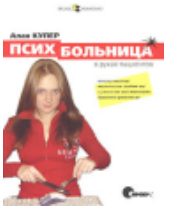
В ближайшее время статья по расширению ARB_shader_image_load_store

Quote of the Day
All change is not growth, as all movement is not forward.
Ellen Glasgow



Основы CUDA

Введение, GPGPU

CUDA (*Compute Unified Device Architecture*) -- это технология от компании NVidia, предназначенная для разработки приложений для массивно-параллельных вычислительных устройств (в первую очередь для GPU начиная с серии G80).

Основными плюсами CUDA являются ее бесплатность (SDK для всех основных платформ свободно скачивается с developer.nvidia.com), простота (программирование ведется на "расширенном C") и гибкость.

Фактически CUDA является дальнейшим развитием GPGPU (*General Purpose computation on GPU*). Дело в том, что уже с самого начала GPU активно использовали параллельность (как вершины, так и отдельные фрагменты могут обрабатываться параллельно и независимо друг от друга, т.е. очень хорошо ложатся на параллельную архитектуру).

По мере развития GPU росла как степень распараллеливания, так и гибкость самих GPU. Самые первые GPU для PC (Voodoo) фактически представляли собой просто растеризатор с возможностью наложения текстуры и буфером глубины. Довольно быстро появились GPU с T&L, т.е. полной обработкой вершин на самом GPU - на вход поступают трехмерные данные и на выходе получаем готовое изображение (например, Riva TNT). Однако гибкость в них была небольшой - ведь все вычисления велись в рамках фиксированного конвейера (FFP).

Следующим шагом (GeForce 2) стало появления вершинных шейдеров (расширение ARB_vertex_program) - обработку вершин стало возможным задавать в виде программы, написанной на специальном ассемблере. При этом вершины обрабатывались параллельно и независимо друг от друга. Ниже приводится пример простой программы на таком ассемблере.

```
!!ARBvp1.0
ATTRIB pos     = vertex.position;
PARAM  mat [4] = { state.matrix.mvp };

# transform by concatenation of modelview and projection matrices

DP4 result.position.x, mat [0], pos;
DP4 result.position.y, mat [1], pos;
DP4 result.position.z, mat [2], pos;
DP4 result.position.w, mat [3], pos;

# copy primary color

MOV result.color, vertex.color;
END
```

Вполне логичным следующим шагом стало появление фрагментных программ (расширение ARB_fragment_program), позволяющим задавать расчет каждого пиксела также при помощи программы, на ассемблере. Важным моментом является то, что все эти вычисления (как для вершин, так и для фрагментов) ведутся с использованием 32-битовых *floating-point* чисел.

В архитектуре GPU появились отдельные вершинные и фрагментные процессоры, выполняющие соответствующие программы. Данные процессоры вначале были крайне просты - можно было выполнять лишь простейшие операции, практически полностью отсутствовало ветвление и все процессоры одного типа одновременно выполняли одну и ту же команду (классическая SIMD-архитектура).

За счет большого числа вершинных и фрагментных процессоров, выполняющих такие программы, оказалось, что по быстродействию (измеряемому в количестве *floating-point* операций в секунду) GPU в разы обгоняют CPU.

Заключительным шагом, превратившим GPU в мощные параллельные вычислители, стало поддержка *floating-point* текстур, т.е. стало возможным хранить значения в текстурах как 32-битовые *floating-point* числа.

В результате GPU фактически стало устройством, реализующим потоковую вычислительную модель (stream computing model) - есть потоки входных и выходных данных, состоящие из одинаковых элементов, которые могут быть обработаны независимо друг от друга. Обработка элементов осуществляется ядром (*kernel*) (см рис 1.).

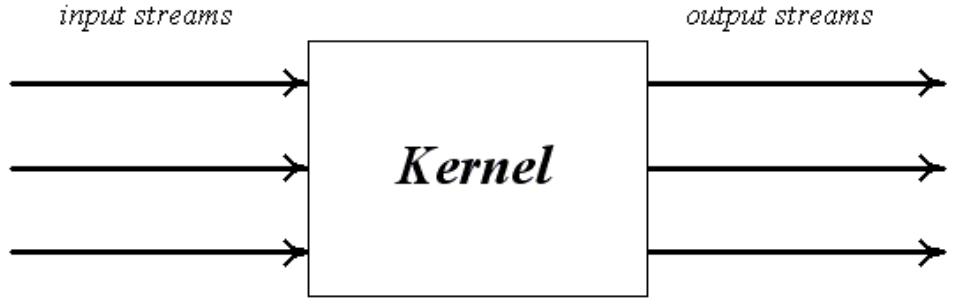


Рис 1. Потоковый вычисления.

Фактически GPU оказалось мощным SIMD (*Single Instruction Multiple Data*) процессором. В результате появилось GPGPU - использование огромной вычислительной мощности GPU для решения неграфических задач. Несмотря на значительные результаты GPGPU обладало рядом недостатков:

- вся работа шла через графический API, код для GPU писался на GLSL/HLSL/Cg, остальной код - на традиционном языке программирования
- наличие ограничений на размеры и размерность текстур
- полностью отсутствовала возможность взаимодействия между параллельно обрабатываемыми пикселами
- отсутствовала поддержка так называемого *scatter*'а (хотя были найдены обходные пути)

Кроме того на GPU GeForce 6xxx/7xxx отсутствовала нативная поддержка целых чисел и побитовых операций над ними.

Появление CUDA (а также GPU G80) полностью сняло все эти ограничения, предложив для GPGPU простую и удобную модель. В этой модели GPU рассматривается как специализированное вычислительное устройство (называемое *device*), которое:

- является сопроцессором к CPU (называемому *host*)
- обладает собственной памятью (DRAM)
- обладает возможностью параллельного выполнения огромного количества отдельных нитей (*threads*)

При этом между нитями на CPU и нитями на GPU есть принципиальные различия -

- нити на GPU обладают крайне "небольшой стоимостью" - их создание и управление требует минимальных ресурсов (в отличии от CPU)
- для эффективной утилизации возможностей GPU нужно использовать многие тысячи отдельных нитей (для CPU обычно нужно не более 10-20 нитей)

Сами программы пишутся на "расширенном" C, при этом их параллельная часть (ядра) выполняется на GPU, а обычная часть - на CPU. CUDA автоматически осуществляет разделением частей и управлением их запуском.

CUDA использует большое число отдельных нитей для вычислений, часто каждому вычисляемому элементами соответствует одна нить. Все нити группируются в иерархию - *grid/block/thread* (см. рис. 2).

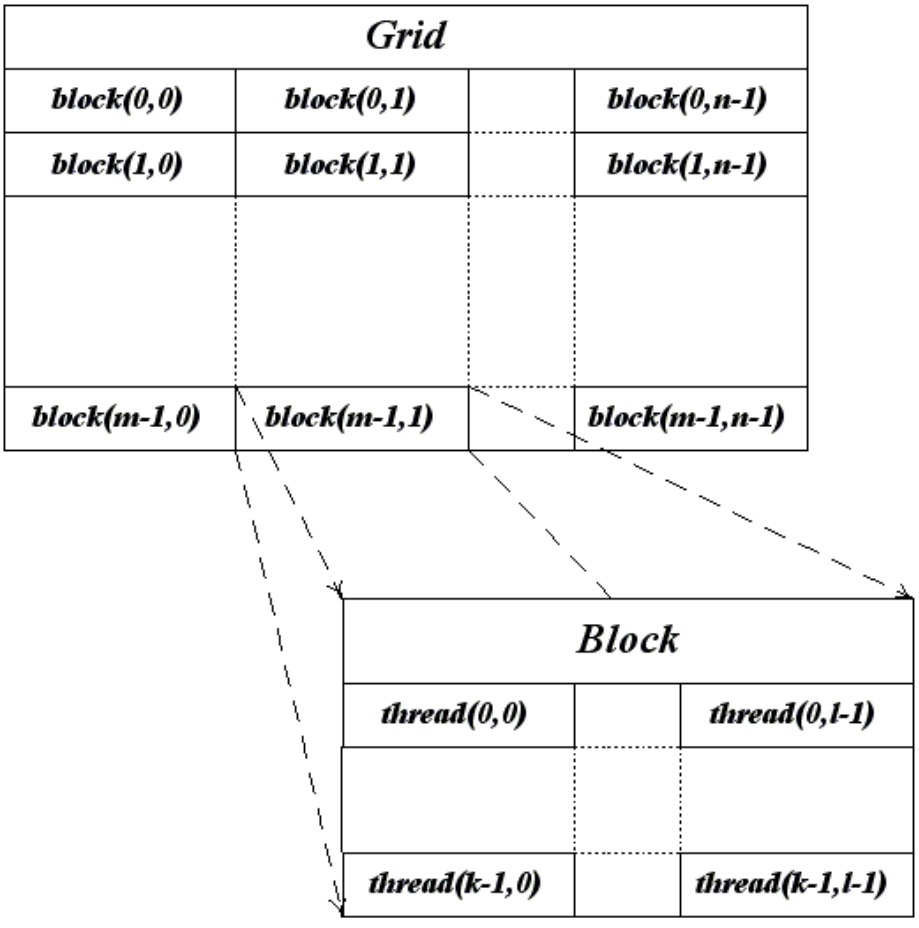


Рис 2. Иерархия нитей в CUDA.

Верхний уровень - *grid* - соответствует ядру и объединяет все нити выполняющие данное ядро. *grid* представляет собой одномерный или двухмерный массив блоков (*block*). Каждый блок (*block*) представляет из себя одно/двух/трехмерный массив нитей (*threads*).

При этом каждый блок представляет собой полностью независимый набор взаимодействующих между собой нитей, нити из разных блоков не могут между собой взаимодействовать.

Фактически блок соответствует независимо решаемой подзадаче, так например если нужно найти произведение двух матриц, то матрицу-результат можно разбить на отдельные подматрицы одинакового размера. Нахождение каждой такой подматрицы может происходить абсолютно независимо от нахождения остальных подматриц. Нахождение такой подматрицы - задача отдельного блока, внутри блока каждому элементу подматрицы соответствует отдельная нить.

При этом нити внутри блока могут взаимодействовать между собой (т.е. совместно решать подзадачу) через

- общую память (*shared memory*)

- функцию синхронизации всех нитей блока (**__synchronize**)

Подобная иерархия довольно естественна - с одной стороны хочется иметь возможность взаимодействия между отдельными нитями, а с другой - чем больше таких нитей, тем выше оказывается цена подобного взаимодействия.

Поэтому исходная задача (применение ядра к входным данным) разбивается на ряд подзадач, каждая из которых решается абсолютно независимо (т.е. никакого взаимодействия между подзадачами нет) и в произвольном порядке.

Сама же подзадача решается при помощи набора взаимодействующих между собой нитей.

С аппаратной точки зрения все нити разбиваются на так называемые *warp*'ы - блоки подряд идущих нитей, которые одновременно (физически) выполняются и могут взаимодействовать друг с другом. Каждый блок нитей разбивается на несколько *warp*'ов, размер *warp*'а для всех существующих сейчас GPU равен 32.

Важным моментом является то, что нити фактически выполняют одну и ту же команды, но каждая со своими данными. Поэтому если внутри *warp*'а происходит ветвление (например в результате выполнения оператора **if**), то все нити *warp*'а выполняют все возникающие при этом ветви. Поэтому крайне желательно уменьшить ветвление в пределах каждого отдельного *warp*'а.

Также используется понятие *half-warp*'а - это первая или вторая половина *warp*'а. Подобное разбиение *warp*'а на половины связано с тем, что обычно обращение к памяти делаются отдельно для каждого *half-warp*'а.

Кроме иерархии нитей существует также несколько различных типов памяти. Быстродействие приложения очень сильно зависит от скорости работы с памятью. Именно поэтому в традиционных CPU большую часть кристалла занимают различные кэши, предназначенные для ускорения работы с памятью (в то время как для GPU основную часть кристалла занимают ALU).

В CUDA для GPU существует несколько различных типов памяти, доступных нитям, сильно различающихся между собой (см. табл. 1).

Таблица 1. Типы памяти в CUDA.

Тип памяти	Доступ	Уровень выделения	Скорость работы
регистры (registers)	R/W	per-thread	высокая (on chip)
local	R/W	per-thread	низкая (DRAM)
shared	R/W	per-block	высокая (on-chip)
global	R/W	per-grid	низкая(DRAM)
constant	R/O	per-grid	высокая(on chip L1 cache)
texture	R/O	per-grid	высокая(on chip L1 cache)

При этом CPU имеет R/W доступ только к глобальной, константной и текстурной памяти (находящейся в DRAM GPU) и только через функции копирования памяти между CPU и GPU (предоставляемые CUDA API).

Установка CUDA для Windows и Linux

Для установки CUDA просто зайдите на страницу для скачивания CUDA и выберите свою операционную систему. Вам необходимо будет скачать и установить CUDA SDK и CUDA Toolkit. Также может понадобиться обновление видеодрайвера.

На сайте доступны версии для основных дистрибутивов Linux, достаточно подробные инструкции по установке CUDA на Ubuntu 8.10 (под которую еще нет готовой версии CUDA) можно найти здесь.

При этом на компьютер устанавливается все, необходимое для работы с CUDA, включая *runtime* и компилятор **nvcc**. Сам компилятор фактически представляет из себя препроцессор, обрабатывающий код и строящий отдельный код для GPU и CPU. Для компиляции кода для CPU (включая код, необходимый для запуска ядра) **nvcc** использует обычный C/C++-компилятор (на Linux'е он использует gcc).

Основными опциями команды **nvcc** являются:

- **-deviceemu** - компиляция в режиме эмуляции, весь код будет выполняться в многонитевом режиме на CPU и можно использовать обычный отладчик (хотя не все ошибки могут проявится в таком режиме)
- **--use_fast_math** - заменить все вызовы стандартных математических функций на их быстрые (но менее точные) аналоги
- **-o <outputFileName>** - задать имя выходного файла

Из известных глюков CUDA 2 - странные сообщения об ошибке при запуске компилятора **nvcc** из-под Far'а или из make-файла. Однако они легко решаются созданием **.bat**-файла, вызывающего **nvcc** и передающему ему все параметры (пример подобного файла есть в прилагаемом к статье исходном коде).

Расширения языка C

Программы для CUDA (соответствующие файлы обычно имеют расширение ***.cu***) пишутся на "расширенном" C и компилируются при помощи команды **nvcc**.

Вводимые в CUDA расширения языка C состоят из

- спецификаторов функций, показывающих где будет выполняться функция и откуда она может быть вызвана
- спецификаторы переменных, задающие тип памяти, используемый для данной переменных
- директива, служащая для запуска ядра, задающая как данные, так и иерархию нитей
- встроенные переменные, содержащие информацию о текущей нити
- *runtime*, включающий в себя дополнительные типы данных

Спецификаторы

Таблица 2. Спецификаторы функций в CUDA.

Спецификатор	Выполняется на	Может вызываться из
__device__	device	device
__global__	device	host
__host__	host	host

При этом спецификаторы **__host__** и **__device__** могут быть использованы вместе (это значит, что соответствующая функция может выполняться как на GPU, так и на CPU - соответствующий код для обеих платформ будет автоматически сгенерирован компилятором). Спецификаторы **__global__** и **__host__** не могут быть использованы вместе.

Спецификатор **__global__** обозначает ядро и соответствующая функция должна возвращать значение типа **void**.

```
__global__ void myKernel ( float * a, float * b, float * c )
{
    int index = threadIdx.x;

    c [i] = a [i] * b [i];
}
```

На функции, выполняемые на GPU (**__device__** и **__global__**) накладываются следующие ограничения:

- нельзя брать их адрес (за исключением **__global__** функций)
- не поддерживается рекурсия
- не поддерживаются **static**-переменные внутри функции
- не поддерживается переменное число входных аргументов

Для задания размещения в памяти GPU переменных используются следующие спецификаторы - **__device__**, **__constant__** и **__shared__**. На их использование также накладывается ряд ограничений:

- эти спецификаторы не могут быть применены к полям структуры (**struct** или **union**)
- соответствующие переменные могут использоваться только в пределах одного файла, их нельзя объявлять как **extern**
- запись в переменные типа **__constant__** может осуществляться только CPU при помощи специальных функций
- **__shared__** переменные не могут инициализироваться при объявлении

Добавленные переменные

В язык добавлены следующие специальные переменные

- *gridDim* - размер *grid*'а (имеет тип **dim3**)
- *blockDim* - размер блока (имеет тип **dim3**)
- *blockIdx* - индекс текущего блока в *grid*'е (имеет тип **uint3**)
- *threadIdx* - индекс текущей нити в блоке (имеет тип **uint3**)
- *warpSize* - размер *warp*'а (имеет тип **int**)

Добавленные типы

В язык добавляются 1/2/3/4-мерные вектора из базовых типов - **char1**, **char2**, **char3**, **char4**, **uchar1**, **uchar2**, **uchar3**, **uchar4**, **short1**, **short2**, **short3**, **short4**, **ushort1**, **ushort2**, **ushort3**, **ushort4**, **int1**, **int2**, **int3**, **int4**, **uint1**, **uint2**, **uint3**, **uint4**, **long1**, **long2**, **long3**, **long4**, **ulong1**, **ulong2**, **ulong3**, **ulong4**, **float1**, **float2**, **float3**, **float2**, и **double2**.

Обращение к компонентам вектора идет по именам - *x*, *y*, *z* и *w*. Для создания значений-векторов заданного типа служит конструкция вида *make_<typeName>*.

```
int2   a = make_int2   ( 1, 7 );
float3 u = make_float3 ( 1, 2, 3.4f );
```

Обратите внимание, что для этих типов (в отличии от шейдерных языков GLSL/Cg/HLSL) не поддерживаются векторные покомпонентные операции, т.е. нельзя просто сложить два вектора при помощи оператора "+" - это необходимо явно делать для каждой компоненты.

Также для задания размерности служит тип **dim3**, основанный на типе **uint3**, но обладающий нормальным конструктором, инициализирующим все не заданные компоненты единицами.

Директива вызова ядра

Для запуска ядра на GPU используется следующая конструкция:

kernelName <<<Dg,Db,Ns,S>>> (args)

Здесь *kernelName* это имя (адрес) соответствующей **__global__** функции, *Dg* - переменная (или значение) типа **dim3**, задающая размерность и размер *grid*'а (в блоках), *Db* - переменная (или значение) типа **dim3**, задающая размерность и размер блока (в нитях), *Ns* - переменная (или значение) типа **size_t**, задающая дополнительный объем *shared*-памяти, которая должна быть динамически выделена (к уже статически выделенной *shared*-памяти), *S* - переменная (или значение) типа **cudaStream_t** задает поток (*CUDA stream*), в котором должен произойти вызов, по умолчанию используется поток 0. Через *args* обозначены аргументы вызова функции *kernelName*.

Также в язык C добавлена функция **__syncthreads**, осуществляющая синхронизацию всех нитей блока. Управление из нее будет возвращено только тогда, когда все нити данного блока вызовут эту функцию. Т.е. когда весь код, идущий перед этим вызовом, уже выполнен (и, значит, на его результаты можно смело рассчитывать). Эта функция очень удобная для организации безконфликтной работы с *shared*-памятью.

Также CUDA поддерживает все математические функции из стандартной библиотеки C, однако с точки зрения быстродействия лучше использовать их *float*-аналоги (а не *double*) - например *sinf*. Кроме этого CUDA предоставляет дополнительный набор математических функций (*__sinf*, *__powf* и т.д.) обеспечивающие более низкую точность, но заметно более высокое быстродействие чем *sinf*, *powf* и т.п.

Основы CUDA host API

CUDA API для CPU (host) выступает в двух формах - низкоуровневый CUDA drievr API и CUDA runtime API (реализованный через CUDA driver API). В своем приложении Вы можете использовать только один из них, далее мы рассмотрим CUDA runtime API, как более простой и удобный.

Все функции CUDA driver API начинаются с префикса *cu*, а все функции CUDA runtime API начинаются с префикса *cuda*. Каждый из этих API предоставляет основной набор базовых функций, таких как перебор всех доступных устройств (GPU), работа с контекстами и потоками, работа с памятью GPU, взаимодействие с OpenGL и D3D (поддерживается только 9-я версия DirectX).

CUDA runtime API не требует явной инициализации - она происходит автоматически при первом вызове какой-либо его функции. Важным моментом работы с CUDA является то, что многие функции API являются асинхронными, т.е. управление возвращается еще до реального завершения требуемой операции.

К числу асинхронных операций относятся

- запуск ядра
- функции копирования памяти, имена которых оканчиваются на **Async**
- функции копирования памяти *device* <-> *device*
- функции инициализации памяти.

CUDA поддерживает синхронизацию через потоки (*streams*) - каждый поток задает последовательность операций, выполняемых в строго определенном порядке. При этом порядок выполнения операций между разными потоками не является строго определенной и может изменяться.

Каждая функция CUDA API (кроме запуска ядра) возвращает значение типа **cudaError_t**. При успешном выполнении функции возвращается **cudaSuccess**, в противном случае возвращается код ошибки.

Получить описание ошибки в виде строки по ее коду можно при помощи функции *cudaGetErrorString*:

```
char * cudaGetErrorString ( cudaError_t code );
```

Также можно получить код последней ошибки при помощи функции *cudaGetLastError*:

```
cudaError_t cudaGetLastError ();
```

Обратите внимание, что в силу асинхронности выполнения многих вызовов, для получения кода ошибки лучше использовать функцию *cudaThreadSynchronize*, которая дожидается завершения выполнения на GPU всех переданных запросов и возвращает ошибку, если один из этих запросов привел к ошибке.

```
cudaError_t cudaThreadSynchronize ();
```

Работа с памятью в CUDA

Самым простым способом выделения и освобождения памяти (речь идет исключительно о памяти GPU, причем только линейной памяти, другой тип - CUDA-arrays будет рассмотрен в следующей статье) является использование функций **cudaMalloc**, **cudeMallocPitch** и **cudaFree**.

```
float * devPtr;         // pointer to device memory
                        // allocate linear memory for 256 floats
cudaMalloc ( (void **)&devPtr, 256*sizeof(float) );

. . .

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

Для выделения памяти под двухмерные массивы более подходящей является функция **cudeMallocPitch**, которая осуществляет выравнивание (путем добавления к каждой строке дополнительной) строк массива для более эффективного доступа к памяти. При этом в параметре *pitch* возвращается размер строки в байтах.

```
float * devPtr;         // pointer to device memory
int     pitch;          // size of row in bytes
                        // allocate linear memory for width*height 2D array of floats
cudaMallocPitch ( (void **)&devPtr, &pitch, width*sizeof(float), height );

. . .

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

В приведенном фрагменте кода осуществляется выделение памяти на GPU под двухмерный массив из *float*'ов размером *width* и *height*. Элемент с индексом *[col,row]* будет находиться по смещению *col*sizeof(float)+row*pitch*.

$a_{0,0}$	$a_{0,1}$		$a_{0,n-1}$	pad
$a_{m-1,0}$	$a_{m-1,1}$		$a_{m-1,n-1}$	pad

Рис 3. Выравнивание двухмерных массивов в памяти.

Рассмотренные выше функции управляют выделением памяти на GPU, к которой CPU не имеет непосредственного доступа. Поэтому API предоставляет функции копирования памяти, которые позволяют копировать память как между CPU и GPU, так и в пределах GPU.

```
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 );
```

Здесь аргументы *dst* и *src* задают адреса куда и откуда необходимо произвести копирование памяти, параметр *count* задает количество байт памяти, которое необходимо переписать.

Параметр *kind* задает тип копирования памяти и принимает одно из следующих значений - *cudaMemcpyHostToHost*, *cudaMemcpyHostToDevice*, *cudaMemcpyDeviceToHost* и *cudaMemcpyDeviceToDevice*.

Использование event'ов для синхронизации на CPU

Для отслеживания выполнения кода на GPU в CUDA используются *event*'ы (аналогичные *fence*'объектам в расширении NV_fence). Для работы с ними служат следующие функции.

```
cudaError_t cudaEventCreate      ( cudaEvent_t * event );
cudaError_t cudaEventRecord      ( cudaEvent_t event, CUstream stream );
cudaError_t cudaEventQuery       ( cudaEvent_t event );
cudaError_t cudaEventSynchronize ( cudaEvent_t event );
cudaError_t cudaEventElapsedTime ( float * time, cudaEvent_t startEvent, cudaEvent_t stopEvent );
cudaError_t cudaEventDestroy     ( cudaEvent_t event );
```

Функции *cudaEventCreate* и *cudaEventDestroy* служат для создание и уничтожения *event*'ов.

Функция *cudaEventRecord* служит для задания места, прохождение которого должен сигнализировать данный *event*. Если параметр *stream* не равен нулю, то отслеживается только завершение выполнения всех операций в данном потоке. Обратите внимание, что этот запрос является асинхронным - он фактически только обозначает место в потоке команд, прохождение которого потом будет запрашиваться.

Функция *cudaEventQuery* выполняет мгновенную проверку "прохождения" данного *event*а - управление из нее сразу же возвращается. В случае, если все операции, предшествующие данному *event*'у были закончены, то возвращается *cudaSuccess*, иначе возвращается значение *cudaErrorNotReady*.

Явная синхронизация, т.е. ожидание пока все операции для данного *event*'а не будут завершены, обеспечивается командой *cudaEventSynchronize*.

При помощи функции *cudaEventElapsedTime* можно узнать время в миллисекундах (с точностью до половины микросекунды), прошедшее между данными *event*'ами (между моментами, когда каждый из этих *event*'ов был "записан").

Ниже приводится фрагмент кода, запускающий ядро на обработку данных, и замеряющих затраченное на обработку время.

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

                    // asynchronously issue work to the GPU (all to stream 0)
cudaEventRecord ( start, 0 );

                    // call a kernel on data
incKernel<t;<<blocks, threads>>>(dev);

                    // get data back
cudaEventRecord ( stop, 0 );

                    // force synchronization
cudaEventSynchronize ( stop );
cudaEventElapsedTime ( &gpuTime, start, stop );

                    // print the cpu and gpu times
printf("time spent executing by the GPU: %.2f milliseconds\n", gpuTime );
```

Получение информации об имеющихся GPU и их возможностях.

CUDA runtime API предоставляет простой способ получить информацию об имеющихся GPU, которые могут быть использованы CUDA, и обо всех их возможностях. Информация о возможностях GPU возвращается в виде структуры **cudaDeviceProp**.

```
struct cudaDeviceProp
{
    char   name[256];
    size_t totalGlobalMem;
    size_t sharedMemPerBlock;
    int    regsPerBlock;
    int    warpSize;
    size_t memPitch;
    int    maxThreadsPerBlock;
    int    maxThreadsDim [3];
    int    maxGridSize   [3];
    size_t totalConstMem;
    int    major;
    int    minor;
    int    clockRate;
    size_t textureAlignment;
    int    deviceOverlap;
    int    multiProcessorCount;
}
```

Для обозначение возможностей CUDA использует понятие *Compute Capability*, выражаемое парой чисел - *major.minor*. Первое число обозначает глобальную архитектурную версию, второе - небольшие изменение. Так GPU GeForce 8800 Ultra/GTX/GTS имеют *Compute Capability* равную 1.0, GPU GeForce 8800 GT/GS и GeForce 9600 GT имеют *Compute Capability* равную 1.1, GPU GeForce GTX 260 и GeForce GTX 280 имеют *Compute Capability* равную 1.3.

Compute Capability 1.1 поддерживает атомарные операции над 32-битовыми словами в глобальной памяти, *Compute Capability* 1.2 поддерживает атомарные операции в *shared*-памяти и атомарные операции над 64-битовыми словами в глобальной памяти, *Compute Capability* 1.3 поддерживает операции над числами типа *double*.

Ниже приводится исходный текст простой программы, перечисляющей все доступные GPU и их основные возможности.

```
#include <stdio.h>

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    : %d\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 );
    s}

    return 0;
}
```


Примеры использования CUDA

Рассмотрим несколько простых примеров использования CUDA, демонстрирующих основные примером будет простое увеличение каждого элемента одномерного массива на единицу - программ

```
#include <stdio.h>

__global__ void incKernel ( float * data )
{
   int idx = blockIdx.x * blockDim.x + threadIdx.x;

   data [idx] = data [idx] + 1.0f;
}

int main ( int argc, char *  argv [] )
{
    int n        = 16 * 1024 * 1024;
    int numBytes = n * sizeof ( float );

                  // allocate host memory
    float * a = new float [n];

    for ( int i = 0; i < n; i++ )
        a [i] = 0.0f;
```

```
                // allocate device memory
    float * dev = NULL;

    cudaMalloc ( (void**)&dev, numBytes );

                // set kernel launch configuration
    dim3 threads = dim3(512, 1);
    dim3 blocks  = dim3(n / threads.x, 1);

                // create cuda event handles
    cudaEvent_t start, stop;
    float gpuTime = 0.0f;

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

                // asynchronously issue work to the GPU (all to stream 0)
    cudaEventRecord ( start, 0 );
    cudaMemcpy      ( dev, a, numBytes, cudaMemcpyHostToDevice );

    incKernel<<<blocks, threads>>>(dev);

    cudaMemcpy      ( a, dev, numBytes, cudaMemcpyDeviceToHost );
    cudaEventRecord ( stop, 0 );

    cudaEventSynchronize ( stop );
    cudaEventElapsedTime ( &gpuTime, start, stop );

                    // print the cpu and gpu times
    printf("time spent executing by the GPU: %.2f millseconds\n", gpuTime );

                    // check the output for correctness
    printf("------------------------------------------------------------\n");

    for ( int i = 0; i < n; i++ )
        if ( a [i] != 1.0f )
        {
            printf ( "Error in pos %d, %f\n", i, a [i] );

            break;
        }
                // release resources
    cudaEventDestroy ( start );
    cudaEventDestroy ( stop  );
    cudaFree         ( dev   );

    delete a;

    return 0;
}
```

Проще всего устроено ядро - каждой нити соответствует одна нить, блоки и *grid* одномерны. Ядро (функция *incKernel*) на вход получает только указатель на массив с данными в глобальной памяти. Задача ядра - по *threadIdx* и *blockIdx* определить какой именно элемент соответствует данной нити и увеличить именно его.

Поскольку и блоки и *grid* одномерны, то номер нити будет определятся как номер блока, умноженный на количество нитей в блоке, плюс номер нити внутри блока, т.е. *blockIdx.x * blockDim.x + threadIdx.x*.

Функция *main* несколько сложнее - она должна подготовить массив с данными в памяти CPU, после этого необходимо при помощи *cudaMalloc* выделить память под копию массива с данными в глобальной памяти (DRAM GPU). Далее данные копируются функцией *cudaMemcpy* из памяти CPU к глобальную память GPU.

После окончания копирования данные в глобальную память можно запустить ядро для обработки данных и после его вызова скопировать результаты вычислений обратно из глобальной памяти GPU в память CPU.

Также в этом примере производится замер затраченного на копирование и вычисления времени и проверяется корректность полученного результата. После этого освобождается вся выделенная память.

Перемножение двух матриц - простейший подход

Следующий пример будет сложнее (и актуальнее) - мы рассмотрим использование CUDA для перемножения двух квадратных матриц размера *N*N*.

Пусть у нас есть две квадратные матрицы *A* и *B* размера *N*N* (будем считать, что *N* кратно 16). Простейший вариант использует по одной нити на каждый элемент получающейся матрицы *C*, при этом нить извлекает все необходимые элементы из глобальной памяти и производит требуемые вычисления.

Элемент $c_{i,j}$ произведения двух матриц *A* и *B* вычисляется следующим фрагментом псевдокода:

```
c [i][j] = 0
for k in 0..N-1:
   c [i][j] += a [i][k] * b [k][j]
```

Тем самым для вычисления одного элемента произведения матриц нужно выполнить *2*N* арифметических операций и *2*N* чтений из

глобальной памяти. Понятно, что в данном случае основным лимитирующим фактором является скорость доступа к глобальной памяти, которая весьма низка. Каким образом отдельные нити группируются в блоки не важно и не оказывает значительного влияния на быстродействие, которое в данном случае оказывается весьма невысоким.

Ниже приводится листинг соответствующей программы.

```
#include <stdio.h>

#define BLOCK_SIZE  16          // submatrix size
#define N           1024        // matrix size is N*N

__global__ void matMult ( float * a, float * b, int n, float * c )
{
    int   bx  = blockIdx.x;     // block index
    int   by  = blockIdx.y;
    int   tx  = threadIdx.x;        // thread index
    int   ty  = threadIdx.y;
    float sum = 0.0f;           // computed subelement
    int   ia  = n * BLOCK_SIZE * by + n * ty;   // a [i][0]
    int   ib  = BLOCK_SIZE * bx + tx;

                            // Multiply the two matrices together;
    for ( int k = 0; k < n; k++ )
        sum += a [ia + k] * b [ib + k*n];

                            // Write the block sub-matrix to global memory;
                            // each thread writes one element
    int ic = n * BLOCK_SIZE * by + BLOCK_SIZE * bx;

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

int main ( int argc, char *  argv [] )
{
    int numBytes = N * N * sizeof ( float );

                    // allocate host memory
    float * a = new float [N*N];
    float * b = new float [N*N];
    float * c = new float [N*N];

    for ( int i = 0; i < N; i++ )
        for ( int j = 0; j < N; j++ )
        {
                        int     k = N*i + j;

            a [k] = 0.0f;
            b [k] = 1.0f;
        }

                    // allocate device memory
    float * adev = NULL;
    float * bdev = NULL;
    float * cdev = NULL;

    cudaMalloc ( (void**)&adev, numBytes );
    cudaMalloc ( (void**)&bdev, numBytes );
    cudaMalloc ( (void**)&cdev, numBytes );

                    // set kernel launch configuration
    dim3 threads ( BLOCK_SIZE, BLOCK_SIZE );
    dim3 blocks  ( N / threads.x, N / threads.y);

                    // create cuda event handles
    cudaEvent_t start, stop;
    float gpuTime = 0.0f;

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

                    // asynchronously issue work to the GPU (all to stream 0)
    cudaEventRecord ( start, 0 );
    cudaMemcpy      ( adev, a, numBytes, cudaMemcpyHostToDevice );
    cudaMemcpy      ( bdev, b, numBytes, cudaMemcpyHostToDevice );

    matMult<<<blocks, threads>>> ( adev, bdev, N, cdev );

    cudaMemcpy      ( c, cdev, numBytes, cudaMemcpyDeviceToHost );
    cudaEventRecord ( stop, 0 );

    cudaEventSynchronize ( stop );
    cudaEventElapsedTime ( &gpuTime, start, stop );

                        // print the cpu and gpu times
    printf("time spent executing by the GPU: %.2f millseconds\n", gpuTime );
```

```
                // release resources
    cudaEventDestroy ( start );
    cudaEventDestroy ( stop  );
    cudaFree         ( adev  );
    cudaFree         ( bdev  );
    cudaFree         ( cdev  );

    delete a;
    delete b;
    delete c;

    return 0;
}
```

Перемножение двух матриц с использованием *shared*-памяти

Можно заметно повысить быстродействие нашей программы за счет использования *shared*-памяти. Для этого разобьем результирующую матрицы на подматрицы 16*16, вычислением каждой такой подматрицы будет заниматься один блок. Обратите внимание, что для вычисления такой подматрицы нужны только небольшие "полосы" матриц *A* и *B* (см. рис. 4).

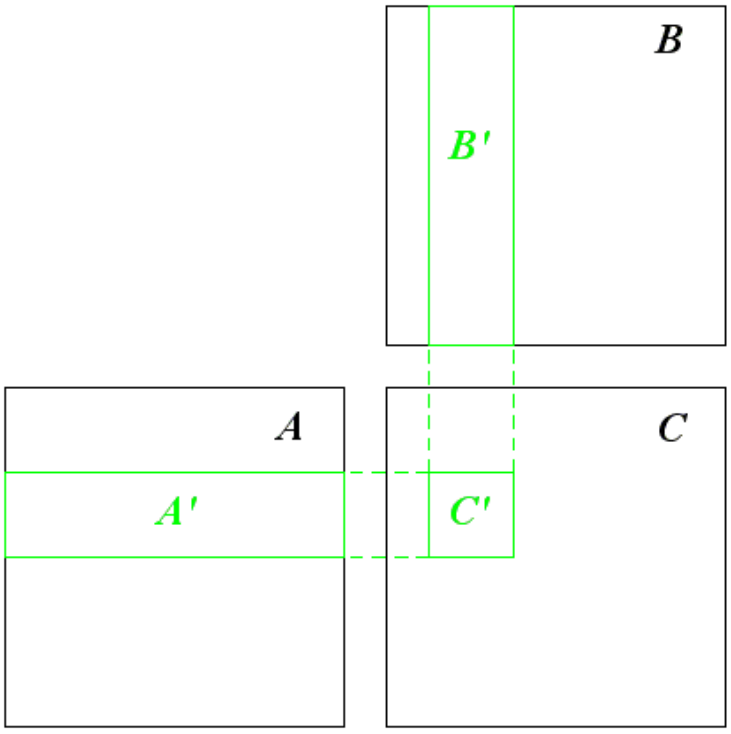


Рис 4. Части матриц *A* и *B*, используемые для вычисления подматрицы *C*.

К сожалению целиком копировать эти "полосы" в *shared*-память практически нереально из-за довольно небольшого объема *shared*-памяти. Поэтому можно поступить другим образом - разобьем эти "полосы" на матрицы 16*16 и вычисление подматрицы произведения матриц будем проводить в *N/16* шагов.

Для этого обратим внимание, что расчет элемента $c_{i,j}$ можно переписать следующим образом, используя разбиение полос на квадратные подматрицы

```
c [i][j] = 0
for step in 0..N/16:
   for k in 0..16:
       c [i][j] += a [i][k+step*16] * b [k+step*16][j]
```

Обратите внимание, что для каждого значения *step* значения из матриц *A* и *B* берутся из двух подматриц размером 16*16. Фактически полосы с рис. 4 просто поделены на квадратные подматрицы и каждому значению *step* соответствует по одной такой подматрице *A* и одной подматрице *B* (см. рис. 5).

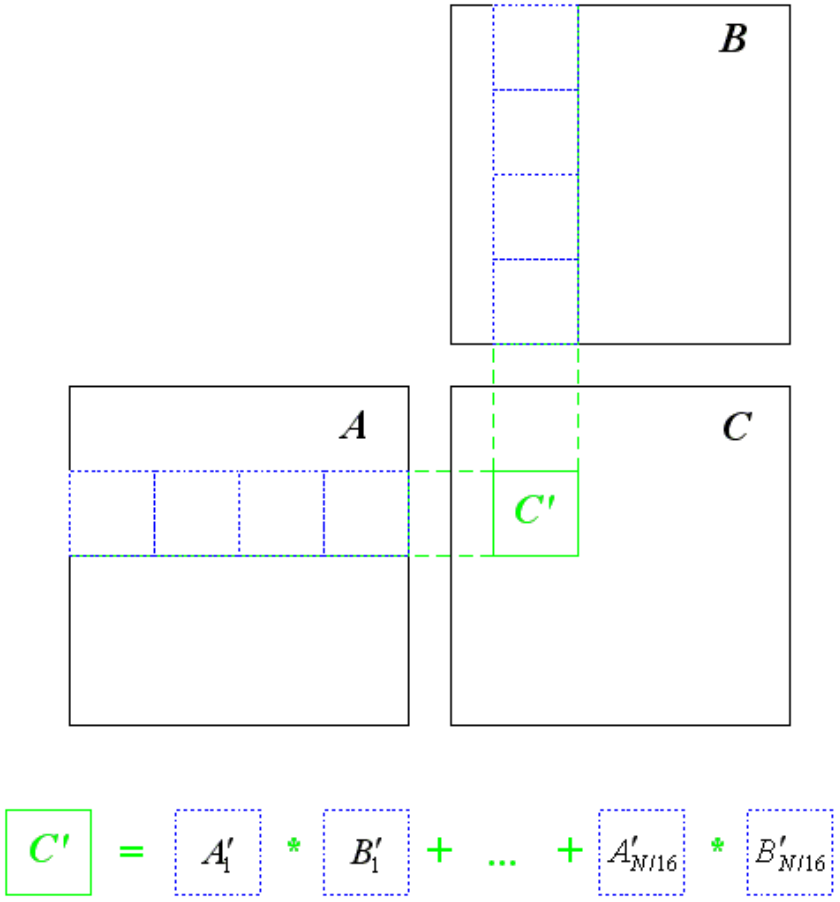


Рис 5. Разбиение полос матриц *A* и *B* на подматрицы 16*16.

На каждом шаге будем загружать в *shared*-память по одной 16*16 подматрице *A* и одной 16*16 подматрице *B*. Далее будем вычислять соответствующую им сумму для элементов произведения, потом загружаем следующие 16*16-подматрицы и т.д.

При этом на каждом шаге одна нить загружает ровно по одному элементу из каждой из матриц *A* и *B* и вычисляет соответствующую им сумму членов. По окончании всех вычислений производится запись элемента в итоговую матрицу.

Обратите внимание, что после загрузки элементов из *A* и *B* нужно выполнить синхронизацию нитей при помощи вызова **__synchronize** для того, чтобы к моменту начала расчетов все нужные элементы (загружаемые остальными нитями блока) были бы уже загружены. Точно также по окончании обработки загруженных подматриц и перед загрузкой следующих также нужна синхронизация (чтобы убедится что текущие 16*16 подматрицы больше не нужно и можно загружать новые).

Ниже приводится соответствующий исходный код.

```
#include <stdio.h>

#define BLOCK_SIZE  16          // submatrix size
#define N           1024        // matrix size is N*N

__global__ void matMult ( float * a, float * b, int n, float * c )
{
    int bx = blockIdx.x;        // block index
    int by = blockIdx.y;

    int tx = threadIdx.x;       // thread index
    int ty = threadIdx.y;

                                // Index of the first sub-matrix of A processed by the block
    int aBegin = n * BLOCK_SIZE * by;
    int aEnd = aBegin + n - 1;
                                // Step size used to iterate through the sub-matrices of A
    int aStep = BLOCK_SIZE;
                                // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * bx;
                                // Step size used to iterate through the sub-matrices of B
    int bStep = BLOCK_SIZE * n;
    float sum = 0.0f;           // computed subelement

    for ( int ia = aBegin, ib = bBegin; ia <= aEnd; ia += aStep, ib += bStep )
    {
                            // Shared memory for the sub-matrix of A
        __shared__ float as [BLOCK_SIZE][BLOCK_SIZE];
                            // Shared memory for the sub-matrix of B
        __shared__ float bs [BLOCK_SIZE][BLOCK_SIZE];

                            // Load the matrices from global memory to shared memory;
        as [ty][tx] = a [ia + n * ty + tx];
```

```
        bs [ty][tx] = b [ib + n * ty + tx];

        __syncthreads();    // Synchronize to make sure the matrices are loaded

                            // Multiply the two matrices together;
        for ( int k = 0; k < BLOCK_SIZE; k++ )
            sum += as [ty][k] * bs [k][tx];

                            // Synchronize to make sure that the preceding
                            // computation is done before loading two new
                            // sub-matrices of A and B in the next iteration
        __syncthreads();
    }

                            // Write the block sub-matrix to global memory;
                            // each thread writes one element
    int ic = n * BLOCK_SIZE * by + BLOCK_SIZE * bx;

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

int main ( int argc, char *  argv [] )
{
    int numBytes = N * N * sizeof ( float );

                    // allocate host memory
    float * a = new float [N*N];
    float * b = new float [N*N];
    float * c = new float [N*N];

    for ( int i = 0; i < N; i++ )
        for ( int j = 0; j < N; j++ )
        {
            a [i] = 0.0f;
            b [i] = 1.0f;
        }

                    // allocate device memory
    float * adev = NULL;
    float * bdev = NULL;
    float * cdev = NULL;

    cudaMalloc ( (void**)&adev, numBytes );
    cudaMalloc ( (void**)&bdev, numBytes );
    cudaMalloc ( (void**)&cdev, numBytes );

                    // set kernel launch configuration
    dim3 threads ( BLOCK_SIZE, BLOCK_SIZE );
    dim3 blocks  ( N / threads.x, N / threads.y);

                    // create cuda event handles
    cudaEvent_t start, stop;
    float gpuTime = 0.0f;

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

                    // asynchronously issue work to the GPU (all to stream 0)
    cudaEventRecord ( start, 0 );
    cudaMemcpy      ( adev, a, numBytes, cudaMemcpyHostToDevice );
    cudaMemcpy      ( bdev, b, numBytes, cudaMemcpyHostToDevice );

    matMult<<<blocks, threads>>> ( adev, bdev, N, cdev );

    cudaMemcpy      ( c, cdev, numBytes, cudaMemcpyDeviceToHost );
    cudaEventRecord ( stop, 0 );

    cudaEventSynchronize ( stop );
    cudaEventElapsedTime ( &gpuTime, start, stop );

                        // print the cpu and gpu times
    printf("time spent executing by the GPU: %.2f millseconds\n", gpuTime );

                    // release resources
    cudaEventDestroy ( start );
    cudaEventDestroy ( stop  );
    cudaFree         ( adev  );
    cudaFree         ( bdev  );
    cudaFree         ( cdev  );

    delete a;
    delete b;
    delete c;
```

```
    return 0;
}
```

Теперь для вычисления одного элемента произведения матриц нам нужно всего *2*N/16* чтений из глобальной памяти. И по результатам сразу видно за счет использования *shared*-памяти нам удалось поднять быстродействие более чем на порядок.

В следующей статье будет рассмотрена работа с текстурами и OpenGL, а также будут рассмотрены основы архитектуры GPU G80 и то, как они соотносятся с CUDA.

Также планируется третья статья, посвященная оптимизации и использованию библиотек CUBLAS и CUFFT.

По этой ссылке можно скачать весь исходный код к этой статье.