РАБОТА С РАЗДЕЛЯЕМОЙ ПАМЯТЬЮ. СИНХРОНИЗАЦИИ.

АФАНАСЬЕВ ИЛЬЯ
AFANASIEV_ILYA@ICLOUD.COM

ЧЕМУ МЫ НАУЧИЛИСЬ ЗА ПЕРВЫЙ ДЕНЬ?

- GPU позволяет крайне эффективно ускорять массивно-параллельные задачи. *(Вопрос - за счет каких аппаратных особенностей?)*

- Для написания GPU-программ мы используем программную архитектуру CUDA.

- CUDA-код состоит из хост- и девайс- частей.

- Хост-код выполняется на ядре центрального процессора и использует специализированное CUDA Runtime API для выделений памяти на GPU, копирований данных, получения свойств GPU и др.

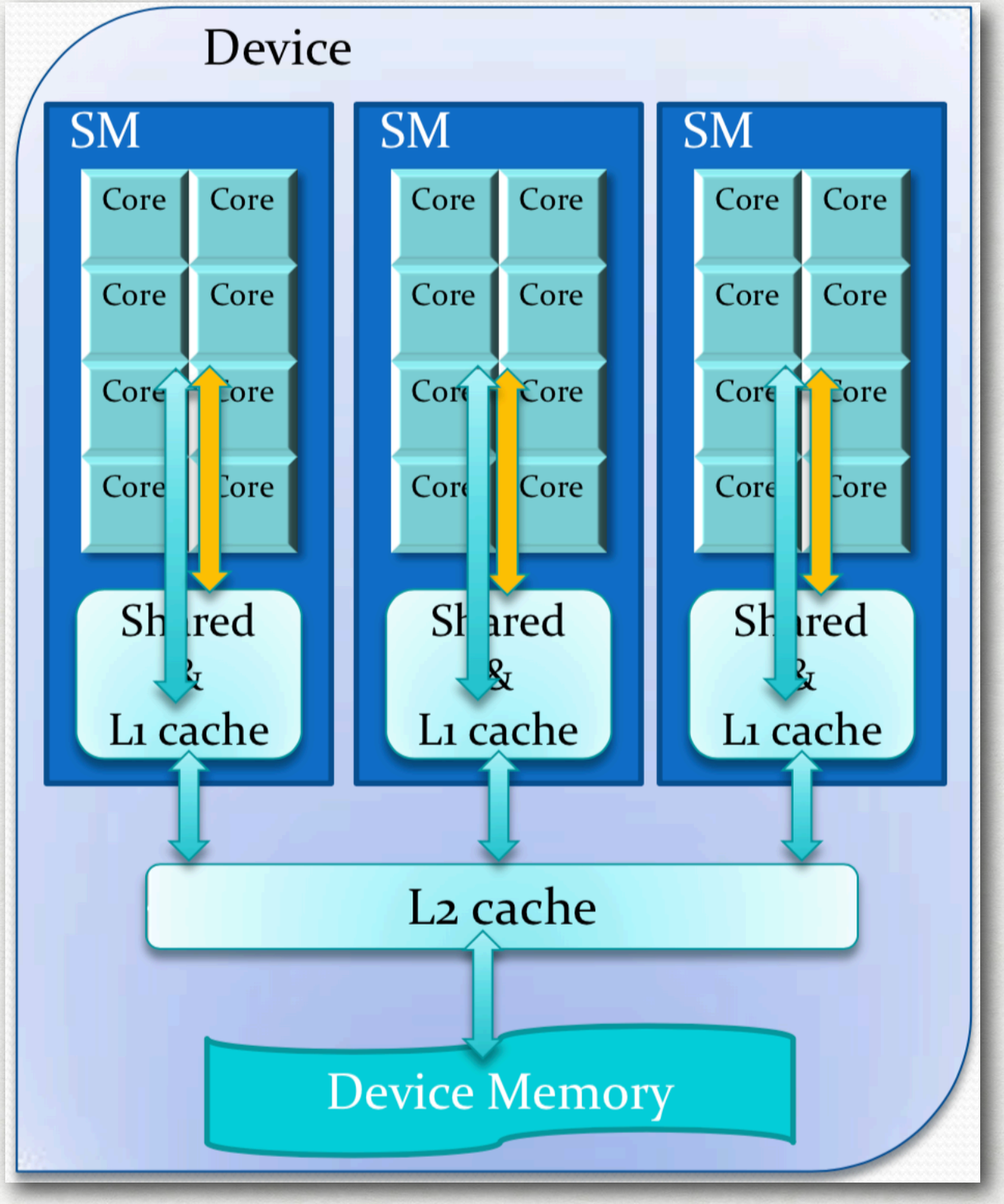
- Девайс-код выполняется на графическом ускорителе и реализуется при помощи написания специальных ядер + задания грида.

ЧЕМУ МЫ НАУЧИЛИСЬ ЗА ПЕРВЫЙ ДЕНЬ?

- Грид состоит из блоков и нитей, и определяет, **сколько** работы должен выполнить девайс-код.
- CUDA-ядра являются специализированными C++ функциями, описывающими, **что делать** каждой из нитей.
- Грид может иметь одномерную, двумерную, и трехмерную конфигурацию, которую нужно выбирать исходя из размеров и типа задачи (работа с векторами - одномерная, с матрицами - двумерная).
- CUDA-нити имеют SIMT (SIMD+MIMD) модель вычислений на аппаратуре
- Обращения к памяти нитями имеют транзакционный характер, что крайне важно учитывать для создания эффективных приложений.

РАЗДЕЛЯЕМАЯ 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 ограниченное адресное пространство общей памяти (64 KB)
- Конфигурации:
  - 16KB общая память, 48KB L1
  - 48KB общая память, 16KB L1 – по умолчанию



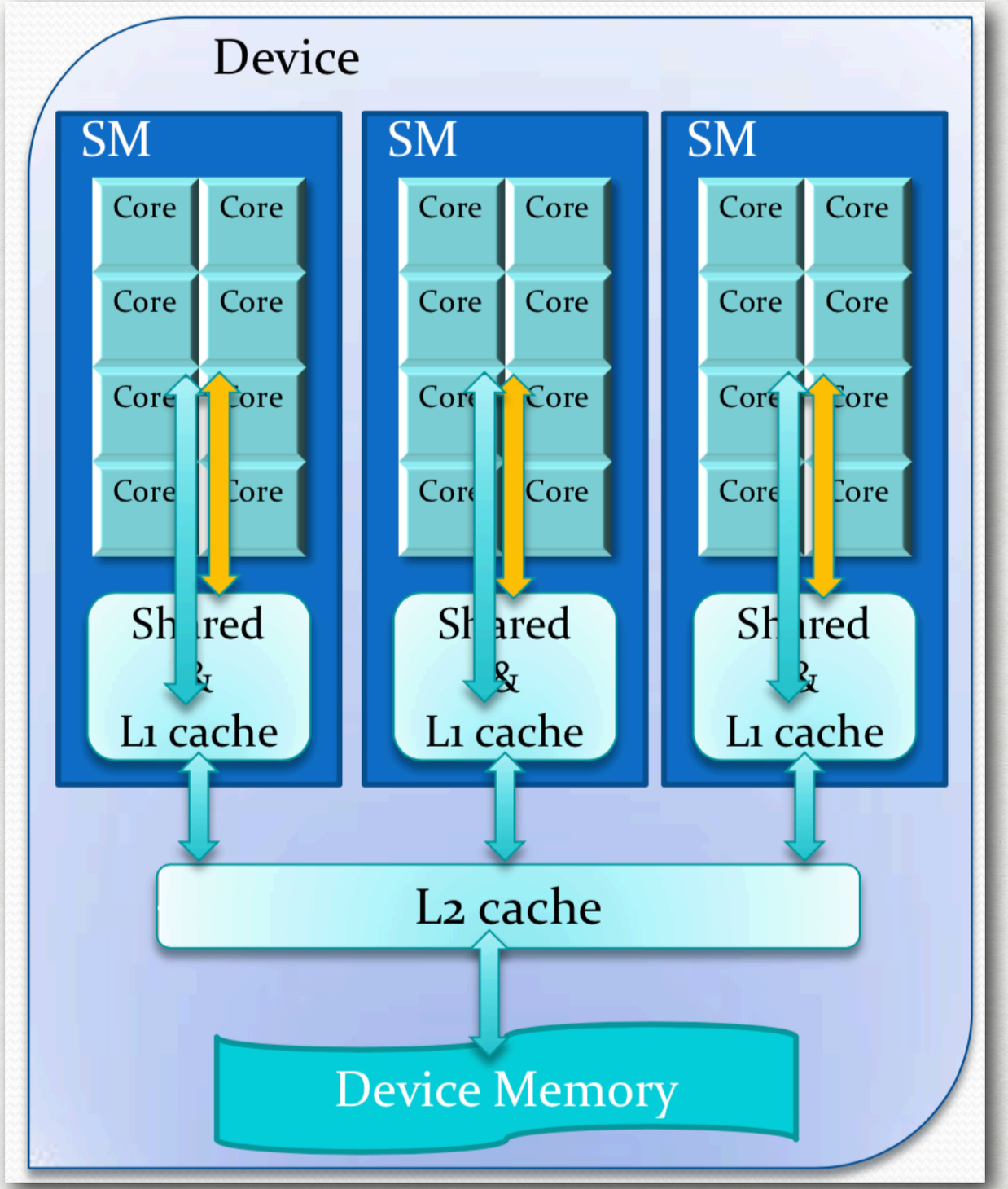
ЗАЧЕМ ИСПОЛЬЗОВАТЬ РАЗДЕЛЯЕМУЮ ПАМЯТЬ?

- Обмены данными между ядрами (рассмотрим для начала)
- Быстрый программный кэш

ОБМЕНЫ ДАННЫМИ ЧЕРЕЗ РАЗДЕЛЯЕМАЯ ПАМЯТЬ

- Через глобальную память возможны обмены данными между любыми двумя ядрами GPU (даже с **различных мультипроцессоров**)

- Через разделяемую память возможно обмены между ядрами **одного мультипроцессора**



ВЫДЕЛЕНИЕ РАЗДЕЛЯЕМОЙ ПАМЯТИ

(СТАТИЧЕСКОЕ)

- 2 способа
    - статически
    - динамически
- Статически: В ядре (kernel) объявляем статический массив или переменную с атрибутом __shared__
- Пример:

```
#define SIZE 1024
__global__ void kernel()
   {
    __shared__ int array[SIZE];  // можно выделять как массив
    __shared__ float varSharedMem;  // так и переменную
}
```

ВЫДЕЛЕНИЕ РАЗДЕЛЯЕМОЙ ПАМЯТИ

(ДИНАМИЧЕСКОЕ)

Динамически:

- 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)
```

- Все переменные extern __shared__ type var[] указывают на начало динамической общей памяти, выделенной блоку

ОСОБЕННОСТИ ИСПОЛЬЗОВАНИЯ

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

ПРОБЛЕМА СИНХРОНИЗАЦИИ

(ОБМЕНЫ ДАННЫМИ ЧЕРЕЗ РАЗДЕЛЯЕМУЮ ПАМЯТЬ)

Рассмотрим пример ядра, запускаемого на одномерном линейном гриде:

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

- Каждая нить:
    - Записывает элемент global_data от своего индекса в соответствующую ей ячейку массива разделяемой памяти
    - Читает из массива элемент, записанный соседней нитью
- Чем этот код лучше в сравнении с вариантом global_data[(threadIdx.x + 1)% BLOCK_SIZE] ?

ПРОБЛЕМА СИНХРОНИЗАЦИИ

(ОБМЕНЫ ДАННЫМИ ЧЕРЕЗ РАЗДЕЛЯЕМУЮ ПАМЯТЬ)

- Варпы одного блока 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
- разделяемая память «делится» между нитями одного блока, а значит необходима синхронизация внутри нитей одного и того же блока

СИНХРОНИЗАЦИЯ НИТЕЙ БЛОКА

- Для явной синхронизации внутри блока предусмотрена встроенная функция `void __syncthreads();`
- При вызове этой функции нить блокирует ся 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
- `__syncthreads()` можно вызвать в ветвях условного оператора только если результат его условия одинаков во всех нитях блока, иначе выполнение может зависнуть или стать непредсказуемым

СИНХРОНИЗАЦИЯ НИТЕЙ БЛОКА

(КОРРЕКТНЫЙ ПРИМЕР)

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

- Каждая нить:
    1. Записывает данные от своего индекса в соответствующую ей ячейку массива
    2. Ожидает завершения операций в других нитях
    3. Читает из массива элемент, записанный соседней нитью

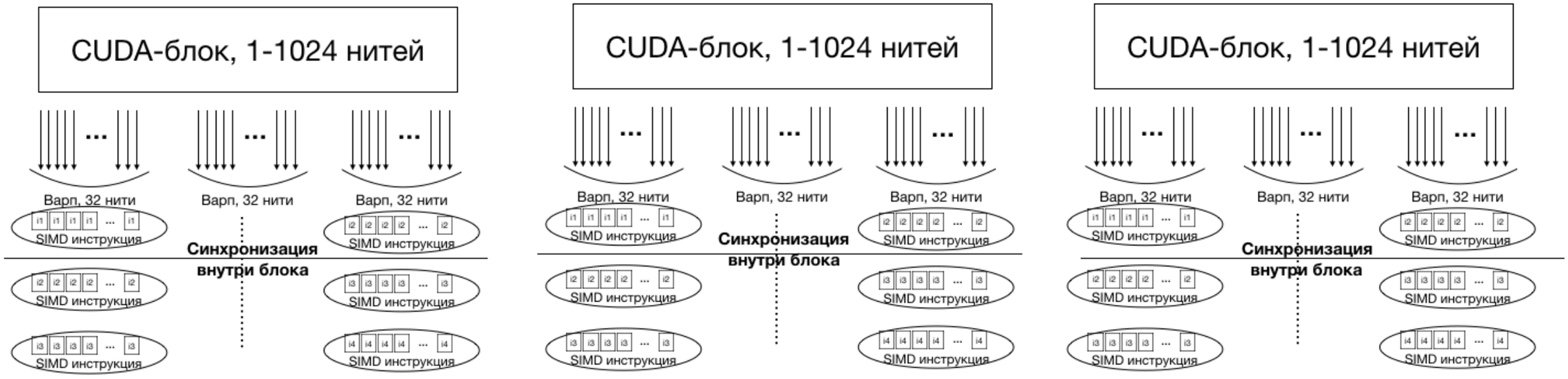
2.

СИНХРОНИЗАЦИЯ ВСЕХ CUDA-НИТЕЙ?

- Итак, конструкция __syncthreads() позволяет синхронизировать нити **внутри одного блока**
- Возможно ли синхронизировать нити различных блоков (или все нити)?
- Порядок вычисления блоков на мультипроцессорах не детерминирован
- Синхронизация нитей возможна только после завершения ядра и запуска нового (!)

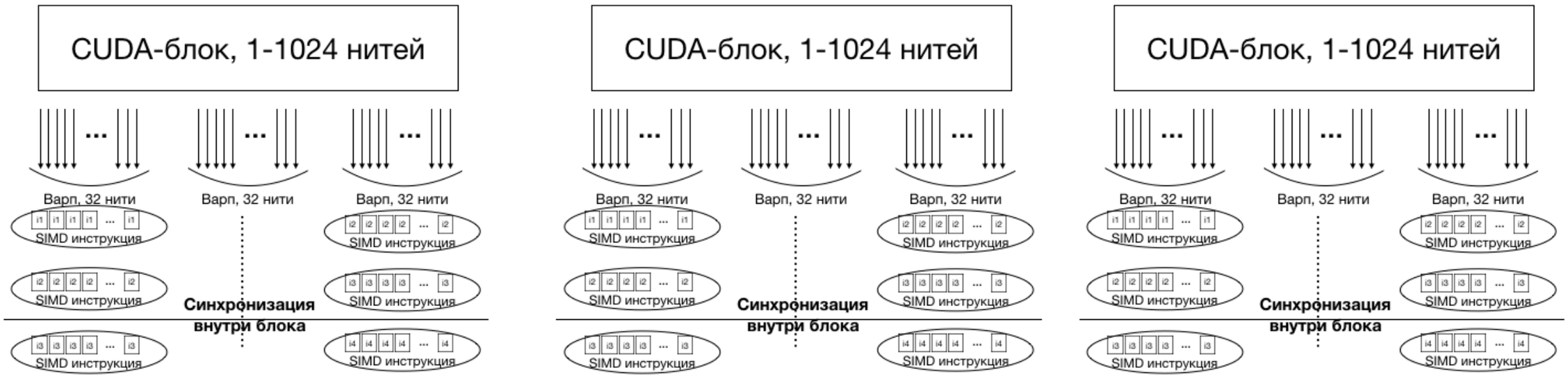
СИНХРОНИЗАЦИЯ НИТЕЙ

ядро 1



Синхронихация после выполнения всех блоков, вызов нового ядра

ядро 2



СТРАТЕГИЯ ИСПОЛЬЗОВАНИЯ РАЗДЕЛЯЕМОЙ ПАМЯТИ (1)

- 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 РАЗДЕЛЯЕМОЙ ПАМЯТИ (2)

- Даже если аппаратный кеш L1 «справляется» с запросами (L1 load hit ~ 100%) использование общей памяти позволяет его дополнительно разгрузить и полностью использовать аппаратуру
- Иначе 16KB (или даже 48KB, если забыли выставить режим) быстрой памяти простаивают

СТРАТЕГИЯ ИСПОЛЬЗОВАНИЯ РАЗДЕЛЯЕМОЙ ПАМЯТИ (3)

- 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

ПРИМЕР ИСПОЛЬЗОВАНИЯ ОБЩЕЙ ПАМЯТИ

(ЯДРО)

```
__global__ void kernel(int sizeOfArray1, int sizeOfArray2, int *devPtr, int *res)
{
    extern __shared__ int dynamicMem[]; // указатель на динамическую общую память
     __shared__ int staticMem[1024]; // статический массив в общей памяти
    __shared__ int var; // переменная в общей памяти
    int *array1 = dynamicMem;//адрес первого массива в динамической общей памяти
    int *array2 = array1 + sizeOfArray1; //адрес второго массива в динамической общей памяти
    staticMem[threadIdx.x] = devPtr[threadIdx.y];// загрузка данных в общую память
    __syncthreads(); // подождать пока все нити завершат запись
    array2[threadIdx.x] = 2 * staticMem[(threadIdx.x - 10) % blockDim.x]; // обратится к элементу, //записанному другой нитью
    __syncthreads(); // подождать пока все нити завершат запись
    res[threadIdx.x] = array2[threadIdx.x]; // записать результаты в глобальную память
}
```

ПРИМЕР ИСПОЛЬЗОВАНИЯ ОБЩЕЙ ПАМЯТИ

(ЗАПУСК ЯДРА НА ХОСТЕ)

```
int *devPtr;
cudaMalloc(&devPtr, 1024*sizeof(int));
kernel<<<3,1024,1024*sizeof(int)>>>(512,512, devPtr);
```

- Запускаются три блока по 1024 нити
- Третий параметр - размер общей памяти динамически выделяемой каждому блоку в байтах
- Таким образом, каждому блоку динамически выделяется по 4KB общей памяти
- Если суммарный (статически + динамически) запрашиваемый объём общей памяти превосходит доступный (16KB или 48KB), произойдёт ошибка запуска ядра

ШАБЛОНЫ ЗАПИСИ В РАЗДЕЛЯЕМУЮ ПАМЯТЬ

Что произойдет, если несколько нитей варпа пытаются записать по одному и тому же адресу ?

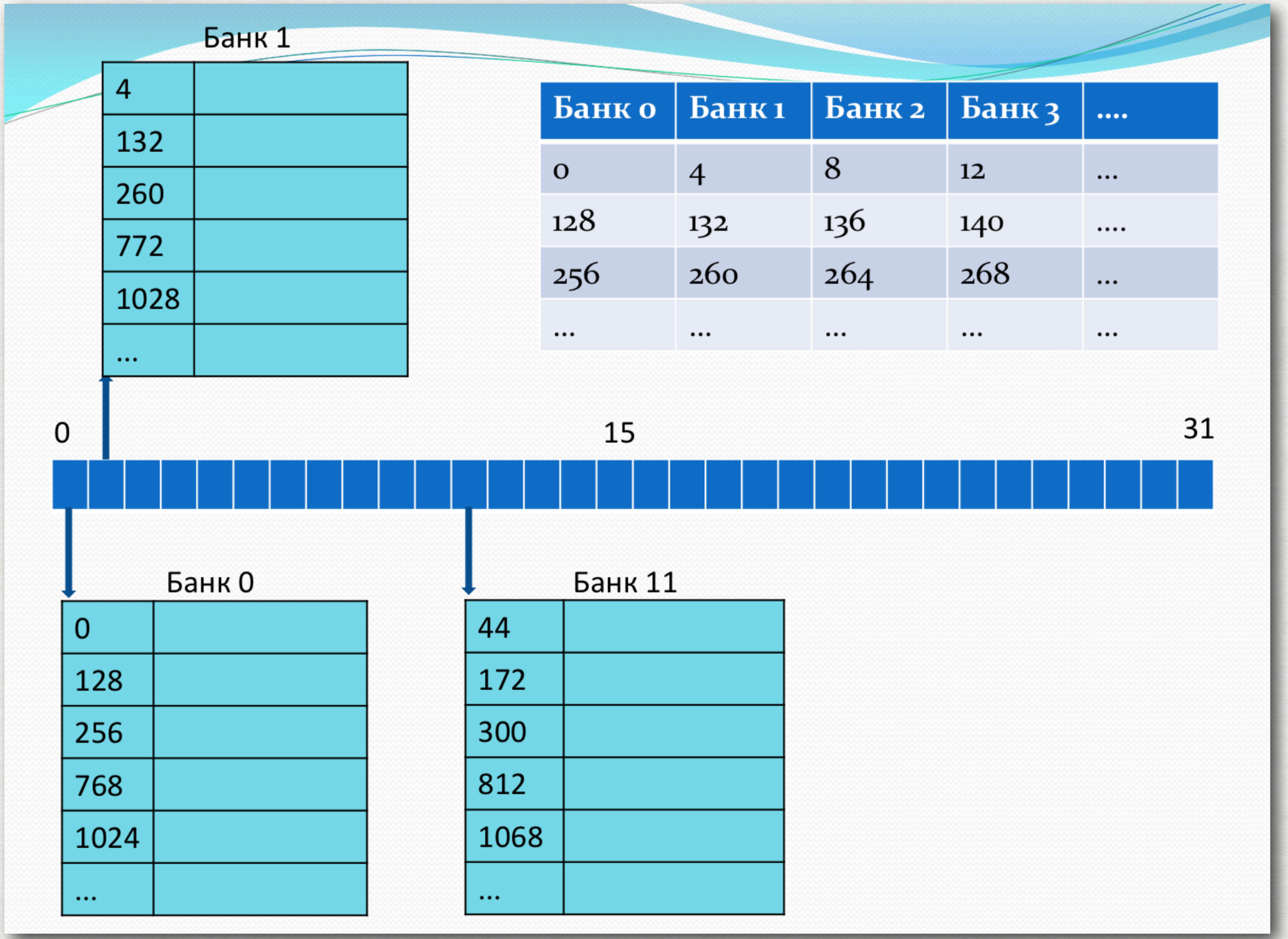
- Запись будет выполнена только одной нитью
- Какой именно - неизвестно
- Если по одному и тому же адресу пишут нити из разных варпов, то результат непредсказуем, т.к. непредсказуем порядок варпов и неизвестно какой варп будет писать последним

БАНКИ ОБЩЕЙ ПАМЯТИ

Общая память разделена на независимые модули одинакового размера, называемые «банками». В рассматриваемой архитектуре 32 банка.

- Последовательные 4-байтные данные располагаются в разных банках
- Номер банка для слова по адресу addr: (addr / 4) % 32
- Каждый банк может выдать за 2 такта одно 32-битное слово (4 байта)

БАНКИ ОБЩЕЙ ПАМЯТИ



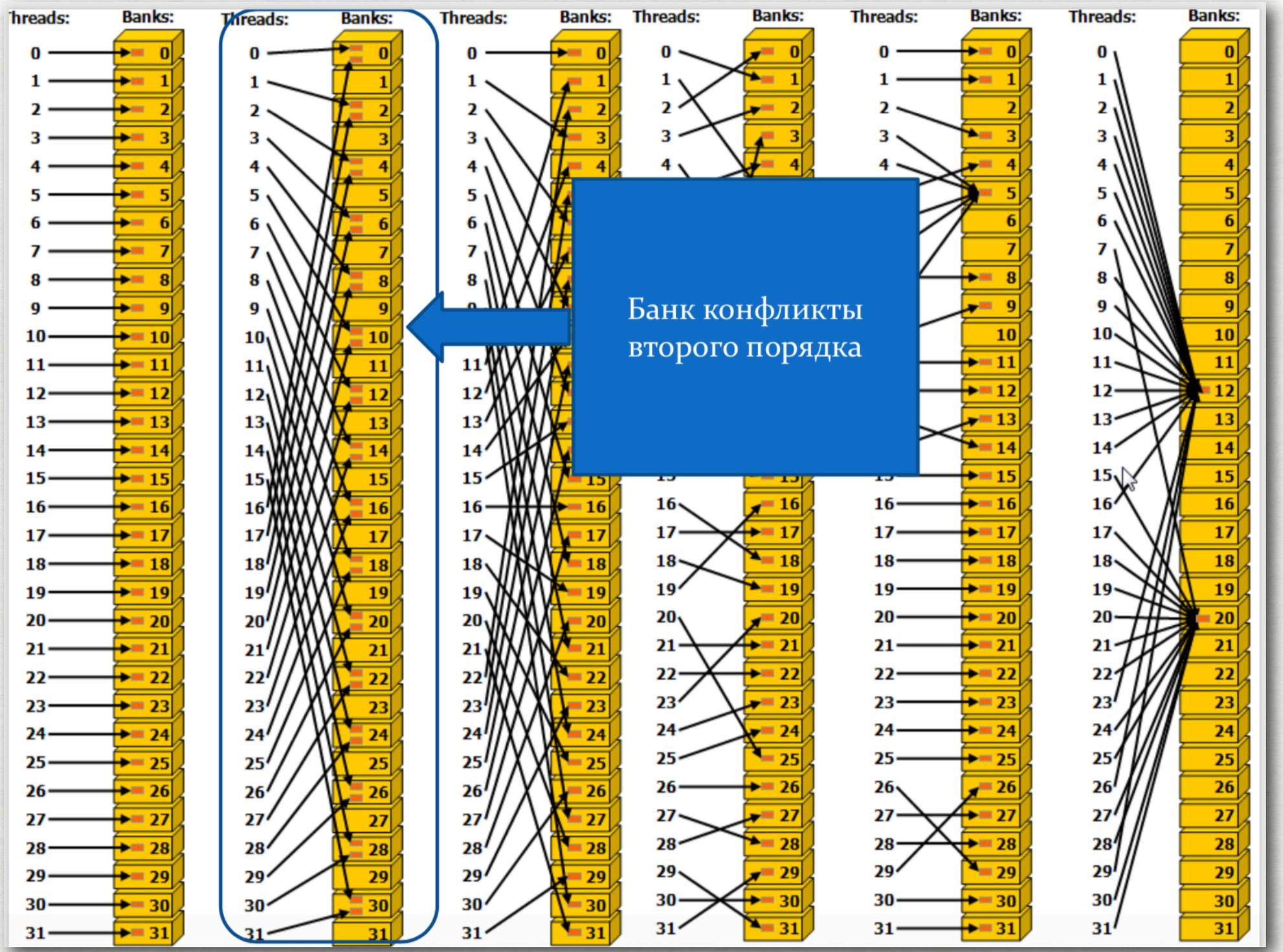
ОБРАЩЕНИЯ В ОБЩУЮ ПАМЯТЬ И БАНКИ

- 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 разделяемой памяти = 32 х пропускная способность банка
- 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

БАНК КОНФЛИКТЫ



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] // конликта 2-го порядка`

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

ПРИМЕРЫ БАНК КОНФЛИКТОВ

(ЧАСТАЯ 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;
```

- Решение: набивка

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

ЗАДАНИЕ 1

- Сделаем «переворачивание» массива небольшого размера в разделяемой памяти
- Опреация: a[i] = a[size - 1 - i]
- Разделяемая память - как быстрый временный буфер данных
- Реализовать 2 варианта:
    - (1) динамическое выделение
    - (2) статическое выделение
- Встроенная проверка корреткности в шаблоне: shared_reverse_template.cu (dropbox)
- Массивы каких размеров можно обработать на одном блоке архитектуры Pascal?

ВОПРОСЫ?