Введение в параллельные вычисления

Лекция 11. NVIDIA CUDA (часть 1)

КС-40, КС-44
РХТУ

Преподаватель
Митричев Иван Игоревич, к.т.н.,
ассистент кафедры ИКТ

2017

Что такое CUDA?

CUDA – архитектура параллельных вычислений от NVIDIA, позволяющая существенно увеличить вычислительную производительность благодаря использованию GPU (графических процессоров).

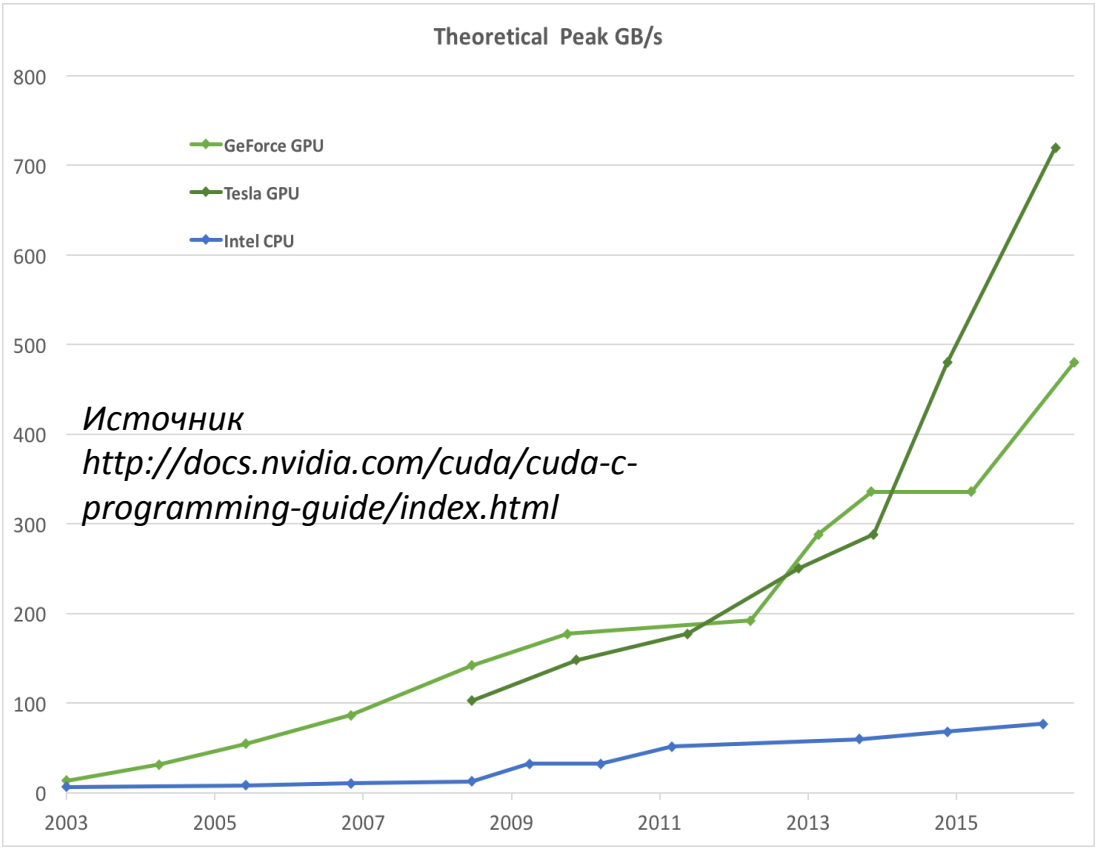
CUDA SDK позволяет программистам реализовывать на специальном упрощённом диалекте языка программирования C алгоритмы, выполнимые на графических процессорах NVIDIA, и включать специальные функции в текст программы на Си.

Архитектура CUDA даёт разработчику возможность по своему усмотрению организовывать доступ к набору инструкций графического ускорителя и управлять его памятью.

Хорошее руководство на английском (официальная документация): http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

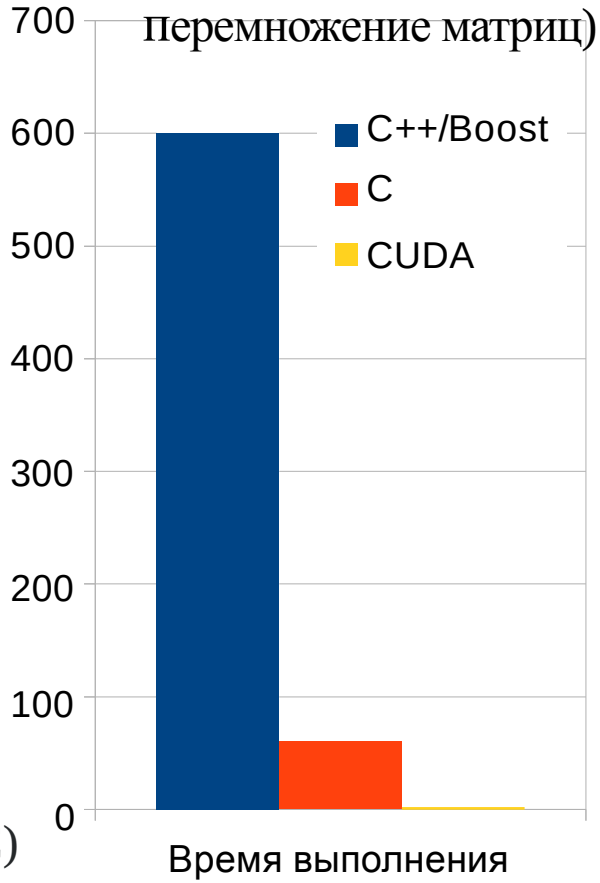
Преимущества и недостатки CUDA

+ Производительность в гигафлопсах одного устройства



Источник
http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

+Высокая скорость работы на некоторых задачах (параллельной и однотипной обработки данных – пример: перемножение матриц)

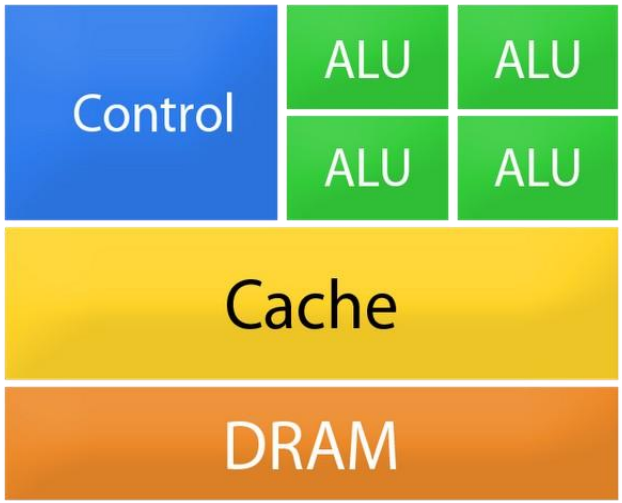


+ Использование языка общего назначения (язык C) вместо программирования под шейдеры

- Требуется специализированное оборудование (NVIDIA-только. OpenCL как альтернатива. Однако NVIDIA создало большое API, библиотеки CUFFT, CUBLAS).

Отличие графических процессоров от обычных

Много мелких ядер



CPU GPU

Источник http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

Архитектура ускорителя Kepler

Блок-схема GK10:

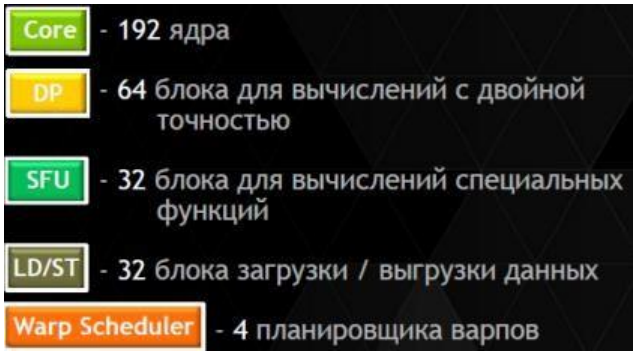
http://www.ixbt.com/video3/gk104-part1.shtml

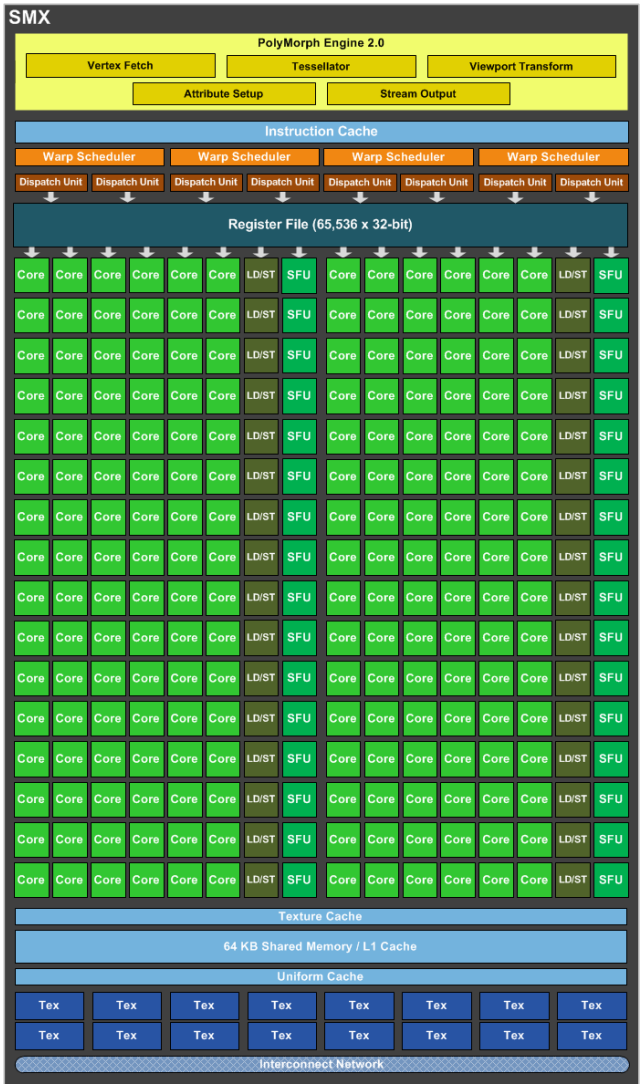
Структура SMX архитектуры Kepler

GPU GK107 (Kepler) имеет четыре блока GPC, каждый из этих блоков содержит по два потоковых мультипроцессора, отличающихся от предыдущих чипов Nvidia.

Используются потоковые мультипроцессоры SMX (Streaming Multiprocessor – SMX), в отличие от SM в предыдущих чипах. В последующих чипах Maxwell – новый вид потокового мультипроцессора – SMM.

Tesla < Fermi < Kepler < Maxwell < Pascal < Volta





Ключевые понятия

Ядро (kernel) CUDA-программы – обычная С-функция, использующая особый способ вызова языка CUDA C.

- ядро, как функция языка С, может принимать параметры;
- из ядра нельзя получить какие либо данные простой передачей указателя на память выделенную на CPU;
- так же, в ядро нельзя передавать массивы, и указатели на них, если они находятся в памяти CPU;
- ядро может вызывать любые другие функции, кроме другого ядра, невозможна рекурсия ядра, но возможна рекурсия функции;
- ядро создает локальные копии переменных объявляемых внутри него для каждой нити.

host – центральный процессор;

device – графический ускоритель;

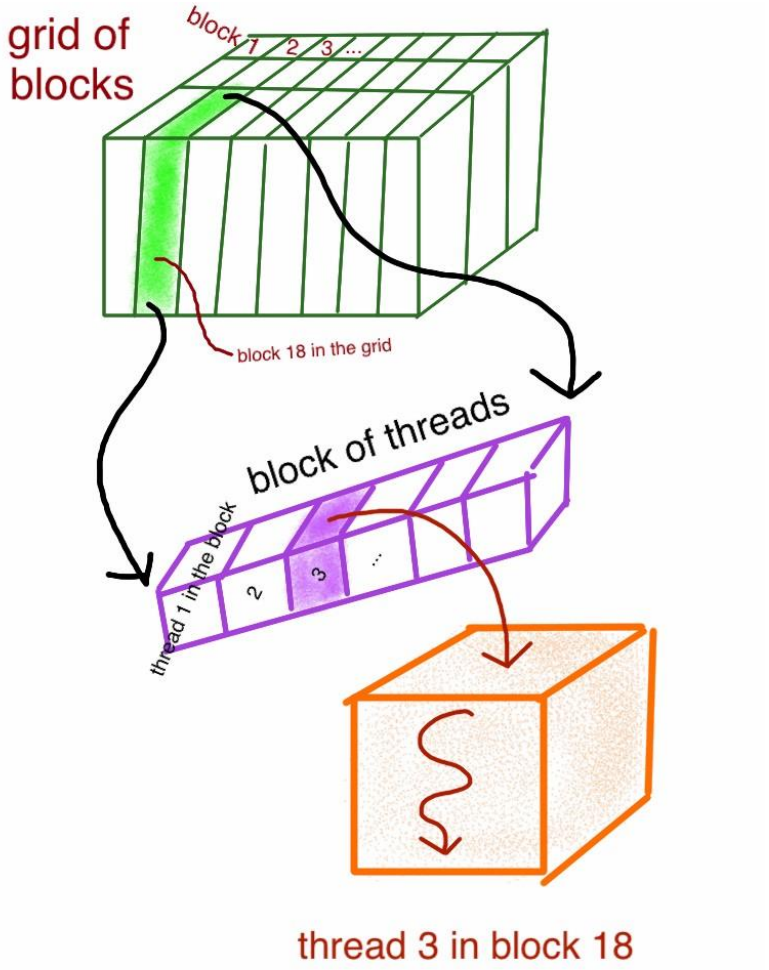
block (блок) – параллельно выполняемый экземпляр ядра;

grid (сетка) – группа параллельных блоков (одномерная, двухмерная, трехмерная);

thread (нить) – при программировании под CPU называется потоком исполнения;

warp – объединение 32 нитей в пучок.

Сетка, блоки, потоки



- Операцию над каждым элементом массива, выполняет отдельная нить (thread)
- В коде указываем, на какое число блоков разделяется массив, и какое число нитей в каждом блоке
- Для вычисления индекса массива используется специальная формула

Размерность сетки и блока

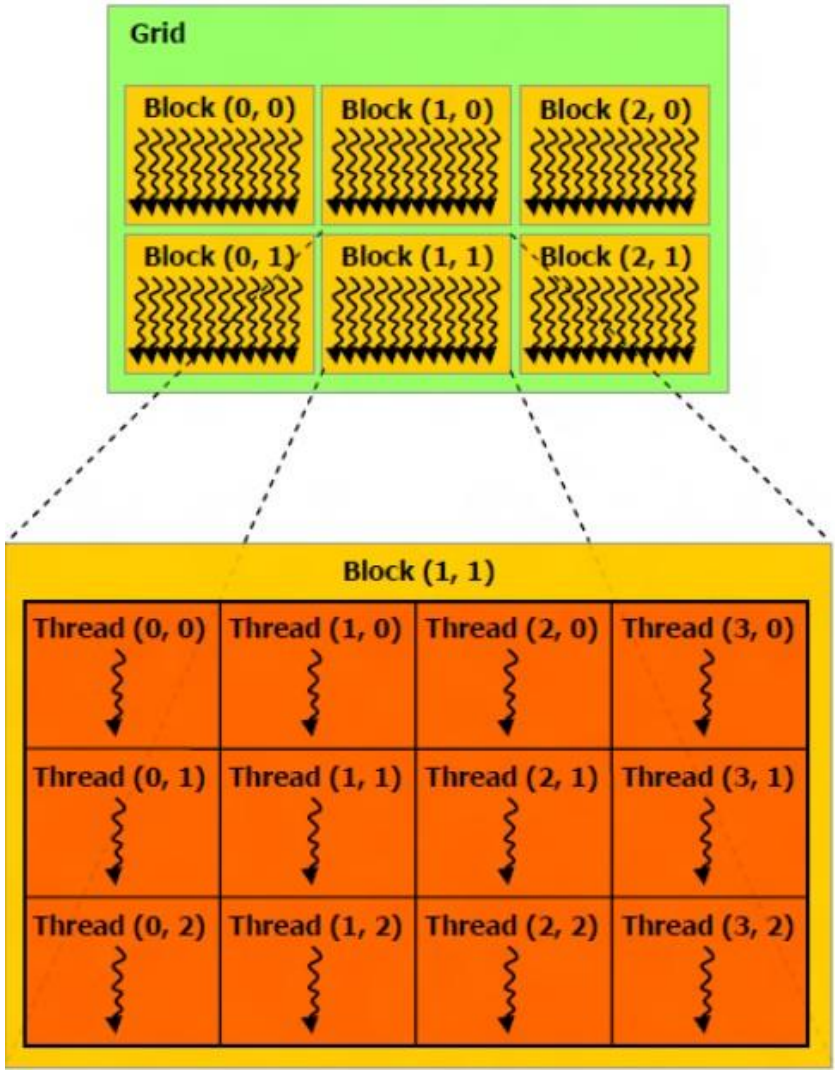
```
// ...

dim3 block(BLOCK_SIZE, BLOCK_SIZE);
dim3 grid(N / (BLOCK_SIZE * BLOCK_SIZE));

my_kernel<<<grid, block>>>(...);

// ...
```

- Сетка может быть одномерной или двухмерной
- Блок может быть одно-, двух- и трёхмерным

Блоки и нити

При вызове __global__ функции в конструкции
<<<…>>> в общем случае указываются два трехмерных вектора типа dim3, обозначающие **размерность решетки блоков (grid)** и **размерности блока (block)**.

Пример для решетки блоков размерностью (2, 3, 1) и размерности блока (3, 4, 1).



Источник http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

Специальные переменные, доступные на GPU

```
// размер сетки
dim3 gridDim;

// размер блока
dim3 blockDim;

// индекс текущего блока в сетке
uint3 blockIdx;

// индекс текущей нити в блоке
unit3 threadIdx;

// размер warpa
int warpSize;
```

Работа с памятью на GPU

```
#include <stdio.h>
__global__void add(int a, int b, int *c) {
        *c = a + b;
}
int main() {
        int c; int *dev_c;
        cudaMalloc( (void**)&dev_c, sizeof(int) );
        add<<<1,1>>>(2, 7, dev_c);
        cudaMemcpy(&c, dev_c, sizeof(int), cudaMemcpyDeviceToHost);
        printf("2 + 7 = %d\n", c);
        cudaFree(dev_c);
        return 0;
}
```

l10_01.cu

В программе:

- Передаются параметры ядру.
- Выделяется память, чтобы устройство через нее вернуло данные CPU.

cudaMalloc() выделяет память на GPU. Первый аргумент – указатель на указатель, в котором будет возвращен адрес выделенной области памяти, второй – размер этой памяти.

Получение информации об устройствах

```
int main() {
        cudaDeviceProp prop;
        int count;
        cudaGetDeviceCount(&count);
        for(int i = 0; i < count; i++) {
                cudaGetDeviceProperties(&prop, i);
                // Сделать что-то со свойствами устройства
}
```

Структура cudaDeviceProp содержит поля с информацией:

- имя GPU,
- объём памяти,
- максимальное количество нитей в блоках и т.д.

Измерение производительности

Событие CUDA – временная метка GPU, запомненная пользователем в определенный момент времени. События позволяют измерять производительность.

```
cudaEventCreate( &start );
cudaEventCreate( &stop );

    cudaEventRecord( start, 0 );
    // Выполнение ядра, работа с памятью GPU
    cudaEventRecord( stop, 0 );
    cudaEventSynchronize( stop );

float elapsedTime;
cudaEventElapsedTime( &elapsedTime, start, stop );
printf( "Time to generate: %3.1f ms\n", elapsedTime );

    cudaEventDestroy( start );
    cudaEventDestroy( stop );
```

Причина использования cudaEventSynchronize() – из-за асинхронности некоторых обращений к исполняющей среде CUDA, например, при запуске ядра GPU начинает исполнять код ядра, но CPU продолжает выполнять дальше свой код, не дожидаясь окончания работы GPU. После возвращения управления функцией cudaEventSynchronize() есть уверенность, что вся работа до окончания события завершена.

Сложение векторов

```
__global__void add(int *a, int *b, int *c) {
        int i = threadIdx.x;
                if(i < N)
                        c[i] = a[i] + b[i];
}
int main() {
        int a[N], b[N], c[N];
        int *dev_a, *dev_b, *dev_c;
// Инициализация a, b, выделение памяти GPU, копирование a и b в память GPU
        // складываем вектора
        add<<<1,N>>>( dev_a, dev_b, dev_c );
        // Копирование dev_c в c, вывод результатов
}
```

Число N в записи <<<1,N>>> определяет число нитей в пределах одного блока. В функции add каждая нить узнает свой индекс (координату x трехмерной решетки (x, y, z)), на основе координаты вычисляет соответствующий элемент вектора.

Использование памяти устройства

```
#define BLOCKS_NUM 10
#define BLOCK_SIZE 256
#include <iostream>
using namespace std;
__global__ void my_kernel(float *in, float *out){
int n = threadIdx.x + blockIdx.x * BLOCK_SIZE;
out[n] = in[n] * in[n]; // квадрат
}
int main() {
float data[BLOCKS_NUM * BLOCK_SIZE]; //     CPU
память
for (int i=0; i<BLOCKS_NUM*BLOCK_SIZE;
i+=BLOCK_SIZE)
   data[i]=i;
cudaSetDevice(0); // выбор устройства
float *in, *out; // GPU память
// GPU выделение памяти
uint memory_size = sizeof(float) * BLOCKS_NUM *
BLOCK_SIZE;
cudaMalloc((void **)&in, memory_size);
cudaMalloc((void **)&out, memory_size);
//       копируем память на устройство
cudaMemcpy(in, data, memory_size,
cudaMemcpyHostToDevice);
dim3 block(BLOCK_SIZE);
dim3 grid(BLOCKS_NUM);
//    запускаем ядро
my_kernel<<<grid, block>>>(in, out);
cudaThreadSynchronize(); //      ждем окончания
расчета
//         копируем результаты обратно на хос
cudaMemcpy(data, out, memory_size,
cudaMemcpyDeviceToHost);
for (int i=0; i<BLOCKS_NUM*BLOCK_SIZE;
i+=BLOCK_SIZE)
                std::cout<<data[i]<<"
";std::cout<<std::endl; //! не забываем отчистить
память на GPU
cudaFree(in);
cudaFree(out);
return 0;
}
```

l10_02.cu

Спецификаторы функций

Спецификатор	Выполняется на	Может вызываться из	Возвращаемый тип
__device___	Device (GPU)	Device (GPU)	*любой*
__global___	Device (GPU)	Host (CPU)	void
__host___	Host (CPU)	Host (CPU)	*любой*

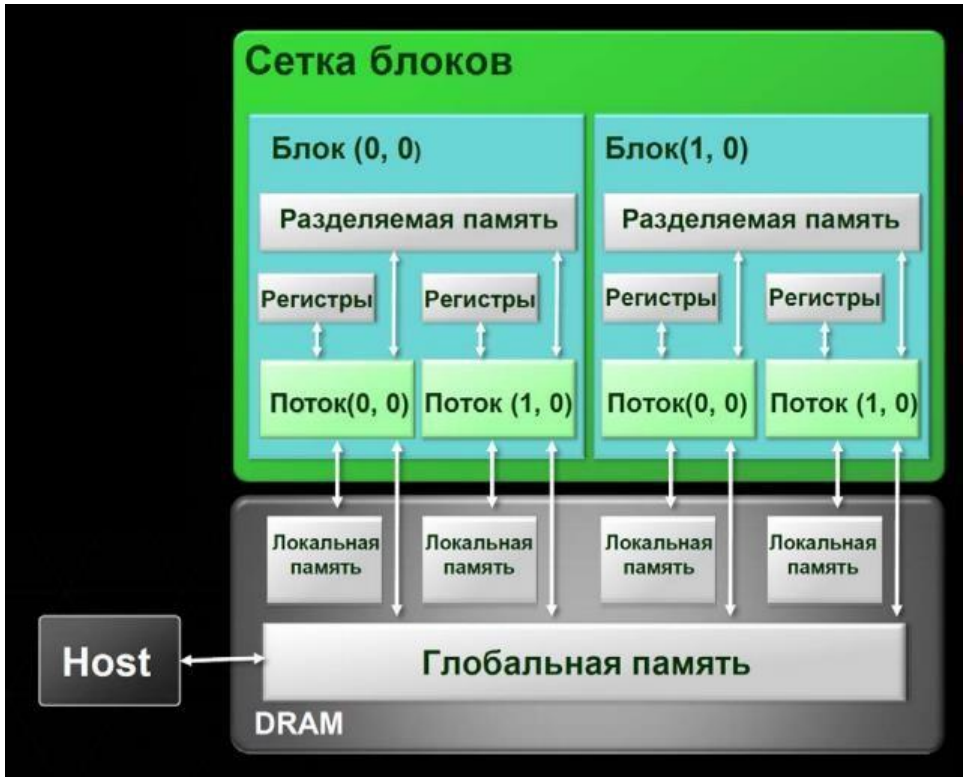
__global__ void AplusB(int *ret, int a, int b) ……

О размерности GPU

Каждая видеокарта имеет свои ограничения для максимальных размерностей решеток блока и размерности блока: 512 x 512 x 64 для размерности блока (или 1024 x 1024 x 64 в более поздних версиях Compute capability) и 65535 x 65535 x 1
для решетки блока (2^{31}-1 x 65535 x 65535 в более поздних версиях Compute capability).

Существует предел количества потоков на блок (512 / 1024), поскольку все потоки блока располагаются на одном процессорном ядре GPU и должны разделять ограниченные ресурсы памяти этого ядра. Стандартно используют 256 потоков в блоке.

Размерность блока можно узнать с помощью встроенной переменной blockDim, размерность сетки блоков – с помощью gridDim.



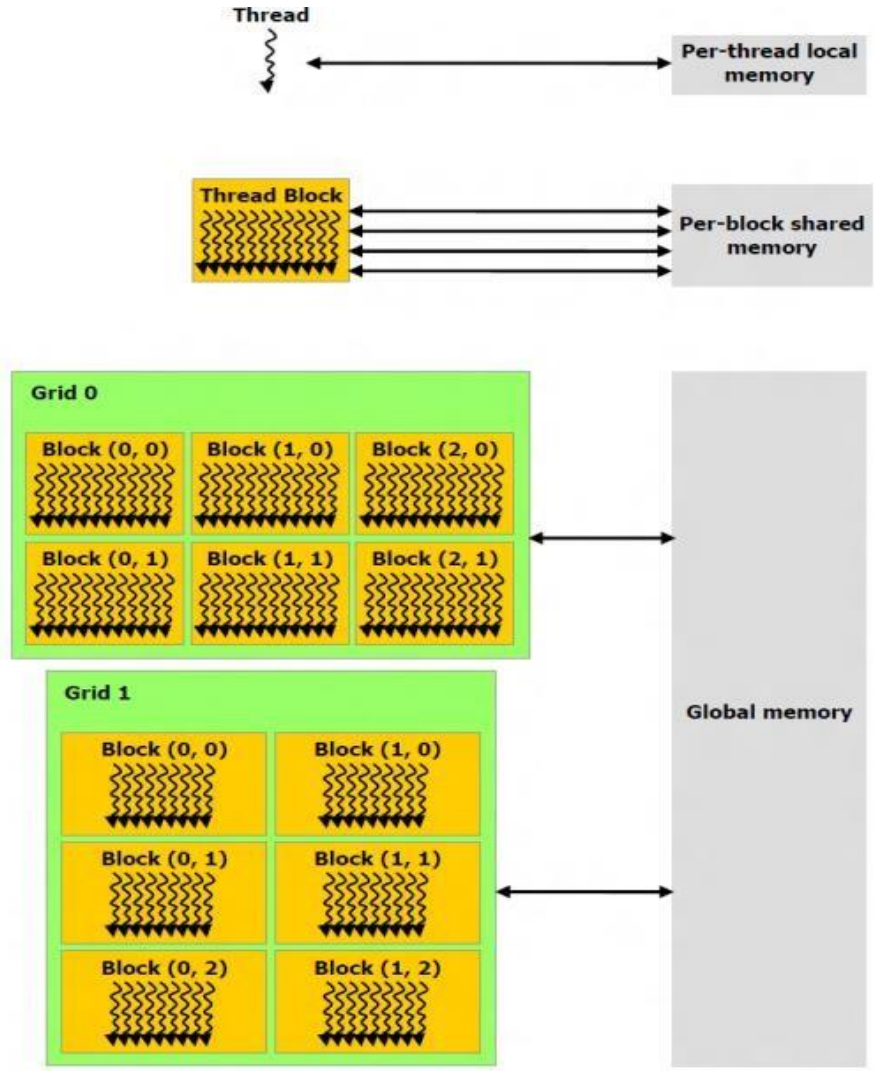
Источник http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

Иерархия памяти

Каждая нить имеет собственную локальную память.

Каждый блок имеет разделяемую память, видимую для всех потоков этого блока. Все нити имеют доступ к глобальной памяти.

К глобальной памяти относится все, что в рассмотренных примерах выделялось с помощью функций cudaMalloc().

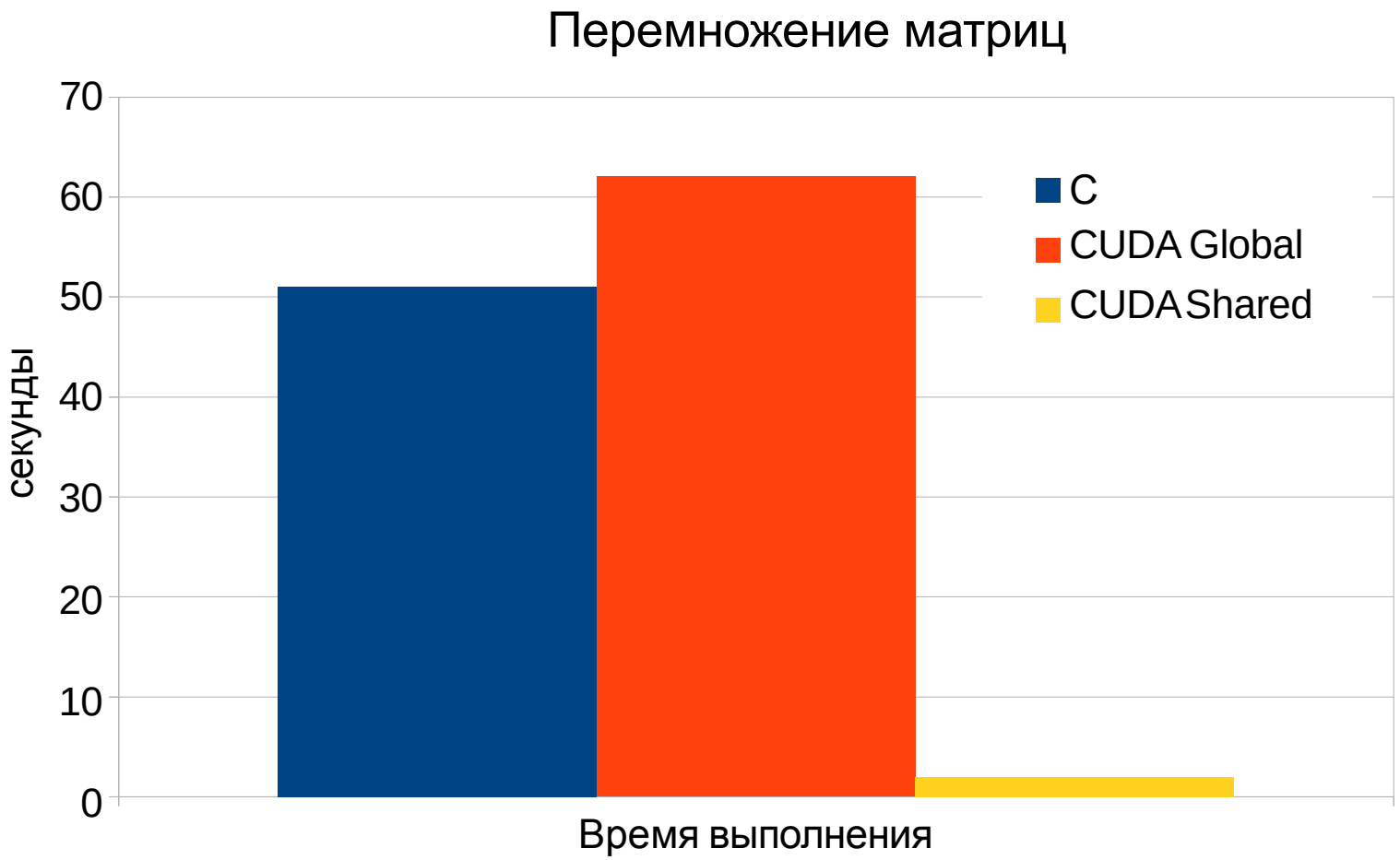


Источник http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

CUDA память

Тип памяти	**Тип доступа**	**Уровень выделения**	**Расположение**	**Скорость работы**
Register (Регистровая)	RW (запись\чтение)	Per-thread (На нить)	On-chip (На чипе)	Самая высокая
Local (Локальная)	RW (запись\чтение)	Per-thread (На нить)	DRAM (В памяти GPU)	Низкая
Global (Глобальная)	RW (запись\чтение)	Per-grid (На сеть)	DRAM (В памяти GPU)	Самая низкая
Shared (Разделяемая)	RW (запись\чтение)	Per-block (На блок)	On-chip (На чипе)	**Высокая (2 по скорости)**
Constant (Константная)	RO (только чтение)	Per-grid (На сеть)	L1 cache (в кеше первого ур.)	Высокая (3 по скорости)
Texture (Текстурная)	RO (только чтение)	Per-grid (На сеть)	L1 cache (в кеше первого ур.)	Высокая (4 по скорости)

Скорость вычислений



Разделяемая память

Если при объявлении переменной указать ключевое слово __shared__, то она будет размещена в разделяемой памяти.

Для каждой такой переменной создается копия в каждом блоке. Все нити, работающие в одном блоке, разделяют эту переменную, но не видят ее копии, размещенные в других блоках.

Буферы разделяемой памяти находятся на самом GPU, а не на DRAM, что уменьшает время доступа к ним, фактически они играют роль программно управляемого кэша.

```
__global__void kernel() {
        …
__shared__float cache[threadBerBlock];
// Действия с кэшем
__syncthreads(); // Синхронизация потоков
        …
}
```

Функция __syncthreads() выполняет барьерную синхронизацию потоков. Предостережение: не стоит делать условный вызов __syncthreads().

Что еще доступно на устройстве?

и необходимо для решения практических задач в РХТУ с использованием CUDA

Математические функции

__fsqrt_[rn, rz, ru, rd] (x)

__expf (x)

__exp10f (x)

__logf (x)

__log2f (x)

__log10f (x)

__sinf (x)

__cosf (x)

__tanf (x)

__powf (x, y)

и другие...

rn – округление к ближайшему
rz – округление к нулю
ru – округление вверх
rd – округление вниз

Типы данных

- 1/2/3/4-мерные векторы базовых типов:
  char, uchar, short, ushort, int, uint, long, ulong, float
- 1 или 2-мерные векторы больших типов:
  longlong, double
- Специальный тип размерности dim3

Например:

```
uint4 a(1, 2); // a.x = 1, a.y = 2, a.z = 0, a.w = 0
dim3 gr(32, 16); // gr.x = 32, gr.y = 16, gr.z = 1
```

Унифицированная память (unified memory)

Это удобный механизм (API) одновременной работы с GPU/CPU. Это **не** реальный отдельный вид памяти на устройстве.

+ Не надо выделять отдельно память на устройстве и хосте и копировать данные туда-сюда.

- Работает только на CUDA Kepler (Compute Capability 3.0) и выше

Унифицированная память (unified memory) - пример

БЕЗ

```
__global__ void AplusB( int *ret, int a, int b)
{ ret[threadIdx.x] = a + b + threadIdx.x; }
int main() {
int *ret;
cudaMalloc(&ret, 1000 * sizeof(int));
AplusB<<< 1, 1000 >>>(ret, 10, 100);
int *host_ret = (int *)malloc(1000
* sizeof(int));
cudaMemcpy(host_ret, ret, 1000
* sizeof(int), cudaMemcpyDefault);
for(int i=0; i<1000; i++)
    printf("%d: A+B = %d\n", i, host_ret[i]);
free(host_ret);
cudaFree(ret);
return 0; }
```

http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-unified-memory-programming-hd

С

```
__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; }
```

Вместо зеленой строчки можно в начало программы добавит другую:

```
__device__ __managed__  int  ret[1000];
```

с аналогичным эффектом

Компиляция программ CUDA

- Традиционно используемое расширение файлов **.cu**
- Комилятор **nvcc** (gcc, умеющий компилировать для CUDA)