Эффективный доступ к памяти

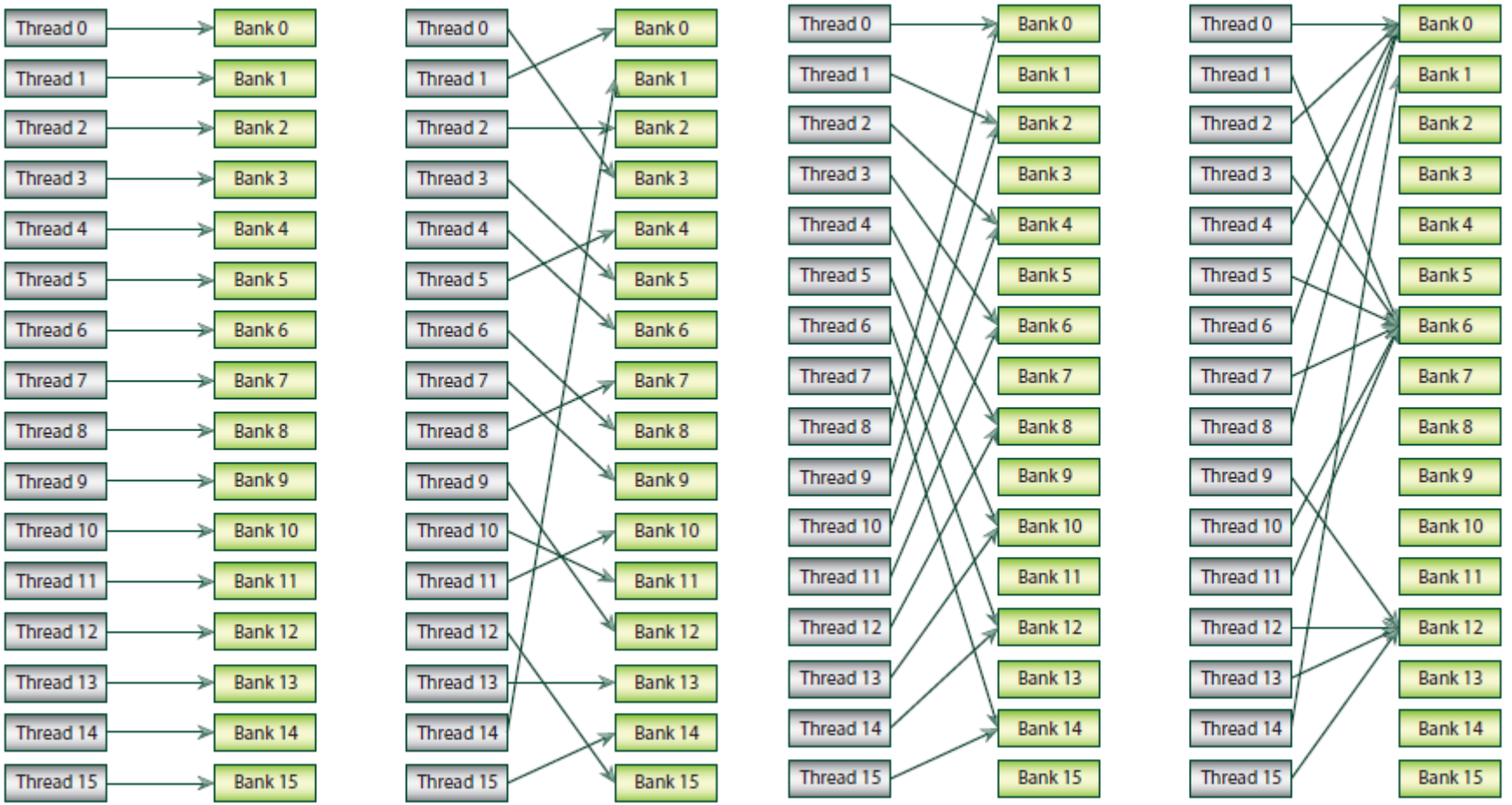
Разделяемая память

- Вся память разбита на 32 банка для повышения пропускной способности
- Подряд идущие 32-битные слова попадают в разные подряд идущие банки
- Все нити варпа обращаются в память совместно
- Каждый банк работает независимо от других
- Одновременно можно выполнить до 32 обращений к shared памяти

Разделяемая память

- Обращения к одному банку выполняются только последовательно
- Одновременный запрос из одного банка несколькими нитями называется конфликт банка
- Порядок конфликта – максимальное число обращений в один банк
- Конфликт 2 порядка снижает скорость доступа к памяти вдвое
- Обращение 32 нитей варпа конфликта не вызывает (!)

Разделяемая память



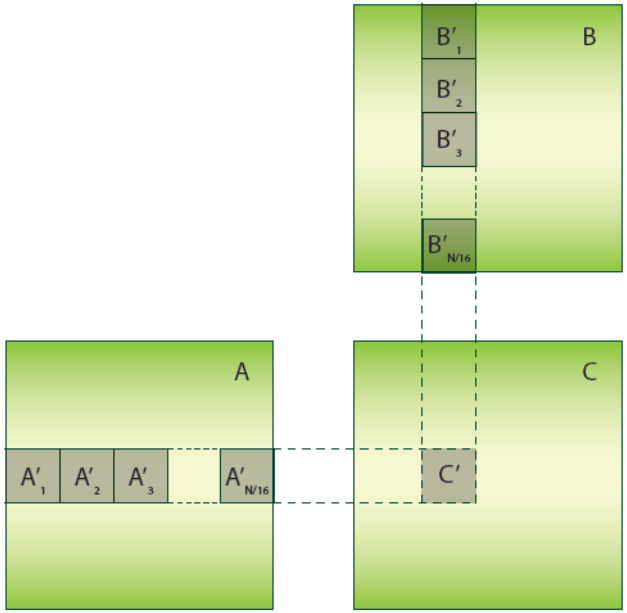
Разделяемая память

```
// Нет конфликтов
__shared__ float buf [128];
float v = buf [baseIndex + threadIdx.x];

// Конфликт 4-го порядка.
__shared__ char buf [128];
char v = buf [baseIndex + threadIdx.x];

// Конфликт 2-го порядка.
__shared__ short buf [128];
short v = buf [baseIndex + threadIdx.x];
```

Разделяемая память

$C = A A^T$

В разделяемой памяти нужно хранить две подматрицы 32x32




```
__shared__ float as [BLOCK_SIZE][BLOCK_SIZE];
__shared__ float ats [BLOCK_SIZE][BLOCK_SIZE];
// Загрузить подматрицы в разделяемую память.
as [ty][tx] = a [ia + n * ty + tx];
ats [ty][tx] = a [iat + n * ty + tx];
// Синхронизация, чтобы убедиться, что обе подматрицы загружены.
__syncthreads();
// Находим нужный элемент произведения подматриц
for (int k = 0; k < BLOCK_SIZE; k++)
sum += as [ty][k] * ats [tx][k];
```

Разделяемая память

```
__shared__ float as [BLOCK_SIZE][BLOCK_SIZE];
__shared__ float ats [BLOCK_SIZE][BLOCK_SIZE];
// Загрузить подматрицы в разделяемую память.
as [ty][tx] = a [ia + n * ty + tx];
ats [ty][tx] = a [iat + n * ty + tx];
// Синхронизация, чтобы убедиться, что обе подматрицы загружены.
__syncthreads();
// Находим нужный элемент произведения подматриц
for (int k = 0; k < BLOCK_SIZE; k++)
sum += as [ty][k] * ats [tx][k];
```

Доступ осуществляется по столбцам транспонированной матрицы

Матрица имеет размер 16x16, столбец располагается в одном банке

Конфликт 16 порядка

Разделяемая память

Добавить в транспонированную матрицу фиктивный столбец

Конфликт банков	Без конфликта
596.60 мс	92.50 мс

(2048x2048 Tesla C2070)

Глобальная память

Выравнивание

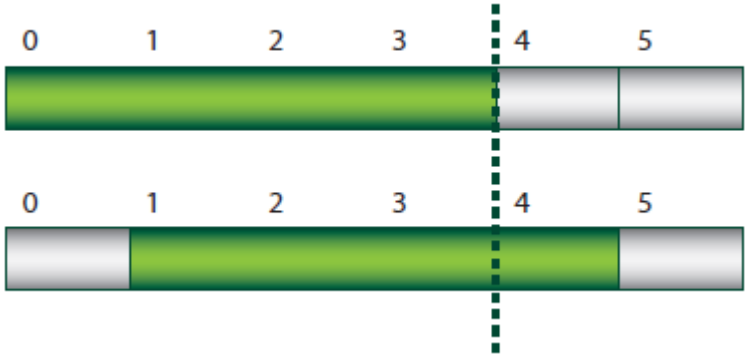
При чтении/записи используются
32-,64-,128-битные слова

Функции выделения памяти CUDA API возвращают адреса, выровненные по 256 байтам

Глобальная память

выровненный и

невыровненный блок



Аналогично для массивов – решается выделением фиктивного элемента / директивой выравнивания

```
struct __attribute__((aligned(16))) vec3
{
    float x, y, z;
};
```

Глобальная память

Объединение запросов

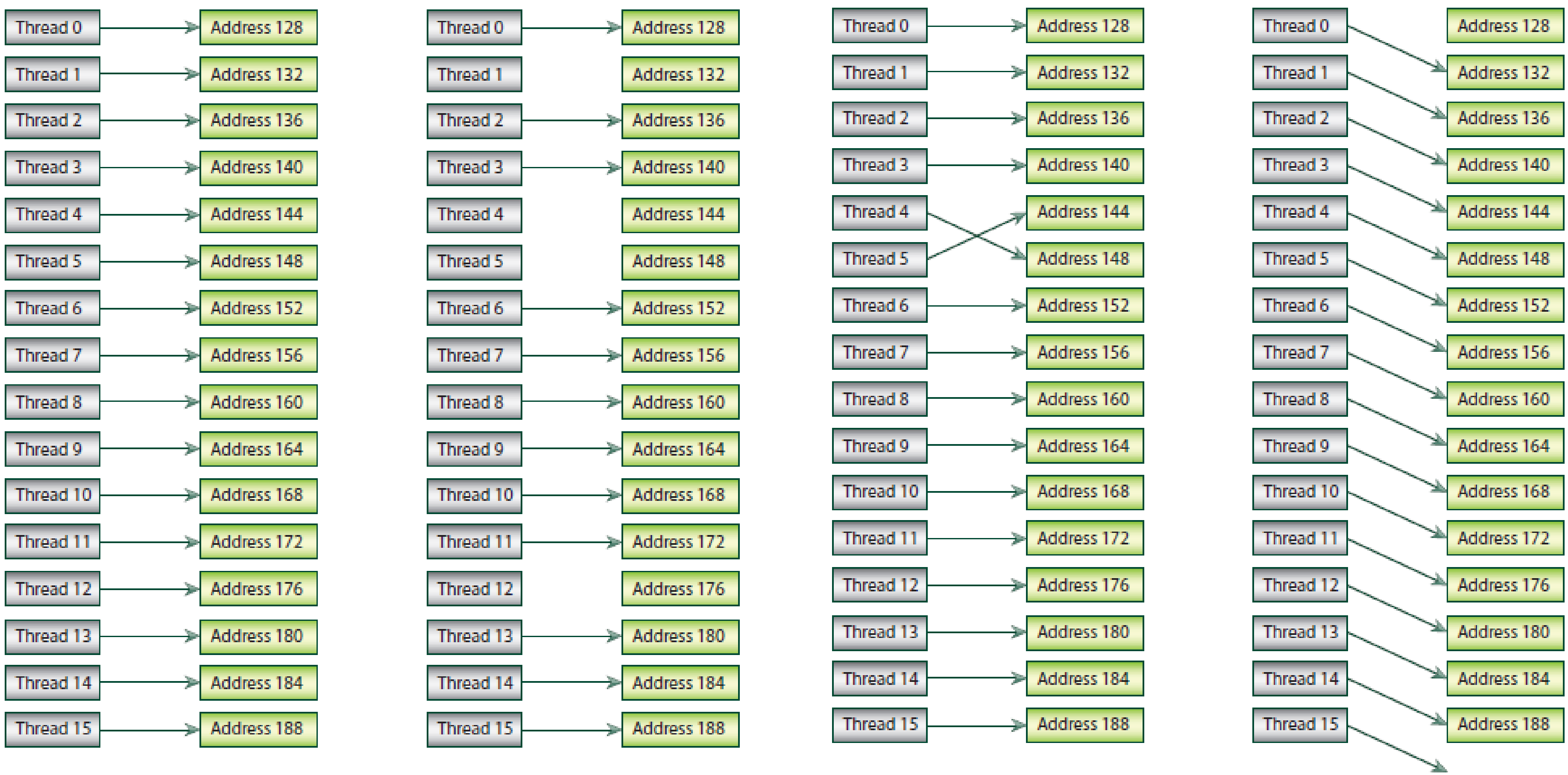
Возможность объединять запросы всех нитей полуварпа или варпа в одно обращение к непрерывному блоку памяти

Глобальная память

- Все нити должны обращаться к 32-битным словам, давая в результате один 64-байтовый блок, или к 64-битным словам, давая один 128-байтовый блок
- Полученный блок должен быть выровнен по своему размеру (адрес 64-байтового кратен 64)
- Все 16 слов, к которым обращаются нити, должны находиться внутри этого блока
- Нити должны обращаться к словам последовательно (допускается пропуск обращения)

compute capability 1.0, 1.1

Глобальная память



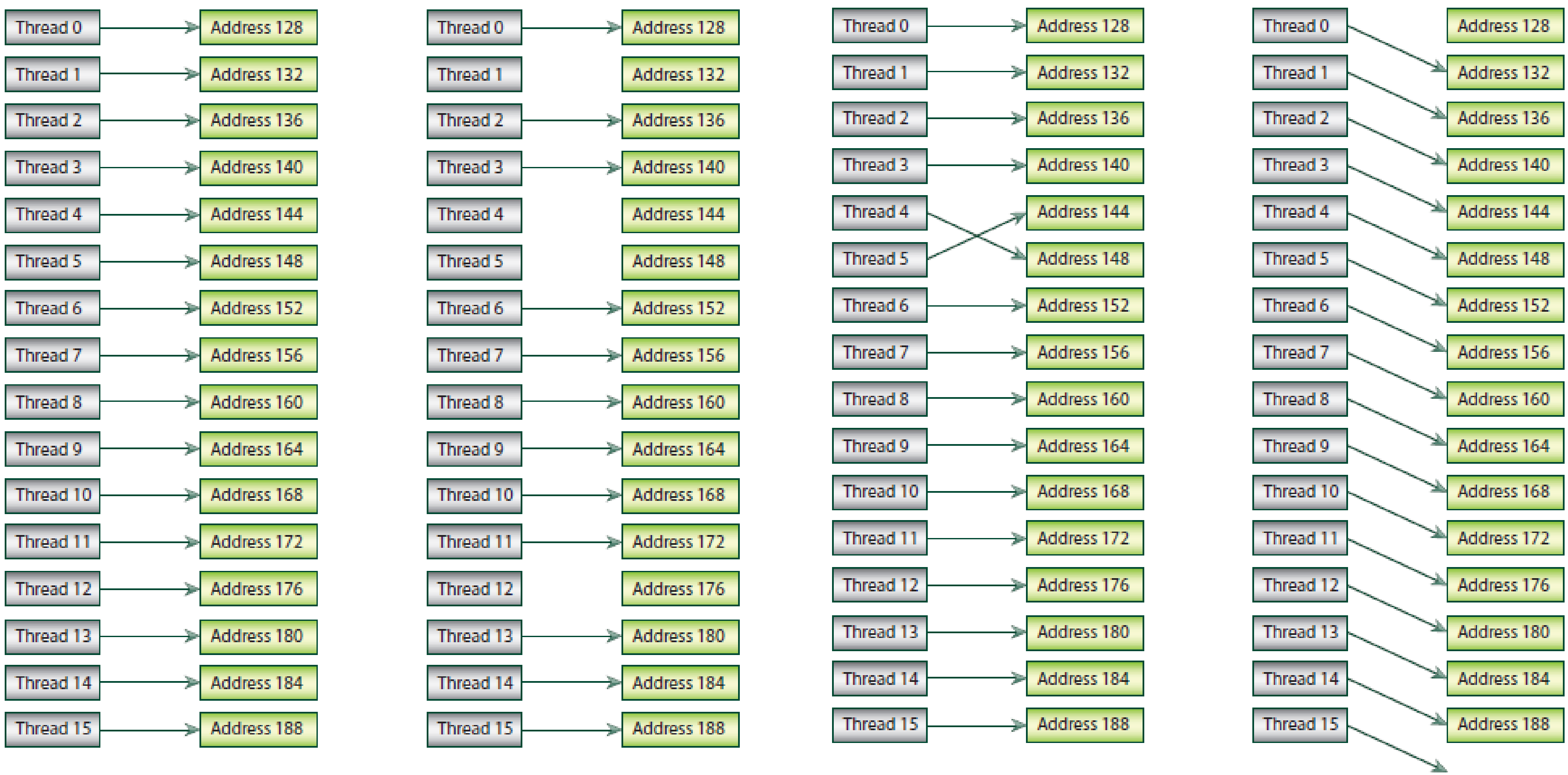
compute capability 1.0, 1.1

Глобальная память

Compute Capability 1.2, 1.3

- Слова к которым происходит обращение лежат в одном сегменте размером 32 байта (8-битовые слова), 64 байта (16-битовые слова), 128 байт (32- или 64-битовые слова)
- Блок выровнен по 32, 64, 128 байтам соответственно
- Порядок обращений роли не играет

Глобальная память



compute capability 1.2, 1.3

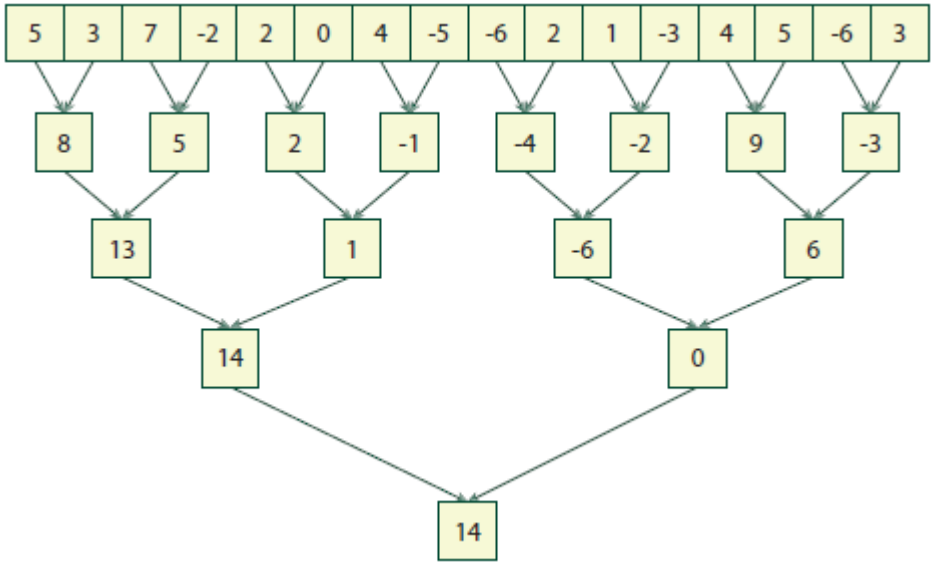
Редукция

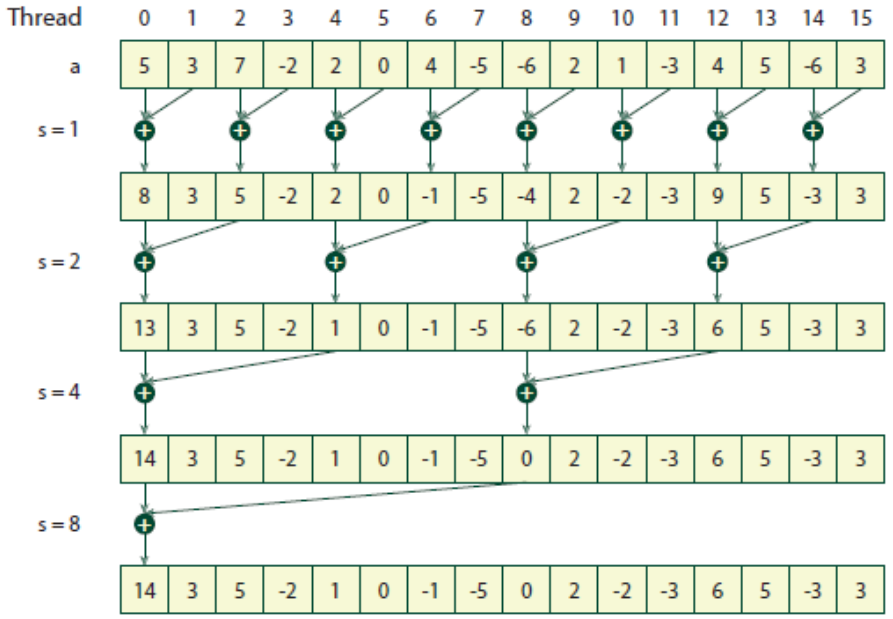
Массив $[a_0, a_1, \ldots, a_n]$

Необходимо: найти сумму

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

- разобьем массив на части и поставим в соответствие блок
- продолжим делить подмассивы на пары элементов между нитями





```
__global__ void reduce1 (int* inData, int* outData)
{
   // Суммируемые данные в разделяемой памяти.
   __shared__ int data [BLOCK_SIZE];
   int tid = threadIdx.x;
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   // Загрузить данные в разделяемую память.
   data [tid] = inData [i];
   // Заблокировать нити блока до окончания загрузки данных.
   __syncthreads ();
   // Выполнять попарное суммирование.
   for ( int s = 1; s < blockDim.x; s *= 2 )
   {
      // Проверить, участвует ли нить на данном шаге.
      if ( tid % (2 * s) == 0 )
         data [tid] += data [tid + s];
      __syncthreads ();
   }
   // Первая нить записывает итоговую сумму.
   if ( tid == 0 )
      outData [blockIdx.x] = data [0];
}
```

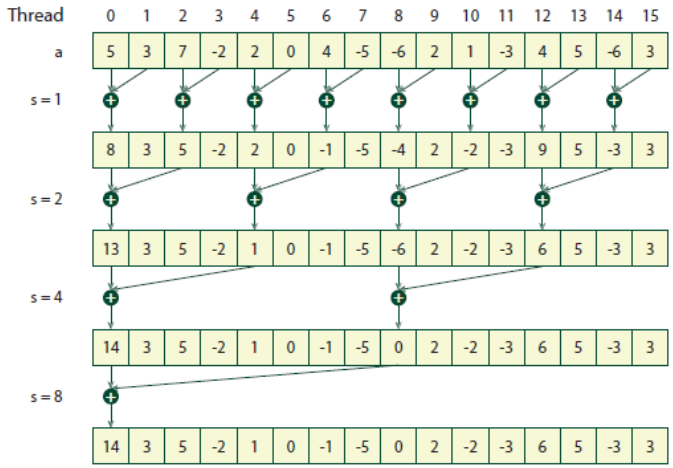
```
__global__ void reduce1 (int* inData, int* outData)
{
   // Суммируемые данные в разделяемой памяти.
   __shared__ int data [BLOCK_SIZE];
   int tid = threadIdx.x;
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   // Загрузить данные в разделяемую память.
   data [tid] = inData [i];
   // Заблокировать нити блока до окончания загрузки данных.
   __syncthreads ();
   // Выполнять попарное суммирование.
   for ( int s = 1; s < blockDim.x; s *= 2 )
   {
      // Проверить, участвует ли нить на данном шаге.
      if ( tid % (2 * s) == 0 )
         data [tid] += data [tid + s];
      __syncthreads ();
   }
   // Первая нить записывает итоговую сумму.
   if ( tid == 0 )
      outData [blockIdx.x] = data [0];
}
```

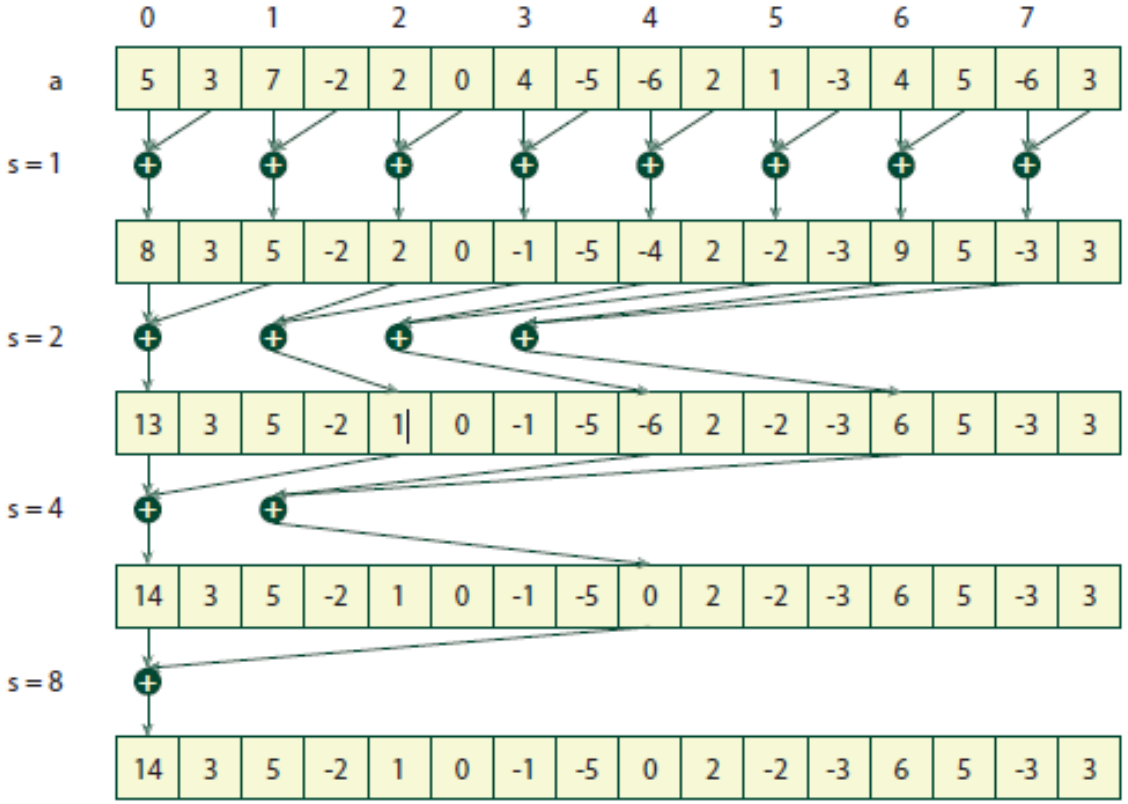
```
for ( int s = 1; s < blockDim.x; s *= 2 )
  {
    // Проверить, участвует ли нить на данном шаге.
    if ( tid % (2 * s) == 0 )
      data [tid] += data [tid + s];
    __syncthreads ();
  }
```

- нити одного варпа переходят в различные ветви условного оператора.

- все нити варпа выполняют все ветви

- как следствие – избыточные вычисления


Thread 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
a 5 3 7 -2 2 0 4 -5 -6 2 1 -3 4 5 -6 3
s = 1 8 3 5 -2 2 0 -1 -5 -4 2 -2 -3 9 5 -3 3
s = 2 13 3 5 -2 1 0 -1 -5 -6 2 -2 -3 6 5 -3 3
s = 4 14 3 5 -2 1 0 -1 -5 0 2 -2 -3 6 5 -3 3
s = 8 14 3 5 -2 1 0 -1 -5 0 2 -2 -3 6 5 -3 3


0 1 2 3 4 5 6 7
a 5 3 7 -2 2 0 4 -5 -6 2 1 -3 4 5 -6 3
s = 1 8 3 5 -2 2 0 -1 -5 -4 2 -2 -3 9 5 -3 3
s = 2 13 3 5 -2 1 0 -1 -5 -6 2 -2 -3 6 5 -3 3
s = 4 14 3 5 -2 1 0 -1 -5 0 2 -2 -3 6 5 -3 3
s = 8 14 3 5 -2 1 0 -1 -5 0 2 -2 -3 6 5 -3 3

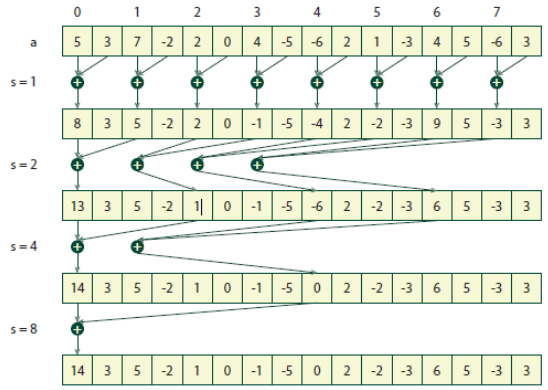
```
__global__ void reduce2 (int* inData, int* outData)
{
    // Суммируемые данные в разделяемой памяти.
    __shared__ int data [BLOCK_SIZE];
    int tid = threadIdx.x;
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    // Загрузить данные в разделяемую память.
    data [tid] = inData [i];
    // Заблокировать нити блока до окончания загрузки данных.
    __syncthreads ();
    for (int s = 1; s < blockDim.x; s*= 2)
    {
        // Проверить участвует ли нить в суммировании.
        int index = 2 * s * tid;
        if (index < blockDim.x)
            data[index] += data[index + s];
        __syncthreads ();
    }
    // Первая нить записывает итоговую сумму.
    if ( tid == 0 )
        outData [blockIdx.x] = data [0];
}
```

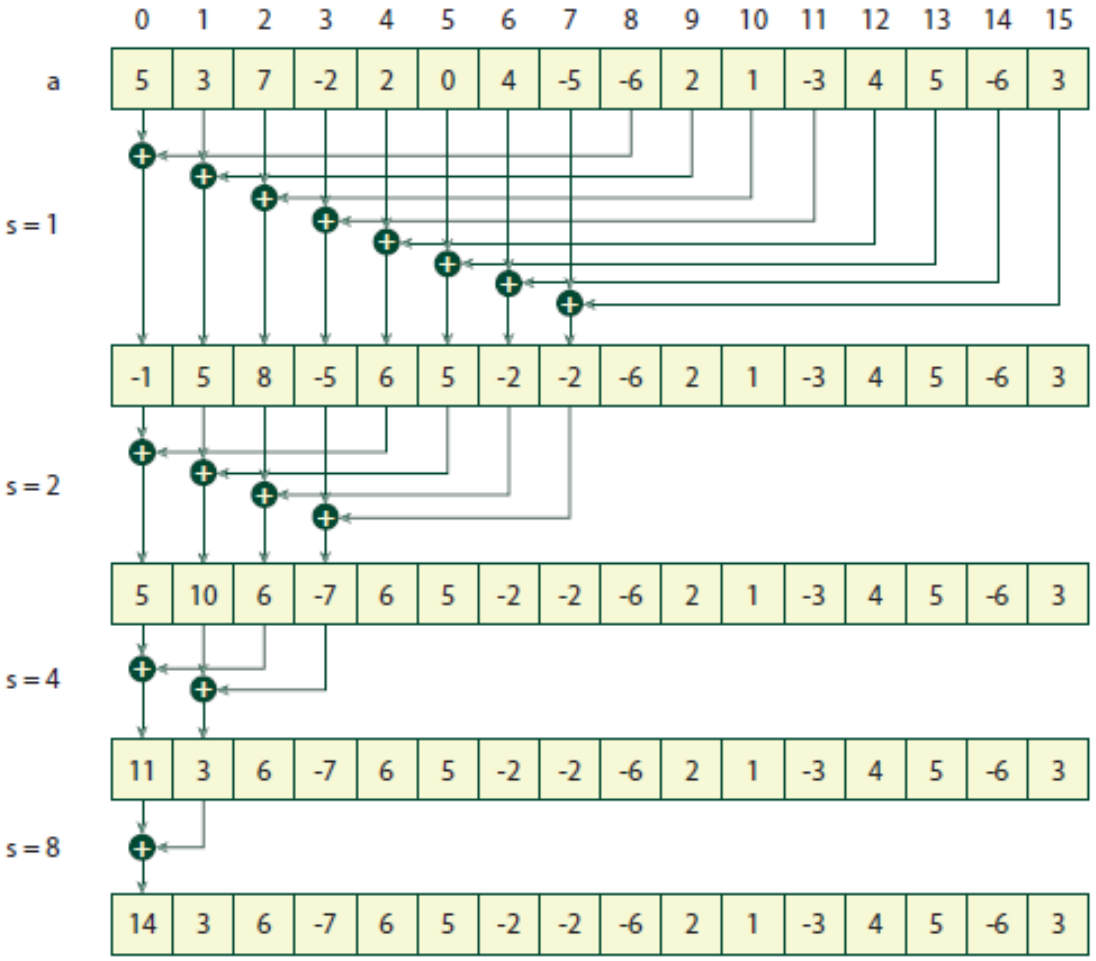
+ почти полностью избавились от ветвления

- много конфликтов по банкам

```
for (int s = 1; s < blockDim.x; s*= 2)
{
    …
    data[index] += data[index + s];
}
```

Для каждого следующего шага цикла степень конфликта удваивается


0 1 2 3 4 5 6 7
a
5 3 7 -2 2 0 4 -5 -6 2 1 -3 4 5 -6 3
s = 1
8 3 5 -2 2 0 -1 -5 -4 2 -2 -3 9 5 -3 3
s = 2
13 3 5 -2 1 0 -1 -5 -6 2 -2 -3 6 5 -3 3
s = 4
14 3 5 -2 1 0 -1 -5 0 2 -2 -3 6 5 -3 3
s = 8
14 3 5 -2 1 0 -1 -5 0 2 -2 -3 6 5 -3 3


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
a
5 3 7 -2 2 0 4 -5 -6 2 1 -3 4 5 -6 3
s = 1
-1 5 8 -5 6 5 -2 -2 -6 2 1 -3 4 5 -6 3
s = 2
5 10 6 -7 6 5 -2 -2 -6 2 1 -3 4 5 -6 3
s = 4
11 3 6 -7 6 5 -2 -2 -6 2 1 -3 4 5 -6 3
s = 8
14 3 6 -7 6 5 -2 -2 -6 2 1 -3 4 5 -6 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;
   // Загрузить данные в разделяемую память.
   data [tid] = inData [i];
   __syncthreads ();
   for (int s = blockDim.x / 2; s > 0; s = s / 2)
   {
      if (tid < s)
         data [tid] += data [tid + s];
      __syncthreads ();
   }
   // Первая нить записывает итоговую сумму.
   if ( tid == 0 )
      outData [blockIdx.x] = data [0];
}
```

+ избавились от ветвления

+ избавились конфликтов по банкам

- на первой итерации простаивает половина нитей

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

```
__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;
   // Записать сумму первых двух элементов в разделяемую память.
   data [tid] = inData [i] + inData [i + blockDim.x];
   __syncthreads ();
   for (int s = blockDim.x / 2; s > 0; s = s / 2)
   {
      if ( tid < s )
         data [tid] += data [tid + s];
      __syncthreads ();
   }
   // Первая нить записывает итоговую сумму.
   if ( tid == 0 )
      outData [blockIdx.x] = data [0];
}
```

+ избавились от ветвления

+ избавились конфликтов по банкам

+ на первой итерации не простаивают нити

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

+ избавились от ветвления

+ избавились конфликтов по банкам

+ на первой итерации не простаивают нити

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

В одном варпе 32 нити, при s <= 32, в каждом блоке выполняется только один варп.

Все нити будут выполняться синхронно.

```
__global__ void reduce5 (int* inData, int* outData)
{
   volatile __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];
   __syncthreads ();
   for (int s = blockDim.x / 2; s > 32; s >>= 1)
   {
      if ( tid < s )
         data [tid] += data [tid + s];
      __syncthreads ();
   }
   // Раскрутить последние итерации.
   if ( tid < 32 )
   {
      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];
   }
   if ( tid == 0 )
      outData [blockIdx.x] = data [0];
}
```

Суммирование массива

1. Разбить на блоки по 512 элементов
2. Суммировать внутри каждого блока
3. В результате массив сумм элементов
4. Повторить 1. если размерность велика

Суммирование массива

Используемый вариант	Время в миллисекундах
reduce1	5.28
reduce2	2.52
reduce3	1.88
reduce4	0.99
reduce5	0.65