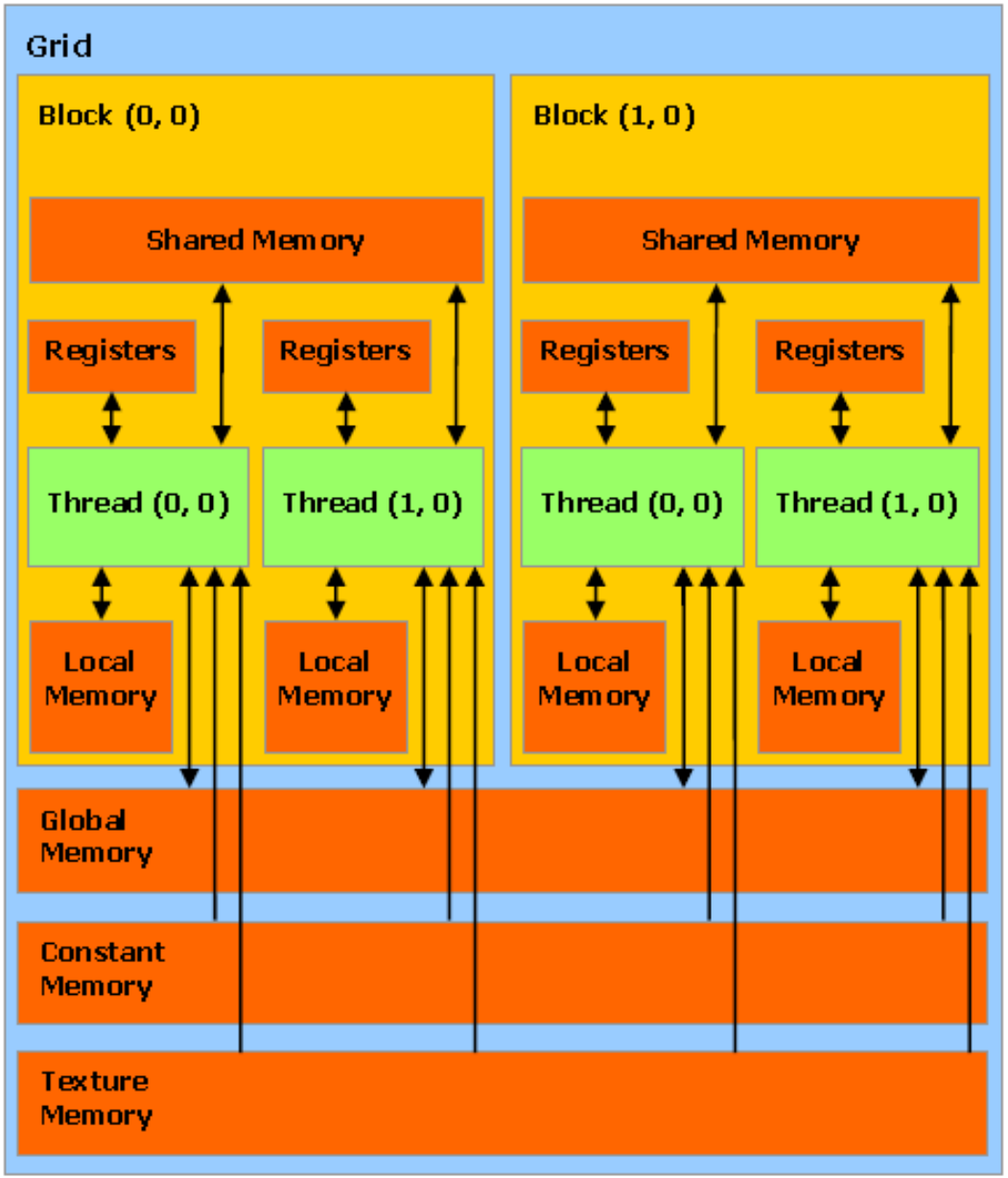
Центр микро- и наномасштабной динамики дисперсных систем

Модель памяти GPU/CUDA
Global memory

Марьин Д. Ф.

Уфа, 2011г.



На GPU/CUDA выделяют 6 видов памяти:

1. регистровая
2. разделяемая
3. локальная
4. глобальная
5. константная
6. текстурная

- *Глобальная, локальная, текстурная* и *память констант* — *локальная видеопамять GPU*.
  Отличие в различных алгоритмах кэширования и моделях доступа.
- *CPU* может обновлять и запрашивать только *внешнюю память*: *глобальную*, *константную* и *текстурную*.

Registers — СОЗУ в мультипроцессоре (*32*-разрядная).
Register file — все регистры мультипроцессора.

- Расположение: **multiprocessor**
- Кэшируемость: **no**
- Уровень выделения: **on chip**
- Доступ: **GPU — R/W, CPU — no**
- Скорость работы: **«максимальная»**
- Уровень доступа: **per-thread, SP**
- Время жизни: **thread**

Распределение регистров

На один мультипроцессор доступно $1024 \div 16384$ регистра.
Регистры распределяются на этапе компиляции.
Пример:
Регистров на мультипроцессор — 16384
Размер блока (число потоков) — 256
Число регистров на один поток:

$$16384/256 = 64$$

Примечание: для большей эффективности надо стараться занимать меньше 64 или даже 32 регистров, чтобы на одном мультипроцессоре могло исполняться несколько блоков!

Constant memory — физически не отделена от глобальной памяти.

Константная память выделяется при помощи спецификатора __constant__.

- Расположение: **DRAM**
- Кэшируемость: **yes, 8 KB at MP**
- Уровень выделения: **on chip L1 cache**
- Доступ: **GPU — R/O, CPU — R/W**
- Скорость работы: **высокая** (cache) / **низкая** (400–600 тактов)
- Уровень доступа: **per-grid, CPU**
- Время жизни: **выделяется CPU**

Кэш существует в единственном экземпляре для одного мультипроцессора, а значит, общий для всех задач внутри блока.

Функции runtime library работы с памятью

```
cudaMemcpyToSymbol (...);
cudaMemcpyFromSymbol(...);
cudaMemcpyToSymbolAsync (...);
cudaMemcpyFromSymbolAsync (...);
cudaGetSymbolAddress (...);
cudaGetSymbolSize (...);
```

```
template<class T>
cudaError_t cudaMemcpyToSymbol (const T& symbol,
  const void* src, size_t count,
  size_t offset = 0,
  enum cudaMemcpyKind kind = cudaMemcpyHostToDevice);
```

count — размер в байтах;
offset — сдвиг в байтах от начала symbol;
kind — cudaMemcpyHostToDevice, cudaMemcpyDeviceToDevice.

```
template<class T>
cudaError_t cudaMemcpyToSymbolAsync (...,
  cudaStream_t stream);
```

stream — дескриптор потока, позволяет организовать несколько потоков команд.

Пример работы с константной памятью

На стороне хоста (в .cu файле):

```
float hostData[256];                          // host memory
__constant__ float constData[256];  // constant memory
// host memory data -> device constant memory data
cudaMemcpyToSymbol(constData, hostData,
                   sizeof(hostData), 0,
                   cudaMemcpyHostToDevice);
```

На стороне устройства:

```
__global__ void kernel(float* pos)
{
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  pos[index] =  pos[index] * constData[index];
}
```

Локальная память — небольшой объём DRAM, используемый компилятором в случаях, когда локальные данные процедур занимают слишком большой размер, или когда компилятор не может вычислить для них некоторый постоянный шаг при обращении.

- Расположение: **DRAM**
- Кэшируемость: **no**
- Уровень выделения: **DRAM**
- Доступ: **GPU — R/W, CPU — no**
- Скорость работы: **низкая** (латентность — 400–600 тактов)
- Уровень доступа: **per-thread, SP**
- Время жизни: **thread**

Примечание: нет механизмов, позволяющих явно запретить компилятору использование локальной памяти для конкретных переменных.

Global memory — DRAM GPU. Обладает высокой пропускной способностью (более 100 ГБ/с) и возможностью произвольной адресации глобальной памяти.
Глобальная память выделяется при помощи спецификатора __device__ или особым образом.

- Расположение: **DRAM**
- Кэшируемость: **no**
- Уровень выделения: **DRAM**
- Доступ: **GPU — R/W, CPU — R/W**
- Скорость работы: **низкая** (латентность — 400–600 тактов)
- Уровень доступа: **per-grid, CPU**
- Время жизни: **выделяется CPU**

Назначение: передача данных между CPU и GPU.

Функции управления памятью

Выделение памяти на устройстве:

```
cudaError_t cudaMalloc(void** devPtr, size_t count);
```

count — размер в байтах.

Освобождение памяти на устройстве:

```
cudaError_t cudaFree(void* devPtr);
```

Для эффективного доступа к глобальной памяти важным требованием является выравнивание данных в памяти.

```
cudaError_t cudaMallocPitch(void** devPtr,
    size_t* pitch, size_t widthInBytes, size_t height);
```

при выделении памяти может увеличить объём памяти под каждую строку, чтобы гарантировать выравнивание всех строк.
pitch — шаг распределения памяти в байтах.
Если при помощи cudaMallocPitch выделялась память под матрицу из элементов типа T, то для получения адреса элемента, расположенного в строке row и столбце col, используется следующая формула:
T * item = (T *) ((char *) baseAddress + row * pitch) + col;

Пример использования cudaMallocPitch для выделения двухмерного массива размерами width x height

```
// host side code
float * devPtr;
size_t pitch;
int height = 10;
int width = 63;

cudaMallocPitch ( ( void ** ) & devPtr, & pitch,
                  width * sizeof(float), height );

myKernel <<< 10, 64 >>> ( devPtr, pitch, width, height);
```

widthInBytes = 252
pitch = 256

```
// device side code
__global__ void
myKernel ( float* devPtr, int pitch,
           int width, int height)
{
  for (int row = 0; row < height; ++row)
  {
    float* rowPtr= (float*)((char*)devPtr + row * pitch);
    for (int col = 0; col < width; ++col)
    {
      float element  = rowPtr[col];
      ...
      rowPtr[col] = row * width + col;
    }
  }
}
```

T * item = (T *) ((char *) baseAddress + row * pitch) + col;

```
cudaError_t cudaMallocHost(void** hostPtr, size_t size);
```

page–locked (pinned) память со стороны хоста, которая доступна напрямую устройству.
Повышает скорость передачи данных между CPU и GPU для функций cudaMemcpy*(). Виртуальная память обладает большей пропускной способностью, чем страничная память (malloc()).
Примечание: Является ограниченным ресурсом и чрезмерное её использование может отрицательно сказаться на быстродействии всей системы.
Лучше использовать как буффер для обмена данными между хостом и устройством.

```
cudaError_t cudaFreeHost(void* hostPtr);
```

Пересылка данных:

```
cudaError_t cudaMemset(void* devPtr, int value,
                       size_t count);
```

count — размер в байтах;

value — значение.

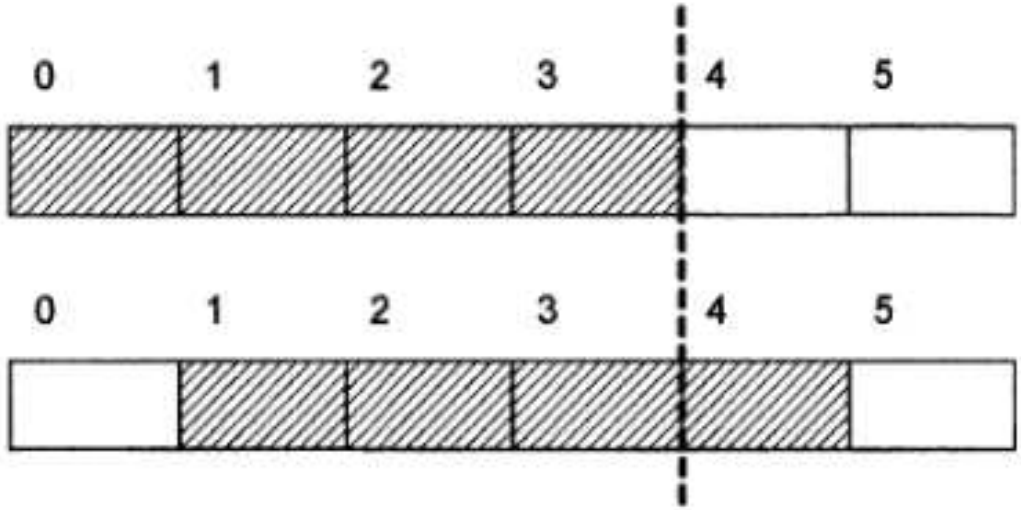
Копирование данных:

```
cudaError_t cudaMemcpy(void* dst, const void* src,
              size_t count, enum cudaMemcpyKind kind);
cudaError_t cudaMemcpyAsync (...
              cudaStream_t stream);
...
```

Пример пересылки данных:

```
int      n = 512;
int      numBytes = n * sizeof ( float );
float * a = NULL;    float * aDev = NULL;

a = (float*) malloc (n*sizeof(float));
cudaError_t error =
      cudaMalloc ( (void**)&aDev, numBytes );

dim3 threads = dim3(128);
dim3 blocks  = dim3(n / threads.x);

cudaMemset ( aDev, 0, n );

vectorsSumKernel <<< blocks, threads >>> (aDev);

cudaMemcpy (a, aDev, numBytes, cudaMemcpyDeviceToHost );

cudaFree (aDev);   free (a);
```

Обращение к глобальной памяти происходит через чтение/запись 32/64/128-битовых слов. Важно, что адрес, по которому происходит доступ, должен быть выровнен по размеру слова (кратен размеру слова в байтах).



Пример выровненного и невыровненного 4-х байтного слова.

Все функции выделяющие глобальную память выделяют её выровненной по 256 байтам.
Пусть есть массив из следующих структур в глобальной памяти:

```
struct vec3
{
  float a;
  float b;
  float c;
}
```

Каждый элемент массива — 12 байт. Адрес первого элемента выровнен по 16 байтам, но адрес второго элемента — нет, и его чтение потребует двух обращений.

Выравнивание:

```
struct __align__(16) vec3
{
  float a;
  float b;
  float c;
}
```

Теперь все элементы массива будут хранится по адресам кратным 16 байтам.

Объединение запросов в глобальную память (coalescing)

GPU имеет возможность объединять несколько запросов к глобальной памяти в один (coalescing).
Все обращения MP к памяти происходят независимо для каждой половины warp'а.
Максимальное объединение — все запросы одного полу-warp'а удается объединить в один большой запрос на чтение из глобальной памяти.

Условия возможности объединения

- все нити обращаются к 32(CP от 1.2)/32/64-битовым словам, давая в результате один 32(CP от 1.2)/64/128-байтный блок;
- получивщийся блок выровнен по своему размеру (адрес кратен 32(CP от 1.2)/64/128);
- все 16 слов, к которым обращаются нити лежат в пределах одного блока;
- нити обращаются к словам последовательно:
  k–ая нить обращается к k–му слову;
  допускается, что отдельные нити пропустят обращение к соответствующим словам.

Если нити полу-warp'а не удовлетворяют любому условию, то каждое обращение к памяти происходит как отдельная транзакция!

Гораздо эффективнее, с точки зрения объединения запросов к памяти, использование не массивов структур, а структуры массивов.

```
struct __align__(16) A
{
  float a;
  float b;
  int c;
}
A array[256];
...
float ga = array[threadIdx.x].a;
A gs = array[threadIdx.x];
```

Не приведет к объединению запросов!
Понадобится 16 транзакций на полу-warp.

```
float a[256];
float b[256];
int c[256];
...
float ga = a[threadIdx.x];
float gb = b[threadIdx.x];
float gc = c[threadIdx.x];
```

Приведет к объединению запросов всех запросов нитей полу-warp'а!

Понадобится 3 транзакции на полу-warp.

Основные способы:

- выравнивание
- объединение запросов в глобальную память (coalescing)
- использование не массивов структуры, а структур массивов