



# NVIDIA CUDA И OPENACC ЛЕКЦИЯ 6

Перепёлкин Евгений

# СОДЕРЖАНИЕ

## Лекция 6

- ▶ Оптимизация работы с разделяемой памятью
- ▶ Пример. Перемножение матриц
- ▶ Пример. Параллельная редукция

*Оптимизация работы с разделяемой памятью*

# РАЗДЕЛЯЕМАЯ ПАМЯТЬ. БАНКИ

Compute Capability 3.x

- ▶ 32 банка, ширина 8 Байт
  - ▶ пропускная способность 8 Байт за такт на SMX
  - ▶ варп (32 потока) считывает 256 Байт за такт на SMX
- ▶ Два режима доступа
  - ▶ 4-Байтовый `cudaSharedMemBankSizeFourByte` ( по умолчанию )
  - ▶ 8-Байтовый `cudaSharedMemBankSizeEightByte`
  - ▶ задается функцией `cudaDeviceSetSharedMemConfig ( )`

# РАЗДЕЛЯЕМАЯ ПАМЯТЬ. БАНКИ

8-Байтовый режим доступа



4 Байта = 32 бита

`__shared__ float A [ N ];`

`float x = A [ threadIdx.x ];`

# РАЗДЕЛЯЕМАЯ ПАМЯТЬ. БАНКИ

4-Байтовый режим доступа



4 Байта = 32 бита

shared float A [ N ];

float x = A [ threadIdx.x ];

# РАЗДЕЛЯЕМАЯ ПАМЯТЬ. БАНК КОНФЛИКТЫ

Compute Capability 3.x

- ▶ Банк конфликты возникают, когда:
  - ▶ две или более нитей одного варпа обращаются к разным 8-Байтовым словам, лежащим в одном банке
  - ▶ банк-конфликт имеет порядок  $N$  когда конфликтуют  $N$  нитей одного варпа
- ▶ Банк конфликтов нет, когда:
  - ▶ разные нити варпа обращаются к одному слову
  - ▶ разные нити варпа обращаются к различным байтам одного и того же слова

# РАЗДЕЛЯЕМАЯ ПАМЯТЬ. ПРИМЕР ДОСТУПА

Compute Capability 3.x

Нет банк конфликтов

В каждый банк по одному обращению

32 нити варпа



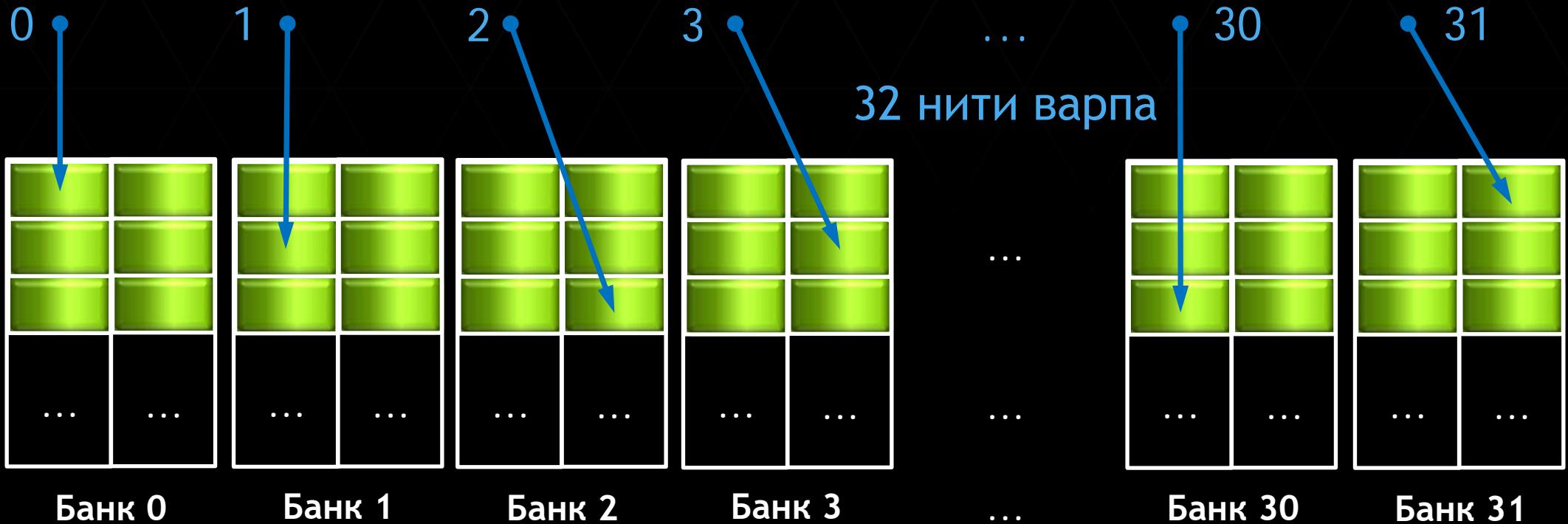
# РАЗДЕЛЯЕМАЯ ПАМЯТЬ. ПРИМЕР ДОСТУПА

Compute Capability 3.x

Нет банк конфликтов

В каждый банк по одному обращению

32 нити варпа



# РАЗДЕЛЯЕМАЯ ПАМЯТЬ. ПРИМЕР ДОСТУПА

Compute Capability 3.x

Нет банк конфликтов

Несколько обращение в один банк, но к одному и тому же слову



# РАЗДЕЛЯЕМАЯ ПАМЯТЬ. ПРИМЕР ДОСТУПА

Compute Capability 3.x

Банк конфликт 2-го порядка

Обращение к двум разным словам, лежащим в одном банке



# РАЗДЕЛЯЕМАЯ ПАМЯТЬ. ПРИМЕР ДОСТУПА

Compute Capability 3.x

Банк конфликт 3-го порядка

Обращение к трем разным словам, лежащим в одном банке



*Пример. Перемножение матриц*

$$C = AB,$$

$$c_{i,j} = \sum_{k=0}^{N-1} a_{i,k} b_{k,j},$$

$$a_{i,j} = 2j + i, \quad b_{i,j} = j - i,$$

$$i, j = 0, \dots, N - 1$$

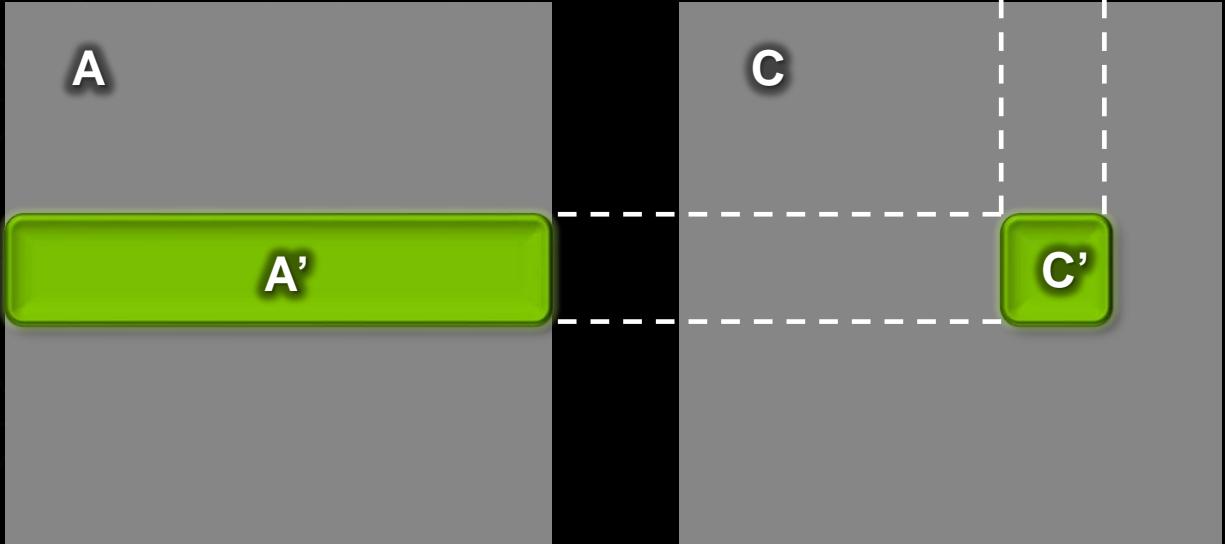
$$N \times N = 2048 \times 2048, \text{BLOCK\_SIZE} = 32$$

( tx, ty ) - ( 32, 32 ) - нити внутри блока

( bx, by ) - ( 64, 64 ) - число блоков

## ПРИМЕР ПЕРЕНОЖЕНИЯ МАТРИЦ

## ВАРИАНТ «GLOBAL»



# КОД ПРОГРАММЫ. ВАРИАНТ «GLOBAL»

## Часть 1. ФУНКЦИЯ-ЯДРО

```
#define BLOCK_SIZE 32

__global__ void kernel_global ( float *a, float *b, int n, float *c )
{
    int bx = blockIdx.x; // номер блока по x
    int by = blockIdx.y; // номер блока по y
    int tx = threadIdx.x; // номер нити в блоке по x
    int ty = threadIdx.y; // номер нити в блоке по y
    float sum = 0.0f;
    int ia = n * ( BLOCK_SIZE * by + ty ); // номер строки из A'
    int ib = BLOCK_SIZE * bx + tx; // номер столбца из B'
    int ic = ia + ib; // номер элемента из C'
    // вычисление элемента матрицы C
    for ( int k = 0; k < n; k++ ) sum += a[ia + k] * b[ib + k * n];
    c[ic] = sum;
}
```

# КОД ПРОГРАММЫ. ВАРИАНТ «GLOBAL»

## Часть 2. Функция main

```
int main()
{int N = 2048;
 int m, n, k;
 // создание переменных-событий
float timerValueGPU, timerValueCPU;
cudaEvent_t start, stop;
cudaEventCreate (&start); cudaEventCreate (&stop);

int numBytes = N * N * sizeof (float );
float *adev, *bdev, *cdev, *a, *b, *c, *cc, *bT;
// выделение памяти на host
a = (float *) malloc (numBytes); //матрица A
b = (float *) malloc (numBytes); //матрица B
bT = (float *) malloc (numBytes); //транспонированная матрица B
c = (float *) malloc (numBytes); //матрица C для GPU-варианта
cc = (float *) malloc (numBytes); //матрица C для CPU-варианта
```

# КОД ПРОГРАММЫ. ВАРИАНТ «GLOBAL»

## Часть 3. Функция main

```
// задание матрицы A, B и транспонированной матрицы BT
for ( n = 0; n < N; n++ )
{for ( m = 0; m < N; m++ )
{a[m + n * N] = 2.0f * m + n; b[m + n * N] = m - n; bT[m + n * N] = n - m;
}
}
// задание сетки нитей и блоков
dim3 threads ( BLOCK_SIZE, BLOCK_SIZE );
dim3 blocks ( N / threads.x, N / threads.y );
// выделение памяти на GPU
cudaMalloc ( (void**)&adev, numBytes );
cudaMalloc ( (void**)&bdev, numBytes );
cudaMalloc ( (void**)&cdev, numBytes );
```

# КОД ПРОГРАММЫ. ВАРИАНТ «GLOBAL»

## Часть 4. Функция main

```
// ----- GPU-вариант -----
// копирование матриц A и B с host на device
cudaMemcpy ( adev, a, numBytes, cudaMemcpyHostToDevice );
cudaMemcpy ( bdev, b, numBytes, cudaMemcpyHostToDevice );
// запуск таймера
cudaEventRecord ( start, 0 );
// запуск функции-ядра
kernel_global <<< blocks, threads >>> ( adev, bdev, N, cdev );
// оценка времени вычисления GPU-варианта
cudaThreadSynchronize ();
cudaEventRecord ( stop, 0 );
cudaEventSynchronize ( stop );
cudaEventElapsedTime ( &timerValueGPU, start, stop );
printf ( "\n GPU calculation time %f msec\n", timerValueGPU );
// копирование, вычисленной матрицы C с device на host
cudaMemcpy ( c, cdev, numBytes, cudaMemcpyDeviceToHost );
```

# КОД ПРОГРАММЫ. ВАРИАНТ «GLOBAL»

## Часть 5. Функция main

```
// ----- CPU-вариант -----
// запуск таймера
cudaEventRecord (start, 0);
// вычисление матрицы C
for ( n = 0; n < N; n++ )
{for ( m = 0; m < N; m++ )
{cc[m+n*N] = 0.f;
 for( k = 0; k < N; k++ ) cc[m+n*N] += a[k+n*N] * bT[k+m*N]; // bT !!!
}
}
// оценка времени вычисления CPU-варианта
cudaEventRecord (stop, 0);
cudaEventSynchronize (stop);
cudaEventElapsedTime (&timerValueCPU, start, stop);
printf ("\n CPU calculation time %f msec\n",timerValueCPU);
printf ("\n Rate %f x\n",timerValueCPU/timerValueGPU);
```

# КОД ПРОГРАММЫ. ВАРИАНТ «GLOBAL»

## Часть 6. Функция main

```
// освобождение памяти на GPU и CPU
cudaFree ( adev );
cudaFree ( bdev );
cudaFree ( cdev );
free ( a );
free ( b );
free ( bT );
free ( c );
free ( cc );
// уничтожение переменных-событий
cudaEventDestroy ( start );
cudaEventDestroy ( stop );

return 0;
}
```

# РЕЗУЛЬТАТ. ВАРИАНТ «GLOBAL»

CPU Core2 Quad Q8300 2.5 ГГц ICC x64 1-ядро GPU Tesla K40c CUDA 6.0

| Precision            | : float   | double  |
|----------------------|-----------|---------|
| GPU calculation time | : 134 ms  | 220 ms  |
| CPU calculation time | : 4622 ms | 9154 ms |
| Rate                 | : 34 x    | 41 x    |

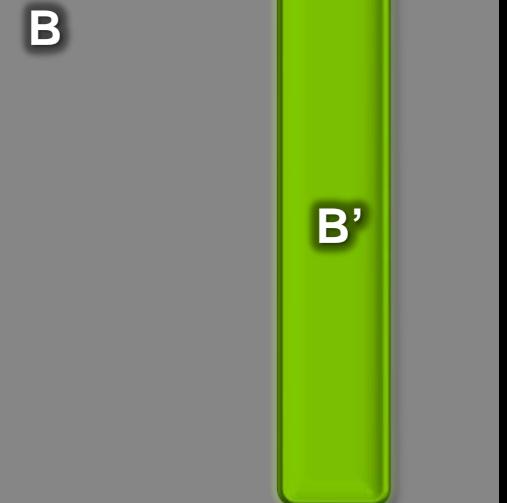
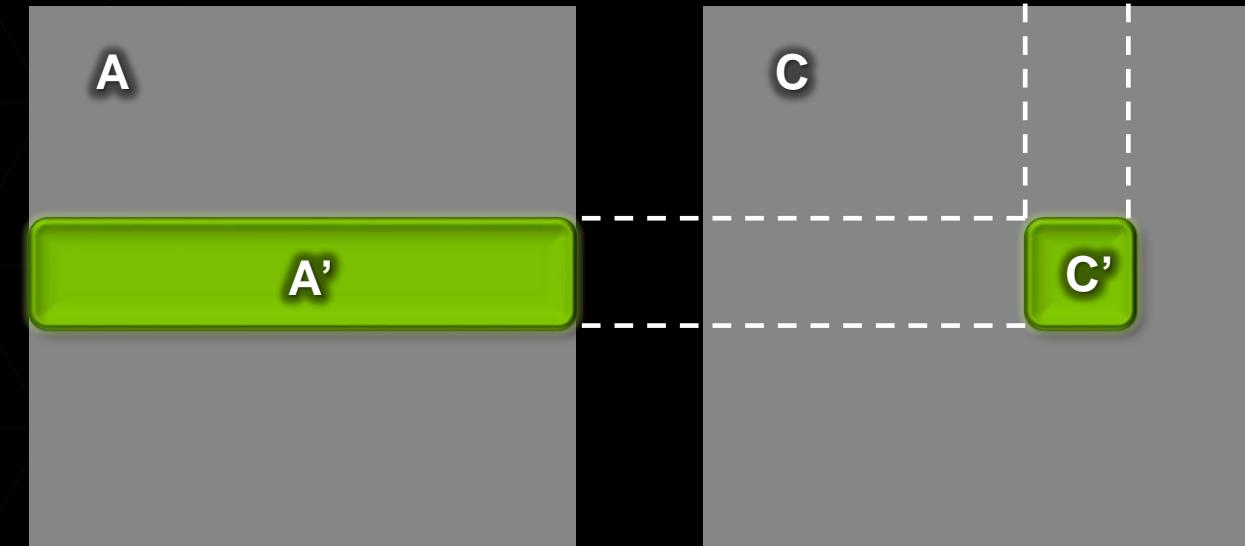
При вычислении

$$c_{i,0}, c_{i,1}, \dots, c_{i,N-1}$$

N-раз считывается строка

$$a_{i,0}, a_{i,1}, \dots, a_{i,N-1}$$

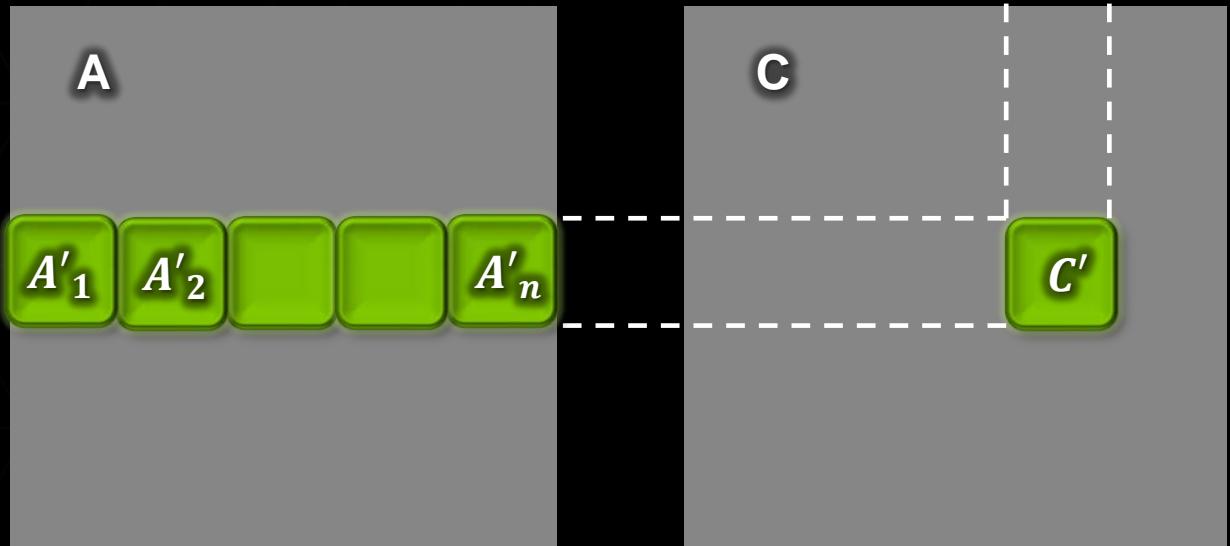
## НЕДОСТАТОК ВАРИАНТА «GLOBAL»



$$C' = A'_1 B'_1 + A'_2 B'_2 + \cdots + A'_n B'_n = \sum_{i=1}^n A'_i B'_i,$$

$$n = \frac{N}{BLOCK\_SIZE}$$

## ВАРИАНТ «SMEM»



# КОД ПРОГРАММЫ. ВАРИАНТ «SMEM-1»

## Часть 1. ФУНКЦИЯ-ЯДРО

```
__global__ void kernel_smem_1 ( 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, aEnd = aBegin + n - 1;
    int bBegin = BLOCK_SIZE * bx, aStep = BLOCK_SIZE, bStep = BLOCK_SIZE * n;
    float sum = 0.0f;
    __shared__ float as [BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float bs [BLOCK_SIZE][BLOCK_SIZE];
    for ( int ia = aBegin, ib = bBegin; ia <= aEnd; ia += aStep, ib += bStep )
    {as [tx][ty] = a [ia + n * ty + tx]; bs [tx][ty] = b [ib + n * ty + tx];
     __syncthreads ();
     for ( int k = 0; k < BLOCK_SIZE; k++ ) sum += as [k][ty] * bs [tx][k];
     __syncthreads ();
    }
    c [aBegin + bBegin + ty * n + tx] = sum;
}
```

# РЕЗУЛЬТАТ. ВАРИАНТ «SMEM-1»

CPU Core2 Quad Q8300 2.5 ГГц ICC x64 1-ядро GPU Tesla K40c CUDA 6.0

| Precision             | : float | double  |
|-----------------------|---------|---------|
| GPU calculation time: | 150 ms  | 476 ms  |
| CPU calculation time: | 4622 ms | 9154 ms |
| Rate                  | : 30 x  | 19 x    |

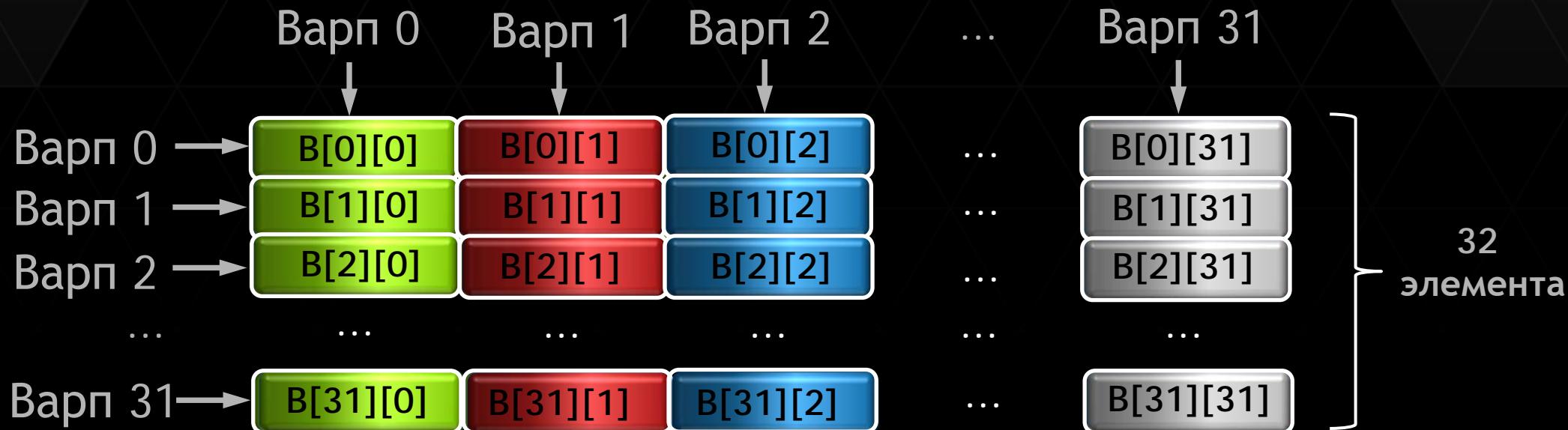
```
as [tx][ty] = a [ia + n * ty + tx]; // копирование из глобальной  
bs [tx][ty] = b [ib + n * ty + tx]; // в разделяемую память
```

(\*) ind = tx + ty \* BLOCK\_SIZE - линейный номер нити

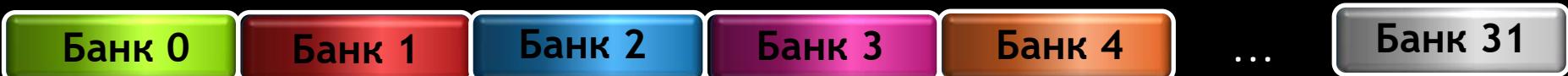
(\*\*) indM = ty + tx \* BLOCK\_SIZE - линейный номер элементов  
в матрицах «as» и «bs»

# БАНК КОНФЛИКТЫ

`__shared__ double в [32][32];`

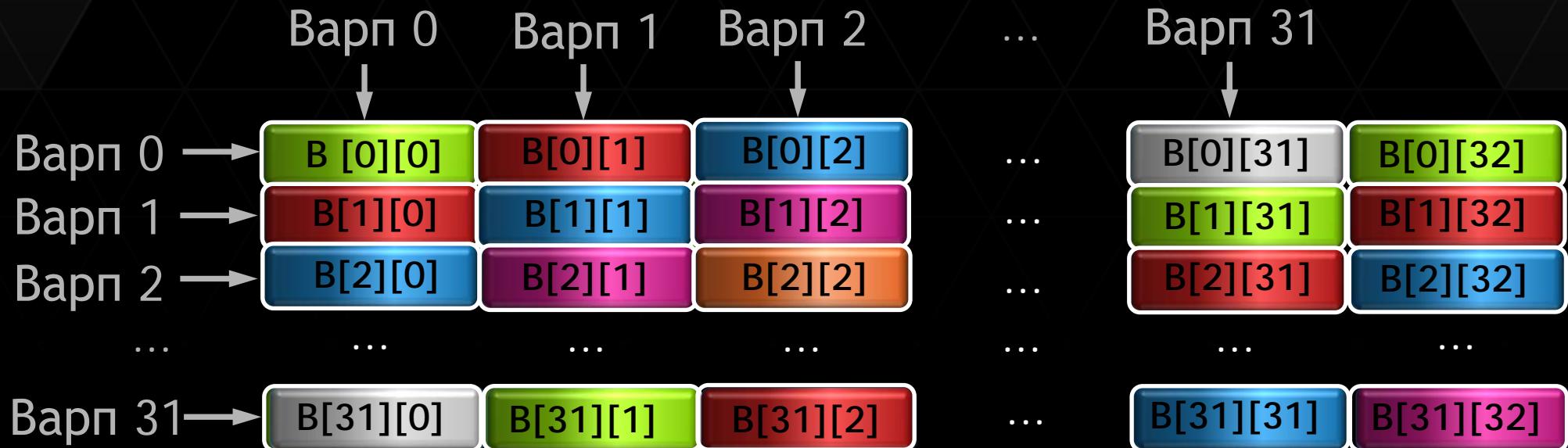


Обращение по столбцу – дает банк конфликт 32 порядка



# НЕТ БАНК КОНФЛИКТОВ

`__shared__ double в [32][33];`



Обращение по столбцу и по строке без банк конфликтов

Банк 0    Банк 1    Банк 2    Банк 3    Банк 4    ...    Банк 31

# РЕЗУЛЬТАТ. ВАРИАНТ «SMEM-2»

CPU Core2 Quad Q8300 2.5 ГГц ICC x64 1-ядро GPU Tesla K40c CUDA 6.0

| Precision             | : float     | double (4B) | double (8B) |
|-----------------------|-------------|-------------|-------------|
| GPU calculation time: | 56 ms (150) | 87 ms (476) | 62 ms (476) |
| CPU calculation time: | 4622 ms     | 9154 ms     | 9154 ms     |
| Rate                  | : 82 x      | 105 x       | 147 x       |

В «функции-ядре» строки (SMEM-1):

```
__shared__ float as [BLOCK_SIZE][BLOCK_SIZE];  
__shared__ float bs [BLOCK_SIZE][BLOCK_SIZE];
```

Заменили на строки (SMEM-2):

```
__shared__ float as [BLOCK_SIZE][BLOCK_SIZE + 1];  
__shared__ float bs [BLOCK_SIZE][BLOCK_SIZE + 1];
```

# КОД ПРОГРАММЫ. ВАРИАНТ «SMEM-3»

## ФУНКЦИЯ-ЯДРО «SMEM-3»

```
__shared__ float as [BLOCK_SIZE][BLOCK_SIZE];
__shared__ float bs [BLOCK_SIZE][BLOCK_SIZE];
```

Вместо строк (SMEM-1, SMEM-2):

```
as [tx][ty] = a [ia + n * ty + tx];
bs [tx][ty] = b [ib + n * ty + tx];
sum += as [k][ty] * bs [tx][k];
```

Поставим стоки (SMEM-3):

```
as [ty][tx] = a [ia + n * ty + tx];
bs [ty][tx] = b [ib + n * ty + tx];
sum += as [ty][k] * bs [k][tx];
```

ind = tx + ty \* BLOCK\_SIZE – линейный номер нити

indM = tx + ty \* BLOCK\_SIZE – линейный номер элементов в матрицах «as» и «bs»

# РЕЗУЛЬТАТ. ВАРИАНТ «SMEM-3»

CPU Core2 Quad Q8300 2.5 ГГц ICC x64 1-ядро GPU Tesla K40c CUDA 6.0

|                       |            |                 |
|-----------------------|------------|-----------------|
| Precision             | : float    | double (4B, 8B) |
| GPU calculation time: | 46 ms (56) | 80 ms (87, 62)  |
| CPU calculation time: | 4622 ms    | 9154 ms         |
| Rate                  | : 100 x    | 114 x           |

# ПАРАЛЛЕЛИЗМ ПО НИТЯМ И ПО ИНСТРУКЦИЯМ

## Thread-Level Parallelism (TLP)

Нить 0

$$X[0] = F(0)$$
$$X[0] = X[0] + G$$
$$X[0] = X[0] * H$$

Нить 1

$$X[1] = F(1)$$
$$X[1] = X[1] + G$$
$$X[1] = X[1] * H$$

Нить 2

$$X[2] = F(2)$$
$$X[2] = X[2] + G$$
$$X[2] = X[2] * H$$

время

Три независимых операции

Instruction-Level  
Parallelism (ILP)

Нить

$$X[0] = F(0)$$
$$X[1] = F(1)$$
$$X[2] = F(2)$$
$$X[0] = X[0] + G$$
$$X[1] = X[1] + G$$
$$X[2] = X[2] + G$$
$$X[0] = X[0] * H$$
$$X[1] = X[1] * H$$
$$X[2] = X[2] * H$$

инструкции

# КОД ПРОГРАММЫ. ВАРИАНТ «SMEM-4»

## ФУНКЦИЯ-ЯДРО «SMEM-4»

Вместо строк ( SMEM-3 ):

```
float sum = 0.0f;  
as [ty][tx] = a [ia + n * ty + tx]; bs [ty][tx] = b [ib + n * ty + tx];  
sum += as [ty][k] * bs [k][tx];  
c [aBegin + bBegin + ty * n + tx] = sum;
```

Поставим строки ( SMEM-4 ):

```
float sum1 = 0.0f, sum2 = 0.0f;  
as [ty][tx] = a [ia + n * ty + tx];  
bs [ty][tx] = b [ib + n * ty + tx];  
as [ty + 16][tx] = a [ia + n * ( ty + 16 ) + tx];  
bs [ty + 16][tx] = b [ib + n * ( ty + 16 ) + tx];  
sum1 += as [ty][k] * bs [k][tx];  
sum2 += as [ty + 16][k] * bs [k][tx];  
c [aBegin + bBegin + ty * n + tx] = sum1;  
c [aBegin + bBegin + ( ty + 16 ) * n + tx] = sum2;
```

# КОД ПРОГРАММЫ. ВАРИАНТ «SMEM-4»

## Функция main

Добавим строки для блока нитей ( SMEM-4 ):

```
dim3 threads_4 ( BLOCK_SIZE, BLOCK_SIZE / 2 );
```

для запуска новой «функции-ядра» ( SMEM-4 ):

```
kernel_smem_4 <<< blocks, threads_4 >>> ( adev, bdev, N, cdev );
```

# РЕЗУЛЬТАТ. ВАРИАНТ «SMEM-4»

CPU Core2 Quad Q8300 2.5 ГГц ICC x64 1-ядро GPU Tesla K40c CUDA 6.0

|                       |            |                 |
|-----------------------|------------|-----------------|
| Precision             | : float    | double (4B, 8B) |
| GPU calculation time: | 33 ms (46) | 50 ms (80, 80)  |
| CPU calculation time: | 4622 ms    | 9154 ms         |
| Rate                  | : 140 x    | 183 x           |

# КОД ПРОГРАММЫ. ВАРИАНТ «SMEM-5»

## ФУНКЦИЯ-ЯДРО «SMEM-5»

Добавим новые строки при копировании данных:

```
float sum1 = 0.0f, sum2 = 0.0f, sum3 = 0.0f, sum4 = 0.0f;  
as [ty][tx] = a [ia + n * ty + tx];  
bs [ty][tx] = b [ib + n * ty + tx];  
as [ty + 8][tx] = a [ia + n * ( ty + 8 ) + tx];  
bs [ty + 8][tx] = b [ib + n * ( ty + 8 ) + tx];  
as [ty + 16][tx] = a [ia + n * ( ty + 16 ) + tx];  
bs [ty + 16][tx] = b [ib + n * ( ty + 16 ) + tx];  
as [ty + 24][tx] = a [ia + n * ( ty + 24 ) + tx];  
bs [ty + 24][tx] = b [ib + n * ( ty + 24 ) + tx];
```

# КОД ПРОГРАММЫ. ВАРИАНТ «SMEM-5»

## ФУНКЦИЯ-ЯДРО «SMEM-5»

Изменения при перемножении матриц:

```
sum1 += as [ty][k] * bs [k][tx];
sum2 += as [ty + 8][k] * bs [k][tx];
sum3 += as [ty + 16][k] * bs [k][tx];
sum4 += as [ty + 24][k] * bs [k][tx];
```

при сохранении данных:

```
c [aBegin + bBegin + ty * n + tx] = sum1;
c [aBegin + bBegin + ( ty + 8 ) * n + tx] = sum2;
c [aBegin + bBegin + ( ty + 16 ) * n + tx] = sum3;
c [aBegin + bBegin + ( ty + 24 ) * n + tx] = sum4;
```

# КОД ПРОГРАММЫ. ВАРИАНТ «SMEM-5»

## Функция «main»

Добавим строки для блока нитей ( SMEM-5):

```
dim3 threads_5 ( BLOCK_SIZE, BLOCK_SIZE / 4);
```

для новой «функции-ядра» ( SMEM-5 ):

```
kernel_smem_5 <<< blocks, threads_5 >>> ( adev, bdev, N, cdev );
```

# РЕЗУЛЬТАТ. ВАРИАНТ «SMEM-5»

CPU Core2 Quad Q8300 2.5 ГГц ICC x64 1-ядро GPU Tesla K40c CUDA 6.0

Precision : float double (4B, 8B)

GPU calculation time: 26 ms (33) 45.5 ms (50, 50)

CPU calculation time: 4622 ms 9154 ms

Rate : 177 x 201 x

Вариант «SMEM-3»

GPU calculation time: 46 ms 80 ms (80, 80)

Rate : 100 x 114 x

## Сравнение вариантов

|        | Global | SMEM-1 | SMEM-2 | SMEM-3 | SMEM-4 | SMEM-5  |
|--------|--------|--------|--------|--------|--------|---------|
| float  | 134 мс | 150 мс | 56 мс  | 46 мс  | 33 мс  | 26 мс   |
|        | 34 x   | 30 x   | 82 x   | 100 x  | 140 x  | 177 x   |
| double | 220 мс | 476 мс | 62 мс* | 80 мс  | 50 мс  | 45.5 мс |
|        | 41 x   | 19 x   | 147 x* | 114 x  | 183 x  | 201 x   |

\* 8-Байтовый режим доступа

*Пример. Параллельная редукция*

# ПАРАЛЛЕЛЬНОЕ СУММИРОВАНИЕ

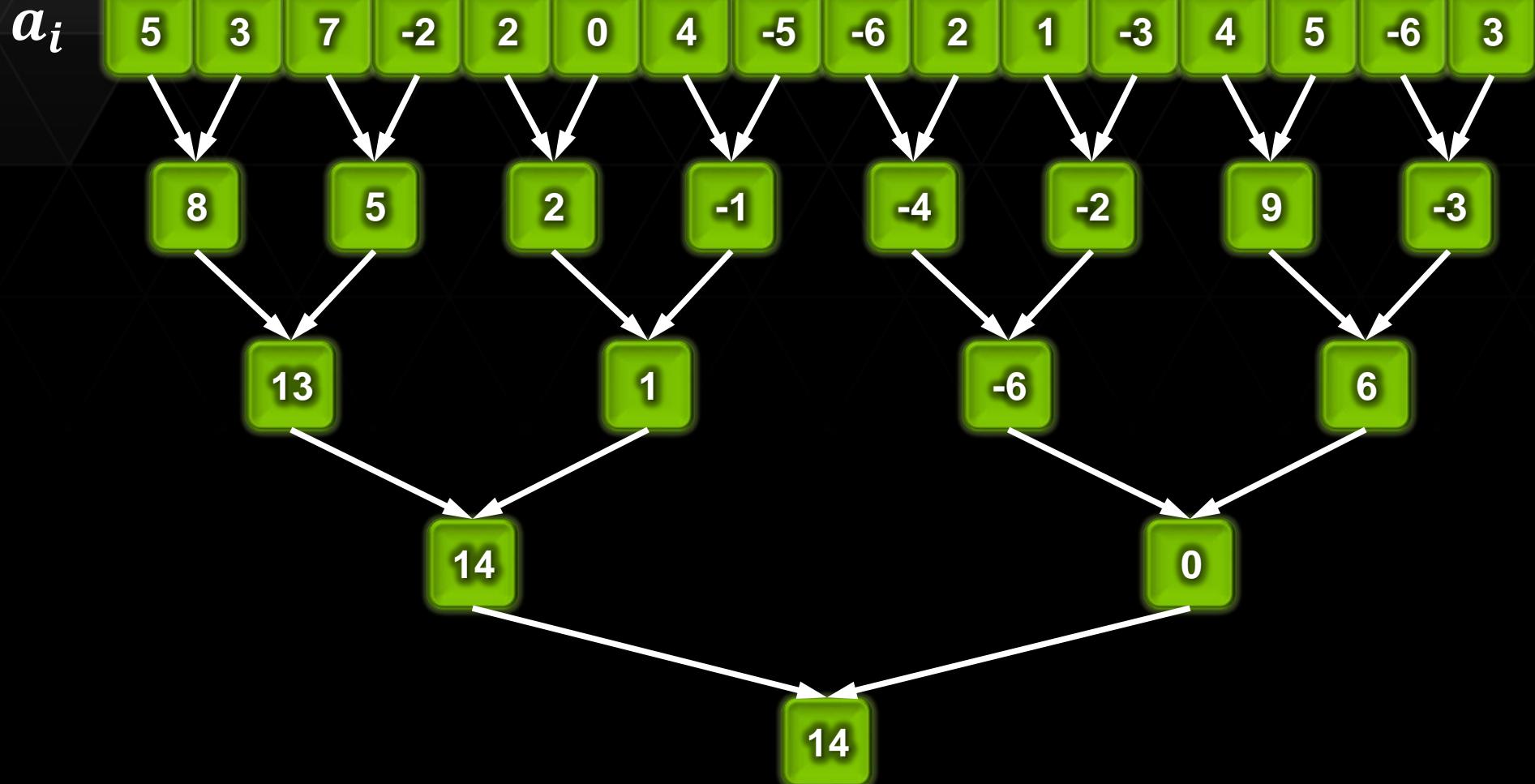
Вычислим сумму элементов массива

$$S = \sum_{i=0}^{N-1} a_i$$

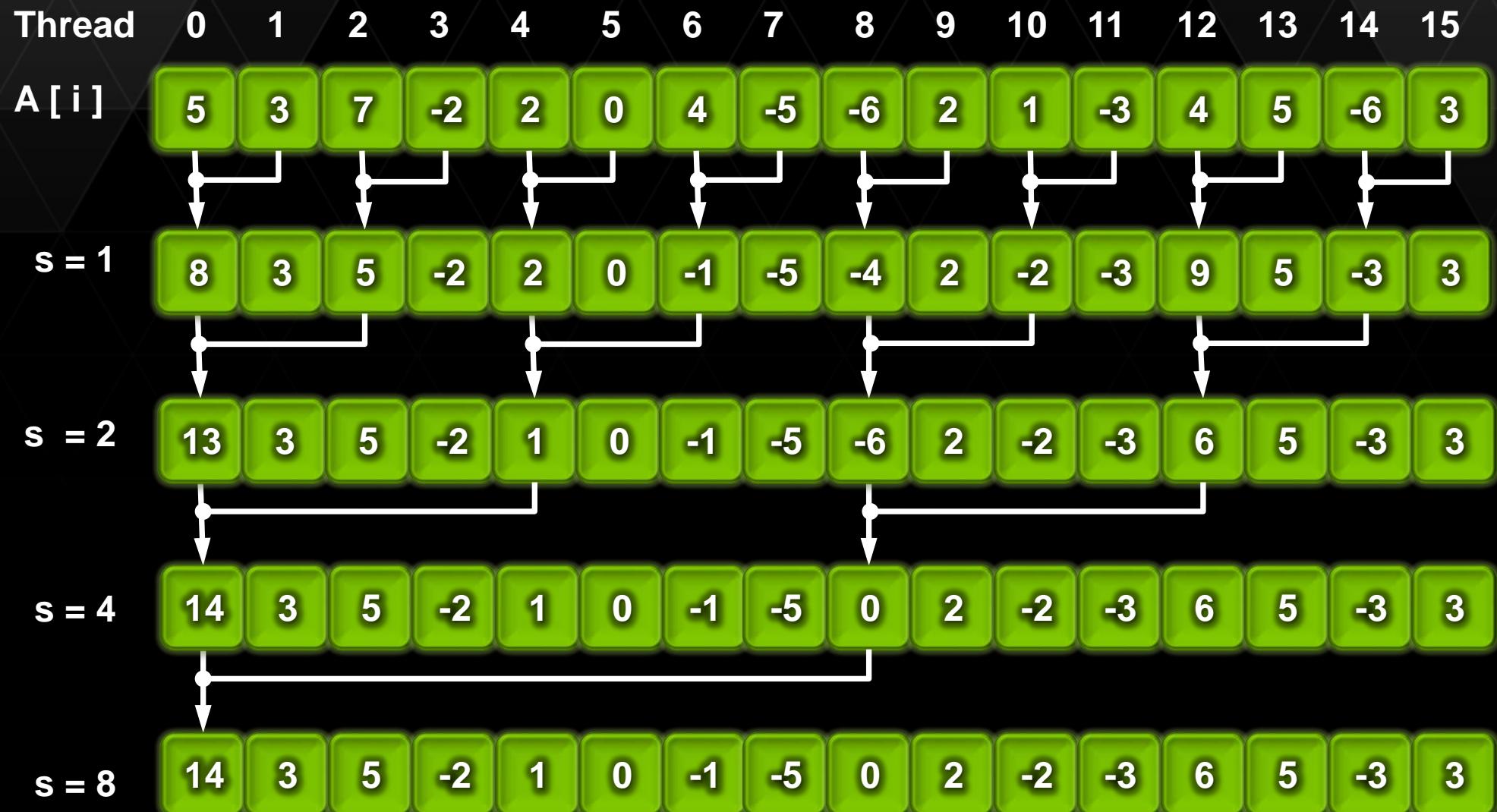
Алгоритм параллельного суммирования

- Массив  $a_i$  разбивается на одинаковые блоки
- Каждый блок нитей суммирует свой блок элементов массива  $a_i$
- Задача сводится к исходной задаче

# ИЕРАРХИЧЕСКОЕ СУММИРОВАНИЕ



# ВАРИАНТ 1

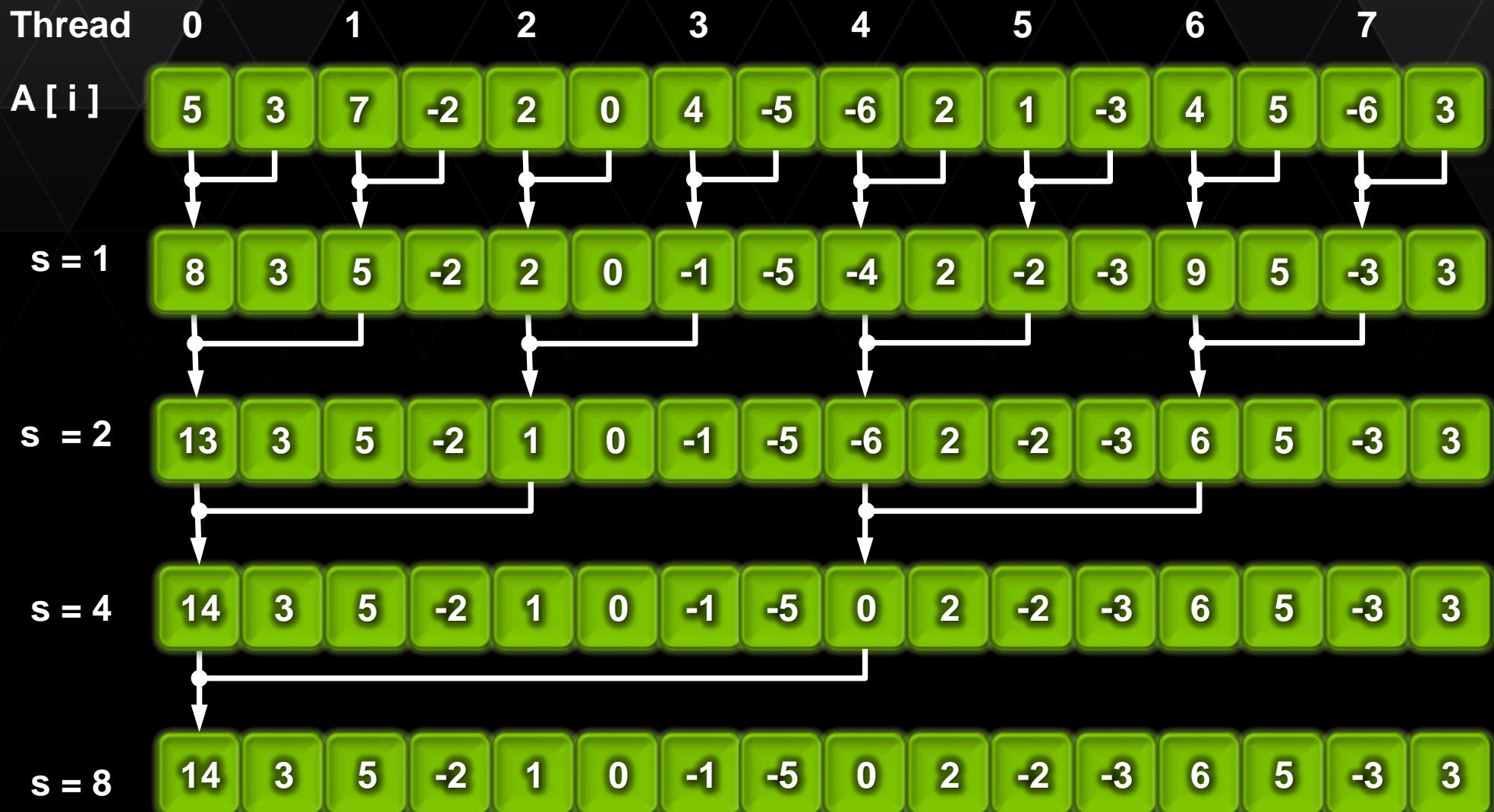


# ВАРИАНТ 1

## ФУНКЦИЯ-ЯДРО «reduce1»

```
__global__ void reduce1 ( float *inData, float *outData )
{
    __shared__ float 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];
}
```

# ВАРИАНТ 2



# ВАРИАНТ 2

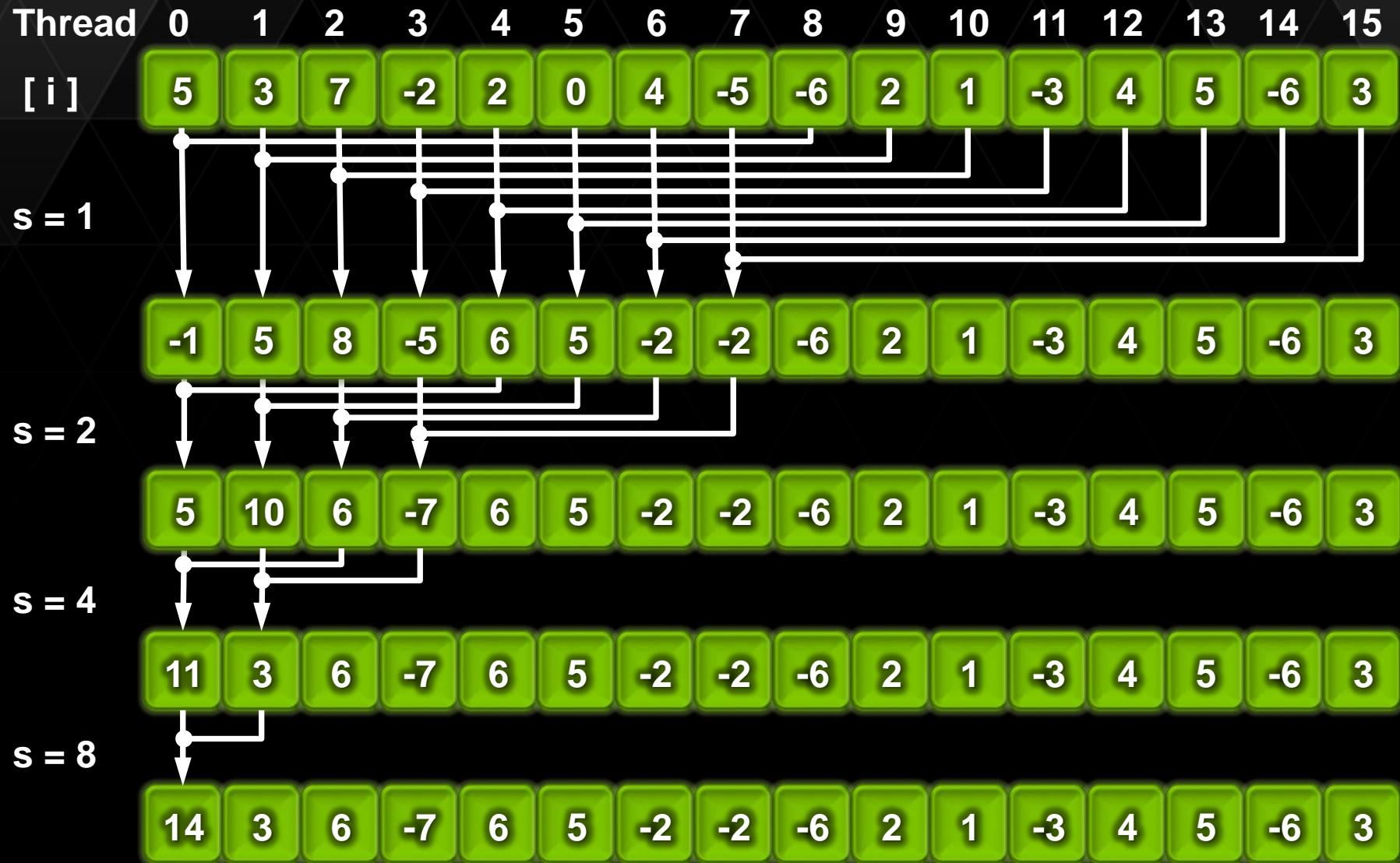
## ФУНКЦИЯ-ЯДРО «reduce2»

```
__global__ void reduce2 ( float *inData, float *outData )
{
    __shared__ float 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];
}
```

## Банк-конфликты

| s   | 16 Banks |       | 32 Banks |       | Bank-conflict |
|-----|----------|-------|----------|-------|---------------|
|     | tid      | index | tid      | index |               |
| 1   | 0        | 0     | 0        | 0     | 2             |
|     | 8        | 16    | 16       | 32    |               |
| 2   | 0        | 0     | 0        | 0     | 4             |
|     | 4        | 16    | 8        | 32    |               |
|     | 8        | 32    | 16       | 64    |               |
|     | 12       | 48    | 24       | 96    |               |
| ... | ...      | ...   | ...      | ...   | ...           |

# ВАРИАНТ 3



# ВАРИАНТ 3

## ФУНКЦИЯ-ЯДРО «reduce3»

```
__global__ void reduce3 ( float *inData, float *outData )
{
    __shared__ float 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];
}
```

# ВАРИАНТ 4

## Фрагмент функции-ядра «reduce4»

```
__global__ void reduce4 ( float *inData, float *outData )
{
...
    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];
    }
    if ( tid == 0 ) outData [blockIdx.x] = data [0];
}
```

## Сравнение

| Функция-ядро | Время GPU, [мс] |
|--------------|-----------------|
|--------------|-----------------|

|         |      |
|---------|------|
| reduce1 | 19.2 |
| reduce2 | 10.6 |
| reduce3 | 8.3  |
| reduce4 | 5.7  |

GPU: Geforce GT9650M, CC 1.1, 32 CUDA-ядра, 16 Banks

<http://nvlabs.github.io/cub>