Иерархия памяти в CUDA. Глобальная память.

Лектор:

Боресков А.В. (ВМК МГУ) (steps3d@narod.ru)

План

- CUDA Compute Capability
- Типы памяти в CUDA
- Основы CUDA C API

План

- CUDA Compute Capability
  - Получение информации о GPU
- Типы памяти
- Основы CUDA C API

CUDA Compute Capability

- Возможности GPU обозначаются при помощи *Compute Capability*, например 1.1
- Старшая цифра соответствует архитектуре
- Младшая – небольшим архитектурным изменениям
- Можно получить из полей *major* и *minor* структуры ***cudaDeviceProp***

CUDA Compute Capability

GPU Family	Compute Capability Major
Tesla (GeForce 8800 GTX)	1
Fermi (GeForce 480 GTX)	2
Kepler (GeForce 660)	3
Maxwell (GeForce 960)	5
Pascal	6
Volta	7

Определяется архитектурой GPU, не имеет ничего общего с версией CUDA

CUDA Compute Capability

- Compute Caps. – доступная версия CUDA
  - Разные возможности HW
  - Пример:
    - В 1.1 добавлены атомарные операции в global memory
    - В 1.2 добавлены атомарные операции в shared memory
    - В 1.3 добавлены вычисления в double
    - В 2.0 добавлены управление кэшем и др. операции
- Сегодня Compute Caps:
  - Влияет на правила работы с глобальной памятью
- На курсе рассмотрим 2.0 – 7.5

Получение информации о GPU

```
int main ( int argc, char *  argv [] )
{
    int             deviceCount;
    cudaDeviceProp  devProp;

    cudaGetDeviceCount ( &deviceCount );
    printf             ( "Found %d devices\n", deviceCount );
    for ( int device = 0; device < deviceCount; device++ )
    {
       cudaGetDeviceProperties ( &devProp, device );
       printf ( "Device %d\n", device );
       printf ( "Compute capability     : %d.%d\n", devProp.major, devProp.minor );
       printf ( "Name                   : %s\n",  devProp.name );
       printf ( "Total Global Memory    : %ld\n", devProp.totalGlobalMem );
       printf ( "Shared memory per block: %d\n",  devProp.sharedMemPerBlock );
       printf ( "Registers per block    : %d\n",  devProp.regsPerBlock );
       printf ( "Warp size              : %d\n",  devProp.warpSize );
       printf ( "Max threads per block  : %d\n",  devProp.maxThreadsPerBlock );
       printf ( "Total constant memory  : %d\n",  devProp.totalConstMem );
    }
    return 0;
}
```

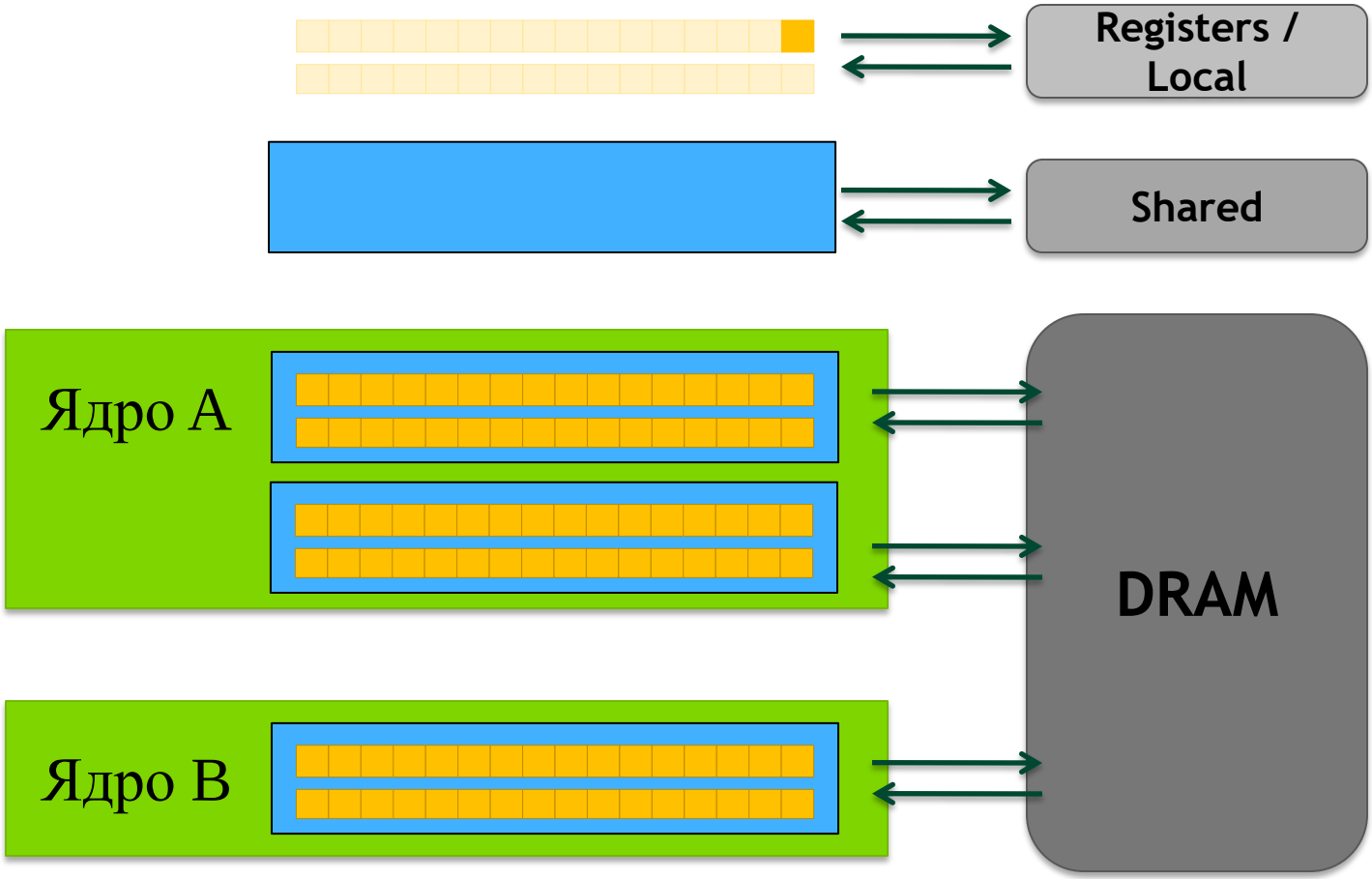
Выбор GPU

- В системе может быть установлено несколько поддерживающими CUDA GPU
  - Возможно с разными Compute Capability
- Описанный выше код позволяет получить информацию обо всех поддерживаемых GPU
- Для выбора GPU как текущего используется вызов
  - `cudaSetDevice ( int device )`
  - Контекст создается неявно при вызове CUDA,
  - Никакой специальной инициализации не нужно
  - Контекст можно также уничтожить
    - `cudaDeviceReset ()`

План

- CUDA Compute Capability
- Типы памяти в CUDA
  - Глобальная
- Основы CUDA C API

Типы памяти в CUDA



Типы памяти в CUDA

Тип памяти	Доступ	Уровень выделения	Скорость работы	Расположение
Регистры	R/W	Per-thread	Высокая	SM
Локальная	**R/W**	**Per-thread**	**Низкая**	**DRAM**
Shared	R/W	Per-block	Высокая	SM
Глобальная	**R/W**	**Per-grid**	**Низкая**	**DRAM**
Constant	R/O	Per-grid	Высокая	DRAM
Texture	R/O	Per-grid	Высокая	DRAM

Легенда:
-интерфейсы доступа

Типы памяти

- Быстродействие приложения очень сильно определяется доступом к памяти и используемыми типами памяти
  - Гораздо серьезнее чем на CPU
- Shared и регистры – самая быстрая
- Глобальная (DRAM) – самая медленная
- В ряде случаев можно использовать константную и текстурную память, но у нее свои особенности
- Доступ к памяти для варпа
  - Целиком для всего варпа для CC >= 2.0
  - По полуварпам для CC 1.x

План

- CUDA Compute Capability
- Типы памяти в CUDA
- Основы CUDA C API
  - Выделение глобальной памяти
    - Пример: умножение матриц
    - Coalescing
    - Pitch linear
    - Pinned
  - Работа с глобальной памятью

Основы CUDA C API

- В CUDA есть два API
  - Высокоуровневый CUDA C API (мы будем разбирать его)
  - Низкоуровневый CUDA Driver API
- Внутри каждого API есть определенные соглашения
  - Все функции CUDA C API начинаются с `cuda`
  - Все функции CUDA C API возвращают код ошибки (`cudaError_t`)
    - `cudaSuccess` - ошибки нет, все в порядке
  - Многие функции на самом деле асинхронны
    - Запрос помещается в очередь GPU и управление сразу же возвращается, хотя запрос даже не начал выполняться
    - Асинхронные запросы
      - Запуск ядра
      - Копирование памяти `*Async`
      - Копирование памяти device <-> device
      - Инициализация памяти (`cudaMemset`)

Основы CUDA C API

- Не требуют явной инициализации
- Все функции возвращают cudaError_t
  - cudaSuccess в случае успеха
- Начинаются с cuda
- Многие функции API асинхронны:
  - Запуск ядра
  - Копирование функциями *Async
  - Копирование device <-> device
  - Инициализация памяти

Основы CUDA C API

```
// Получение информации о существующих в системе GPU
cudaError_t cudaGetDeviceCount    ( int * );
cudaError_t cudaGetDevicePropertis ( cudaDeviceProp * props, int deviceNo );
// Получение информации об ошибках
char      * cudaGetErrorString    ( cudaError_t );
cudaError_t cudaGetLastError      ();
// Синхронизация исполнения в текущем CPU потоке и в CUDA stream'е
cudaError_t cudaDeviceSynchronize ();
cudaError_t cudaStreamSynchronize ( cudaStream_t * stream );
// Средства управления событиями
cudaError_t cudaEventCreate       ( cudaEvent_t * );
cudaError_t cudaEventRecord       ( cudaEvent_t *, cudaStream_t );
cudaError_t cudaEventQuery        ( cudaEvent_t );
cudaError_t cudaEventSynchronize  ( cudaEvent_t );
cudaError_t cudeEventElapsedTime  ( float * time, cudaEvent_t st, cudaEvent_t sp );
cudaError_t cudaEventDestroy      ( cudaEvent_t );
```

Функции для работы с глобальной памятью

```
cudaError_t cudaMalloc      ( void ** devPtr, size_t size );
cudaError_t cudaMallocPitch ( void ** devPtr, size_t * pitch,
                              size_t width, size_t height );
cudaError_t cudaMalloc3D    ( cudaPitchedPtr * pitchedDevPtr,
                    cudaExtent extent );
cudaError_t cudaFree        ( void * devPtr );
cudaError_t cudaMemcpy      ( void * dst, const void * src, size_t count,
                              enum cudaMemcpyKind kind );
cudaError_t cudaMemcpyAsync ( void * dst, const void * src,
                              size_t count,
                              enum cudaMemcpyKind kind,
                              cudaStream_t stream );
cudaError_t cudaMemset      ( void * dst, int value, size_t count );
```

Работа с глобальной памятью

- В 32-бита нет способа отличить указатель в память GPU от указателя в память CPU
  - Обращение не на той стороне ведет к crash
  - Венгерская нотация (тот редкий случай, когда подходит)
- В 64 бита используется унифицированное адресное пространство, в котором находится память CPU и память всех GPU
  - Можно проверить указатель - `cudaPointerGetAttributes`
- В команде копирования памяти указывается направление копирования (в 32 бита иначе его не задать)
  - В 64 бита можно смело писать `cudaMemcpyDefault` - мы просим систему саму понять где какая память

Работа с глобальной памятью в CUDA

- Пример работы с глобальной памятью

```
float * devPtr;           // pointer to device memory

                          // allocate device memory
cudaMalloc ( (void **) &devPtr, 256*sizeof ( float );


                          // copy data from host to device memory
cudaMemcpy ( devPtr, hostPtr, 256*sizeof ( float ), cudaMemcpyHostToDevice );


                          // process data ...


                          // copy results from device to host
cudaMemcpy ( hostPtr, devPtr, 256*sizeof( float ), cudaMemcpyDeviceToHost );


                          // free device memory
cudaFree   ( devPtr );
```

Работа с глобальной памятью

- Вся выделяемая память выровнена как минимум по 256
- Команды для чтения 1/2/4/8/16-байтовых слов за раз
- **Выравнивание данных**
- Работа с памятью зависит от CC
- CC 3.x
  - Кэшируется в L2, может также кэшироваться в L1
  - L1 & L2 – 128 bytes cache line
  - Только L2 – 32 bytes cache line
  - Кэширование в L1 включается опцией компилятора `-Xptxas -dlcm=ca`
  - Запросы к памяти – блоками по 128 байт, затем разбиваются на cache lines
- CC 5.x
  - Кэшируется в L2 также, как и для 3.x
  - Constant data can be cached in unified L1/texture cache (через функцию `__ldg`)
- CC 6.x и 7.x
  - Как и для 5.x

Пример: умножение матриц

- Произведение двух квадратных матриц *A* и *B* размера *N***N*, *N* кратно 16
- Матрицы расположены в глобальной памяти
- По одной нити на каждый элемент произведения
  - 2D блок – 16*16
  - 2D *grid*

Умножение матриц. Простейшая реализация.

```
#define BLOCK_SIZE 16
__global__ void matMult ( float * a, float * b, int n, float * c )
{
  int   bx  = blockIdx.x;
  int   by  = blockIdx.y;
  int   tx  = threadIdx.x;
  int   ty  = threadIdx.y;
  float sum = 0.0f;
  int   ia  = n * BLOCK_SIZE * by + n * ty;
  int   ib  = BLOCK_SIZE * bx + tx;
  int   ic  = n * BLOCK_SIZE * by + BLOCK_SIZE * bx;

  for ( int k = 0; k < n; k++ )
    sum += a [ia + k] * b [ib + k*n];

  c [ic + n * ty + tx] = sum;
}
```

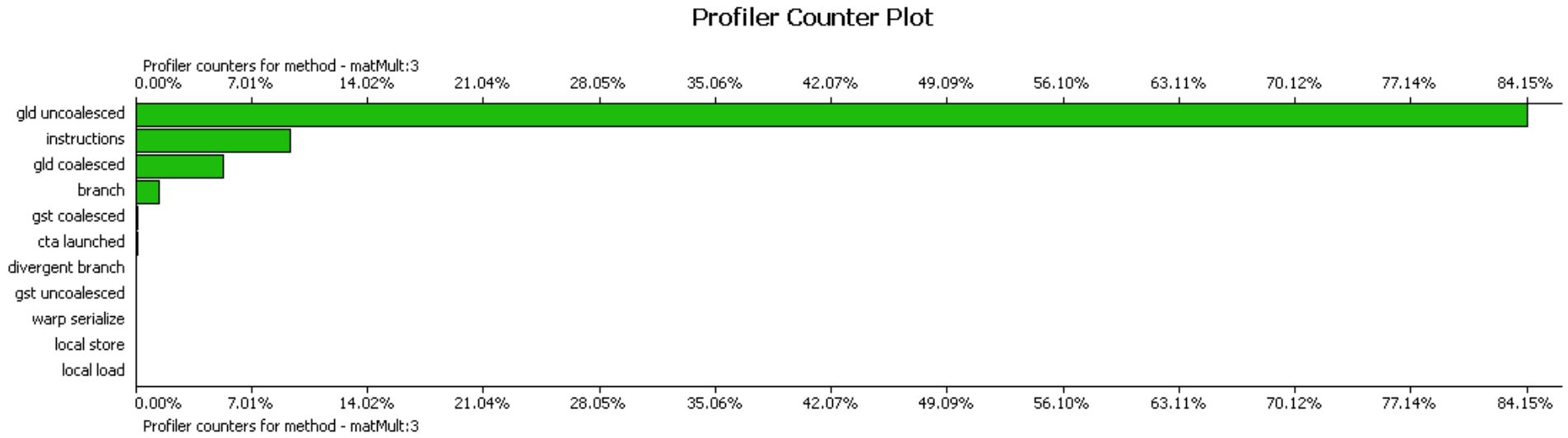
Умножение матриц. Простейшая реализация.

```
int          numBytes = N * N * sizeof ( float );
float      * adev, * bdev, * cdev ;
dim3         threads ( BLOCK_SIZE, BLOCK_SIZE );
dim3         blocks ( N / threads.x, N / threads.y);
cudaMalloc       ( (void**)&adev, numBytes ); // allocate DRAM
cudaMalloc       ( (void**)&bdev, numBytes ); // allocate DRAM
cudaMalloc       ( (void**)&cdev, numBytes ); // allocate DRAM
                          // copy from CPU to DRAM
cudaMemcpy ( adev, a, numBytes, cudaMemcpyHostToDevice );
cudaMemcpy ( bdev, b, numBytes, cudaMemcpyHostToDevice );
matMult<<<blocks, threads>>> ( adev, bdev, N, cdev );
cudaThreadSynchronize();
cudaMemcpy             ( c, cdev, numBytes, cudaMemcpyDeviceToHost );
                           // free GPU memory
cudaFree               ( adev );
cudaFree               ( bdev );
cudaFree               ( cdev );
```

Простейшая реализация.

- На каждый элемент
  - 2*N арифметических операций
  - 2*N обращений к глобальной памяти
- *Узкое место – доступ в память*

Используем CUDA Profiler



- Основное время (84.15%) ушло на чтение из глобальной памяти
- Вычисления заняли всего около 10%

Работа с памятью в CUDA

- Основа оптимизации – правильная работа с памятью:
  - Максимальное использование *shared*-памяти
    - Лекция 4
  - Использование специальных шаблонов доступа к памяти
    - Coalescing

Оптимизация работы с глобальной памятью.

- Команды поддерживают чтение за раз 1/2/4/8/16-байтовых слов
- При обращении к *t[i]*
  - *sizeof(t [0])* равен 4/8/16 байтам
  - *t [i]* выровнен по *sizeof (t [0])*
- Вся выделяемая память всегда выровнена по 256 байт

Использование выравнивания.

```
struct vec3
{
  float x, y, z;
};
```

- **Размер равен 12 байт**
- **Элементы массива не будут выровнены в памяти**

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

- **Размер равен 16 байт**
- **Элементы массива всегда будут выровнены в памяти**
- **Вся структура может прочитана за одну команду**

Объединение запросов к глобальной памяти.

- GPU умеет объединять ряд запросов к глобальной памяти в транзакцию одного сегмента
- Длина сегмента должна быть 32/64/128 байт
- Сегмент должен быть выровнен по своему размеру

Объединение (coalescing) 2.x

- На мультипроцессоре есть L1 кэш
  - Физически там, где разделяемая память
- Мультипроцессоры имеют общий L2 кэш
- Флаги компиляции
  - Использовать L1 и L2 :-Xptxas -dlcm=ca
  - Использовать L2 :-Xptxas -dlcm=cg
- Кэш линия 128B
- Объединение происходит на уровне варпов

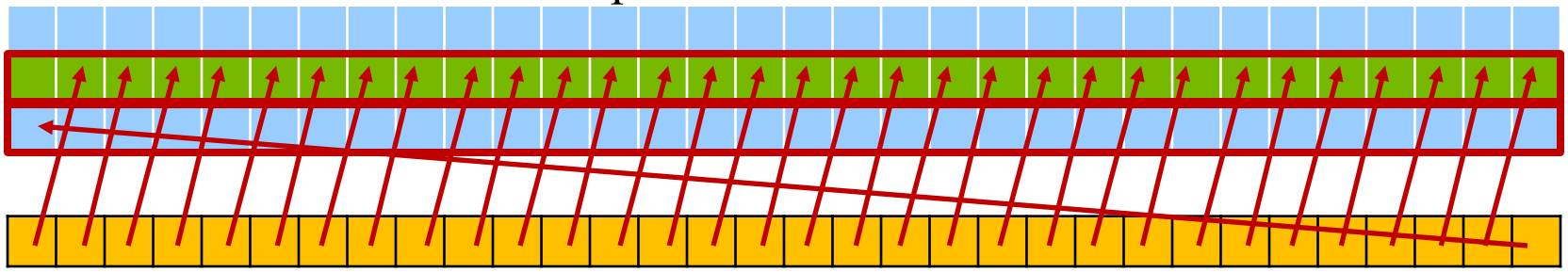
Объединение (coalescing) 3.x

- Обычно кэшируется только в L2 также как и для 2.x
- Объединение происходит на уровне варпов
- Read-only данные могут также кэшироваться константным кэшем

Объединение (coalescing) 2.x

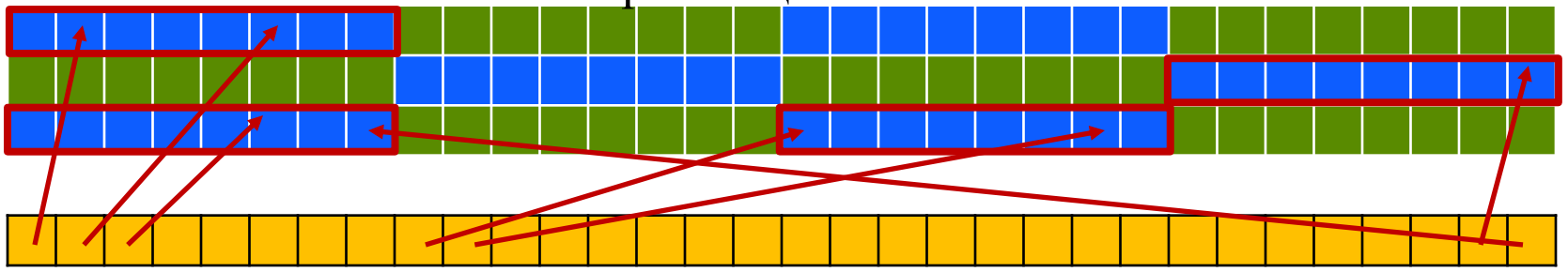
- Если L1 кэш включен: всегда 128B сегменты

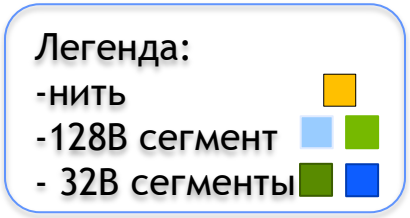
2 транзакция по 128B



- Если L2 кэш выключен: всегда 32B сегменты

4 транзакция по 32B





Объединение (coalescing)

- Увеличения скорости работы с памятью на порядок
- Лучше использовать не массив структур, а набор массивов отдельных компонент
  - Проще гарантировать условия выполнения *coalescing'а*

Использование отдельных массивов

```
struct vec3
{
  float x, y, z;
};
vec3 * a;

float x = a [threadIdx.x].x;
float y = a [threadIdx.x].y;
float z = a [threadIdx.x].z;
```

Не можем использовать *coalescing* при чтении данных

```
float * ax, * ay, * az;

float x = ax [threadIdx];
float y = ay [threadIdx];
float z = az [threadIdx];
```

Поскольку нити одновременно обращаются к последовательно лежащим словам памяти, то будет происходить *coalescing*

Pitch linear

- Для работы с 2D данными
  - `cudaMallocPitch(&ptr, &p, w, h)`
  - В p возвращает ширину выделенной памяти в байтах
  - p ≥ w * sizeof()
- Для 1.x p кратно 64
- Для 2.x p кратно 128

cudaMalloc3D

- Для работы с 3D данными
  - `cudaMalloc3D(&ptr, extent)`

```
cudaExtent extent;
cudaPitchedPtr ptr;
extent = make_cudaExtent(width, height, depth);
cudaMalloc3D ( &ptr, extent);

// Access: ptr.pitch, ptr.ptr
```

Pinned память

X2 производительности

- Для ускорения передачи по PCI-E
  - `cudaMallocHost(&hostPtr, size)`
  - `cudaHostAlloc(&hostPtr, size, flag)`
  - `cudaHostFree(hostPtr)`
- Флаги при выделении памяти
  - **`DEFAULT`** : эквивалентно cudaMallocHost
  - **`PORTABLE`** : для работы со множеством GPU из одного потока
  - **`MAPPED`** : для систем с общей памятью
  - **`WRITE-COMBINED`** : память не кэшируется на CPU, передача по PCI-E быстрее, чтение на CPU медленное

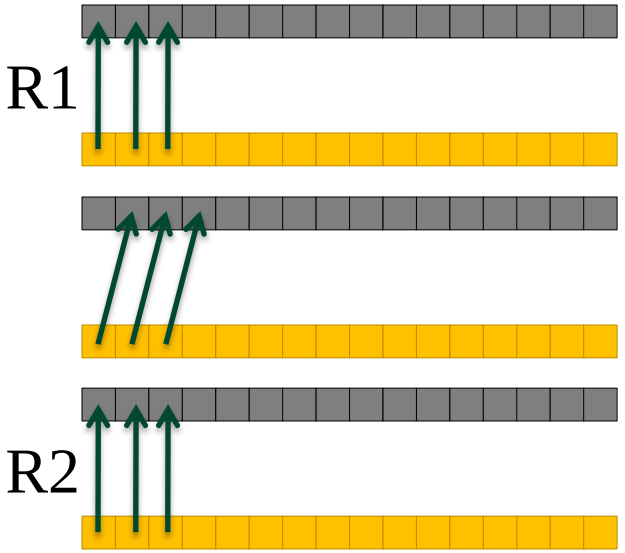
Работа с глобальной памятью

- threadfence() – дождаться, когда для всех активных блоков, текущие обращения в память завершаться
  - Не является средством синхронизации блоков
  - Блоки могут быть в разных состояниях
- threadfence_block() – дождаться пока все операции записи в память завершаться для вызвавшего блока

Работа с глобальной памятью

- volatile указывает, что переменная может быть изменена извне.

```
__global__ foo(float *p)
{
  float R1 = p[threadIdx.x];
  p[threadIdx.x + 1] = 0.0f;
  float R2 = p[threadIdx.x];
}
```



- В примере R1 == R2
- Слишком умный компилятор ☹

Легенда:
-нить
-float *p

Работа с глобальной памятью (2.x)

- Глобальный и локальный контроль за кэшем
  - cudaThreadSetCacheConfig()
  - cudaFuncSetCacheConfig
- cudaFuncCachePreferNone: значение по умолчанию
- cudaFuncCachePreferShared: бОльший объем разделяемой памяти предпочтительней
- cudaFuncCachePreferL1: бОльший объем L1 кэша предпочтительней

Работа с глобальной памятью

- Лучше иметь не менее 512 нитей на SM
- Число нитей на блок должно быть кратно 32
- Слишком маленькие блоки мешают получению высокой occupancy (есть ограничение на число блоков на SM)
- Слишком большие блоки менее гибки
- Vasily Volkov’s GTC2010 talk “Better Performance at Lower Occupancy”
- Выбирайте режим кэширования исходя из паттерна доступа
  - Рандомный паттерн – лучше 32-байтовые блоки
  - Обращения по близлежащий адресам – лучше 128-байтовые блоки

Работа с управляемой памятью (3.x и 64-битная ОС)

- Единое адресное пространство для CPU и GPU
- Выделение через cudaMallocManaged
- Может обращаться и CPU и GPU
- Во время активности GPU CPU не может обращаться
- Прозрачно обрабатывает внутренние ссылки
- cudaMemcpy определяет местоположение по последнему аргументу
- Память выделяется на GPU, неявная синхронизация между CPU и GPU
  - Pascal и выше – есть page faulting
  - Перед запуском ядра идет копирования всей измененной памяти
- Есть prefetch

Работа с управляемой памятью (3.x и 64-битная ОС)

```
__global__ void AplusB(int *ret, int a, int b)
{
   ret[threadIdx.x] = a + b + threadIdx.x;
}
int main()
{
   int *ret;

   cudaMallocManaged(&ret, 1000 * sizeof(int));

   AplusB<<< 1, 1000 >>>(ret, 10, 100);
   cudaDeviceSynchronize();
   for(int i=0; i<1000; i++)
      printf("%d: A+B = %d\n", i, ret[i]);
   cudaFree(ret);
   return 0;
}
```

Работа с управляемой памятью (3.x и 64-битная ОС)

```
__device__ __managed__ int ret[1000];


__global__ void AplusB(int a, int b)
{
   ret[threadIdx.x] = a + b + threadIdx.x;
}
int main()
{
   AplusB<<< 1, 1000 >>>(10, 100);


   cudaDeviceSynchronize();


   for(int i=0; i<1000; i++)
      printf("%d: A+B = %d\n", i, ret[i]);


   return 0;
}
```

Работа с управляемой памятью (3.x и 64-битная ОС)

```
__device__ __managed__ int x, y=2;
__global__ void kernel()
{
   x = 10;
}
int main1()
{
   kernel<<< 1, 1 >>>();
   y = 20; // ERROR: CPU access concurrent with GPU
   cudaDeviceSynchronize();

   kernel<<< 1, 1 >>>();
   cudaDeviceSynchronize();
   y = 20; // Success – GPU is idle so access is OK
   return 0;
}
```

Ресурсы нашего курса

- Steps3d.Narod.Ru
- Google Site CUDA.CS.MSU.SU
- Google Group CUDA.CS.MSU.SU
- Google Mail CS.MSU.SU
- Google SVN
- Tesla.Parallel.Ru
- Twirpx.Com
- Nvidia.Ru

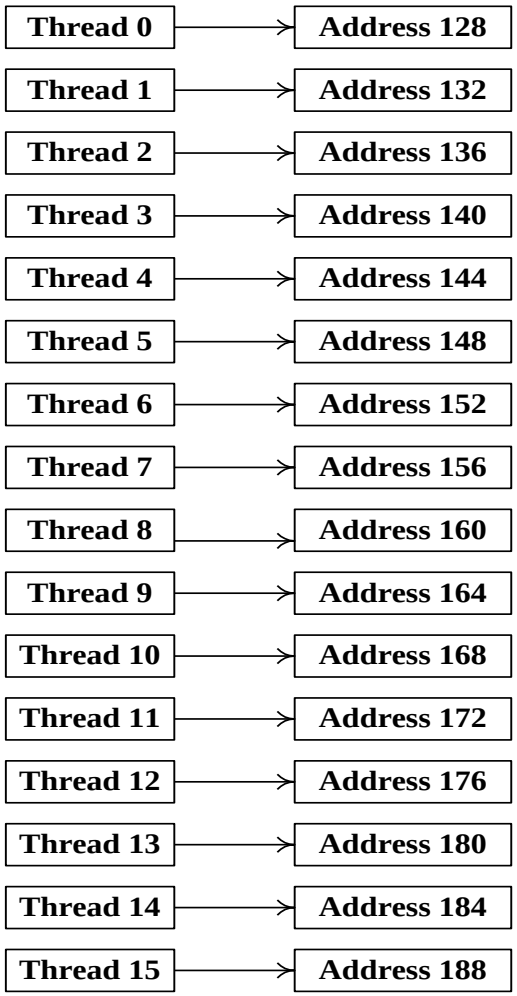
Дополнительные слайды

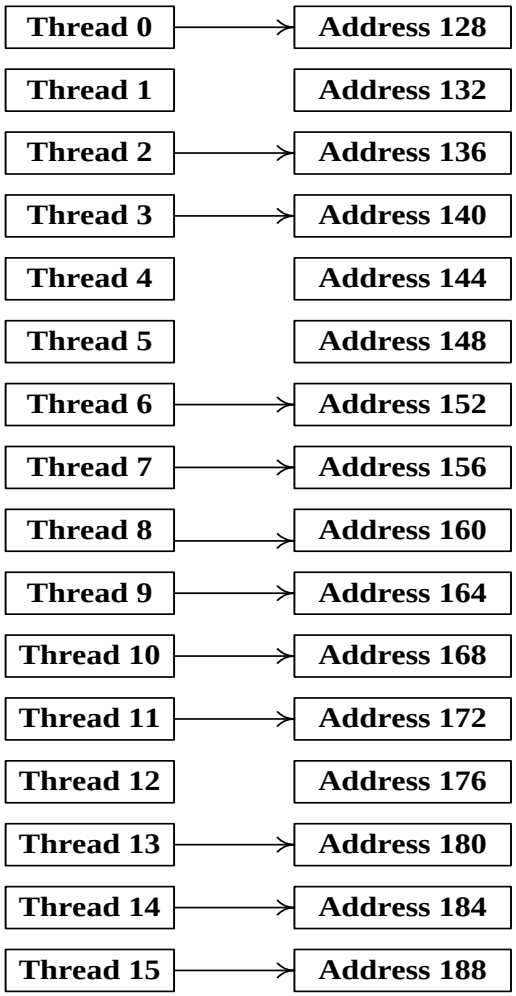
- Объединение (coalescing) для GPU с CC 1.0/1.1

Объединение (coalescing) для GPU с CC 1.0/1.1

- Нити обращаются к
  - 32-битовым словам, давая 64-байтовый блок
  - 64-битовым словам, давая 128-байтовый блок
- Все 16 слов лежат в пределах блока
- *k*-ая нить *half-warp*'а обращается к *k*-му слову блока

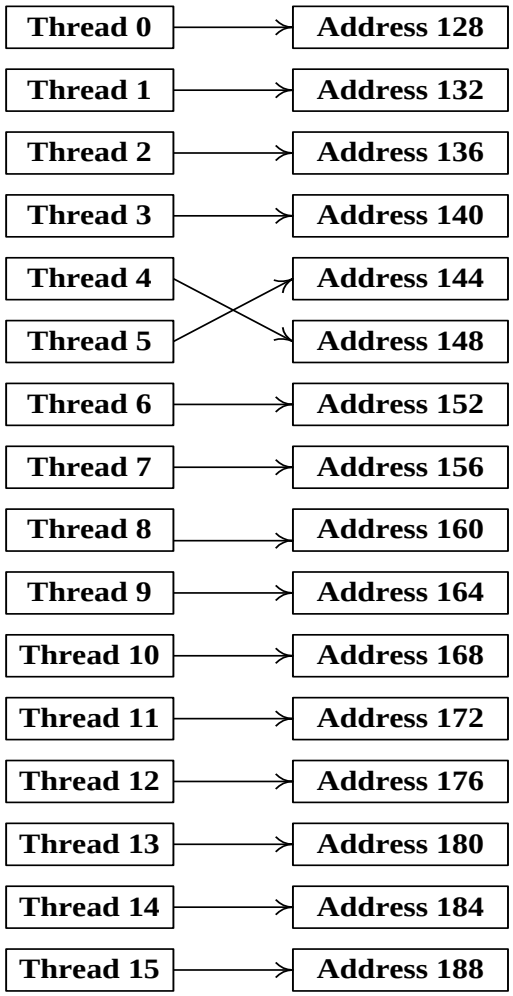
Объединение (coalescing) для GPU с CC 1.0/1.1

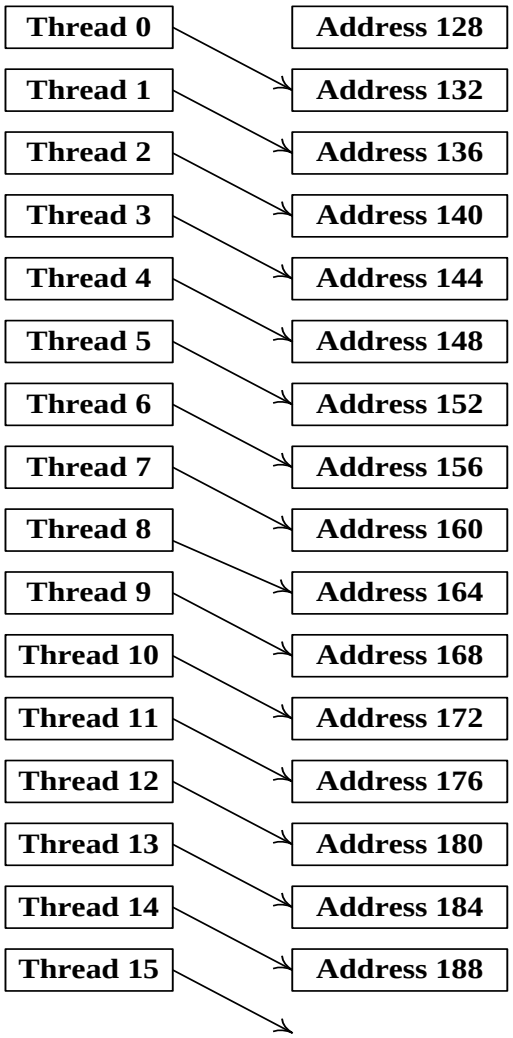




Coalescing

Объединение (coalescing) для GPU с CC 1.0/1.1





No Coalescing