Технология CUDA для высокопроизводительных вычислений на кластерах с GPU

Лихогруд Николай
n.lihogrud@gmail.com

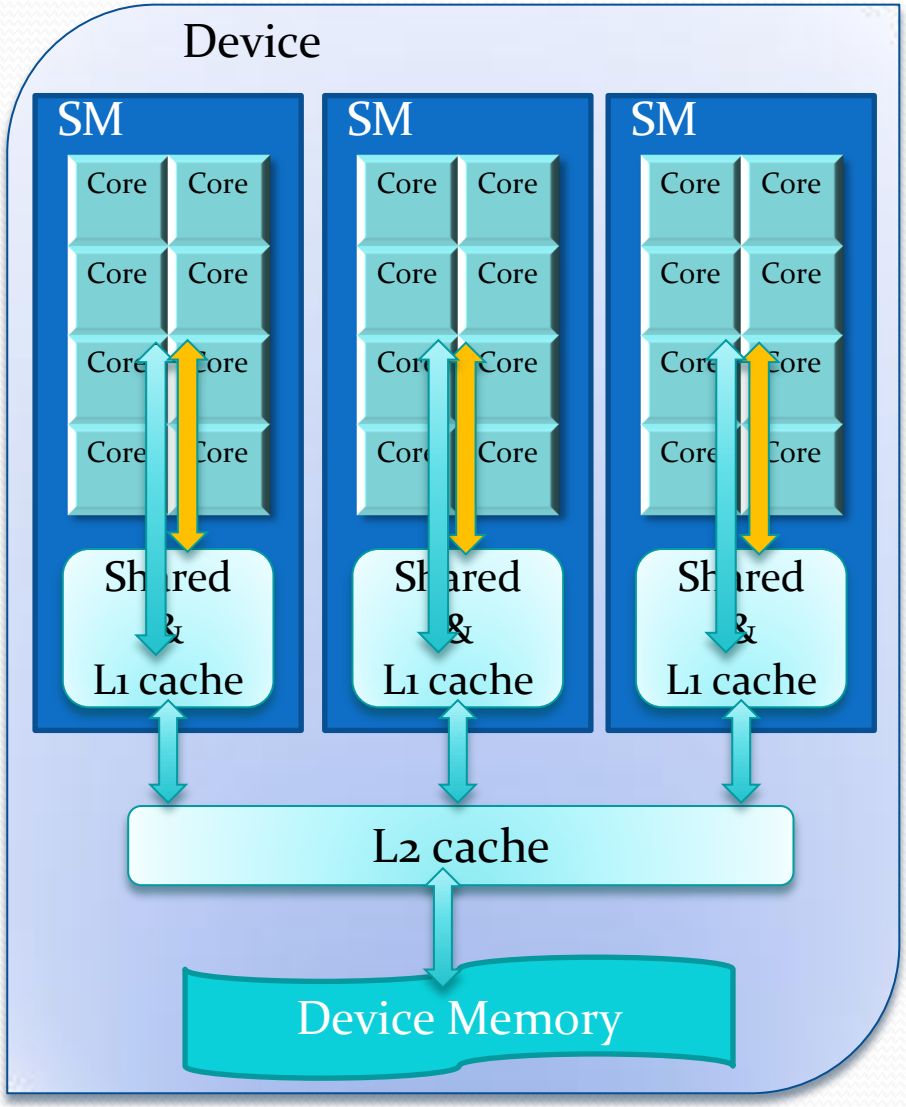
Часть третья

Общая память

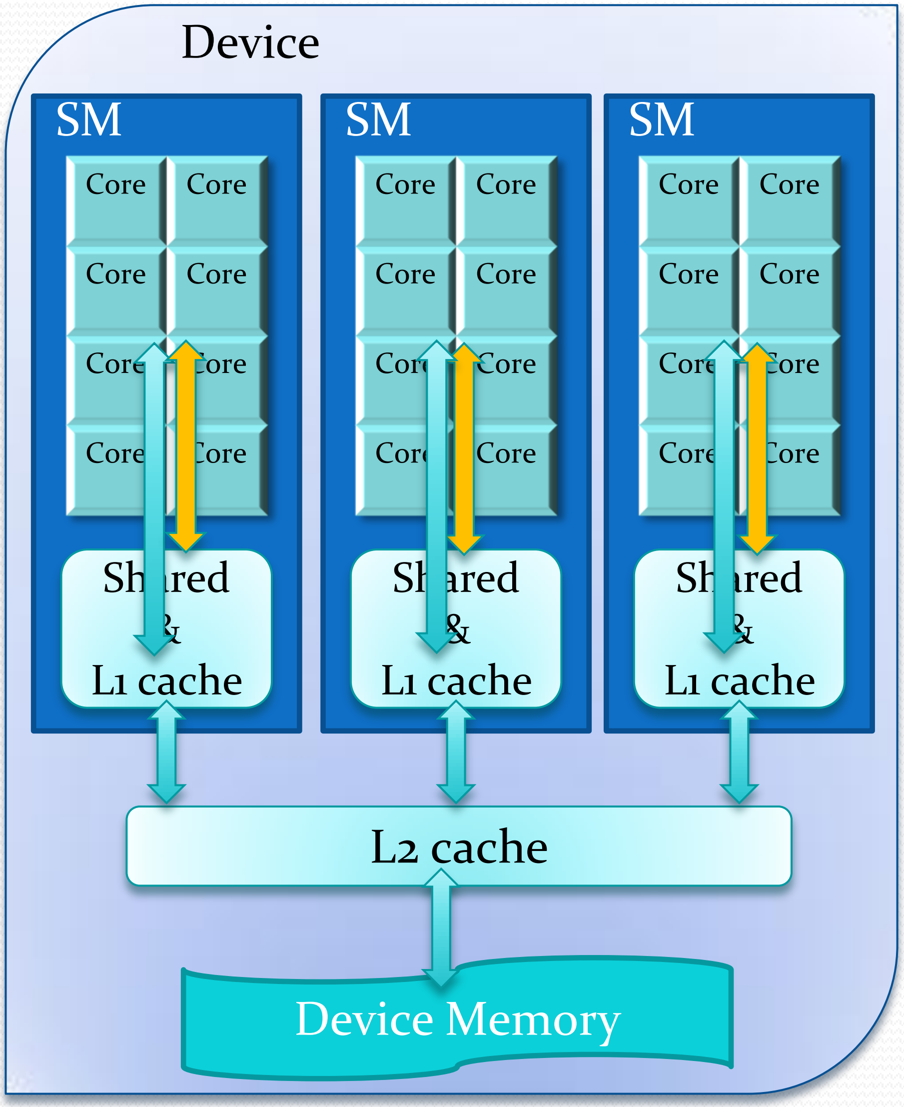
Разделяемая(общая) память

- Расположена в том же устройстве, что и кеш L1
- Совместно используется (разделяется) всеми нитями виртуального блока
- Если на мультипроцессоре работает несколько блоков – общая память делится между ними поровну
- У каждого блока своё адресное пространство общей памяти

- Конфигурации:
  - 16KB общая память, 48KB L1
  - 48KB общая память, 16KB L1 – по умолчанию



Разделяемая(общая) память



Возможные обмены между устройствами при обработке обращений в глобальную память

Возможные обмены между устройствами при обработке обращений в общую память

Выделение общей памяти

- В GPU коде объявляем статический массив или переменную с атрибутом `__shared__`

```
#define SIZE 1024

__global__ void kernel() {
  __shared__ int array[SIZE]; //массив
  __shared__ float varSharedMem; //переменная
  …
}
```

Особенности использования

- Переменные с атрибутом **__shared__** с точки зрения программирования:
  - Существуют только на время жизни блока
    - недоступны с хоста или из других блоков
  - Индивидуальны для каждого блока и привязаны к его личному пространству общей памяти
    - каждый блок нитей видит «своё» значение
  - Не могут быть проинициализированы при объявлении

Раздача указателя нитям блока

```
__global__ void kernel() {
    __shared__ int *memoryOnDevice;
    if (threadIdx.x == 0) {
        // выделяет память только первая нить
        size_t size = blockDim.x * sizeof(float);
        memoryOnDevice = (int *)malloc(size);
        memset(memoryOnDevice, 0, size);
    }

    memoryOnDevice[thredIdx.x] = …;
    …// использование указателя всеми нитями блока
}
```

Раздача указателя нитям блока

```
__global__ void kernel() {
    __shared__ int *memoryOnDevice;
    if (threadIdx.x == 0) {
        // выделяет память только первая нить
        size_t size = blockDim.x * sizeof(float);
        memoryOnDevice = (int *)malloc(size);
        memset(memoryOnDevice, 0, size);
    }
        ??
    memoryOnDevice[thredIdx.x] = …;
    …// использование указателя всеми нитями блока
}
```

Синхронизация

- Рассмотрим пример ядра, запускаемого на одномерном линейном гриде:

```
__global__ void kernel() {
    __shared__ int shmem[BLOCK_SIZE];
    shmem[threadIdx.x] = __sinf(threadIdx.x);
    int a = shmem[(threadIdx.x + 1 )% BLOCK_SIZE];
    …
}
```

- Каждая нить
  - Записывает **__sinf** от своего индекса в соответствующую ей ячейку массива
  - Читает из массива элемент, записанный соседней нитью

Синхронизация

- Рассмотрим пример ядра, запускаемого на одномерном линейном гриде:

```
__global__ void kernel() {
    __shared__ int shmem[BLOCK_SIZE];
    shmem[threadIdx.x] = __sinf(threadIdx.x);
    int a = shmem[(threadIdx.x + 1 )% BLOCK_SIZE];
    …
}
```

- Варпы выполняются в непредсказуемом порядке
  - Может получиться, что нить ещё не записала элемент, соседняя уже пытается его считать!
  - read-after-write, write-after-read, write-after-write конфликты

Синхронизация

- Явная синхронизация нитей одного блока
  - `void __syncthreads();`

    При вызове этой функции нить блокируется до момента, когда:
    - все нити в блоке достигнут данную точку
    - результаты всех инициированных к данному моменту операций с глобальной\общей памятью, станут видны всем нитям блока

Синхронизация

- `__syncthreads()` можно вызывать в ветвях условного оператора только если результат его условия одинаков во всех нитях блока, иначе выполнение может зависнуть или стать непредсказуемым

Синхронизация

```
__global__ void kernel() {
    __shared__ int shmem[BLOCK_SIZE];
    shmem[threadIdx.x] = __sinf(threadIdx.x);
    __syncthreads();
    int a = shmem[(threadIdx.x + 1 )% BLOCK_SIZE];
    …
}
```

- Каждая нить
  - Записывает **__sinf** от своего индекса в соответствующую ей ячейку массива
  - Ожидает завершения операций в других нитях
  - Читает из массива элемент, записанный соседней нитью

Раздача указателя нитям блока

```
__global__ void kernel() {
    __shared__ int *memoryOnDevice;
    if (threadIdx.x == 0) {
        // выделяет память только первая нить
        size_t size = blockDim.x * 64;
        memoryOnDevice = (int *)malloc(size);
        memset(memoryOnDevice, 0, size);
    }
    __syncthreads();

    …// использование указателя всеми нитями блока
}
```

Нужна синхронизация!

Динамическая общая память

- Бывают ситуации, когда нужный размер общей памяти не известен на этапе компиляции
  - Зависит от размер задачи, блока и т.д.
- В этом случае выделить память как статическую переменную невозможно
- Можно указать требуемый размер общей памяти при запуске ядра

Динамическая общая память

- В GPU коде объявляем указатель для доступа к общей памяти:

```
__global__ void kernel() {
      extern __shared__ int array[];
     …
}
```

- В ***третьем параметре конфигурации*** запуска указываем сколько общей памяти нужно выделить ***каждому блоку***

```
kernel<<<gridDim, blockDim, SIZE >>>(params)
```

Динамическая общая память

- Все переменные `extern __shared__ type var[]` указывают на одно и то же начало динамической общей памяти, выделенной блоку
- Ядру может быть одновременно выделена статическая, и динамическая память.
- Если суммарный объем динамической и статической памяти превышает 48кб на блок – произойдет ошибка запуска

Стратегия использования

- Общая память по смыслу является кешем, **управляемым пользователем**
  - Имеет низкую латентность - расположена на том же оборудовании, что и кеш L1, скорость загрузки сопоставима с регистрами
  - Приложение явно выделяет и использует общую память
  - Пользовать сам выбирает что, как и когда в ней хранить
    - Шаблон доступа может быть произвольным, в отличие от L1

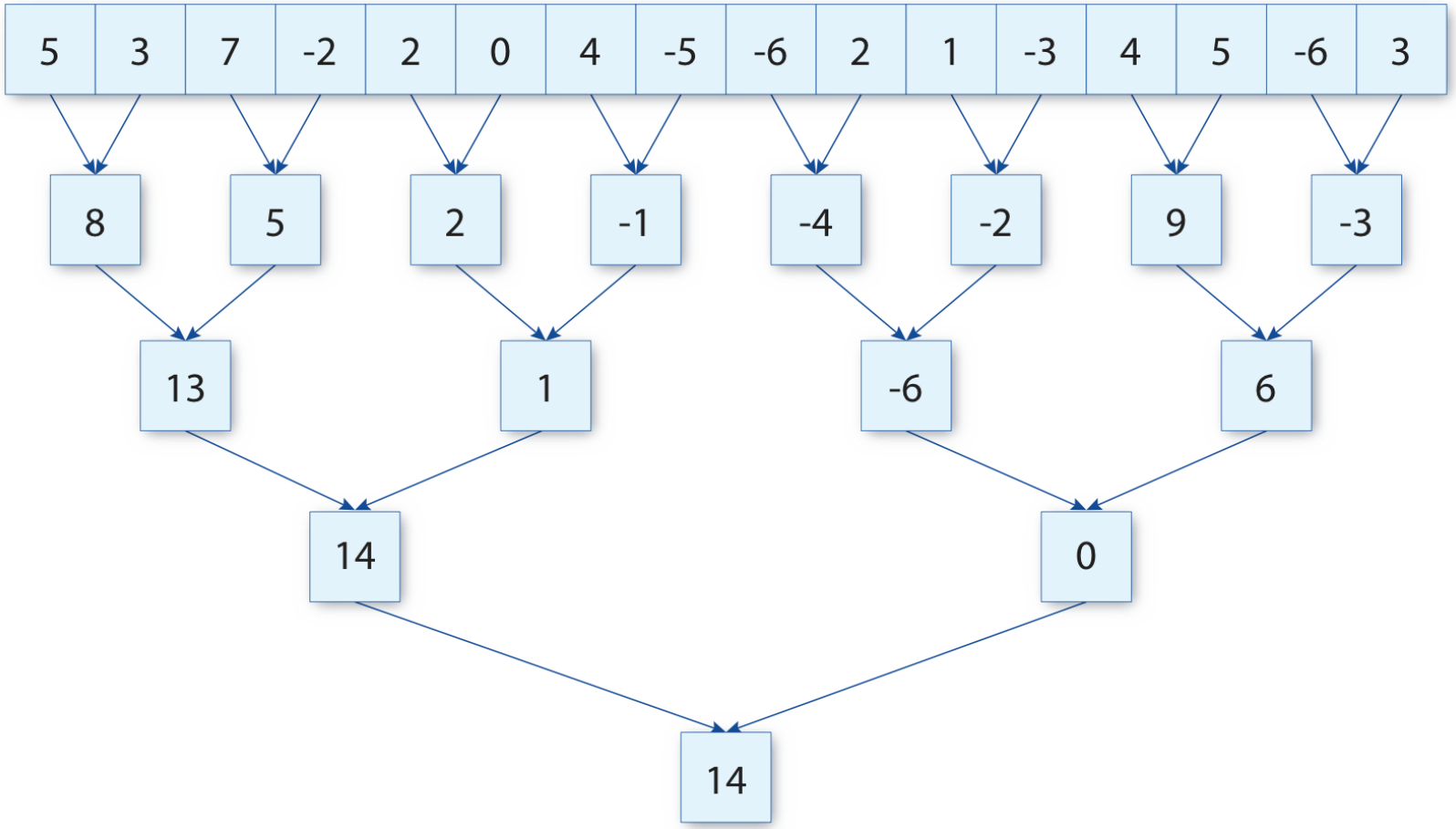
Стратегия использования

- Типичная стратегия использования:
  - Нити блока коллективно
    1. Загружают данные из глобальной памяти в общую
       - Каждая нить делает часть этой загрузки
    2. Синхронизуются
       - Чтобы никакая нить не начинала чтение данных, загружаемых другой нитью, до завершения их загрузки
    3. Используют загруженные данные для вычисления результаты
       - Если нити что-то пишут в общую память, то также может потребоваться синхронизация
    4. Записывают результаты обратно в глобальную память

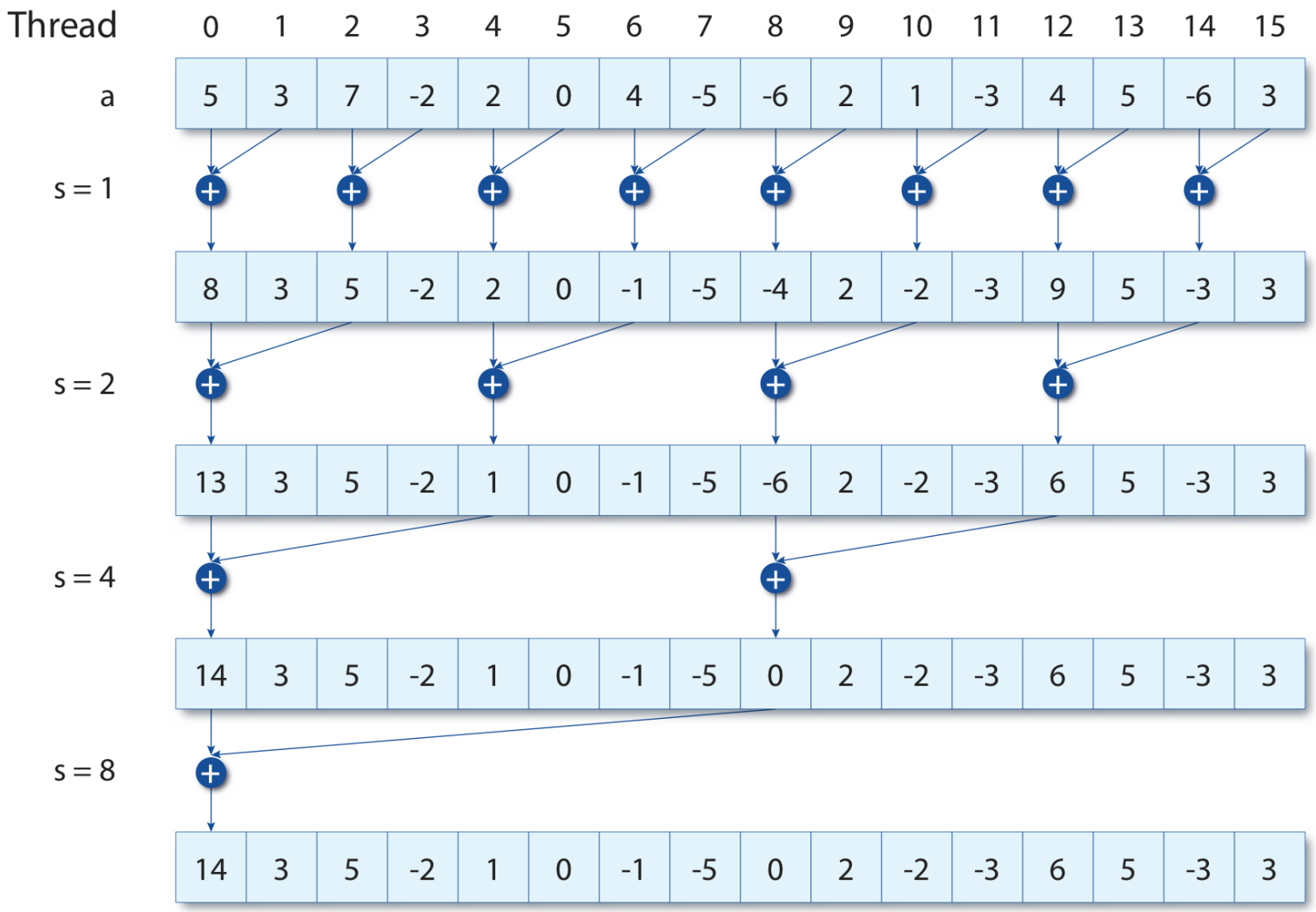
Редукция

- Блоку нитей сопоставляем часть массива
  - Каждый блок нитей суммирует элементы из своей части массива
- Блок нитей
  - Копирует данные в общую память
  - Иерахически суммирует данные в общей памяти
  - Сохраняет результат в глобальной памяти

Иерархическое суммирование



Иерархическое суммирование



Ядро суммирования

```
__global__ void reduce (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  = 1; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0) {
            data [tid] += data [tid + s];
        }
        __syncthreads ();
    }
    if (tid == 0) {
        outData [blockIdx.x] = data [0];
    }
}
```

Банки общей памяти

Банки общей памяти

- Для увеличения полосы пропускания устройство, на котором расположена общая память, разделено на подмодули («**банки**»)
  - **n** – число банков
  - **m** – сколько последовательных байтов может отдать каждый банк за цикл
- Адресное пространство общей памяти разделено на **n** непересекающихся подмножеств, расположенных в разных банках
- Банки работают независимо друг-от-друга и могут вместе выдать максимум **n*m** байтов за один цикл

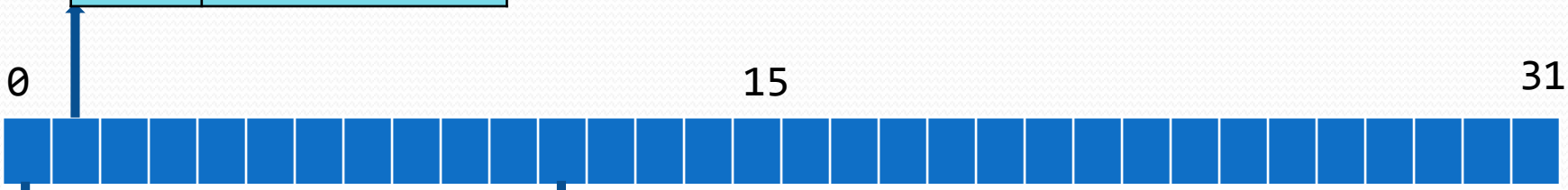
Банки общей памяти на Fermi

- 32 банка, каждый банк может выдать за 2 такта ядер **одно** 32-битное слово (4 последовательных байта)
- Последовательные 32-битные слова располагаются в последовательных банках
  - Номер банка для слова по адресу addr: `(addr / 4) % 32`
- За два такта ядер общая память может отдать 128 байт

Банк 1

4	
132	
260	
388	
516	
…	

Банк 0	Банк 1	Банк 2	Банк 3	…
0	4	8	12	…
128	132	136	140	…
256	260	264	268	…
…	…	…	…	…



Банк 0

0	
128	
256	
384	
512	
…	

Банк 11

44	
172	
300	
428	
556	
…	

Обращения в общую память

- Обращение выполняется одновременно всеми нитями варпа (SIMT)
- Банки работаю параллельно
  - Если варпу нитей нужно получить 32 4-байтных слова, расположенных в разных банках, то такой запрос будет выполнен одновременно всеми банками
    - Каждый банк выдаст соответствующее слово
    - Пропускная способность = **32 х пропускная способность банка**
- Поддерживается рассылка (broadcast):
  - Если часть нитей (или все) обращаются к одному и тому же 4-х байтному слову, то нужное слово будет считано из банка и роздано соответствующим нитям (broadcast) без накладных расходов

Банк конфликты

- Если хотя бы два нужных варпу слова расположены в одном банке, то такая ситуация называется «**банк конфликтом**» и обращение в глобальную память будет «**сериализованно**»:
  - Такое обращение аппаратно разбивается на серию обращений, не содержащих банк конфликтов
    - Если число обращений, на которое разбит исходный запрос, равно n, то такая ситуация называется **банк-конфликтом порядка n**
    - Пропускная способность при этом падает в n раз

Банки общей памяти на Kepler

- 32 банка, каждый банк может выдать за 1 такт ядер 8 байтов
  - На Kepler частота ядер в 2 раза меньше, чем на Fermi
- Два режима разбиения общей памяти на банки:
  - Последовательные 32-битные слова располагаются в последовательных банках: `(addr / 4) % 32`
  - Последовательные 64-битные слова располагаются в последовательных банках: `(addr / 8) % 32`
- За два такта ядер общая память может отдать 256 байт

Банк 1

4	
132	
260	
388	
516	
…	

Банк 0		Банк 1		Банк 2		Банк 3		…
0	128	4	132	8	136	12	138	…
256	384	260	388	264	392	268	396	…
512	640	516	644	520	648	524	652	…
…		…		…		…		…

Банк 0

0	
128	
256	
384	
512	
…	

Банк 11

44	
172	
300	
428	
556	
…	

Последовательные 32-битные слова в последовательных банках

Банк 1

8	
12	
264	
368	
520	
…	

Банк 0		Банк 1		Банк 2		Банк 3		…
0	4	8	12	16	20	24	28	…
256	260	264	268	272	276	280	284	…
512	516	520	524	528	532	536	540	…
…		…		…		…		…

Банк 0

0	
4	
256	
360	
512	
…	

Банк 11

88	
92	
344	
348	
600	
…	

Последовательные 64-битные слова в последовательных банках

Последовательные 32-битные слова в последовательных банках

Банк 0		Банк 1		Банк 2		Банк 3		…
0	128	4	132	8	136	12	138	…
256	384	260	388	264	392	268	396	…
512	640	516	644	520	648	524	652	…
…		…		…		…		…

Могут быть отданы банком 0 за один такт*

Последовательные 64-битные слова в последовательных банках

Банк 0		Банк 1		Банк 2		Банк 3		…
0	4	8	12	16	20	24	28	…
256	260	264	268	272	276	280	284	…
512	516	520	524	528	532	536	540	…
…		…		…		…		…

Могут быть отданы банком 0 за один такт*

*Каждая ячейка соответствует 4-м последовательным байтам

Банк конфликты на Kepler

- Последовательные **32**-битные слова располагаются в последовательных банках:

  Банк-конфликта между двумя нитями нет, если запрашиваются байты 32-битных слов из разных банков, либо запрашиваемые слова находятся в 32-битных словах с адресами `i` и `i + 128`, `256*n <= i < 256*n + 128` (`broadcast`)

- Последовательные **64**-битные слова располагаются в последовательных банках:

  Банк-конфликта между двумя нитями нет, если запрашиваются байты 64-битных слов из разных банков, либо байты 64-битного слова из одного банка (`broadcast`)

Установка режима общей памяти

```
cudaError_t cudaDeviceSetSharedMemConfig (
                                cudaSharedMemConfig config )
```

- Глобально для всех запусков ядер
  - `cudaSharedMemBankSizeDefault` - последовательные **32**-битные слова в последовательных банках
  - `cudaSharedMemBankSizeFourByte` - последовательные **32**-битные слова в последовательных банках
  - `cudaSharedMemBankSizeEightByte` - последовательные **64**-битные слова в последовательных банках

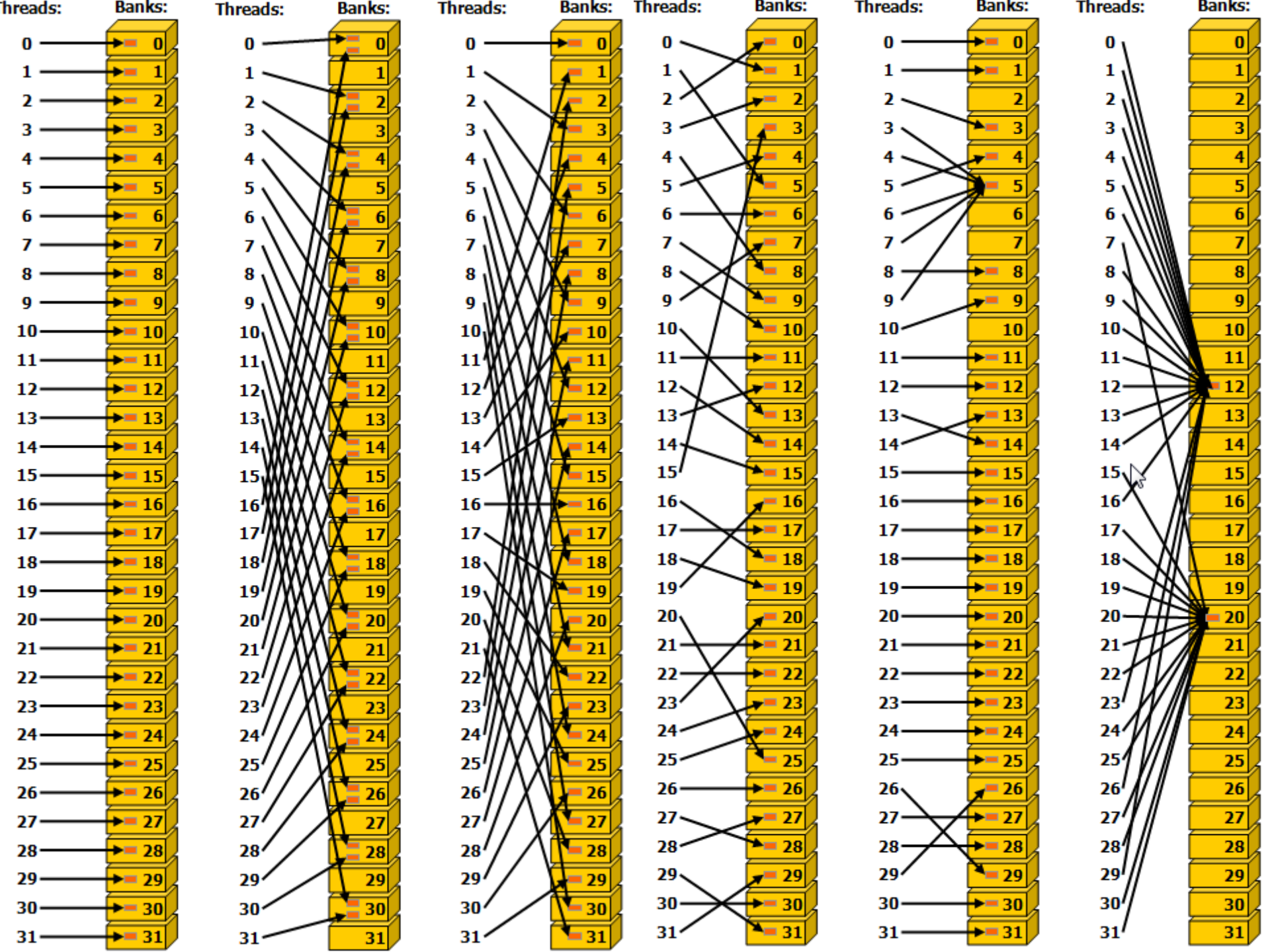
```
cudaError_t cudaFuncSetSharedMemConfig ( const void* func,
                                cudaSharedMemConfig config )
```

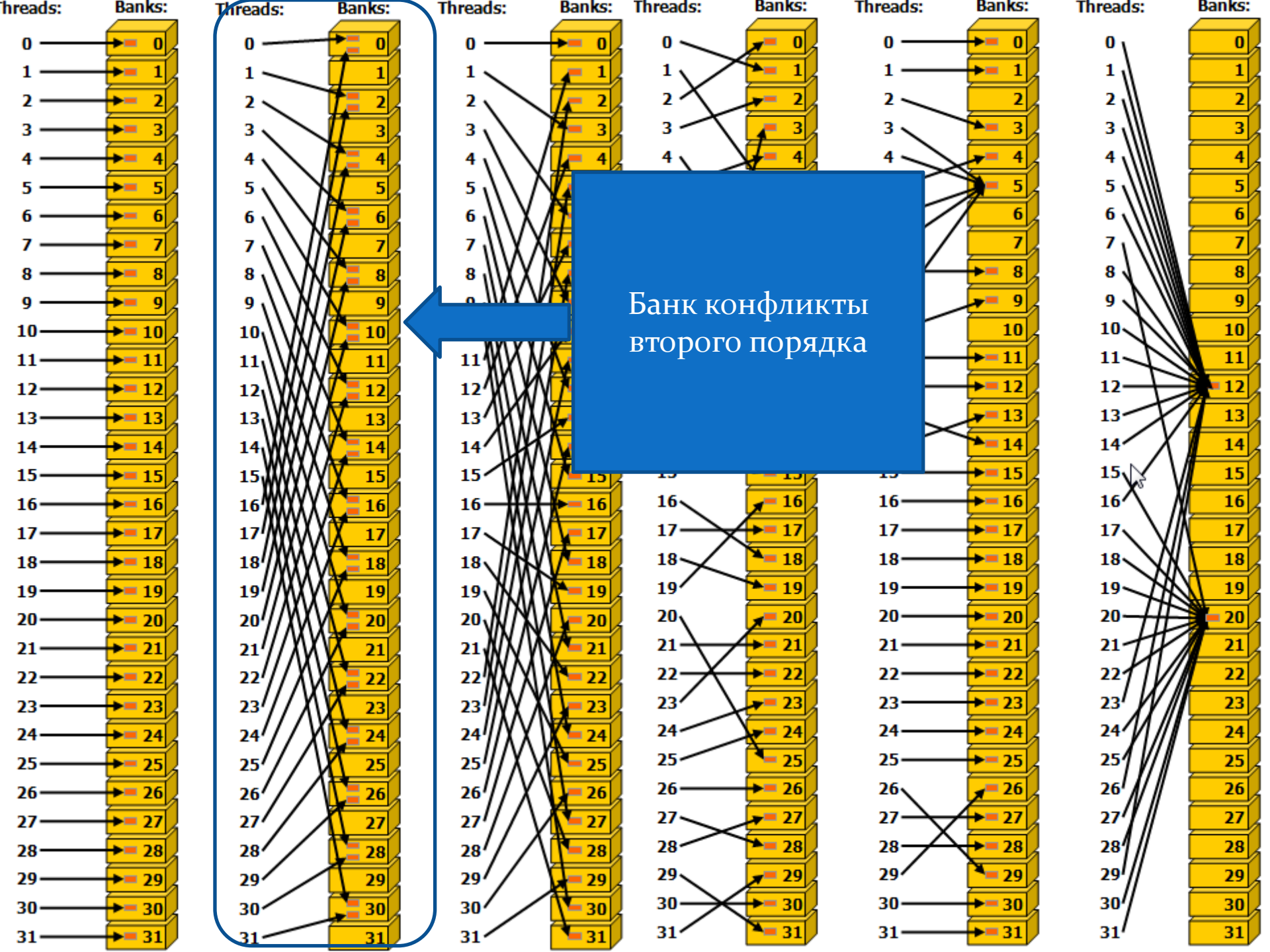
- Для запусков конкретного ядра

Зачем устанавливать режим

```
extern __shared__ double arr[];
double res = sin(arr[thredIdx.x * 3]);
```

- Нить 0 обращается к байту со смещением 0, нить 16 – 384
- cudaSharedMemBankSizeFourByte
  - Оба обращения попадают в один банк, банк-конфликт второго порядка
- cudaSharedMemBankSizeEightByte
  - Обращения попадают в банки 0 и 16, банк-конфликта нет


Threads:
Banks:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31


Threads:
Banks:
Банк конфликты
второго порядка

Примеры банк-конфликтов

```
extern __shared__ float char[];
float data = shared[BaseIndex + s * threadIx.x];// конфликты
                                                зависят от s
```

- Нити `threadIx.x` и `(threadIx.x + n)` обращаются к элементам из одного и того же банка когда s*n делится на 32 (число банков).
  - S = 1 :

    ```
    shared[BaseIndex + threadIx.x] // нет конфликта
    ```
  - S = 2 :

    ```
    shared[BaseIndex + 2*threadIx.x] // конфликт 2-го порядка
    ```

    Например, между нитями `threadIx.x=0` и `(threadIx.x = 16)` – попадают в один варп!

Распространенная проблема

- Пусть в общей памяти выделена плоская плотная матрица шириной, кратной 32, и соседние нити варпа обращаются к соседним элементам столбца

```
__shared__ int matrix[32][32]
matrix[thredIdx.x][4] = 0;
```

Распространенная проблема

- Пусть в общей памяти выделена плоская плотная матрица шириной, кратной 32, и соседние нити варпа обращаются к соседним элементам столбца

```
__shared__ int matrix[32][32]
matrix[thredIdx.x][4] = 0;
```

Банк конфликт 32-го порядка 

Распространенная проблема

```
__shared__ int matrix[32][32]
  matrix[thredIdx.x][4] = 0;
```

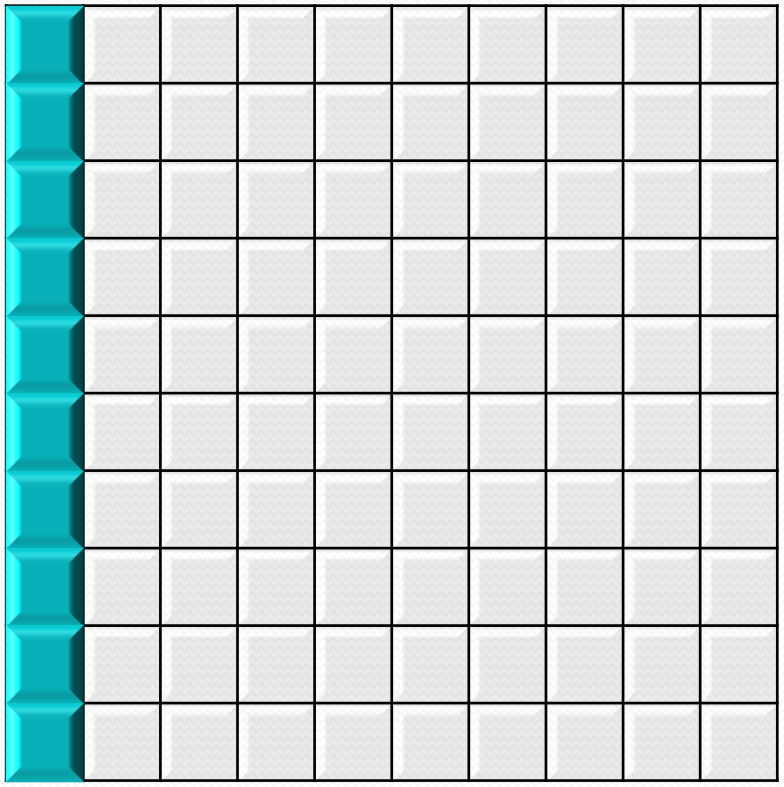
Банк конфликт 32-го порядка 

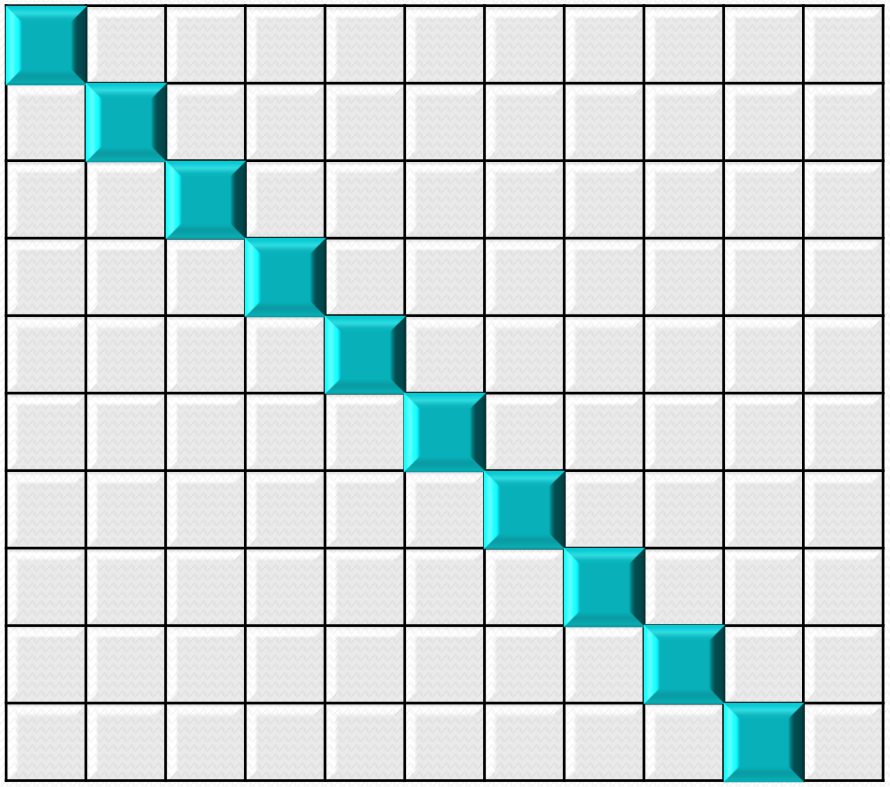
Решение: набивка

```
__shared__ int matrix[32][32 + 1]
matrix[thredIdx.x][4] = 0; //нет конфликта
```

Распространенная проблема

Пусть банков 10, матрица 10x10





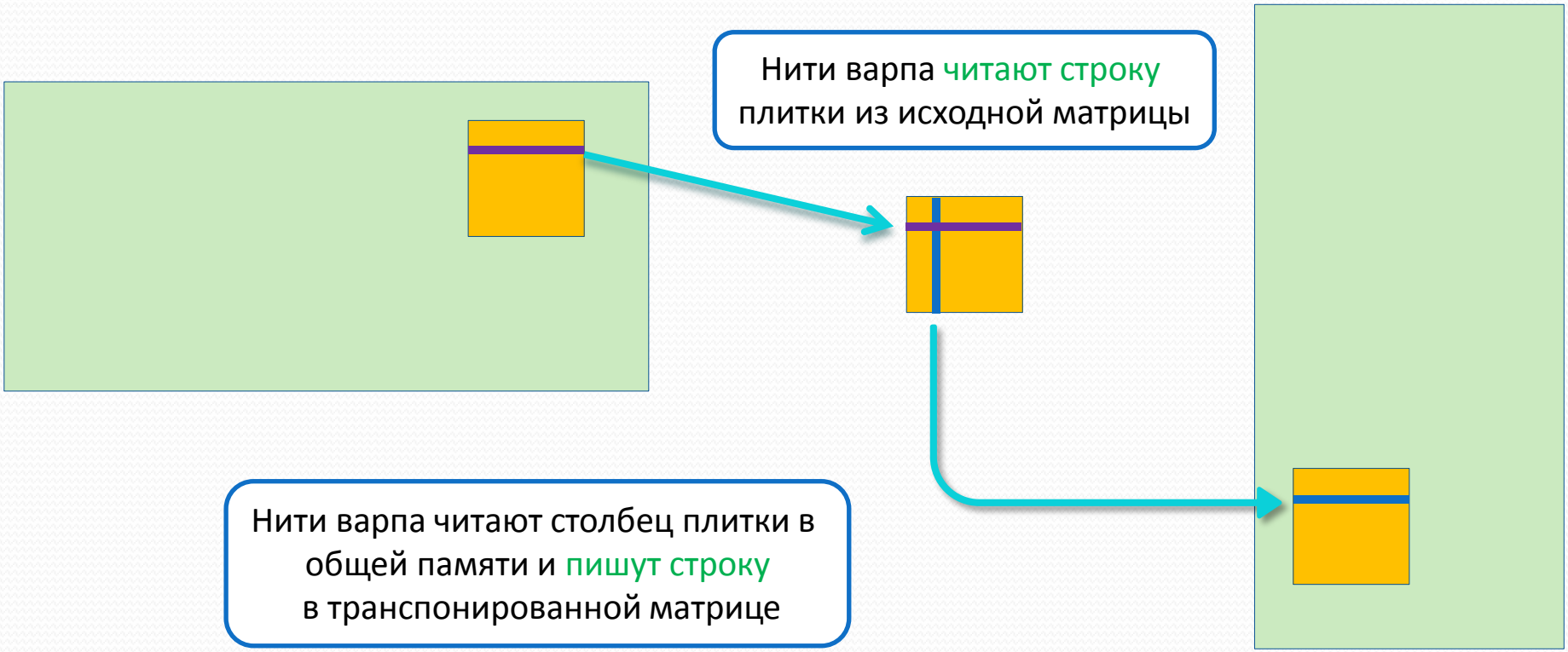
Транспонирование матрицы

```
__global__ void simpleTranspose(ElemType *inputMatrix, ElemType
                                *outputMatrix, int width, int height) {

    int i = threadIdx.y + blockIdx.y * blockDim.y;
    int j = threadIdx.x + blockIdx.x * blockDim.x;

    if ( i < height && j < width) {
        outputMatrix[j * (height) + i] = inputMatrix[ i * (width) + j];
    }
}
```

Нити варпа записывают элементы столбца
32 транзации на одну запись

Транспонирование через общую память

- Считать плитку матрицы в общую память
- Записать в результат транспонированную плитку



Транспонирование через общую память

```
__global__ void shmemTranspose( ElemType *inputMatrix,
                                ElemType *outputMatrix, int width, int height) {

    int i = threadIdx.y + blockIdx.y * blockDim.y;
    int j = threadIdx.x + blockIdx.x * blockDim.x;

    __shared__ ElemType shmem[32][32];

    if ( i < height && j < width) {
        shmem[threadIdx.y][threadIdx.x] = inputMatrix[i * (width) + j];
    }
    __syncthreads();

    if ( i < width && j < height) {
        outputMatrix[i * (height) + j] = shmem[threadIdx.x][threadIdx.y];
    }
}
```

Банк-конфликт 32-го порядка

Транспонирование через общую память

```
__global__ void correctShmemTranspose( ElemType *inputMatrix,
                                       ElemType *outputMatrix, int width, int height) {

    int i = threadIdx.y + blockIdx.y * blockDim.y;
    int j = threadIdx.x + blockIdx.x * blockDim.x;

    __shared__ ElemType shmem[32][32 + 1];

    if ( i < height && j < width) {
        shmem[threadIdx.y][threadIdx.x] = inputMatrix[i * (width) + j];
    }
    __syncthreads();

    if ( i < width && j < height) {
        outputMatrix[i * (height) + j] = shmem[threadIdx.x][threadIdx.y];
    }
}
```

Избавились от банк-конфилкта

Тесты

- Kepler K20c, матрица 16384x16384 элемента

Ядро	Double	Float
Простое	53.942ms	41.040ms
С общей памятью	52.338ms	32.840ms
С общей памятью без банк-конфликтов	36.057ms	21.274ms

Выводы

- Общую память можно использовать как управляемый кеш для реиспользования данных
  - Как в редукции
- Доступ в общую память может быть произвольным, в отличие от кеша L1
  - Можно применять пространственные преобразования к данным, используя общую память как буфер (транспонирование - поворот и отражение)
- Банк-конфликты высокого порядка могут сильно ухудшить пропускную способность общей памяти
- Доступный объем общей памяти ограничен
  - Влияет на **occupancy**

The end