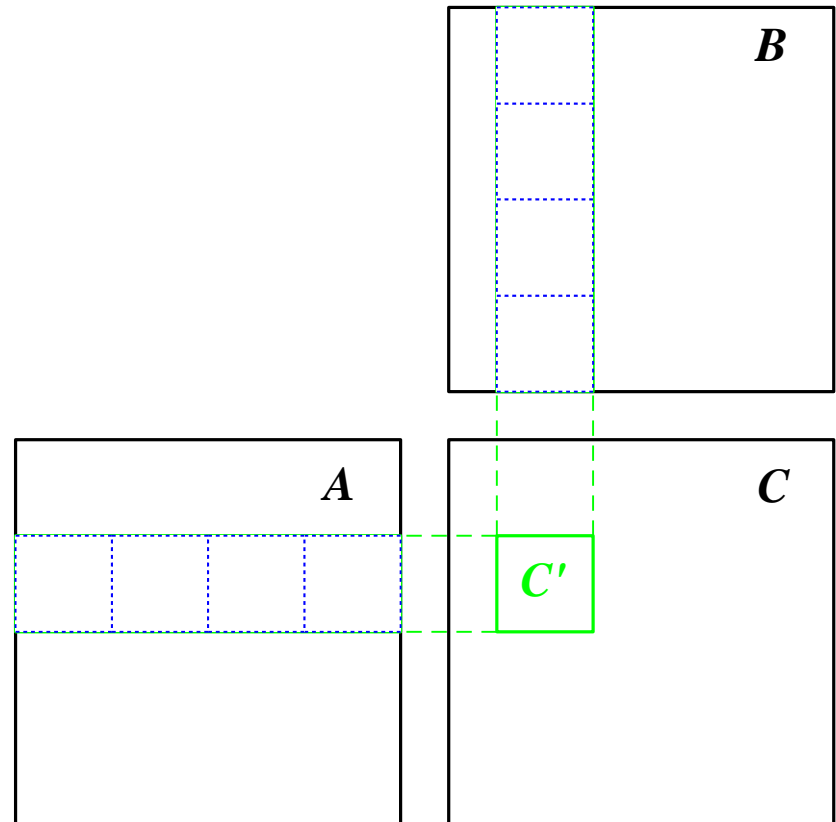
# Умножение матриц (2)

- 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)

- 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

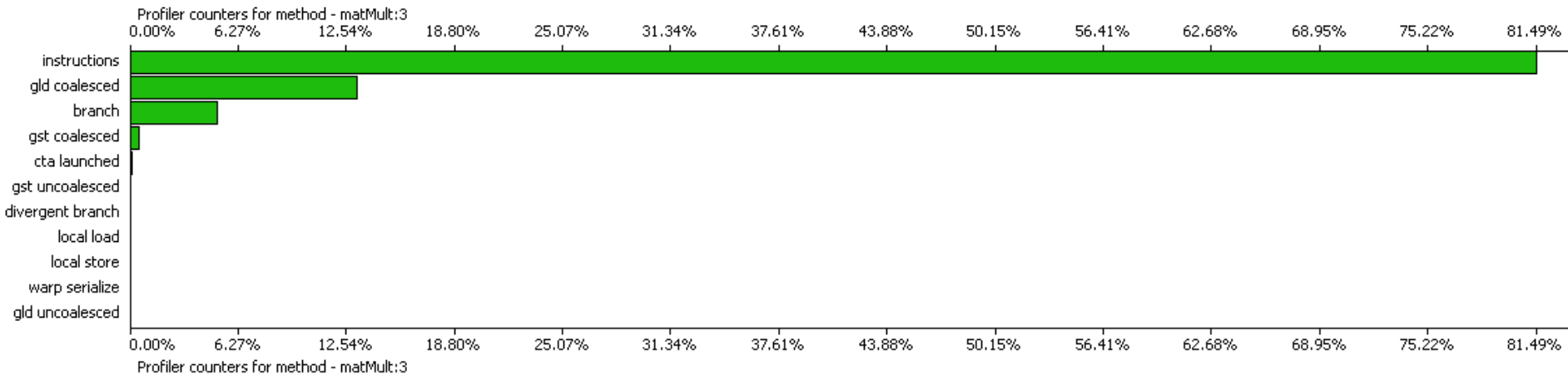
```
__global__ void matMult ( float * a, float * b, int n, float * c ) {
    int bx      = blockIdx.x,  by = blockIdx.y;
    int tx      = threadIdx.x, ty = threadIdx.y;
    int aBegin  = n * BLOCK_SIZE * by;
    int aEnd    = aBegin + n - 1;
    int bBegin  = BLOCK_SIZE * bx;
    int aStep   = BLOCK_SIZE, 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
  - $2*N$  арифметических операций
  - $2*N/16$  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

Profiler Counter Plot



- Теперь основное время (81.49%) ушло на вычисления
- Чтение из памяти стало *coalesced* и заняло всего 12.5 %

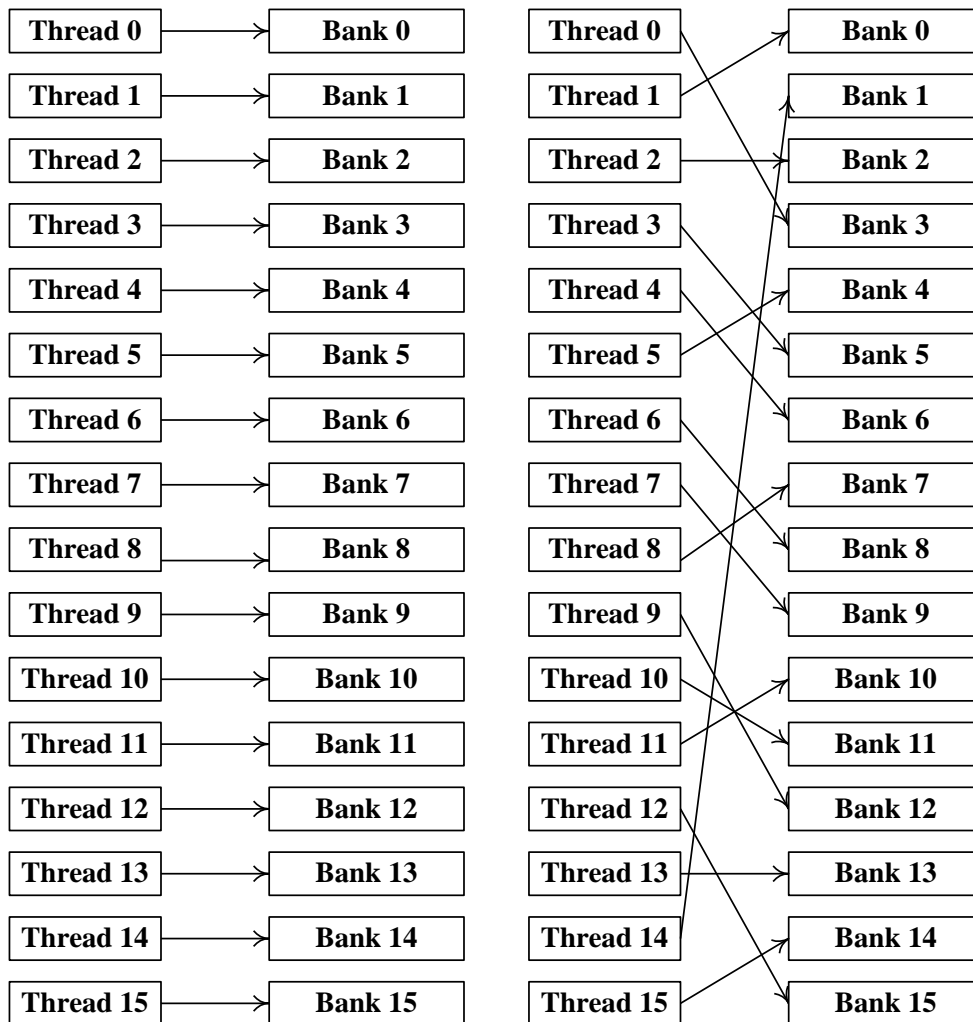
# Эффективная работа с *shared*-памятью Tesla 10

- Отдельное обращение для каждой половины *warp*'а
- 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 Tesla 10

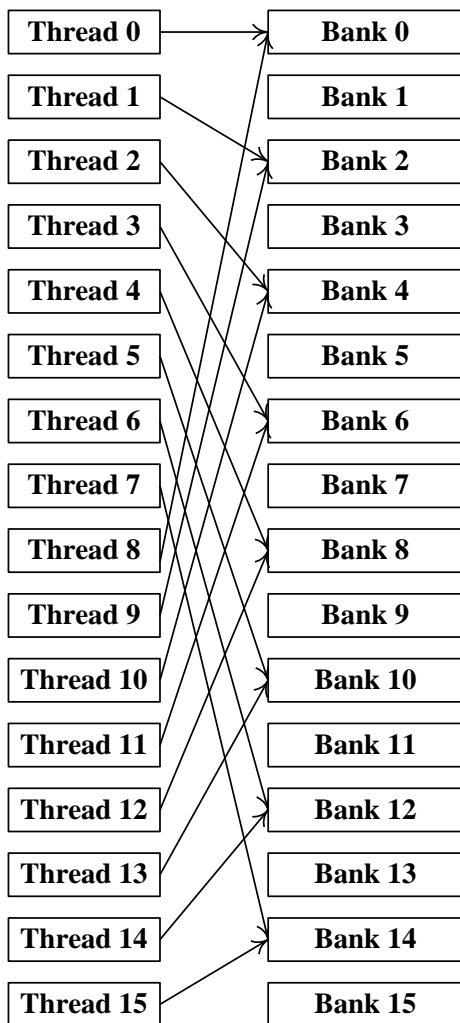
- Банки строятся из 32-битовых слов
- Подряд идущие 32-битовые слова попадают в подряд идущие банки
- *Bank conflict* - несколько нитей из одного *half-warp*'а обращаются к одному и тому же банку
- Конфликта не происходит если все 16 нитей обращаются к одному слову (*broadcast*)

# Бесконфликтные паттерны доступа

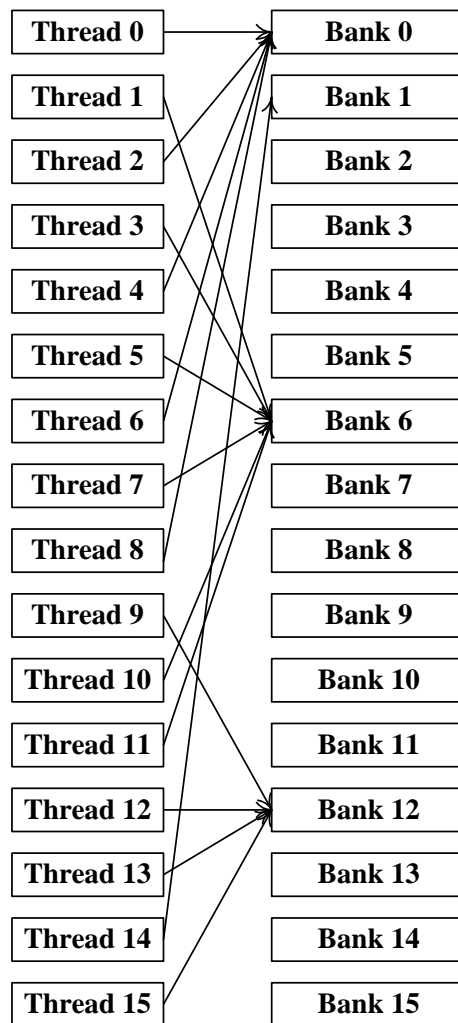


# Паттерны с конфликтами банков

X2



X6



# Доступ к массиву элементов

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

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

Нет конфликтов

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

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

Конфликты 2-го  
порядка

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

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

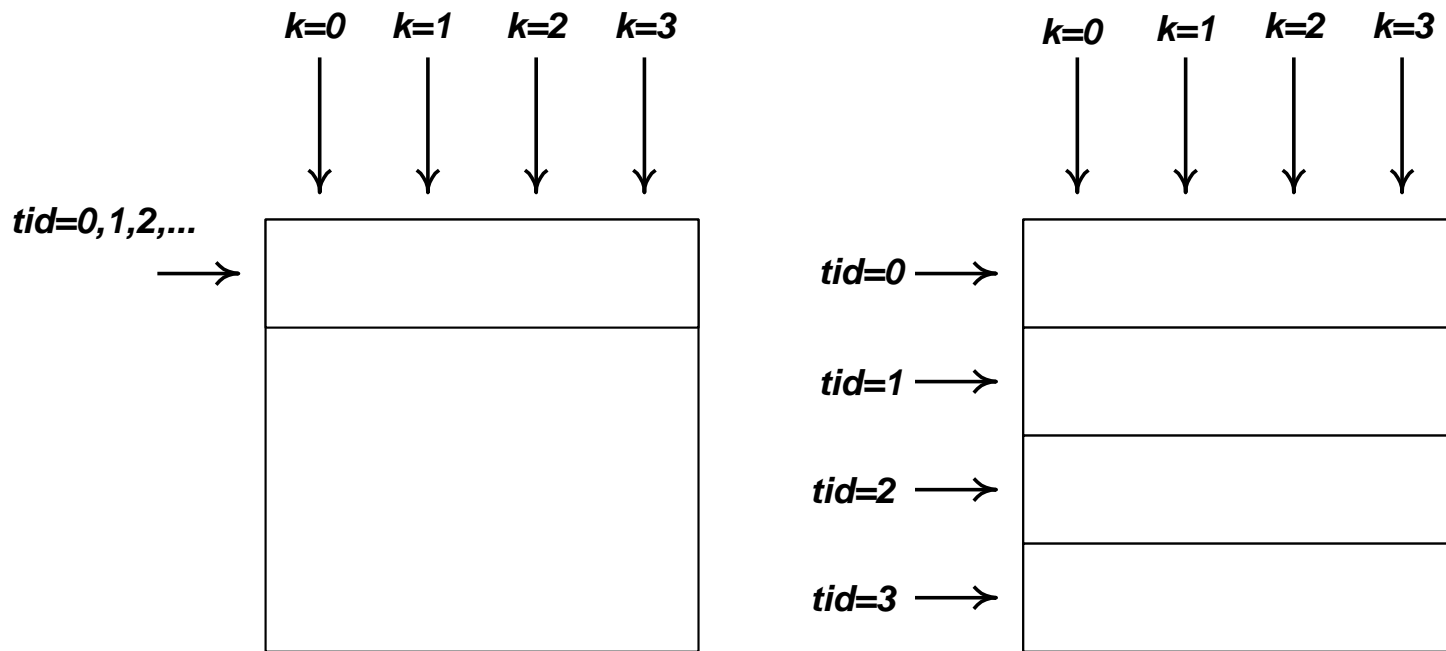
Конфликты 4-го  
порядка



# Пример - матрицы 16\*16

- Перемножение  $A \cdot B^T$  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 $16 \times 16$





# Эффективная работа с *shared*-памятью Tesla 20

- Вся *shared*-память разбита на 32 банка
- Все нити варпа обращаются в память совместно.
- Каждый банк работает независимо от других
- Можно одновременно выполнить до 32 обращений к *shared*-памяти

# Эффективная работа с shared-памятью Tesla 20

- Банк конфликты:
  - При обращении  $>1$  нити варпа к разным 32битным словам из одного банка
- При обращении  $>1$  нити варпа к разным байтам одного 32битного слова, конфликта нет
  - При чтении: операция broadcast
  - При записи: результат не определен

# План

- Разделяемая память
- **Константная память**
- Базовые алгоритмы

# 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 );
```

# План

- Разделяемая память
- Константная память
- **Базовые алгоритмы**



# Базовые алгоритмы на CUDA

- Reduce
- Scan (prefix sum)
- Histogram
- Sort

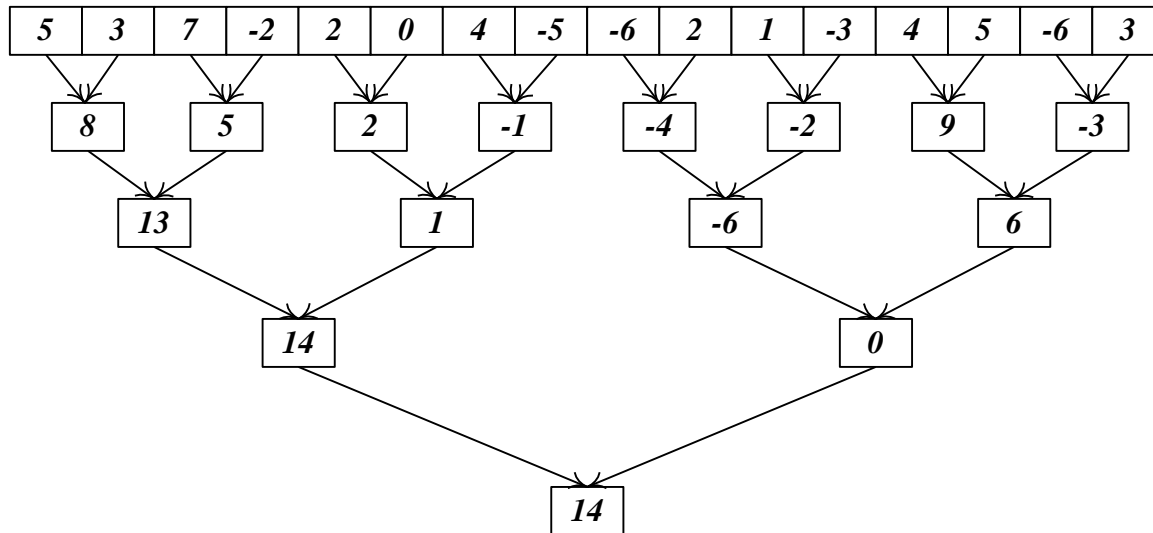
# Параллельная редукция (reduce)

- Дано:
  - Массив элементов  $a_0, a_1, \dots, a_{n-1}$
  - Бинарная ассоциативная операция '+'
- Необходимо найти  $A = a_0 + a_1 + \dots + a_{n-1}$
- Лимитирующий фактор – доступ к памяти
- В качестве операции также может быть *min*, *max* и т.д.

# Реализация параллельной редукции

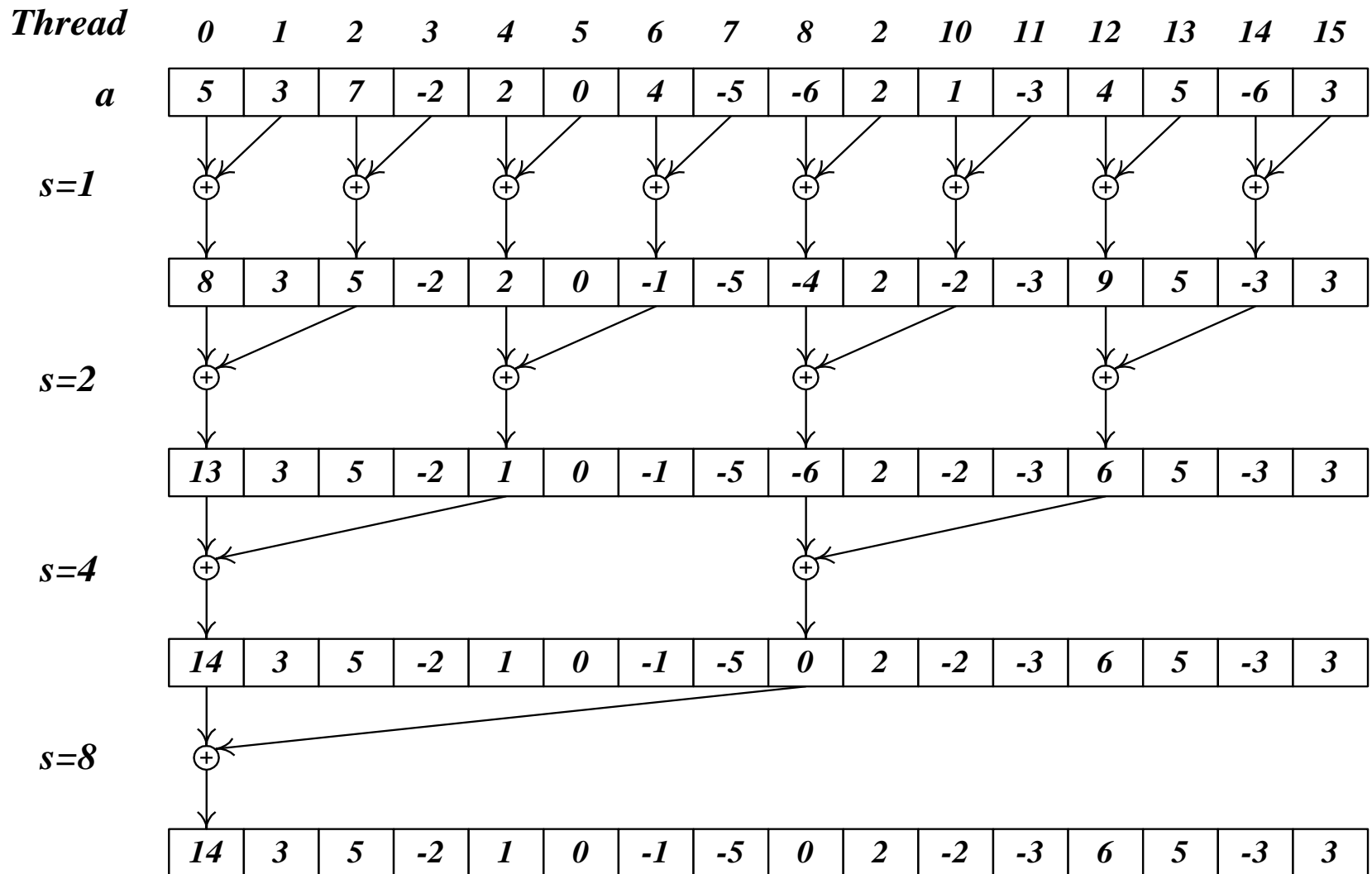
- Каждому блоку сопоставляем часть массива
- Блок
  - Копирует данные в *shared*-память
  - Иерархически суммирует данные в *shared*-памяти
  - Сохраняет результат в глобальной памяти

# Иерархическое суммирование



- Позволяет проводить суммирование параллельно, используя много нитей
- Требуется  $\log(N)$  шагов

# Редукция, вариант 1

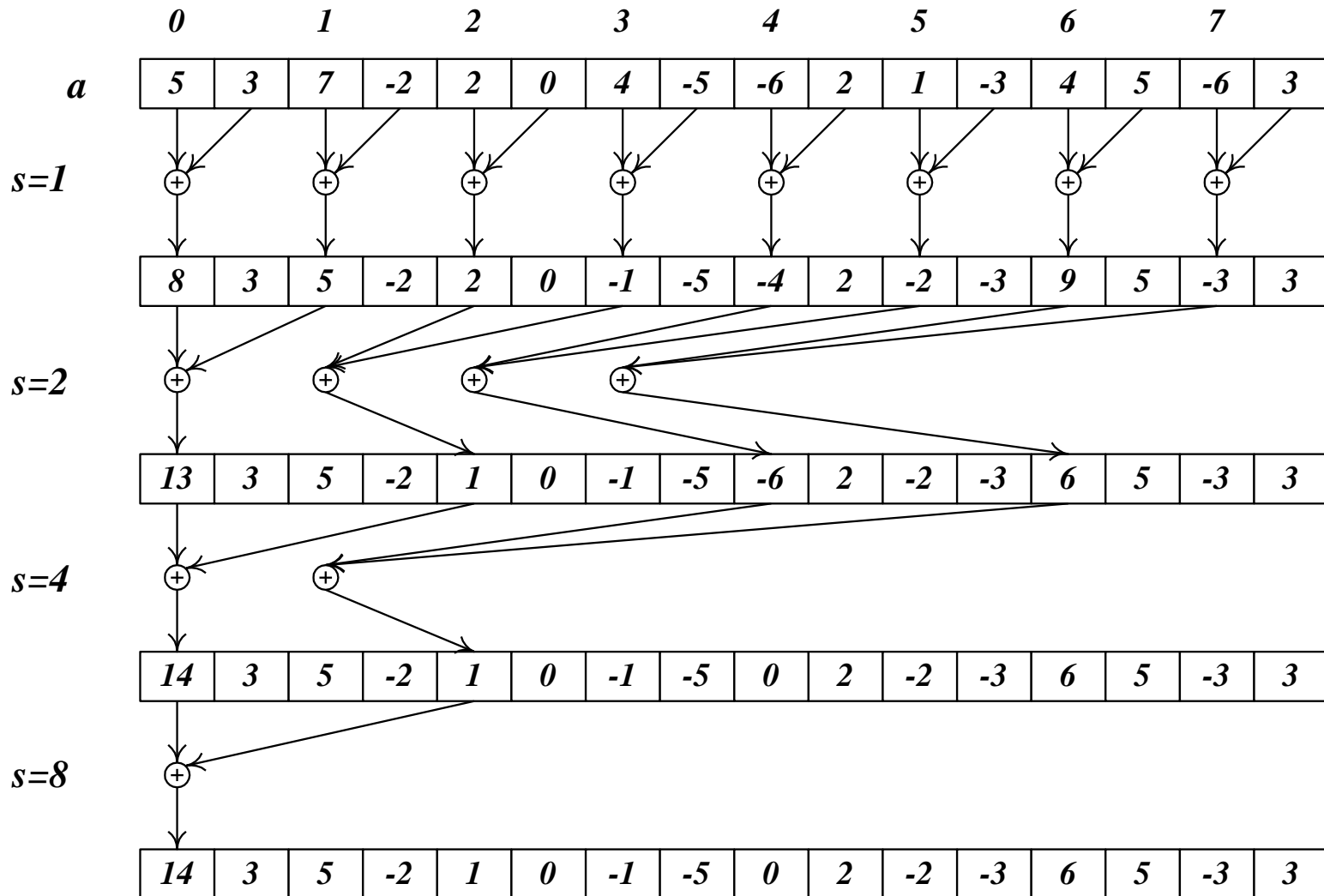


# Редукция, вариант 1

```
__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];
}
```



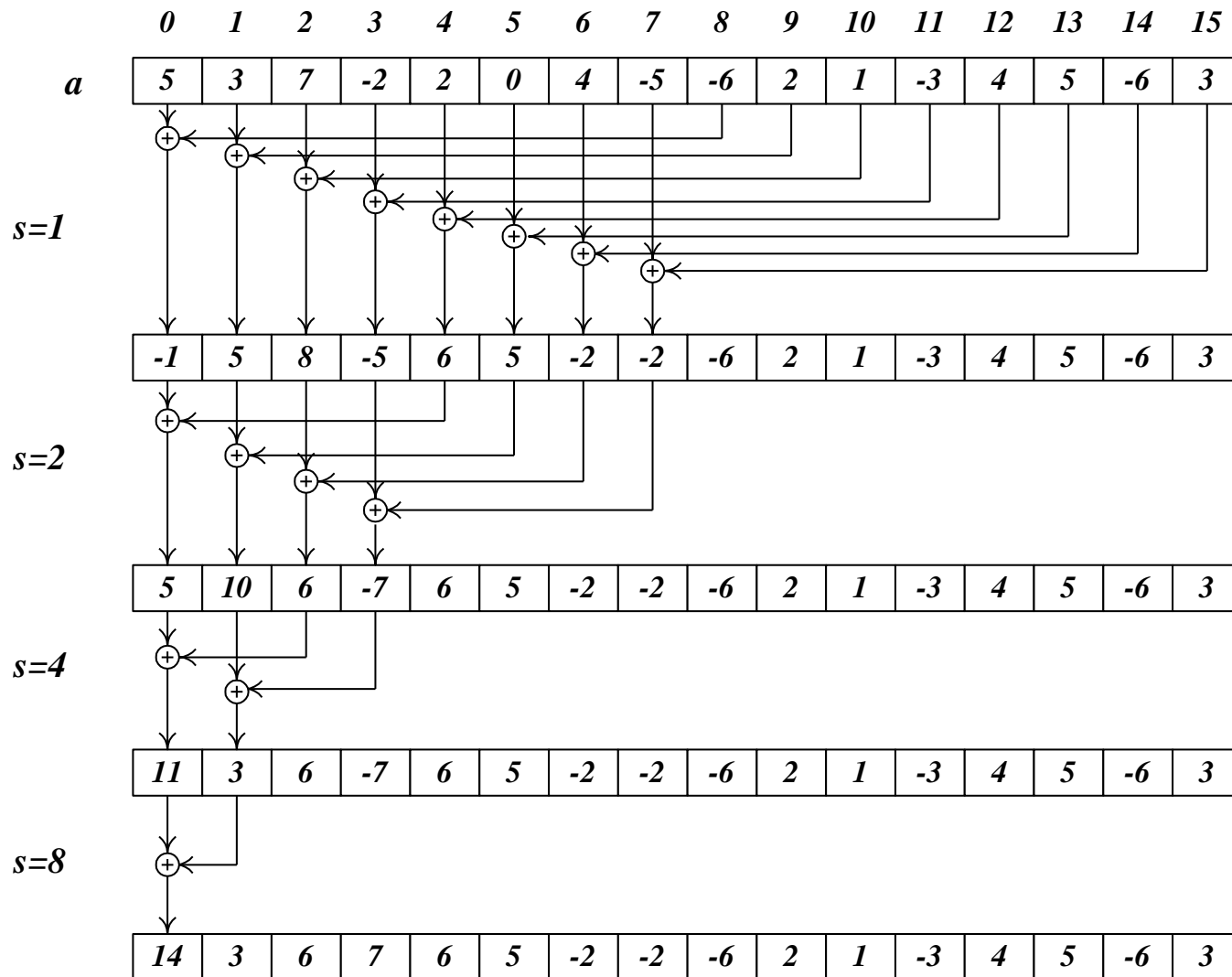
# Редукция, вариант 2

- 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  $dimBlock.x/2$ ) 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, вариант 5 (fixed)

```
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
{
    // compile can be "oversmart here"
    volatile float * smem = data;

    smem [tid] += smem [tid + 32];
    smem [tid] += smem [tid + 16];
    smem [tid] += smem [tid + 8];
    smem [tid] += smem [tid + 4];
    smem [tid] += smem [tid + 2];
    smem [tid] += smem [tid + 1];
}
```

# Редукция, быстроедействие

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



# Ресурсы нашего курса

- [Steps3d.Narod.Ru](#)
- [Google Site CUDA.CS.MSU.SU](#)
- [Google Group CUDA.CS.MSU.SU](#)
- [Google Mail CS.MSU.SU](#)
- [Google SVN](#)
- [Tesla.Parallel.Ru](#)
- [Twirpx.Com](#)
- [Nvidia.Ru](#)