Технология CUDA для высокопроизводительных вычислений на кластерах с графическими процессорами

Колганов Александр
alexander.k.s@mail.ru

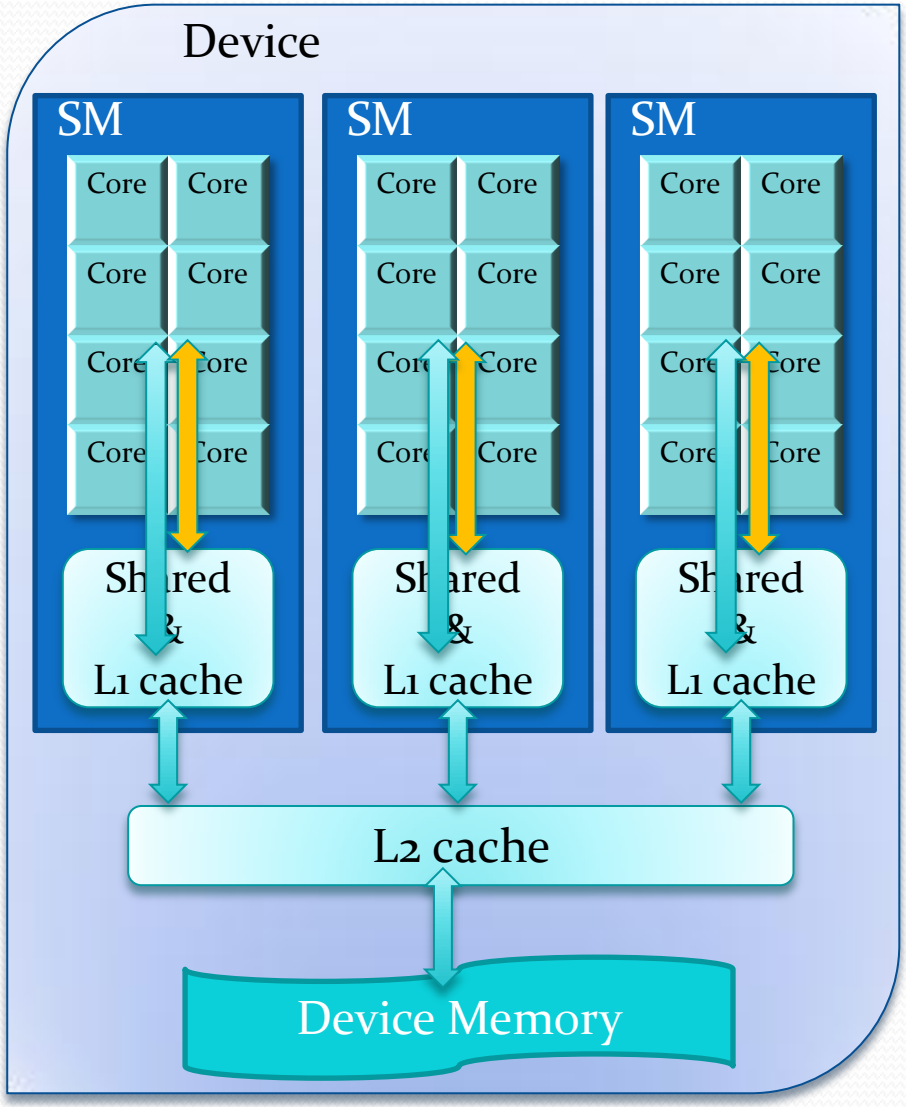
часть 3

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 ограниченное адресное пространство общей памяти

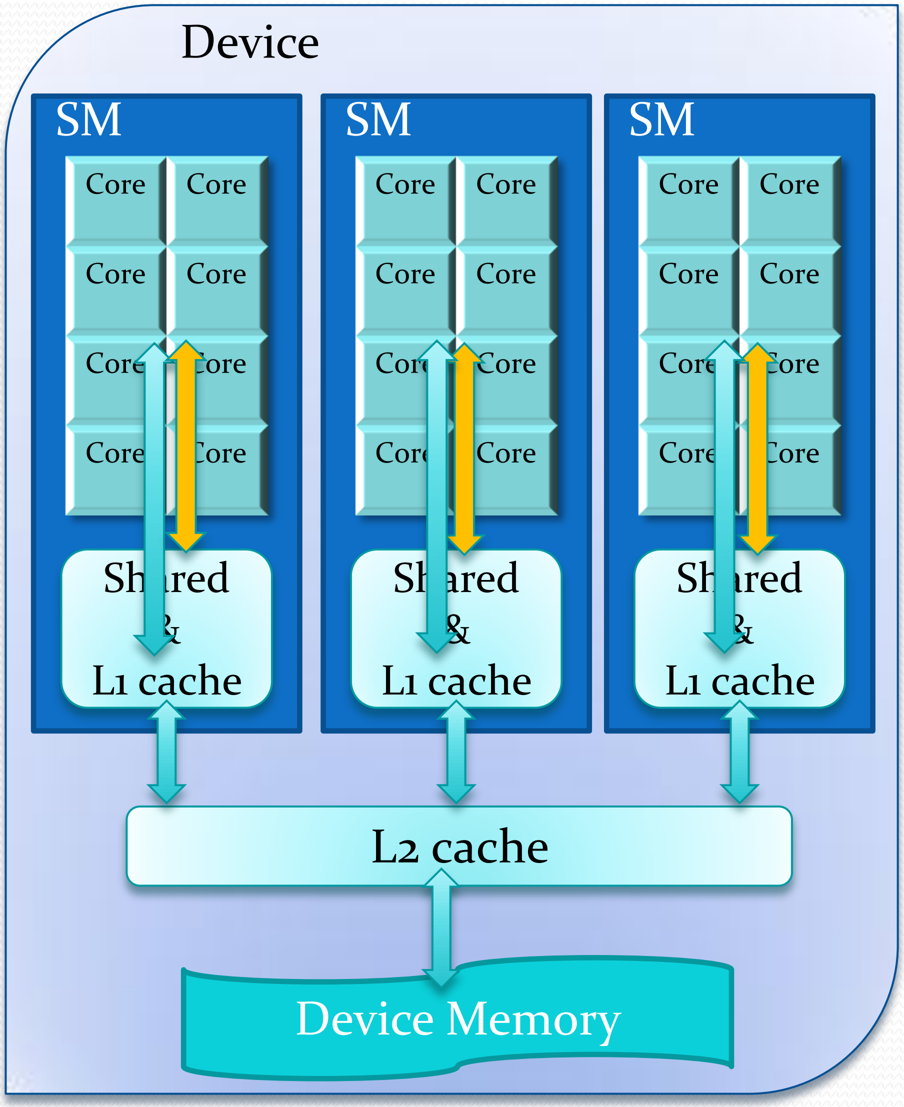
- Конфигурации:
  - 16KB общая память, 48KB L1
  - 48KB общая память, 16KB L1 – по умолчанию



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

- Статически:
  - 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
  …
}
```

Выделение общей памяти

- 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)
```

Особенности использования

- Переменные с атрибутом **__shared__** с точки зрения программирования:
  - Могут быть объявлены в глобальной области видимости или внутри функций
    - При объявлении внутри функции работают как статические, т.е. один экземпляр существует для всех вызовов функции
  - Индивидуальны для каждого блока и привязаны к его личному пространству общей памяти
    - каждый блок нитей видит «своё» значение

Особенности использования

- Переменные с атрибутом **__shared__** с точки зрения программирования:
  - Существуют только на время жизни блока
    - не доступны с хоста или из других блоков
  - Не могут быть проинициализированы при объявлении
- Ядру может быть одновременно выделена и статическая, и динамическая память.

Особенности использования

- Для ядра статически выделяется общая память только если в нём есть использование какой-либо переменной **__shared__**, объявленной без [] в
  - глобальной области видимости
  - в ядре
  - в функциях, которые оно вызывает,
- Все переменные **extern __shared__ type** var**[]** указывают на начало динамической общей памяти, выделенной блоку

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

- Для явной синхронизации внутри блока предусмотрены встроенные функции :

  - `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

```
__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

Синхронизация

- Дополнительные функции:
  - `int __syncthreads_count(int predicate)`

    идентична __syncthreads, но дополнительно возвращает число нитей в блоке, в которых предикат не равен нулю
  - `int __syncthreads_and(int predicate)`

    идентична __syncthreads , но дополнительно возвращает ненулевое значение тогда и только тогда, когда предикат не равен нуля во всех нитях блока
  - `int __syncthreads_or(int predicate)`

    идентична __syncthreads , но дополнительно возвращает ненулевое значение тогда и только тогда, когда предикат не равен нуля хотя бы в одной нити блока

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

- 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

- 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

Пример ядра с общей памятью

```
__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 ядра с общей памятью

```
__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
    //В принципе, тут синхронизация уже не нужна
    res[threadIdx.x] = array2[threadIdx.x]; // записать результаты в глобальную память
}
```

Пример ядра с общей памятью

- На хосте

```
int *devPtr;
cudaMalloc(&devPtr, 1024*sizeof(int));
kernel<<<3,1024,1024*sizeof(int)>>>(512,512, devPtr);
      // запустить три блока по  1024 нити.
      // динамически выделить блокам по 4KB
      // общей памяти
```

Размер общей памяти динамически выделяемой каждому блоку в байтах

- Если суммарный (статически + динамически) запрашиваемый объём общей памяти превосходит доступный (16KB или 48KB), произойдёт ошибка запуска

Пример ядра с общей памятью

```
__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(); // обязательна синхронизация

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

Запись в общую память

- Несколько нитей варпа пытаются записать по одному и тому же адресу
  - Запись будет выполнена только одной нитью
  - Какой именно – неизвестно
    - (Судя по всему – последняя по счёту нить варпа, из тех что должны выполнить запись)
- Если под 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 32-битные слова 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)

Банк 1

4	
132	
260	
772	
1028	
…	

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	
768	
1024	
…	

Банк 11

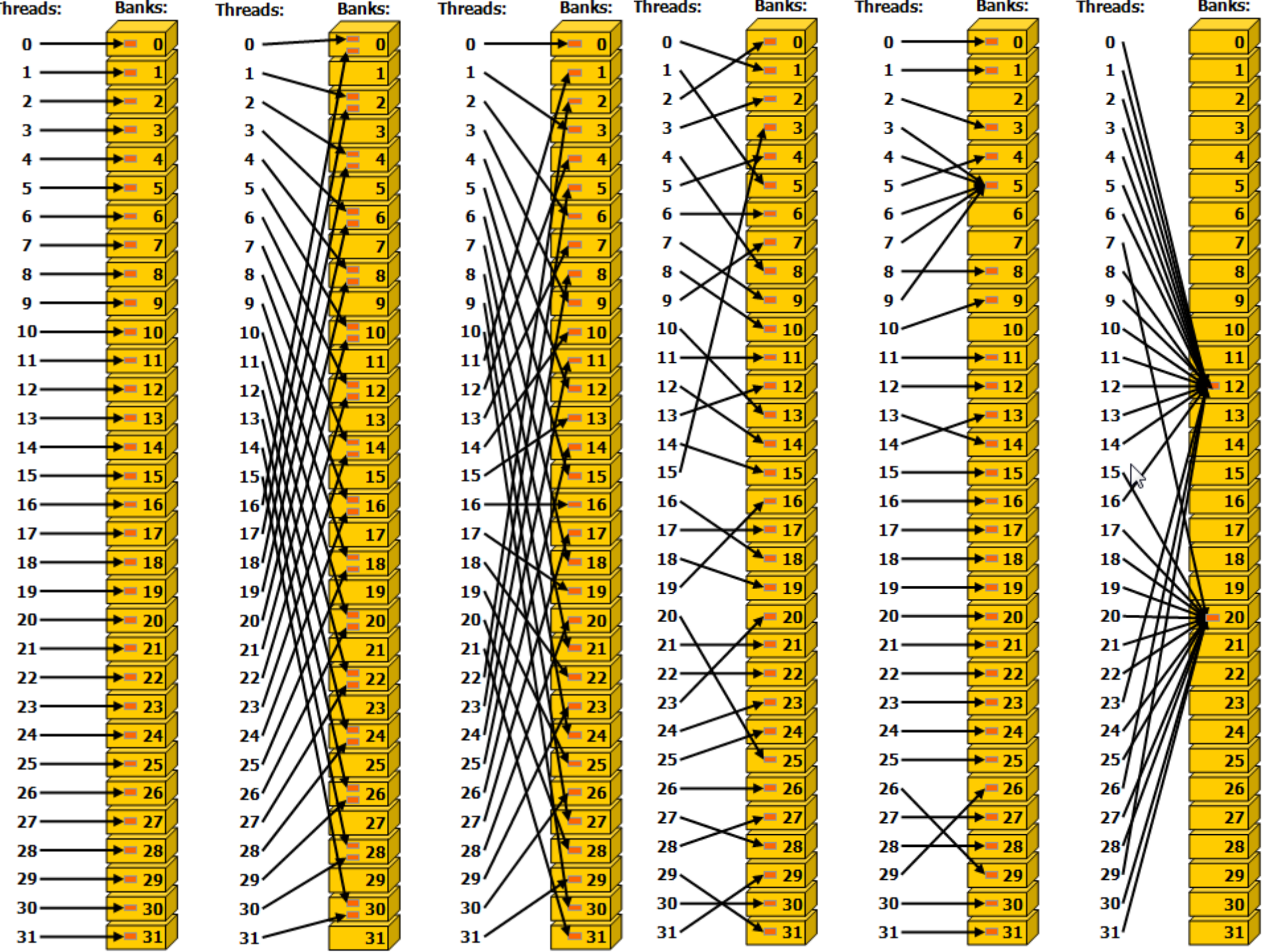
44	
172	
300	
812	
1068	
…	

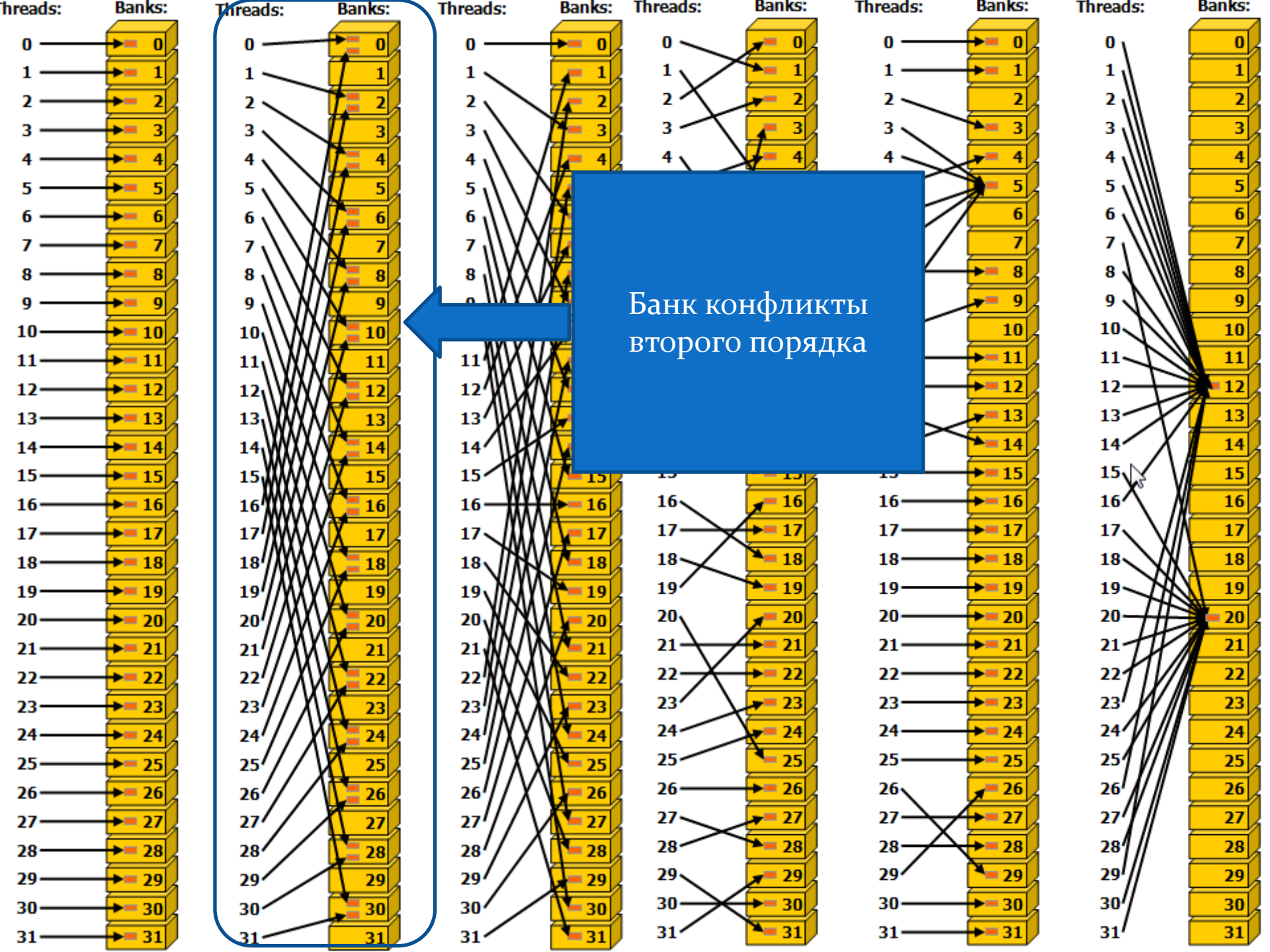
Обращения в общую память

- 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


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 доступа

```
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 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

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

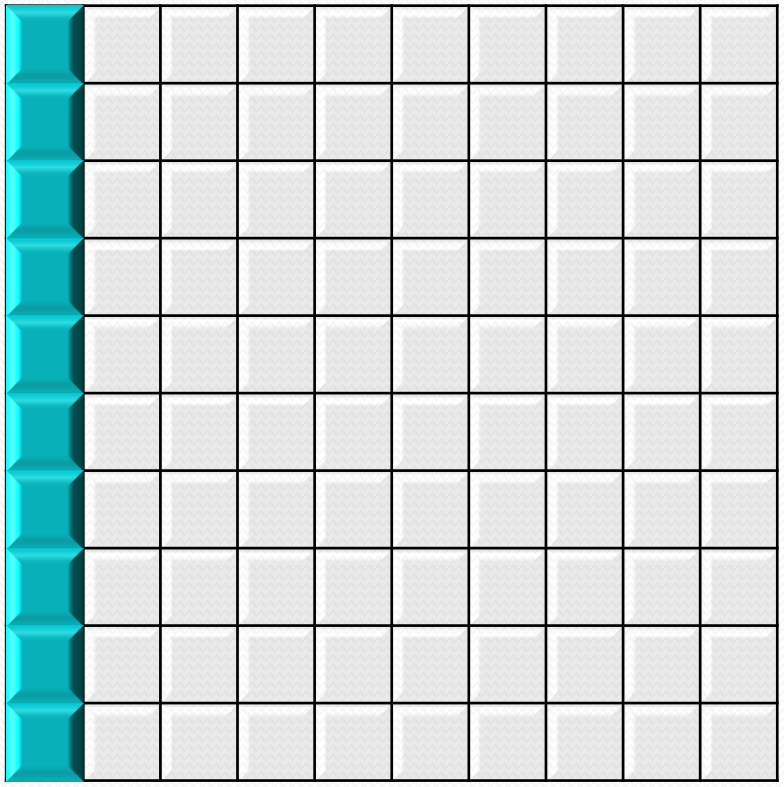
```
__shared__ int matrix[32][32]
matrix[thredIdx.x][4] = 0;
```

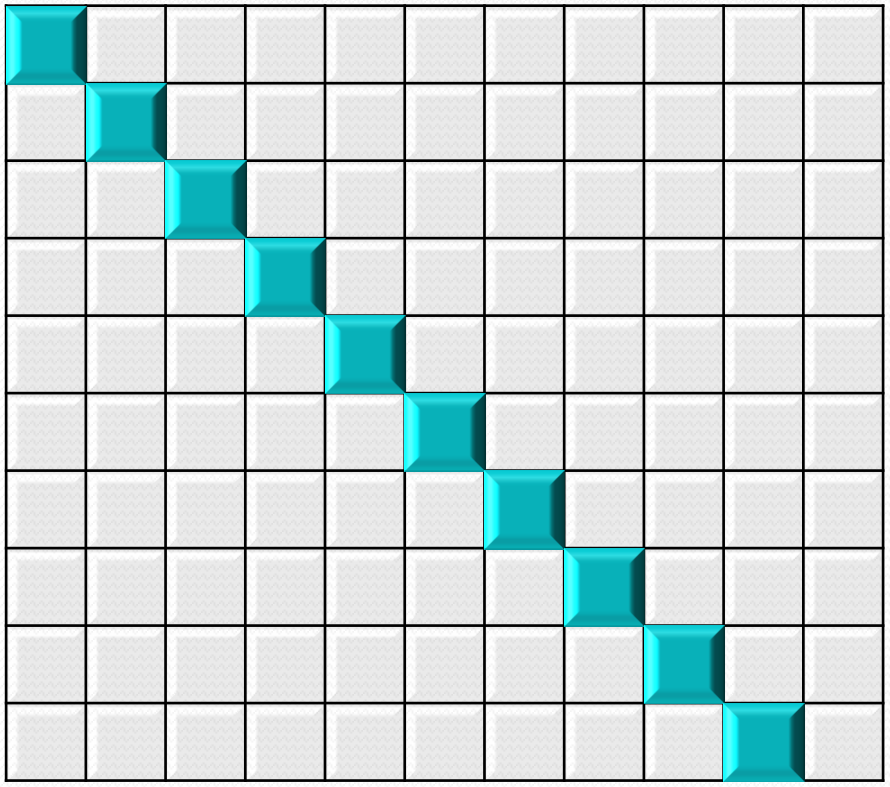
Решение: набивка

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

Распространенная проблема

Пусть банков 10, матрица 10x10

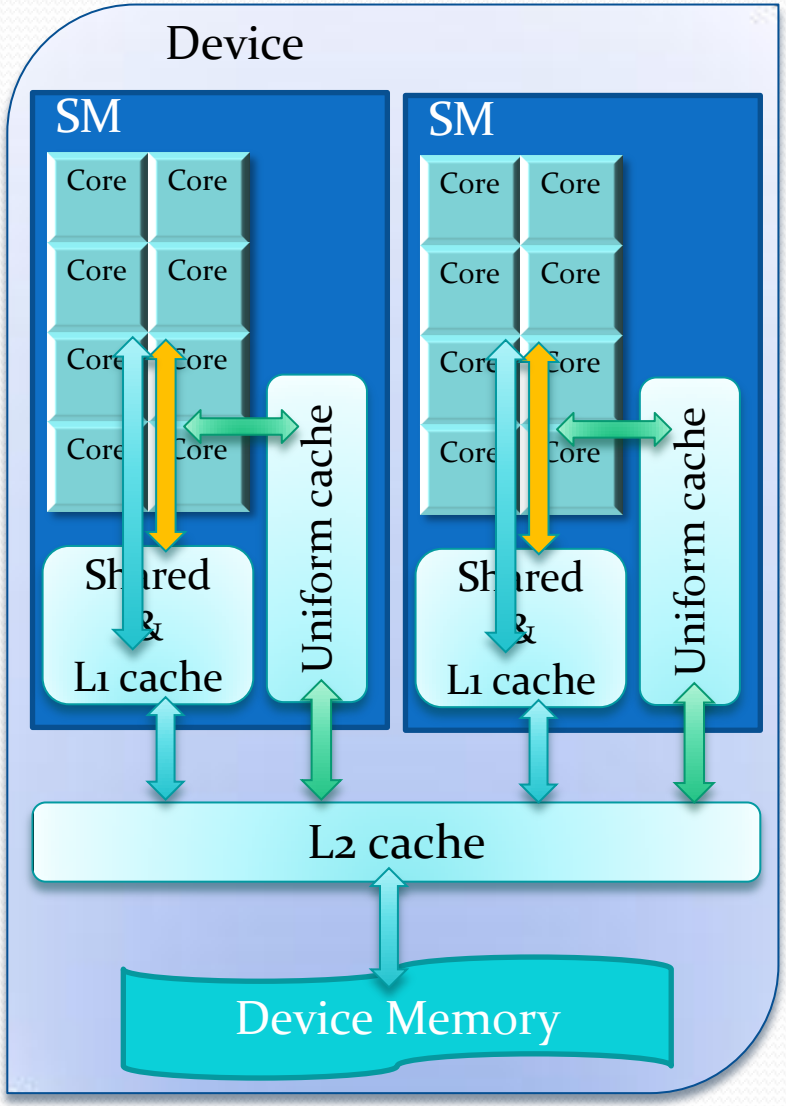




Константная память

Константная память

- Расположена в **DRAM GPU**
- Объём до 64KB
  - Параметр устройства **totalConstMem**
- **Кешируется** в специальном read-only кеше – Unifrom Cache
  - Объём 8 KB

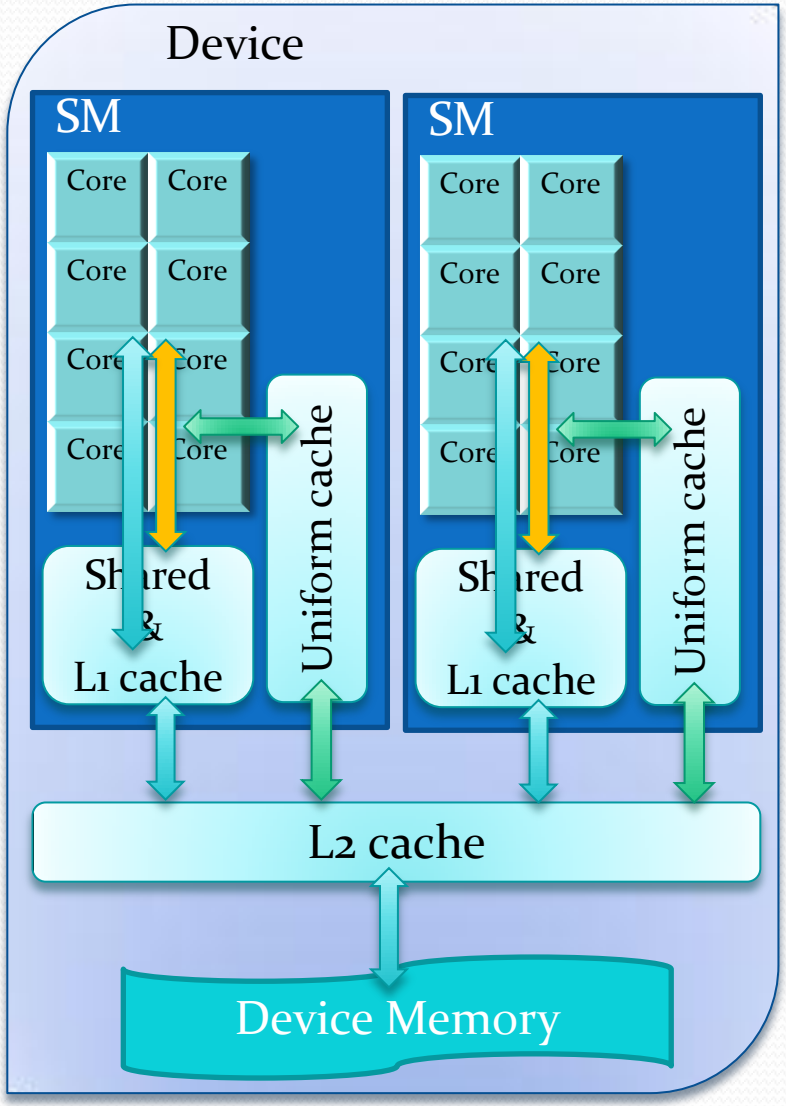


Константная память

Возможные обмены
между устройствами
при обработке обращений в
глобальную память

Возможные обмены
между устройствами
при обработке обращений
в общую память

Возможные обмены
между устройствами
при обработке обращений
в константную память



Объявление

- В глобальной области видимости

```
__constant__ int constMem[1024];
__constant__ int constVar;
```

- Можно ещё дополнительно указать **__device__** , чтобы подчеркнуть, что память выделяется на устройстве :

```
__device__ __constant__ int constVar2;
```

Особенности

- Выделяется при старте приложения, освобождается при завершении приложения

- Доступна **на чтение (и только на чтение!)** из любой нити любого грида обычным способом:

```
__constant__ int constMem[32];
__global__ void kernel() {
    …
    int a = constMem[ threadIdx.x / 32 ];
    …
}
```

- Доступна с хоста при помощи функций тулкита **cudaGetSymbolAddress**() /**cudaGetSymbolSize**() / **cudaMemcpyToSymbol**() / **cudaMemcpyFromSymbol**())

Пример

```
__constant__ float constData[256];
```

- На хосте:

```
float data[256];
cudaMemcpyToSymbol(constData, data, sizeof(data));
cudaMemcpyFromSymbol(data, constData,
sizeof(data));
```

Обращение в константную память

- Обращение выполняется одновременно для всех нитей варпа (SIMT)
- Исходное обращение разбивается на столько запросов, сколько различных адресов в нём было
  - Каждый запрос выполняется либо через запрос к кешу в случае кеш-попадания, либо через глобальную память
  - Если их было n, то пропускная способность уменьшается в n раз

```
__constant__ int constMem[32];
__global__ void kernel() {
    …
    int a = constMem[ threadIdx.x / 32 ]; // 1 запрос в константную
                                             память на варп
    int a = constMem[ threadIdx.x]; // 32 запроса в константную
                                       память на варп
    …
}
```

Однородные обращения

- Помимо обработки запросов в константную память, Uniform Cache обрабатывает «Однородные» обращения (Unifrom Accesses) – когда все нити варпа обращаются в глобальную память по одному адресу
  - При выполнении требований:
    - Доступ только по чтению
    - Адрес не зависит от индекса нити в блоке (**threaIdx**)

```
while(k < 100 ) tmp += a[blockIdx.x + k++];
```

Компилятор заменит в ассемблере обычную инструкцию загрузки из глобальной памяти на инструкцию однородной загрузки, которая будет выполнена через Uniform Cache

Однородные обращения

- При выполнении требований:
  - Доступ только по чтению
  - Адрес не зависит от индекса нити в блоке (**threaIdx**)

```
while(k < 100 ) tmp += a[blockIdx.x + k++];
```

- Второе требование гарантирует, что все нити варпа обращаются по одному адресу

- Чтобы помочь компилятору с первым требованием, можно пометить указатели атрибутом **const**

Передача параметров в ядра

- Параметры передаются в ядра через константную память
  - Параметры передаются в единственном экземпляре для всех нитей грида
  - Это приемлимо, т.к.,
    - в основном, нити варпа обращаются к одному и тому же параметру -> Uniform Access
    - После первого варпа параметры уже будут в кеше
- Суммарный размер передаваемых параметров долен быть **не больше, чем 4 КВ**

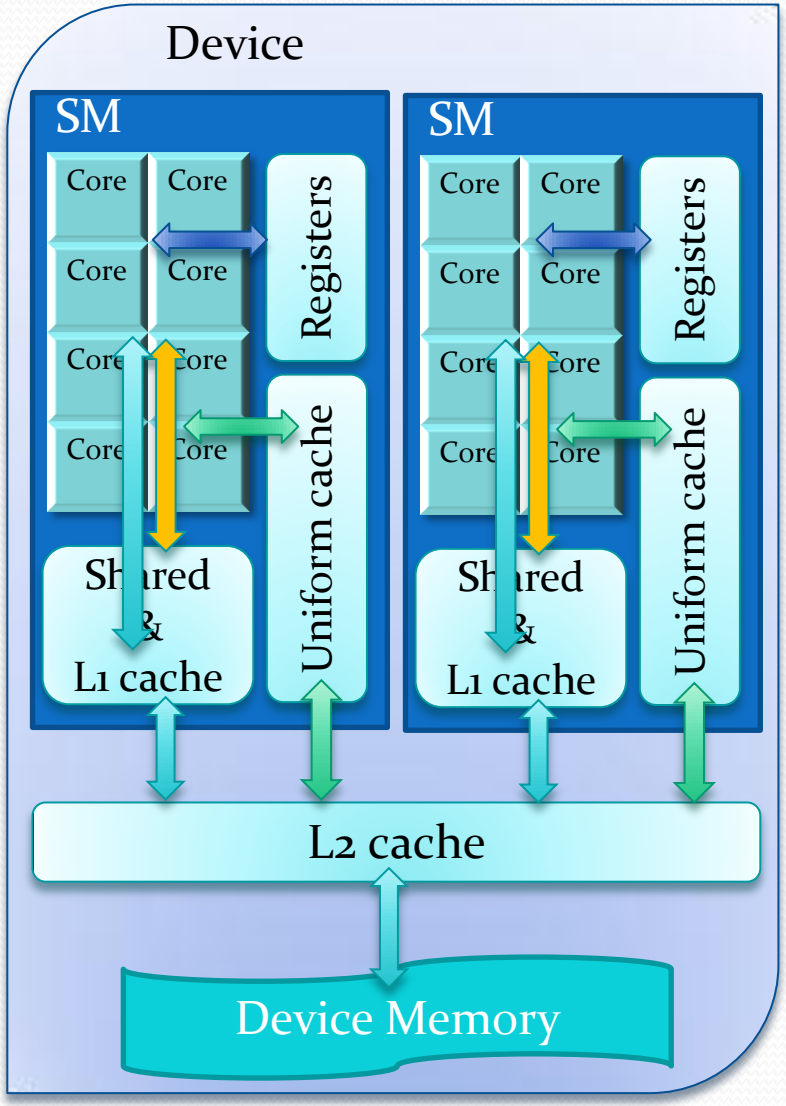
Передача грида в ядра

- Помимо параметров, через константную память передаются размеры грида: **`gridDim`**, **`blockDim`**
  - **`threadIdx`**, **`blockIdx`** нить получает из спец. регистров (заведомо не Uniform)
  - **`gridDim`**, **`blockDim`** нить **считывает из константной памяти** в самом начале работы (Uniform)

Регистры и локальная память

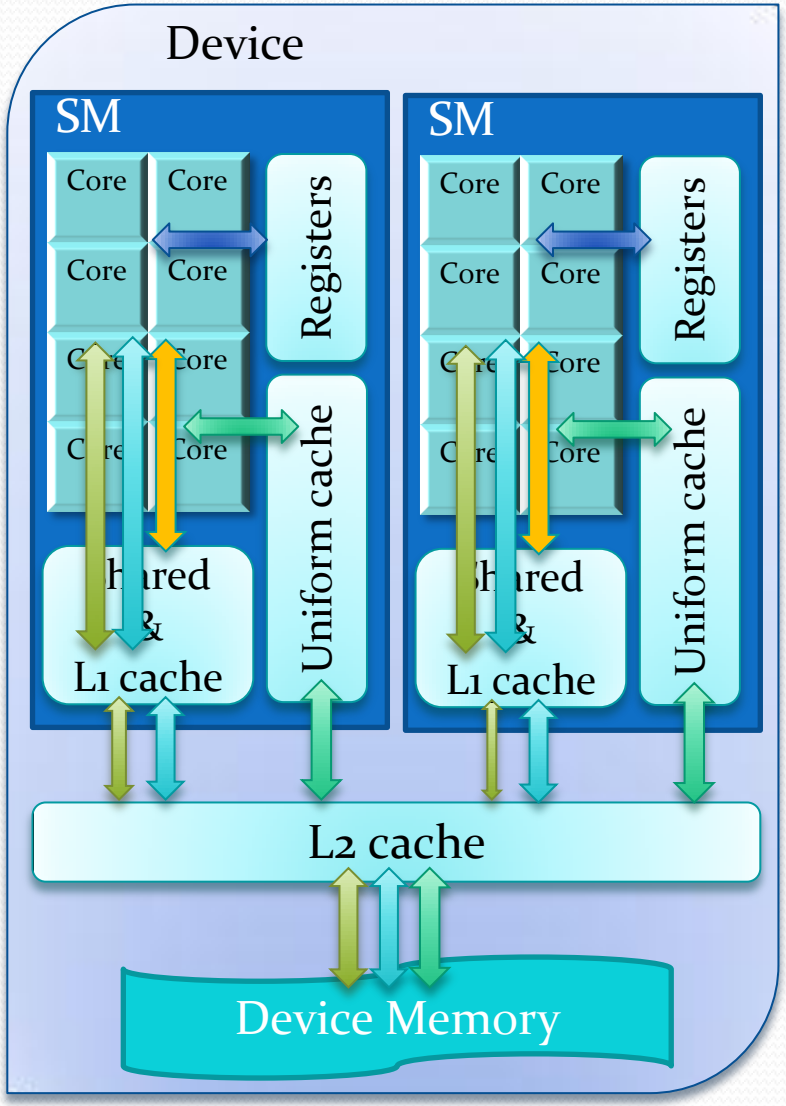
Регистровая память

- Расположена на мультипроцессоре
- Самая быстрая память
- Каждый мультипроцессор содержит 32768 32-битных регитров
  - 128 КВ регистров
  - Параметр устройства **regsPerBlock**
- Отдельной нити доступно максимум 63 регистра
- Распределяются **во время компиляции**
- Каждая нить является эксклюзивным пользователем своих регистров на всё время выполнения ядра. Доступ к регистрам других нитей запрещён



Локальная память

- Расположена в **DRAM**
- Доступ осуществляется по тем же правилам, что и запросы в глобальную память
  - Кеширование в L1
  - Транзакции
- Недоступна явно в программе
- Обладает упрощённой схемой адресации
  - Оптимизирована для минимизации количества транзакций



Когда используется локальная память?

- Обычно, компилятор помещает на регистры все локальные переменные
- Но есть исключения, размещаемые в локальной памяти:
  - Массивы, для которые не всегда можно определить к какому элементу в какой момент времени идёт доступ (не константные индексы)
  - Большие массивы или структуры, которые использовали бы слишком много регистров
  - Любая переменная, если превышен лимит в 63 регистра на нить (так называемый «спиллинг регистров» register spilling)

Когда используется локальная память?

- Некоторые встроенные математические функции могут использовать локальную память
- Через локальную память передаётся часть операндов при вызове функций
  - В т.ч. в локальной памяти моделируется стек фреймов при рекурсивных вызовах

Пример

```
__device__ int deviceFunc(int *a) {
      int x; // скорее всего на регистре
      int array[10]; // точно в локальной
      x = sinf(threadIdx.x) // sinf может
       использовать локальную память
      x = x + array[a[threadIdx.x  + x * 1000]];
     if (x < 100) {
          x = x + deviceFunc(a); // фрейм будет
       //сохранен в стеке, расположенном в лок. памяти
     }
     return x;
}
```

nvcc -Xptxas -v

- Выводит количество регистров, константной памяти, локальной памяти и статической общей памяти, используемые ядром:

```
$:~/programming/testMod$ nvcc -arch=sm_20 -Xptxas -v test.cu
ptxas info    : 0 bytes gmem, 8 bytes cmem[2]
ptxas info    : Compiling entry function '_Z13matmul_kernelv' for
'sm_20'
ptxas info    : Function properties for _Z13matmul_kernelv
    8 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 8 registers, 4 bytes smem, 32 bytes cmem[0]
```

Статические ресурсы и occupancy

- Факторы, влияющие на occupancy:
  - Не более 1536 нитей на sm, не более 8 блоков
  - Не более 48KB общей памяти на sm
  - 32768 регистров на sm

Пусть ядро использует 63 регистра и размер блока 384 нити
32768 / 63 = 520 – максимум нитей

- на sm будет работать всего один блок из 384 нитей (occupancy = 0.25)

Статические ресурсы и occupancy

- Факторы, влияющие на occupancy:
  - Не более 1536 нитей на sm, не более 8 блоков
  - Не более 48KB общей памяти на sm
  - 32768 регистров на sm

Пусть на блок из 512 нитей нужно 32KB общей памяти

- на sm будет работать всего один блок из 512 нитей (occupancy = 0.33)