# Иерархия памяти CUDA. Разделяемая память. Основные алгоритмы.


## Лектор:

Боресков А.В. (ВМК МГУ), steps3d@narod.ru

# 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 |
| **Shared** | **R/W** | **Per-block** | **Высокая** | **SM** |
| Глобальная | R/W | Per-grid | Низкая | DRAM |
| Constant | R/O | Per-grid | Высокая | DRAM |
| Texture | R/O | Per-grid | Высокая | DRAM |

Легенда:
-интерфейсы доступа

# Разделяемая память в CUDA

- Самая быстрая память (кроме регистров) – это разделяемая
- Изначально было всего 16 Кбайт на мультипроцессор
- CC 3.x – shared + L1 cache = 64 Кбайта
  - 16:48, что 16, а что 48 вы определяете сами для каждого ядра
  - cudaFuncSetCacheConfig(MyKernel, cudaFuncCachePreferShared)
  - cudaFuncSetCacheConfig(MyKernel, cudaFuncCachePreferEqual)
  - cudaFuncSetCacheConfig(MyKernel, cudaFuncCachePreferL1)
- Режимы доступа – 32 и 64 битовые (cudaDeviceSetSharedMemConfig)
- CC 5.x 64 (или 96 для 5.2)
- CC 6.x 64 (или 96 для 6.1)
- CC 7.0 128 Кбайт (0, 8, 16, 32, 64 или 96)

# Разделяемая память в CUDA

- Разбита на 32 независимых банка
- Доступ идет по варпам
- Если два и более варпа обращаются к разным 32-битовым словам разных банков – bank conflict
- Вся разделяемая память разбивается на банки следующим образом – подряд идущие слова попадают в подряд идущие банки
- 32 подряд идущих слова попадают в 32 банка
  - В этом (типичном) случае мы получаем бесконфликтный доступ
- Обращения к разным частям одного слова не дает конфликт
- Обращения к одному и тому же слову не дает конфликта
- Конфликт N-го порядка снижает скорость в N раз

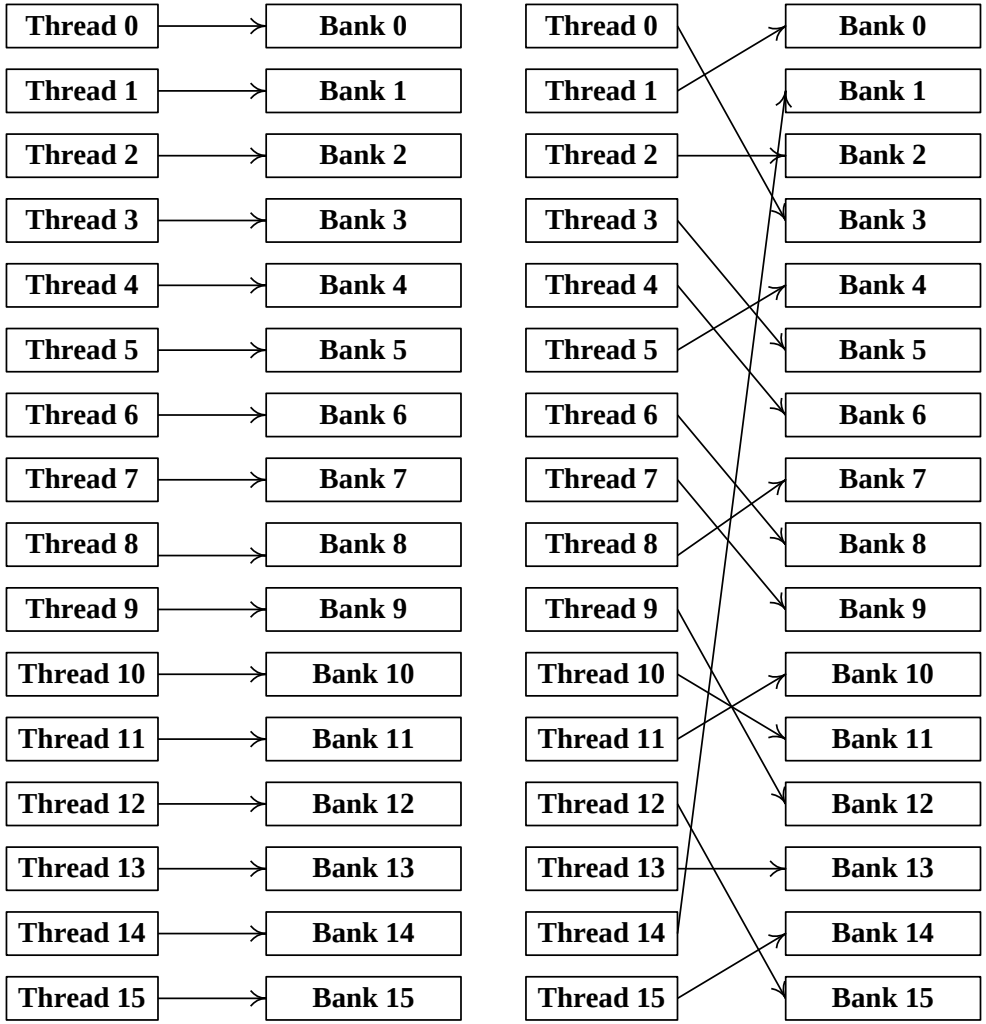
# Типичное использование

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

# Эффективная работа с shared-памятью

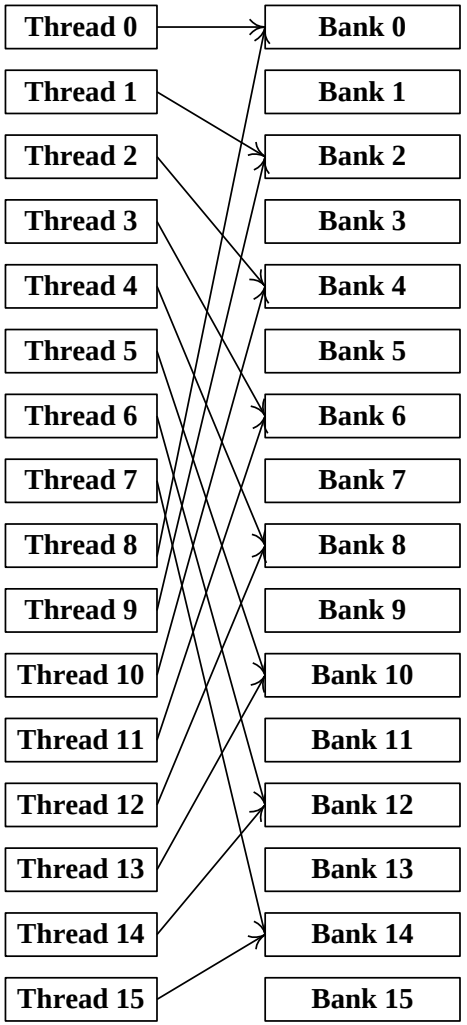
- Банки строятся из 32-битовых слов
- Подряд идущие 32-битовые слова попадают в подряд идущие банки
- *Bank conflict* – несколько нитей из одного *half-warp*'а обращаются к одному и тому же банку
- Конфликта не происходит если все 16 нитей обращаются к одному слову (*broadcast*)

# Бесконфликтные паттерны доступа

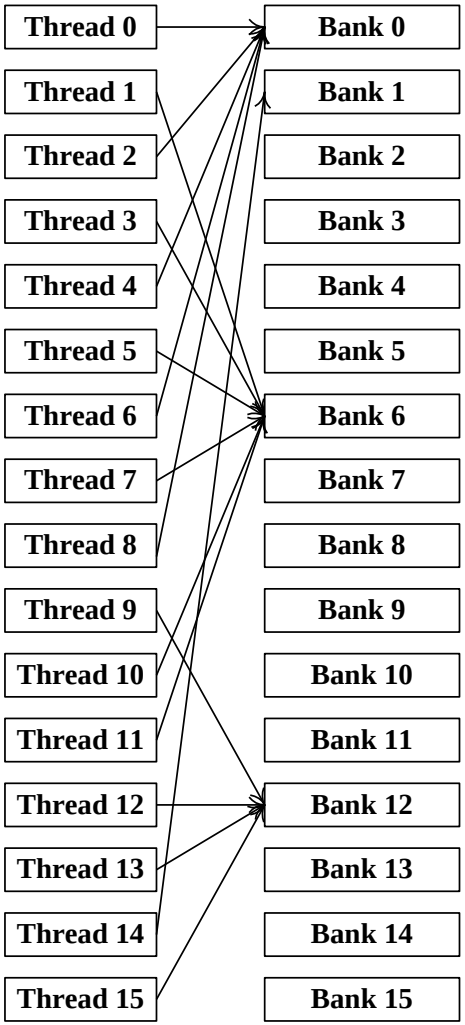


# Паттерны с конфликтами банков

X2




X6



# Эффективная работа с shared-памятью Tesla 20

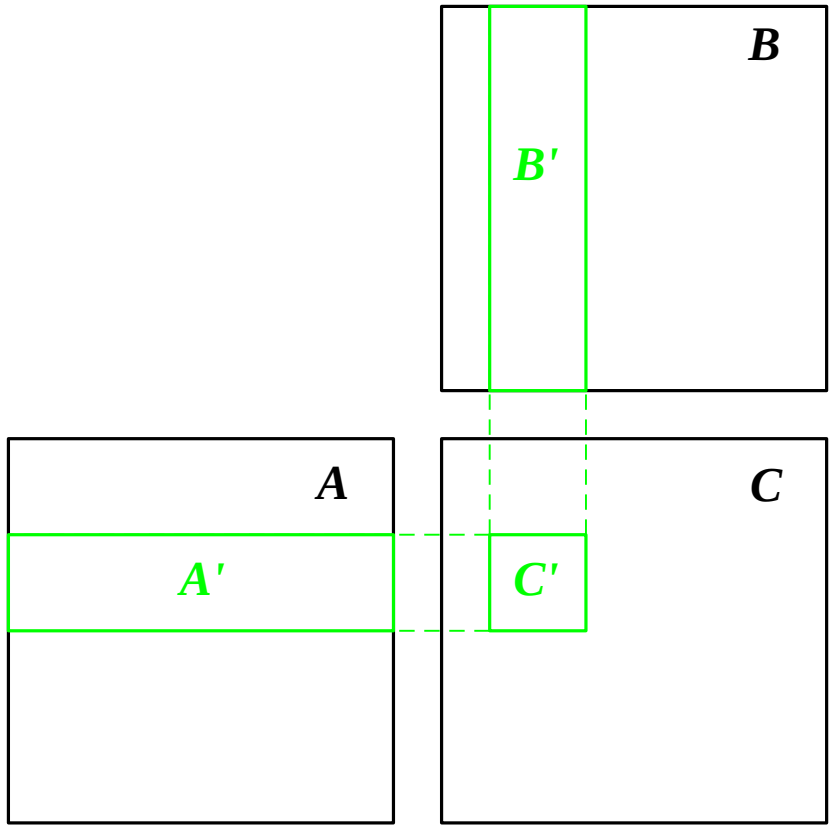
- Вся *shared*-память разбита на 32 банка
- Все нити варпа обращаются в память совместно.
- Каждый банк работает независимо от других
- Можно одновременно выполнить до 32 обращений к *shared*-памяти

# Эффективная работа с shared-памятью Tesla 20

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

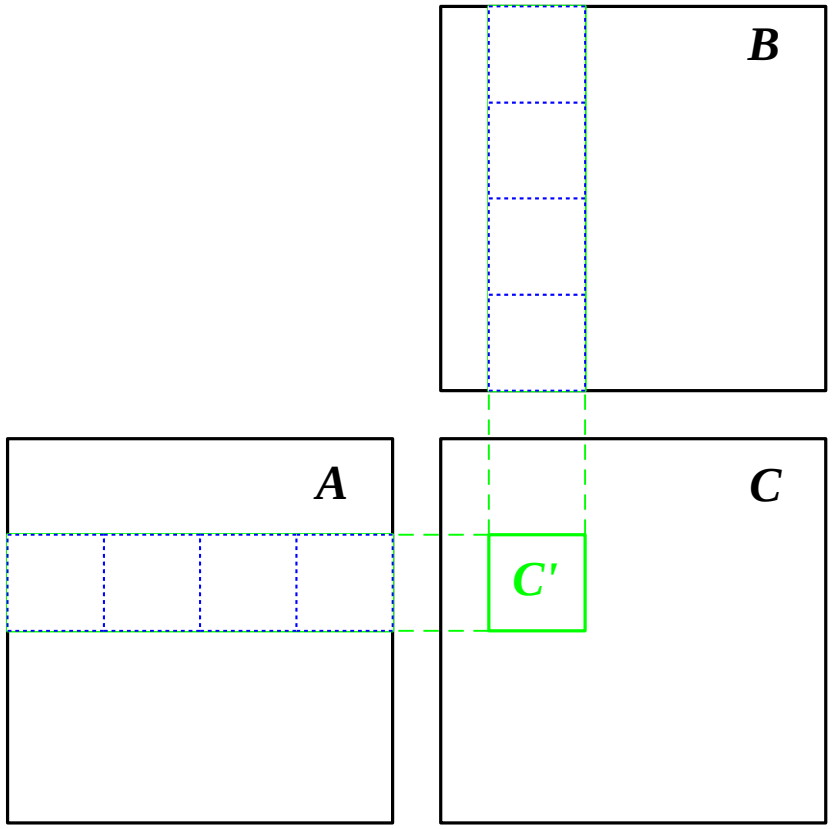
# Умножение матриц (2)

- 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




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

# 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;
}
```

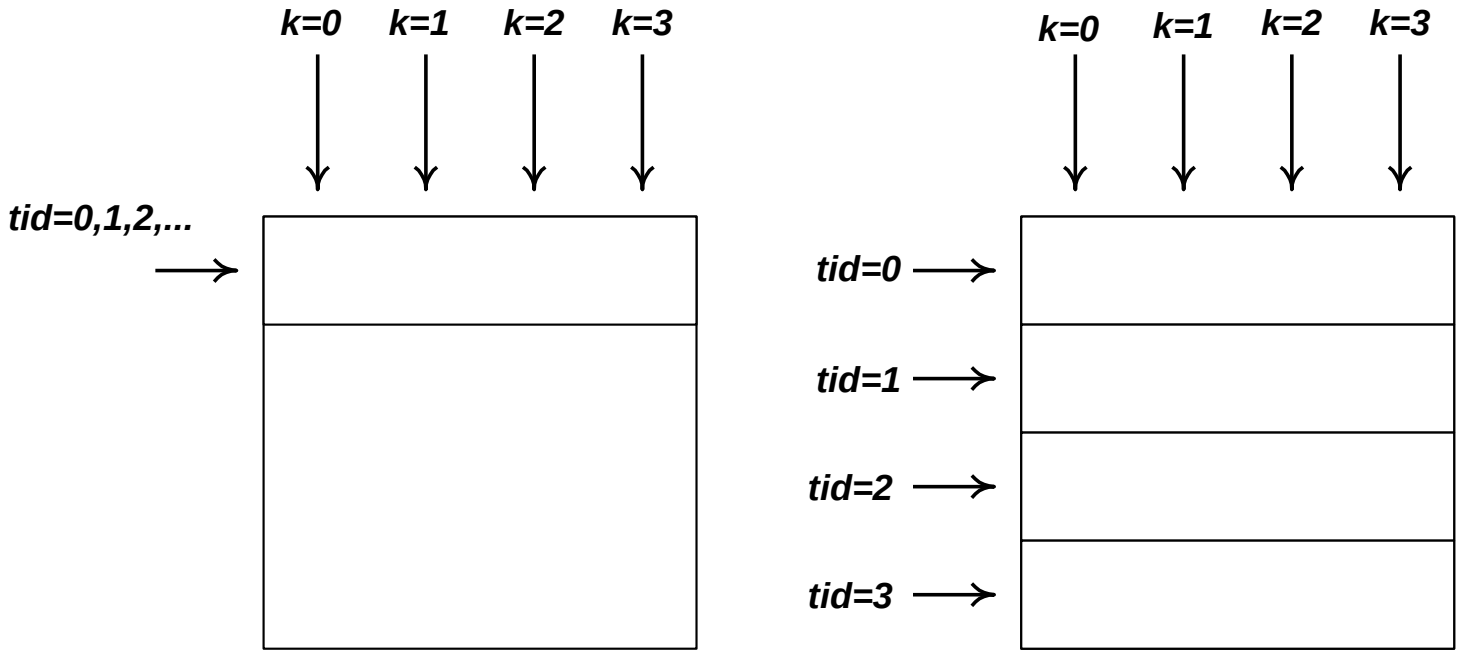
# Эффективная реализация

- На каждый элемент
  - 2*N арифметических операций
  - 2*N/16 обращений к глобальной памяти
- Чтение обоих матриц происходит оптимальным образом
- Поскольку разные *warp*'ы могут выполнять разные команды нужна явная синхронизация всех нитей блока
- Для перемножения 2048*2048 разница (с конфликтами и без конфликтов) – 168.20 vs 67.53 millisecs
  - На первом поколении выигрыш был гораздо больше из-за отсутствия кэша

# Пример – матрицы 32*32

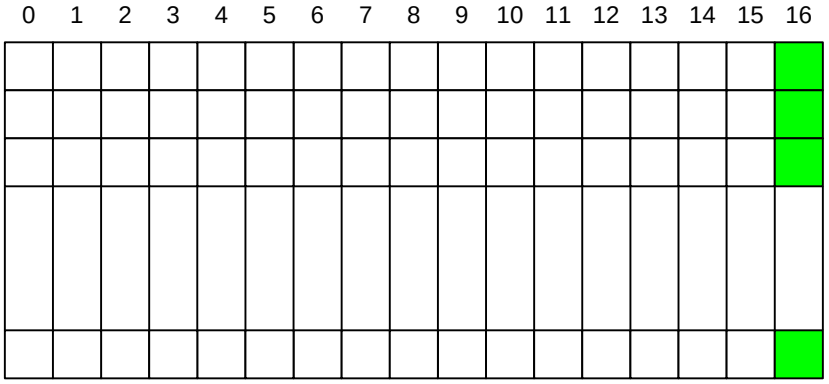
- Перемножение $A*B^T$ двух матриц 32*32, расположенных в *shared*-памяти
  - Доступ к обеим матрицам идет по строкам
- Все элементы строки распределены равномерно по 32 банкам
- Каждый столбец целиком попадает в один банк
- Получаем конфликт 32-го порядка при чтении матрицы *B*

# Пример - матрицы 32*32



# Пример – матрицы 32*32

- 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



# Работа с разделяемой памятью

- Разделяемая память – ограниченный ресурс
  - Чем больше нужно блоку, тем меньше блоков на SM
- Если конфликт 2-4 порядка и нет очевидного фикса – ignore it
- Фактически это просто управляемый кэш
  - Загружаем блок часто используемых данных
  - __syncthreads()
  - Используем этот блок
  - __syncthreads()
  - Можем перейти к следующему блоку
  - Синхронизация обычно бывает необходима – разные варпы могут выполнять разные команды
    - Кто-то уже загрузил данные и хочет их использовать
    - Кто-то еще не начал загружать
    - Если данные общие – то возможна попытка использовать данные до загрузки

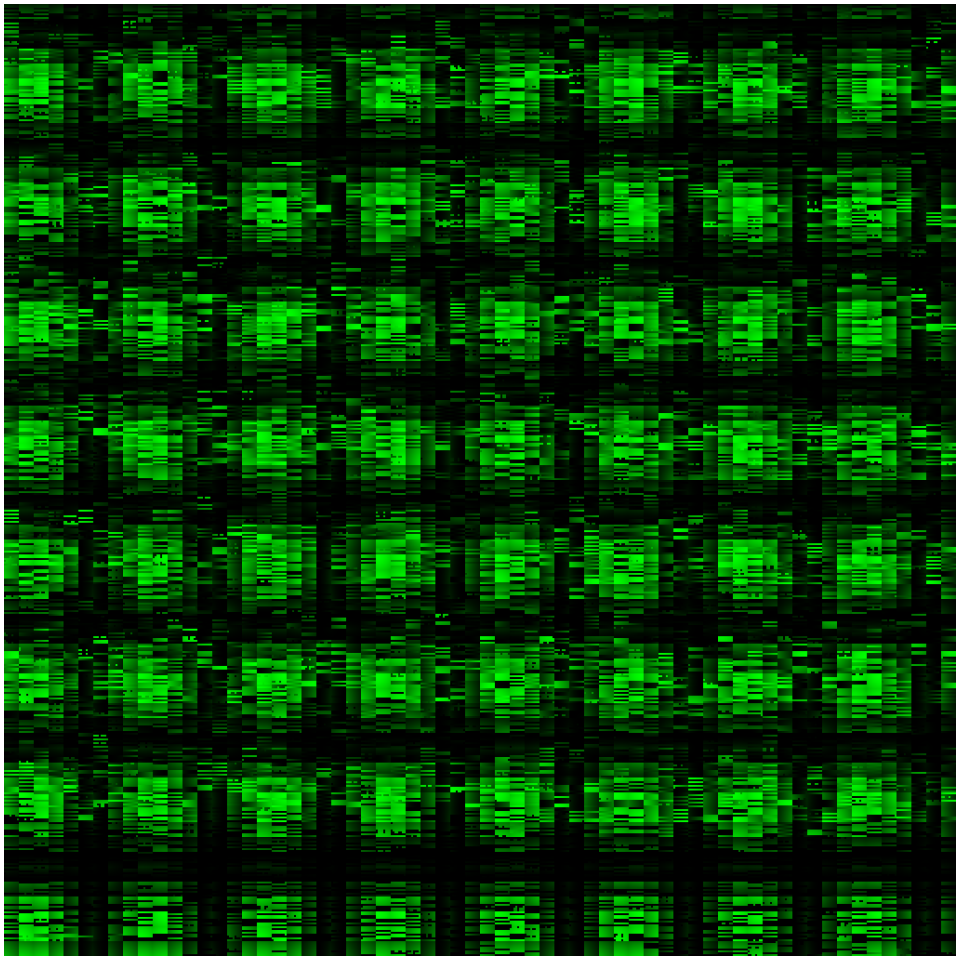
# Работа с разделяемой памятью

- Если не проводить явную синхронизацию то может возникнуть race condition
  - Нить попытается прочесть неготовые данные
  - Мы затрем еще не прочитанные данные
- Следующий пример показывает, что произойдет
  - Считаем изображение
  - Сперва нити считают и складывают в shared-буфер
  - Потом нити копируют результат
  - Копирует пиксел не та нить, которая его считала
  - Поэтому нужная явная синхронизация

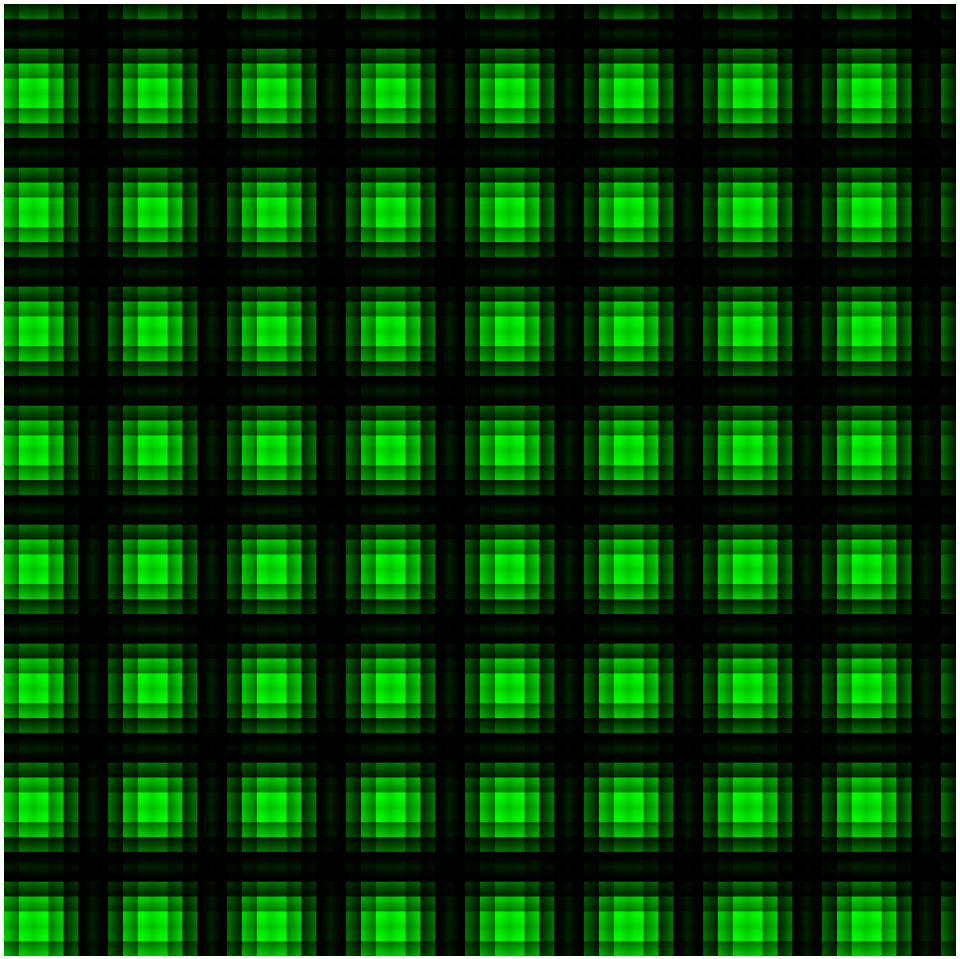
# Работа с разделяемой памятью

```
__global__ void kernel( uint8_t * ptr )
{
   int x      = threadIdx.x + blockIdx.x * blockDim.x;
   int y      = threadIdx.y + blockIdx.y * blockDim.y;
   int offset = x + y * blockDim.x * gridDim.x;
   __shared__ float shared[16][16];

   const float period = 128.0f;
   shared[threadIdx.x][threadIdx.y] = 255 * (sinf(x*2.0f*PI/ period) + 1.0f)*
                                            (sinf(y*2.0f*PI/ period) + 1.0f)/4.0f;
 __syncthreads ();       // !!!

   ptr [offset*3 + 0] = 0;
   ptr [offset*3 + 1] = shared [15-threadIdx.x][15-threadIdx.y];
   ptr [offset*3 + 2] = 0;
}
```

# Работа с разделяемой памятью - no sync



# Работа с разделяемой памятью - sync



# План

- Разделяемая память
- Константная память
- Базовые алгоритмы

# Константная память

```
__constant__ float constData [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 );
```

# Работа с константной памятью

- Фактически это память для uniform-ов в графике
- Кэшируется
- Со стороны GPU возможно только чтение
- Со стороны CPU доступ через специальные функции
- Быстро работает, когда все нити варпа обращаются к одному и тому же слову
- Если паттерн обращения имеет регулярный характер и все нити варпа обращаются за раз к одному и тому же элементу, то имеет смысл использовать
- Если характер обращения нерегулярный (разные нити обращаются к случайным элементам), то возможно использование текстурной памяти будет более удачным вариантом

# План

- Разделяемая память
- Константная память
- Базовые алгоритмы

# Базовые алгоритмы на CUDA

- Reduce
- Scan (prefix sum)
- Histogram
- Sort
- Все это сейчас есть в thrust, цель – рассмотреть оптимизацию алгоритмов и типичные приемы
  - Сам thrust мы рассмотрим позже

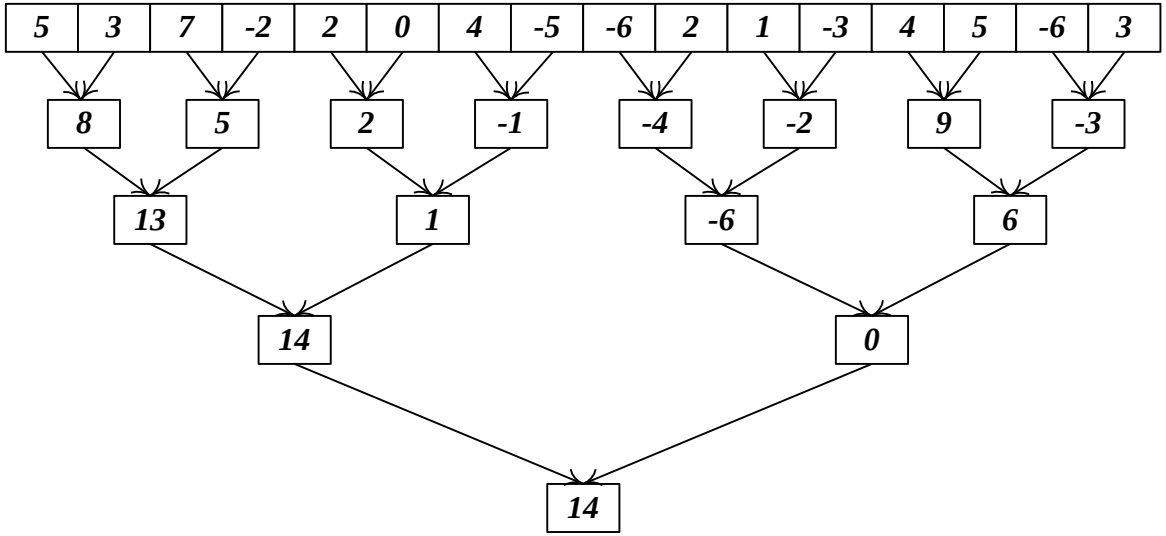
# Параллельная редукция (reduce)

- Дано:
  - Массив элементов $a_0, a_1, ..., a_{n-1}$
  - Бинарная ассоциативная операция ‘+’
- Необходимо найти $A = a_0 + a_1 + ... + a_{n-1}$
- Лимитирующий фактор – доступ к памяти
- В качестве операции также может быть *min, max* и т.д.
- Легко реализовать последовательно за O(N) шагов
  - Мы хотим гораздо быстрее
  - *Divide at impera*
    - сперва разбиваем на массив на части и каждый блок получает свою часть и независимо суммирует ее
    - Внутри блока выполняем иерархическое суммирование (в разделяемой памяти, так как нужно много обращений)

# Реализация параллельной редукции

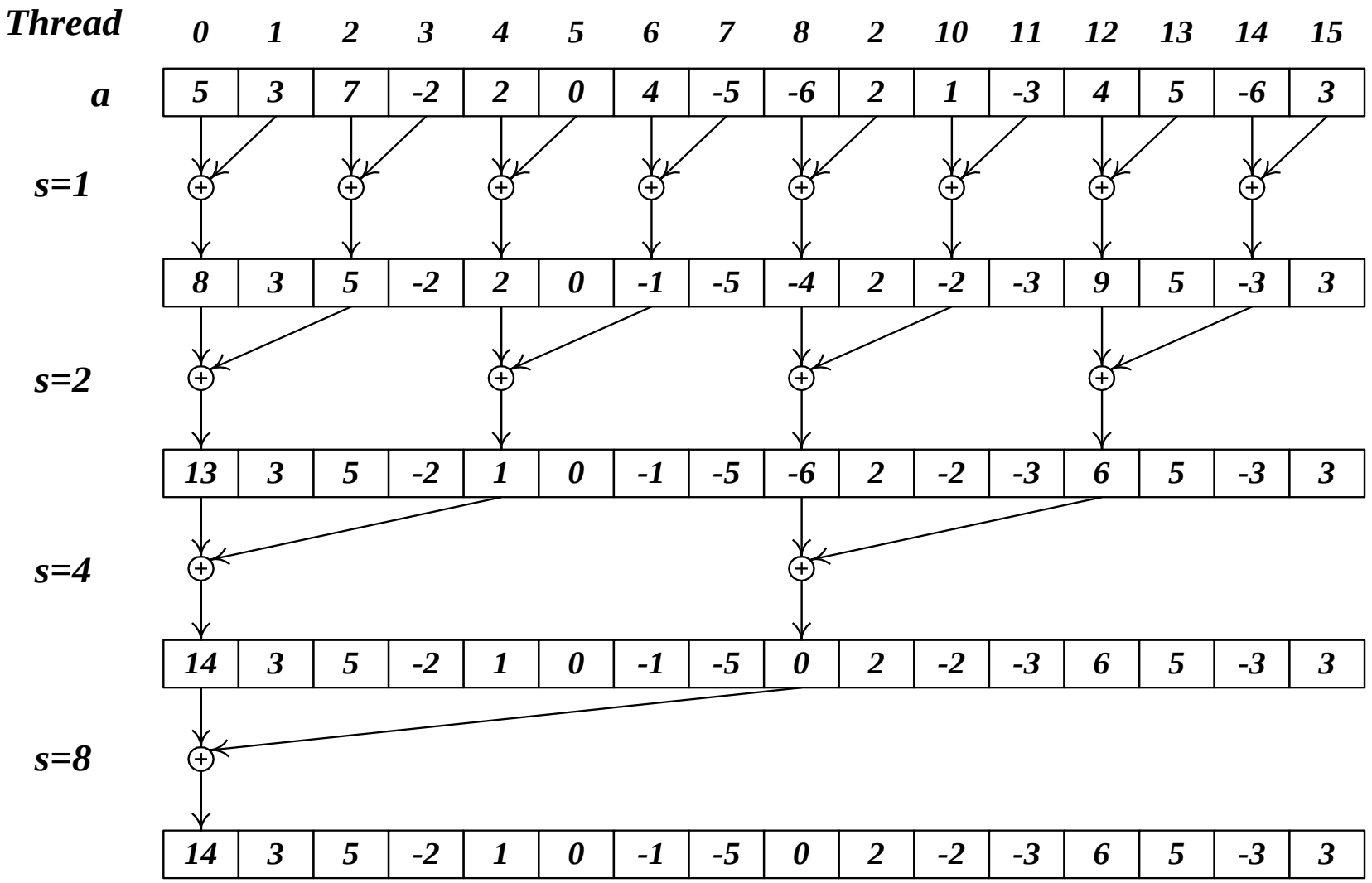
- Каждому блоку сопоставляем часть массива
- Блок
  - Копирует данные в *shared*-память
  - Иерархически суммирует данные в *shared*-памяти
  - Сохраняет результат в глобальной памяти

# Иерархическое суммирование




- Позволяет проводить суммирование параллельно, используя много нитей
- Требует *log(N)* шагов

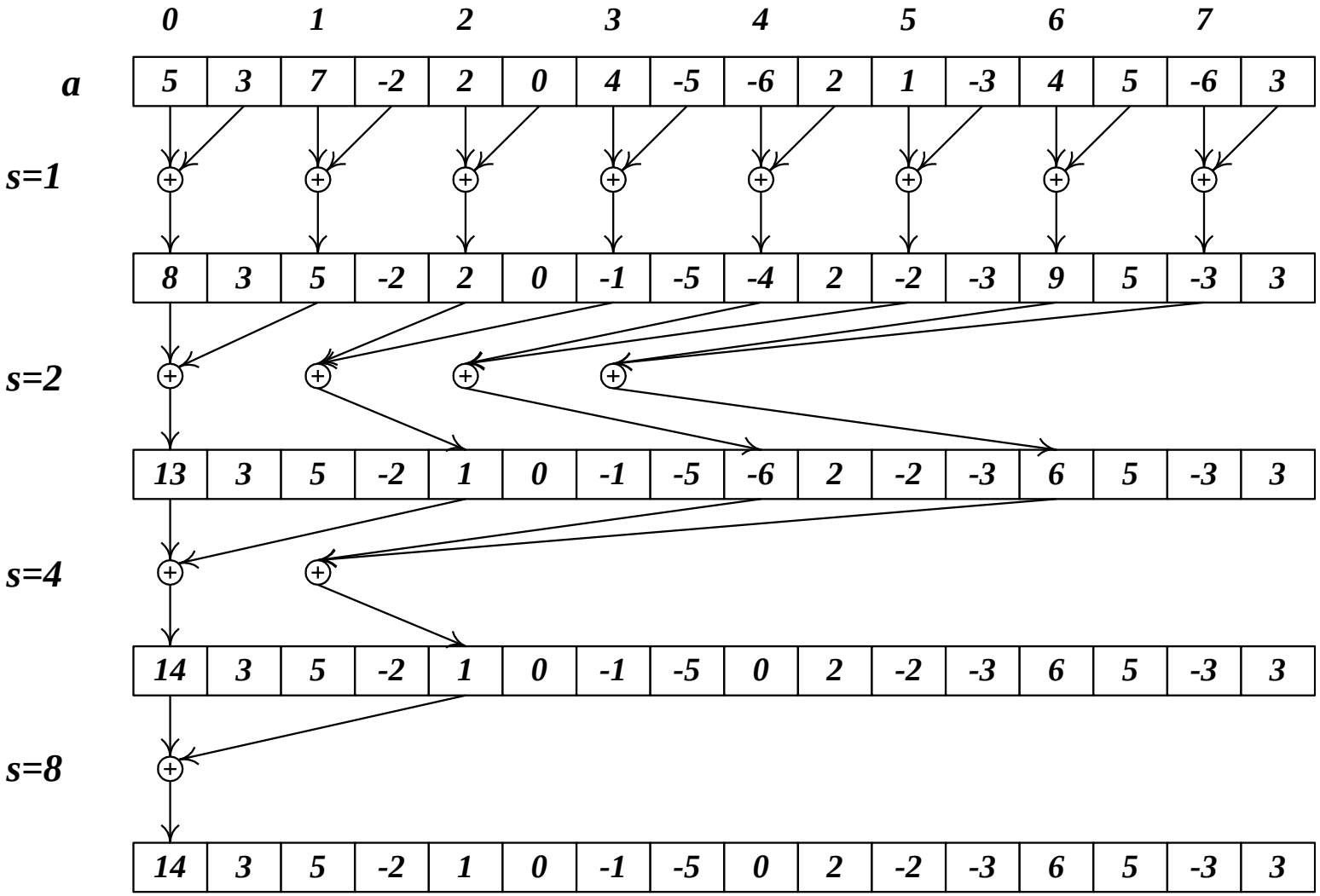
# Редукция, вариант 1



# 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]; // bank conflict !!!
    __syncthreads ();
  }
  if ( tid == 0 )             // write result of block reduction
    outData [blockIdx.x] = data [0];
}
```

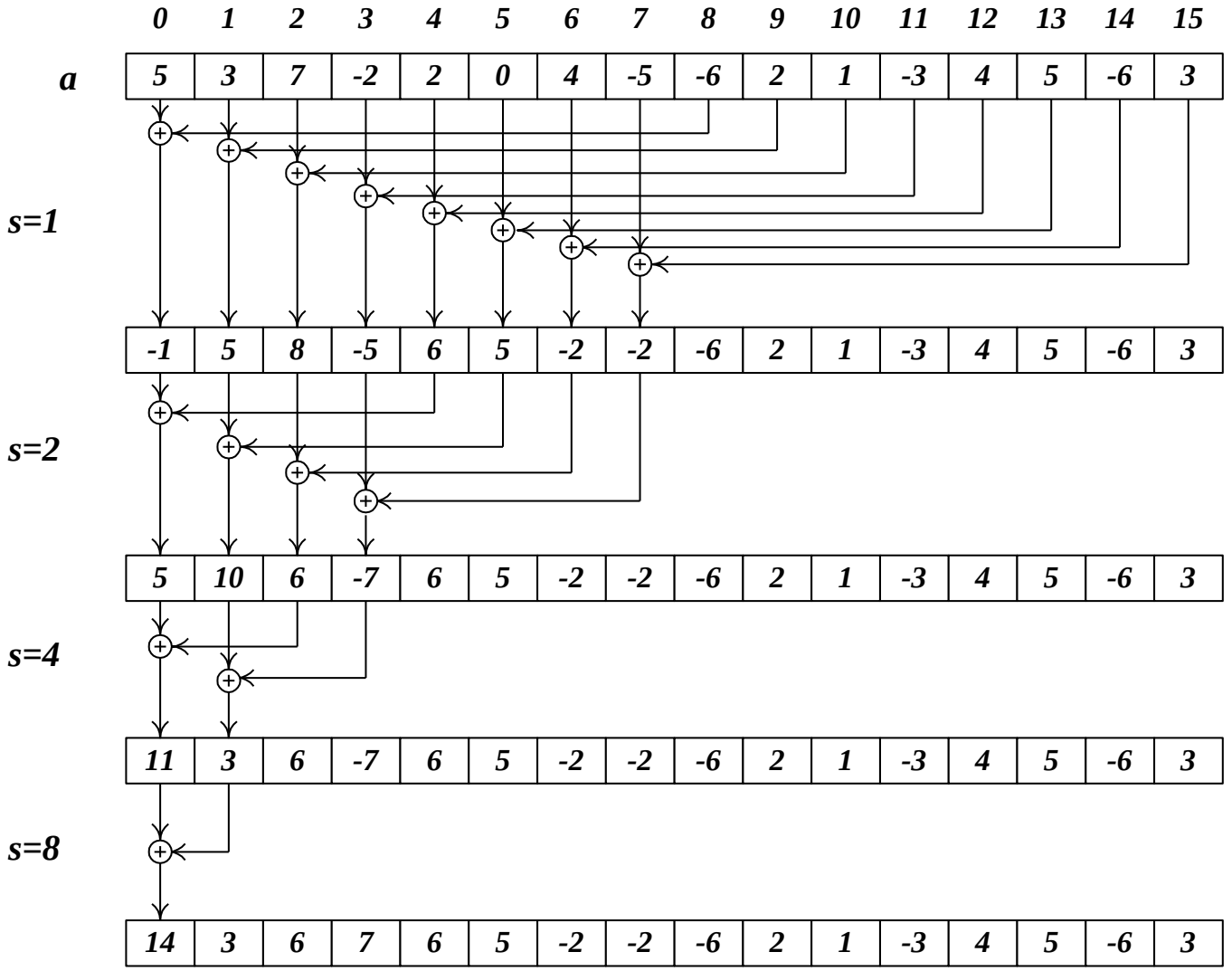
# Редукция, вариант 2

- 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<=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, 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, быстродействие

| **Вариант алгоритма** | **Время выполнения (миллисекунды)** |
|---|---|
| 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