Иерархия памяти CUDA. Shared-память и ее эффективное использование. Параллельная редукция.

- 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)

shared-память в CUDA

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

Типичный паттерн использования

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

- Произведение двух квадратных матриц A и B размера N*N, N кратно 16
- Матрицы расположены в глобальной памяти
- По одной нити на каждый элемент произведения
- 2D блок – 16*16
- 2D grid

Умножение матриц. Старая реализация.

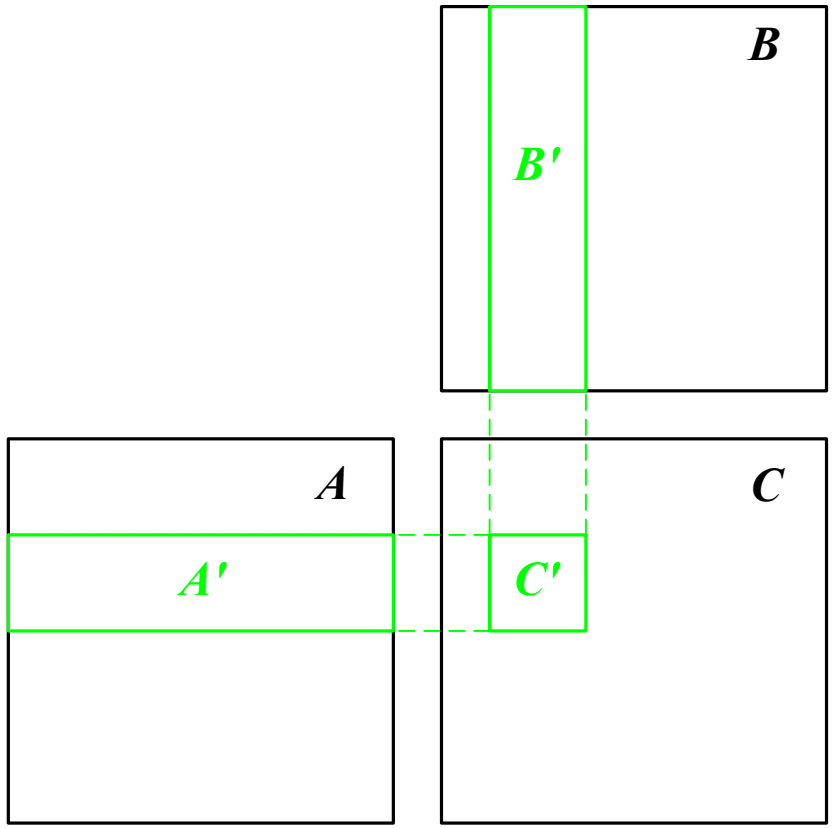
```
#define BLOCK_SIZE 16

__global__ void matMult ( float * a, float * b, int n, float * c )
{
  int   bx = blockIdx.x;
  int   by = blockIdx.y;
  int   tx = threadIdx.x;
  int   ty = threadIdx.y;
  float sum = 0.0f;
  int   ia = n * BLOCK_SIZE * by + n * ty;
  int   ib = BLOCK_SIZE * bx + tx;
  int   ic = n * BLOCK_SIZE * by + BLOCK_SIZE * bx;

  for ( int k = 0; k < n; k++ )
    sum += a [ia + k] * b [ib + k*n];

  c [ic + n * ty + tx] = sum;
}
```

Простейшая реализация.

- На каждый элемент
  - 2*N арифметических операций
  - 2*N обращений к глобальной памяти
- Memory bound (тормозит именно доступ к памяти)

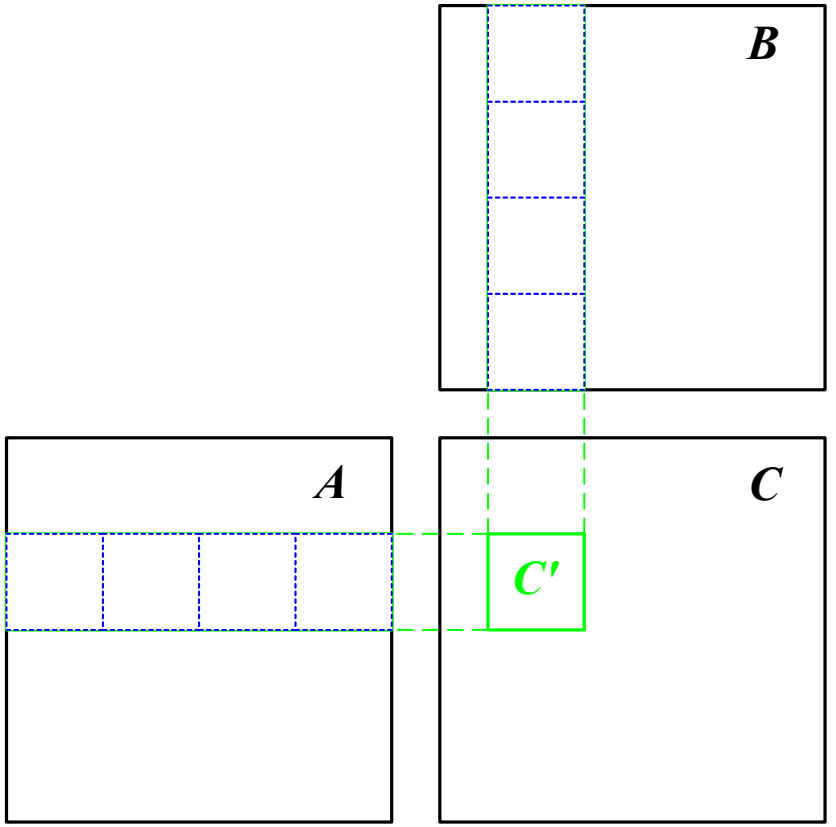
Простейшая реализация.

- 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



Эффективная реализация.

- «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



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

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)

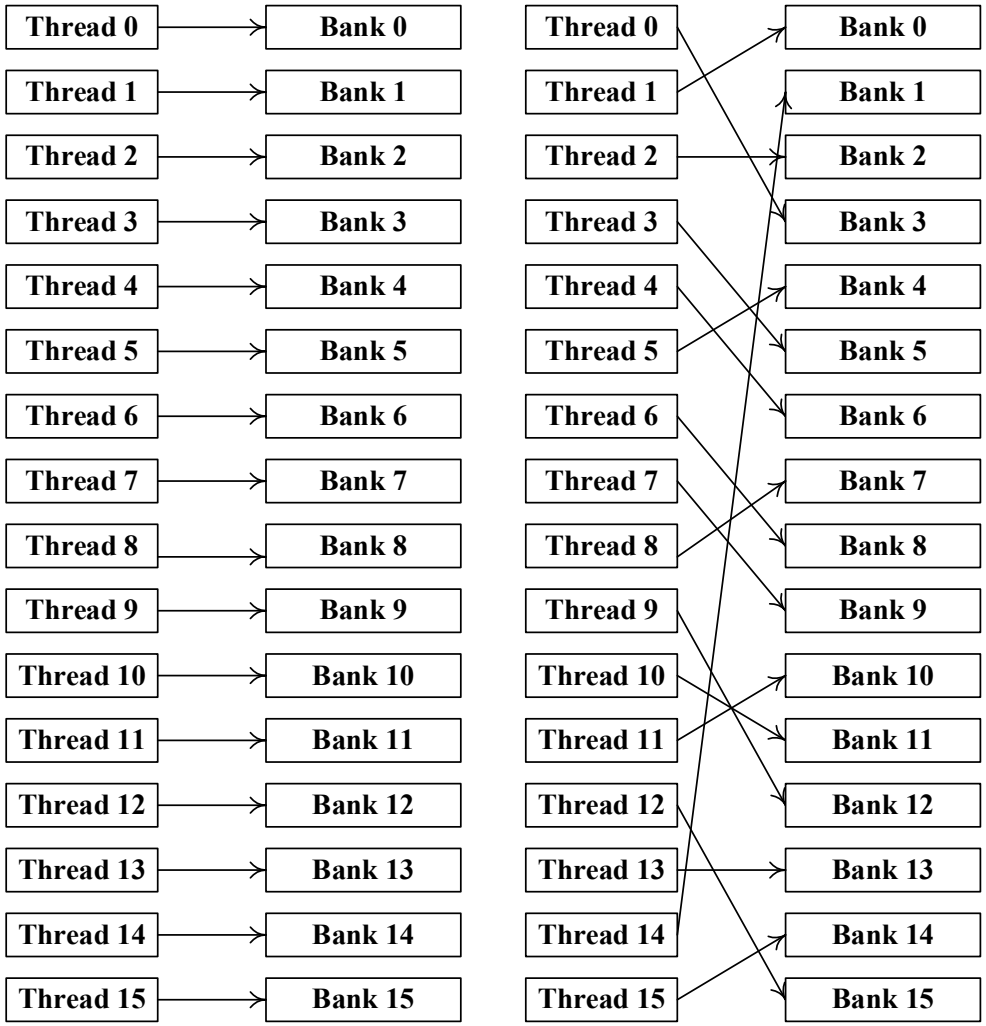
Эффективная работа с shared-памятью.

- 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

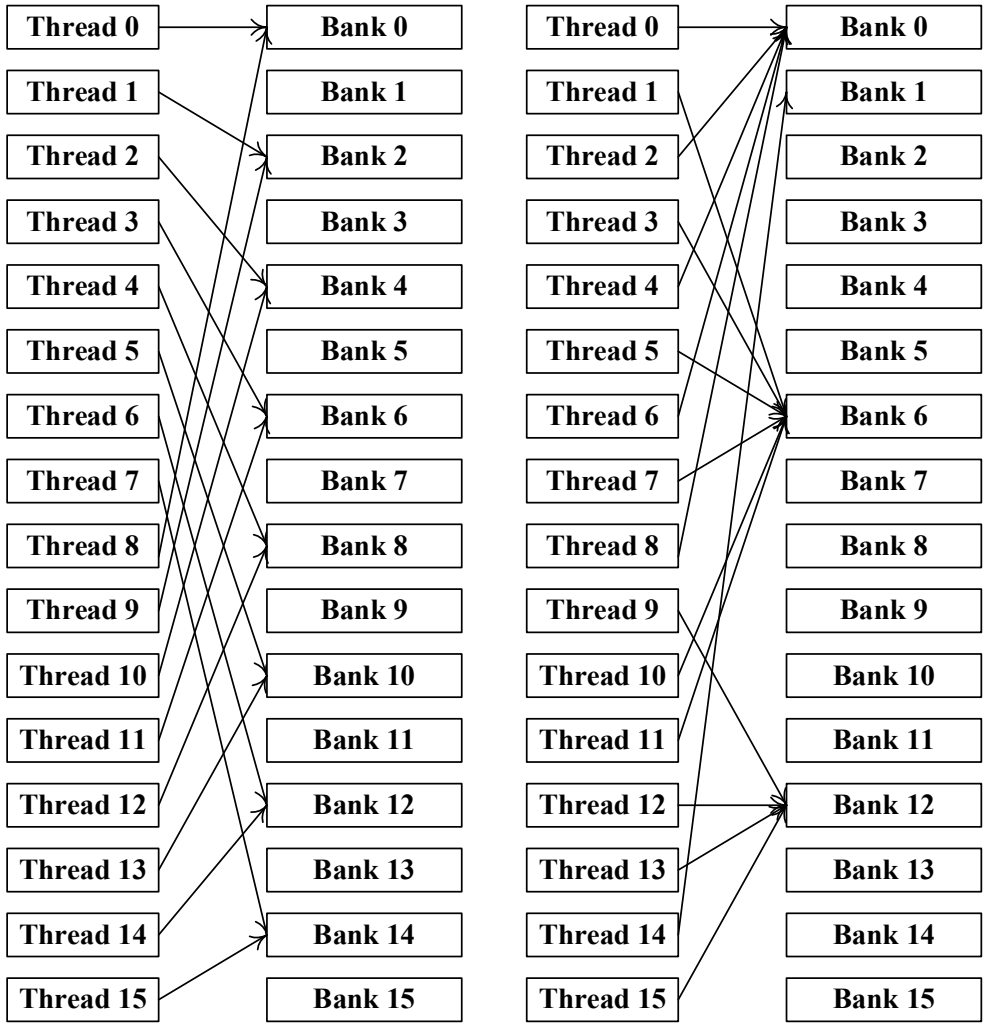
Эффективная работа с shared-памятью.

- Банки строятся из 32-битовых слов
- Подряд идущие 32-битовые слова попадают в подряд идущие банки
- ***Bank conflict*** – несколько нитей из одного half-warp'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



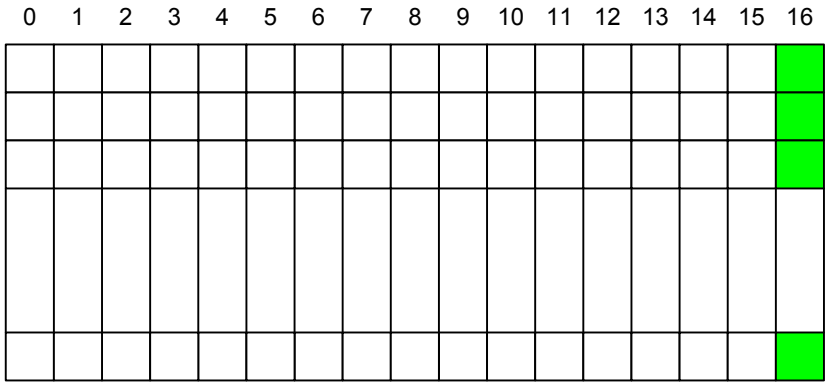
- Слева – конфликт второго порядка – вдвое меньшая скорость
- Несколько конфликтов, до 6-го порядка

Пример – матрицы 16*16

- Перемножение двух матриц 16*16, расположенных в shared-памяти
  - Доступ к одной идет по строкам, к другой – по столбцам
- Все элементы строки распределены равномерно по 16 банкам
  - конфликтов нет
- Все элементы столбца лежат в одном банке
  - Конфликт 16-го порядка

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



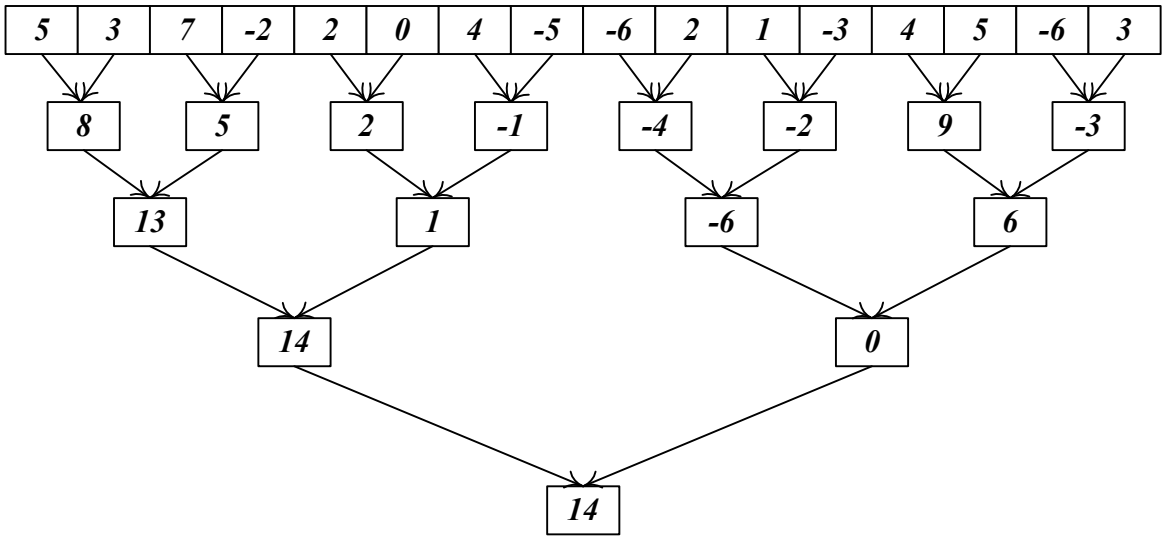
Реализация параллельной редукции

- Дано:
  - Массив элементов $a_0, a_1, \ldots, a_{n-1}$
  - Бинарная ассоциативная операция ‘+’
- Необходимо найти
  - A, $A = a_0 + a_1 + \ldots + a_{n-1}$
- Лимитирующий фактор – доступ к памяти
- В качестве операции может быть min/max

Реализация параллельной редукции

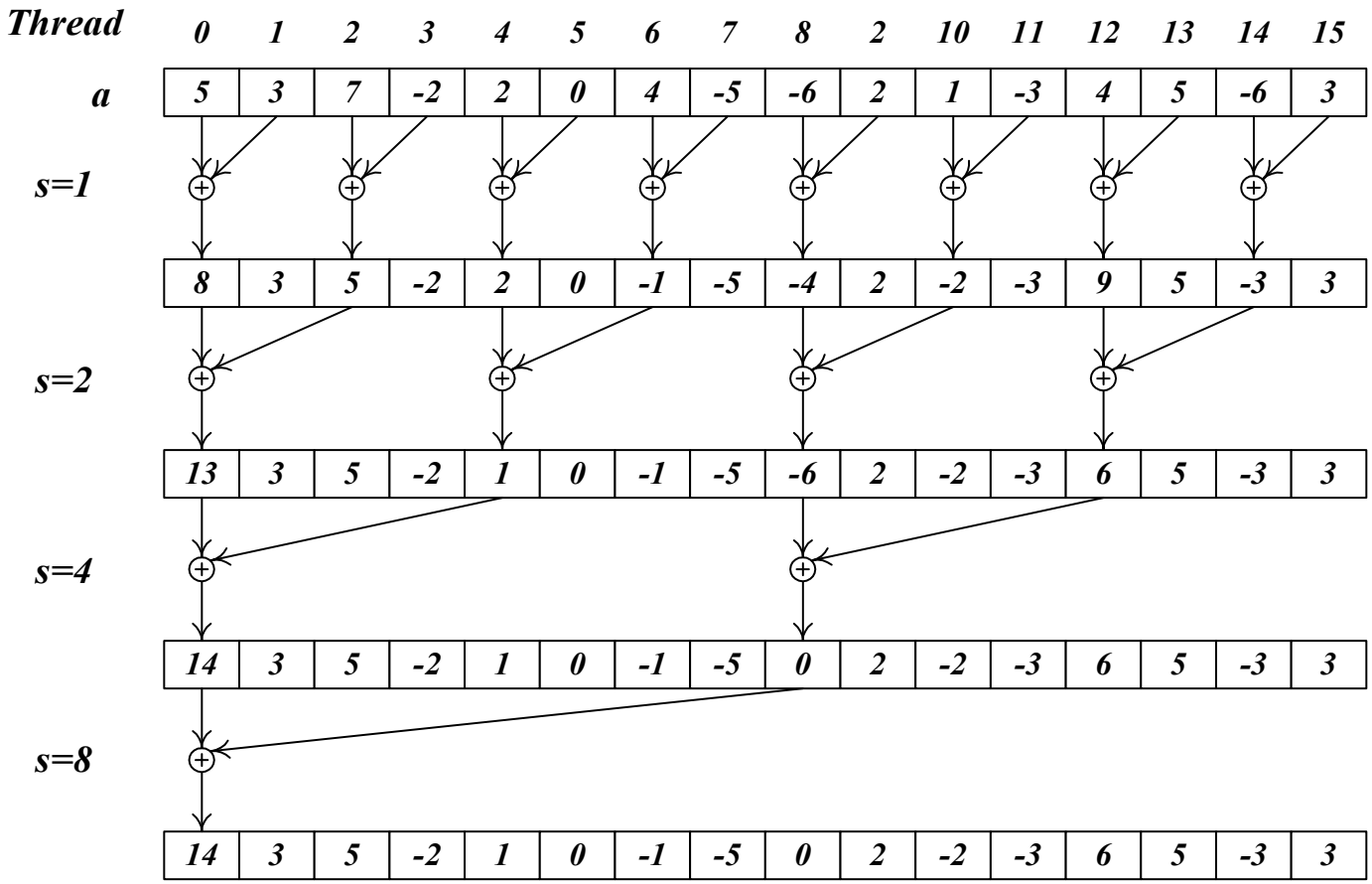
- Каждому блоку сопоставляем часть массива
- Блок
  - Копирует данные в shared-память
  - Иерархически суммирует данные в shared-памяти
  - Сохраняет результат

Иерархическое суммирование



- Позволяет проводить суммирование параллельно, используя много нитей
- Требует logN шагов

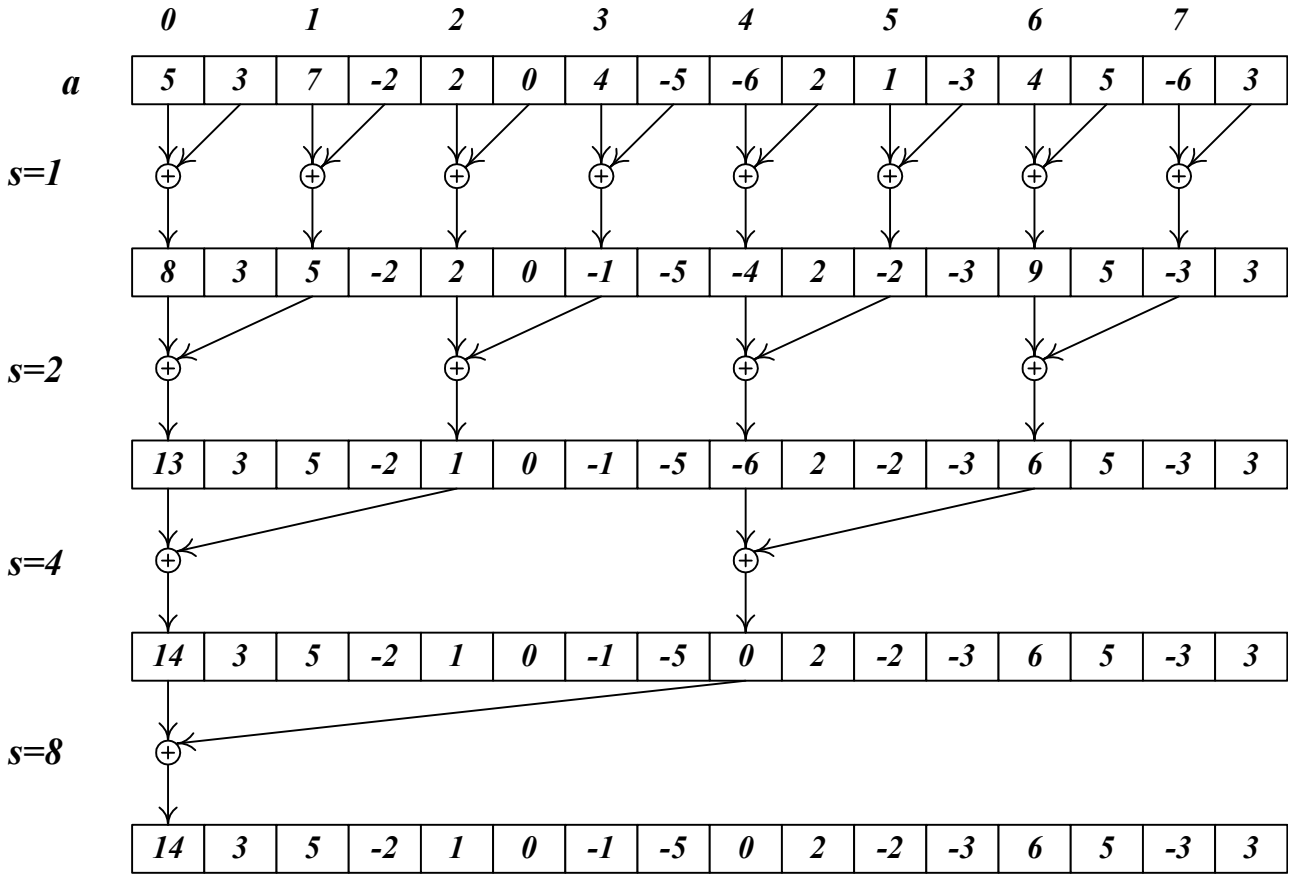
Редукция, вариант 1



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

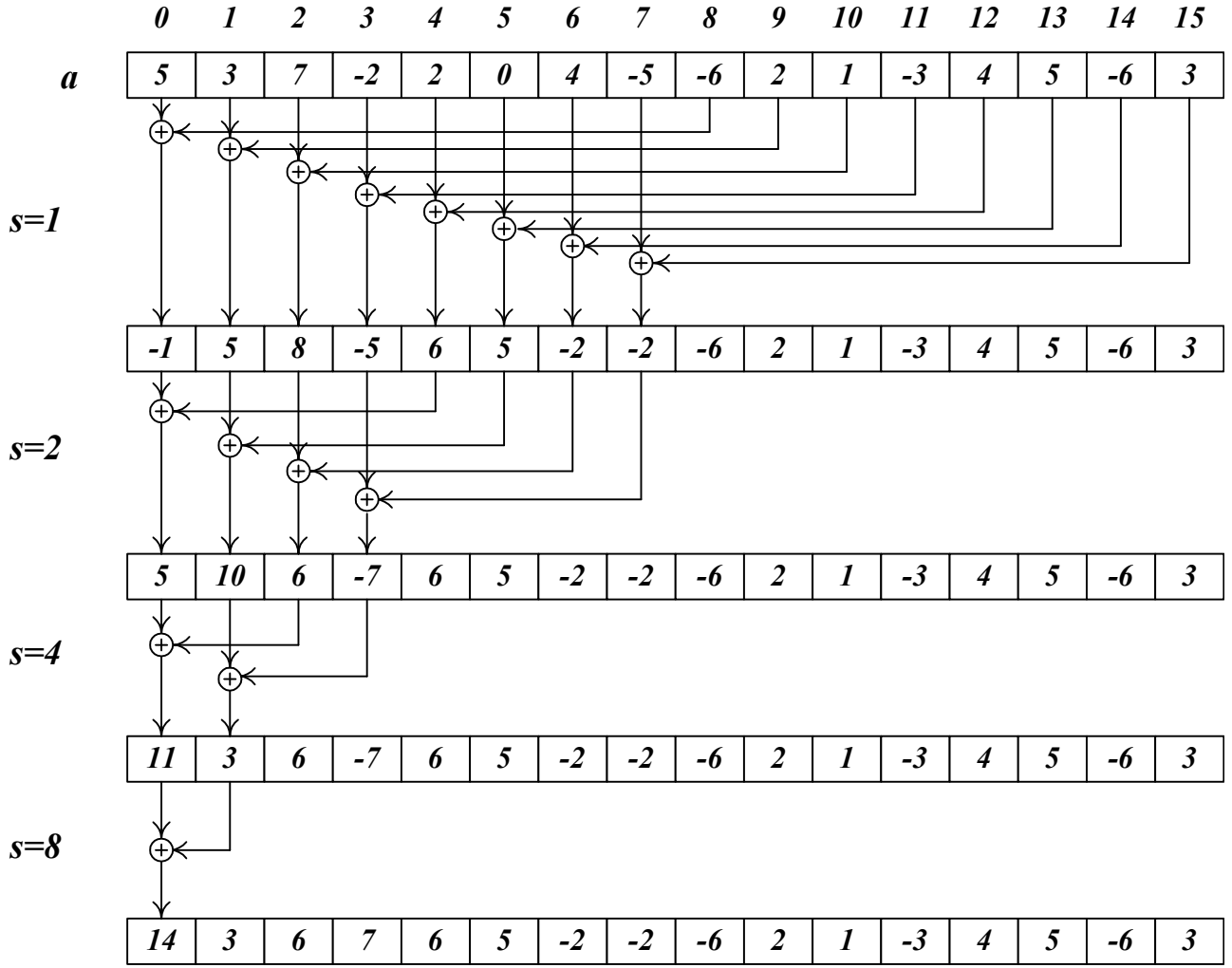
Редукция, вариант 2

- 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

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 [index] += data [index + 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<=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, 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, быстродействие

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

Ресурсы нашего курса

- 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