ЧАСТЬ 1

CUDA: Ускоряем

Константин Калгин и **Евгений Балдин** ухватились за игровые технологии с целью продвинуть решение научных задач.

Наш эксперт

Константин Калгин
Программист, который превращает персоналку в маленький супер-компьютер.

Наш эксперт

Евгений Балдин
Физик, который действительно знает, что такое нехватка вычислительных ресурсов.

Вам нужны гигафлопсы здесь и сейчас? Вы не хотите ждать светлого будущего OpenCL и согласны ради действительно стоящего дела вляпаться в «вендор-лок»? Тогда выбора как бы и нет – CUDA ждет нового адепта.

Предыстория

Как правило, люди тратят деньги либо на хлеб, либо на зрелища. На первое тратят потому, что хочется дожить до второго. Игровой плебс требует красивой картинки – так и возникли графические ускорители.

Изначально графические ускорители (видеокарты, GPU, Graphical Processing Units) предназначались для вывода двумерной или трехмерной графической информации на экран. Долгое время процесс отображения графической информации на экране управлялся лишь структурами данных – массивами примитивных фигур, текстурами и простейшими цветовыми фильтрами.

Только в XXI веке, хоть и в самом его начале, в графических ускорителях появилась поддержка шейдеров или мини-программ обработки данных на различных стадиях графического конвейера. Процесс отображения картинки теперь стал управляться не только данными, но и мини-программами, исполняющимися на самом графическом ускорителе. Шейдеры дали возможность разработчикам создавать свои собственные спецэффекты, а не ограничиваться уже встроенными в железо. Это значительно увеличило сложность и реалистичность компьютерной графики и, соответственно, привлекательность итоговой картинки для избалованного зрителя.

Фактически сразу после первых шейдеров графические ускорители начали использоваться энтузиастами для решения неграфических задач, то есть для задач общего назначения (GPGPU, General Purpose computations on GPU). Начали появляться статьи, в том числе и в научных журналах, о применении графических ускорителей для решения довольно важных и часто используемых задач линейной алгебры, а также задач, которые не имеют аналитического решения в общем виде – вроде моделирования системы N гравитирующих тел. Зачем серьезные, казалось бы, люди тратили силы и время на железо, целиком и полностью ориентированное на игровую аудиторию? Ответ прост: цена на производительность. В силу большого спроса казуальной аудитории на развлечения графические ускорители стали мощными и относительно дешевыми, хотя и узко специализированными вычислительными системами. На пути энтузиастов была только одна проблема: как бы сделать процедуру загрузки всей этой мощи попроще?

С появлением в 2007 году программно-аппаратной архитектуры CUDA графических ускорителей компании Nvidia ситуация кардинально изменилась. Программы стали составляться не на специальном языке описания шейдеров, а на знакомом C/C++. В описании графических ускорителей CUDA практически перестали использоваться графические термины, такие как шейдеры, точки, текстуры, фильтрация, Z-буфер и пр. Потенциально серьезный конкурент в лице OpenCL появился только в 2010 году, поэтому на сегодня CUDA является фактически единственной зрелой технологией использования графических ускорителей. Во множестве учебных центрах существуют обучающие программы по технологиям CUDA, и множество специалистов уже владеют необходимыми знаниями на вполне приемлемом уровне.

К недостаткам CUDA следует отнести закрытость драйверов и непереносимость кода за пределы платформ от Nvidia. Первое может приводить к весьма странным аберрациям в поведении пользовательских программ и необъяснимым падениям производительности в зависимости от версии компонент этой программно-аппаратной платформы. Второе не позволяет отказаться от нее в пользу использования других платформ, которые сейчас достаточно активно развиваются и в перспективе могут обогнать системы Nvidia.

Самый производительный суперкомпьютер в Top500 (http://www.top500.org) на ноябрь 2012 года включает в качестве одного из своих элементов ускорители Nvidia. Почти десять процентов систем из этого списка также указали ускорители Nvidia в описании своей архитектуры, и эта доля в обозримом будущем будет расти.

Установка CUDA

Предполагается, что в случае серьезной работы, например, на ближайшем доступном университетском кластере, все уже установлено. Если же хочется начать изучать технологию CUDA в домашних условиях, то необходимо для начала установить ее программную часть.

Естественно, необходимо убедиться, что в вашем компьютере есть графический ускоритель от компании Nvidia и что он поддерживает CUDA. За информацией можно обратиться к той же Википедии: http://ru.wikipedia.org/wiki/CUDA.

Затем нужно установить закрытый двоичный драйвер посвежее от компании Nvidia. Например, в Ubuntu 12.04 это делается командой

```
sudo aptitude install nvidia-experimental-310
```

При этом устанавливается драйвер версии 310.14. Здесь и далее мы ориентируемся именно на этот дистрибутив. В случае несовпадении предпочтений с нашими в сети легко найти пошаговую инструкцию фактически для любого другого дистрибутива.

После осознания того, что ваш домашний компьютер теперь годен для установки CUDA, нужно закачать двоичный установщик с сайта разработчика https://developer.nvidia.com/cuda-downloads. Для Ubuntu 12.04 годится бинарник, собранный для версии 11.10. Здесь же нужно выбрать между установочными файлами 64-bit и 32-bit. Далее будем действовать в предположении, что у вас 64-битная версия дистрибутива. Размер установочного файла порядка 670 МБ, но качается он достаточно бодро.

Чтобы иметь возможность скомпилировать поставляемое с дистрибутивом CUDA, нужно удостовериться в наличии следующих пакетов:

```
sudo aptitide install freeglut3-dev build-essential libx11-
dev libxmu-dev libxi-dev libgl1-mesa-glx libglu1-mesa
libglu1-mesa-dev
```

Электронная копия журнала Linux Format. Нелегальное распространение преследуется по закону РФ. Заказ LC0. Владелец копии: Балдин Евгений М. , email: E.m.Baldin@inp.nsk.su

графику

Если у вас 64-битная версия дистрибутива, то для установки примеров необходимо добавить символическую ссылку на библиотеку *libglut.so*.

```
sudo ln -s /usr/lib/x86_64-linux-gnu/libglut.so.3 /usr/lib/libglut.so
```

Установочный скрипт ищет эту библиотеку почему-то в **/usr/lib/**.

Теперь запускаем программу установки:

```
chmod +x cuda_5.0.35_linux_64_ubuntu11.10-1.run
sudo ./cuda_5.0.35_linux_64_ubuntu11.10-1.run
```

Аккуратно пролистываем и соглашаемся с EULA, отказываемся от установки драйвера для видеокарты, но соглашаемся с установкой CUDA и примеров. По умолчанию установка идет в **/usr/local/**, и если умолчание не менялось, то для начала работы достаточно переопределить переменные окружения:

```
export PATH=$PATH:/usr/local/cuda/bin
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/cuda/lib64
```

Перед началом экспериментов полезно побродить по директории **/usr/local/cuda**, посмотрев на 100 МБ документации в директории **doc** и поковырявшись в примерах из директории **samples**. Примеры собираются с помощью команды **make**. В директории **samples/0_Simple/template** располагается заготовка для стандартного CUDA-проекта.

Исходники и их компиляция

Программная часть архитектуры CUDA описывает расширение языка C/C++, функциональность и ключи управления работой компилятора *nvcc*, интерфейс оболочки CUDA Runtime системного драйвера графического ускорителя, профилировщик и отладчик.

Во время исполнения на центральном процессоре компьютера программа запускает функции на графическом ускорителе. Таких функций в программе может быть несколько. Функция, исполняемая на графическом ускорителе, называется ядром [kernel]. Во время запуска ядра порождается множество потоков, которые будут исполнять одну и ту же функцию на процессорах графического ускорителя. Поведение вычислительных потоков зависит как от кода функции, так и от их координат.

Файлы исходников имеют расширение **.cu**:

```
// example1.cu
#include <stdio.h>
#include <cuda_runtime_api.h>
int main() {
  int N;
  cudaDeviceProp prop;
  // Подсчитываем число устройств CUDA
  cudaGetDeviceCount(&N);
  for (int i = 0; i < N; i++){
    // Получаем информацию об устройстве
    cudaGetDeviceProperties(&prop, i);
    // Выводим иформацию об устройстве
    printf("Устройство N %d: %s\n",i+1,prop.name);
  }
}
```

Сборка производится с помощью компилятора *nvcc*:

```
nvcc example1.cu -o example1
./example1
Устройство N 1: GeForce GTS 450
```

В приведенном выше примере пока нет описания ядра, и все исполнение идет на центральном процессоре без привлечения ускорителя.

Расширение языка C/C++ позволяет описывать в одном исходном файле как основную программу, так и ядра с переменными и массивами, которые будут располагаться в памяти графического ускорителя. Кроме того, в расширении языка имеется компактная конструкция для запуска ядер, скрывающая вызов функции драйвера и упаковку аргументов.

Взаимодействие с графическим ускорителем осуществляется через интерфейс CUDA Runtime системного драйвера: запуск ядра, динамическое выделение памяти в графическом ускорителе, копирование данных из/в память графического ускорителя.

Компилятор *nvcc* разделяет входной файл на две части: одна будет исполняться на центральном процессоре и компилироваться стандартным компилятором gcc, а вторая – на графическом ускорителе, и компилироваться самим *nvcc*. Расширениями CUDA языка C/C++ можно пользоваться только в файлах с расширением **.cu**.

Иерархическая организация потоков

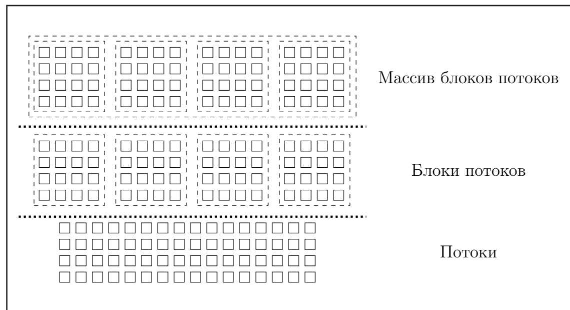
При программировании графического ускорителя необходимо быть в курсе диктуемой архитектурой иерархии исполняемых потоков и памяти. Это ключ к ускорению параллельных вычислений. К сожалению, в этом месте очень легко ошибиться и убить весь эффект большого числа процессоров неэффективным копированием данных.

При запуске программы на графическом ускорителе порождается множество потоков [thread]. Все потоки поделены на группы одного размера – блоки потоков [block]. Максимальный размер блока потоков на современных графических ускорителях равен 1024. У каждого потока и блока потоков имеются свои уникальные идентификаторы, называемые координатами. Тем самым, для разных потоков аргументы исполняемых инструкций и их последовательность могут различаться, поскольку могут зависеть от координат потока и блока потоков. Множества координат потоков и блоков потоков образуют одно-, дву- или трехмерные массивы-сетки [grid]. Размеры блока потоков и массива блоков потоков задаются при запуске ядра.

Во время исполнения ядра потоки одного блока могут синхронизироваться между собой посредством барьеров (механизм __syncthreads()), а потоки разных блоков исполняются независимо. Кроме возможности барьерной синхронизации, потоки одного блока могут взаимодействовать посредством разделяемой памяти. Потоки разных блоков могут взаимодействовать лишь через глобальную память, аналог оперативной памяти в компьютере. Кроме разделяемой и глобальной, есть константная и текстурная памяти, которые доступны из потоков только на чтение. На аппаратном уровне глобальная, константная, текстурная и разделяемая памяти оптимизированы под различные варианты использования. »

» Не хотите пропустить номер? Подпишитесь на www.linuxformat.ru/subscribe/!

Электронная копия журнала Linux Format. Нелегальное распространение преследуется по закону РФ. Заказ LC0. Владелец копии: Балдин Евгений М. , email: E.m.Baldin@inp.nsk.su



› **Иерархическая организация потоков.**

Внутри каждого потока доступны структуры, которые позволяют его идентифицировать:

- **threadIdx** – координаты потока в блоке потоков;
- **blockIdx** – координаты блока потоков в сетке;
- **blockDim** – размеры блока потоков;
- **gridDim** – размеры сетки блоков потоков.

Азбука вызовов

В этом разделе перечислены простейшие языковые структуры, которые смогут пригодиться в процессе введения в среду CUDA.

Функции

Описание произвольной функции может предваряться следующими ключевыми словами:

- **__global__** – это функция-ядро, которая запускается на графическом ускорителе. Тип возвращаемого результата должен быть void.
- **__device__** – функция, которая вызывается из ядра.
- **__host__** – функция, которая вызывается с центрального процессора.

Возможно одновременное использование __device__ и __host__ – это означает, что функция может быть вызвана как из ядра, так и из программы на центральном процессоре. По умолчанию, при отсутствии вышеперечисленных ключевых слов, считается, что функция будет вызываться только с центрального процессора (__host__).

Функция-ядро декларируется примерно так:

```
__global__ void MyKernel(int *a,int *b,int *c) {
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  c[i] = a[i] + b[i];
}
```

Здесь в качестве аргументов передаются три указателя в область памяти видеокарты. Каждый порожденный поток складывает i-е элементы массивов a и b и записывает результат в массив c; номер элемента в массиве i вычисляется в зависимости от координаты потока threadIdx.x, блока потоков blockIdx.x и размера блока потоков blockDim.x по оси Ox.

То же самое можно изобразить и с помощью вспомогательной функции __device__:

```
__device__ int MyDev(int a,int b) {
  return a + b;
}
__global__ void MyKernel(int *a,int *b,int *c) {
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  c[i] = MyDev( a[i], b[i] );
}
```

Чтобы вызвать функцию-ядро, необходимо указать размер блока потоков и размер сетки блоков. Для этого используются тройные угловые скобки:

```
MyKernel<<< 256, N/256 >>>( a, b, c );
```

В случае порождения дву- или трехмерных сеток, что удобно при расчете двумерных и трехмерных задач соответственно, для передачи информации о размере размерах блока и сетки используется специальный тип данных – dim3:

```
__global__ void MyKernel2D(int *a, int *b, int *c) {
  int x = threadIdx.x + blockIdx.x * blockDim.x;
  int y = threadIdx.y + blockIdx.y * blockDim.y;
  int i = y * Nx + x; c[i] = b[i] + a[i];
}
dim3 blockSize( 16, 16, 1 );
dim3 gridSize( Nx/16, Ny/16, 1 );
MyKernel2D<<< blockSize, gridSize >>>( a, b, c );
```

Здесь происходит сложение двух матриц размера Nx×Ny. Количество порождаемых потоков в точности соответствует количеству элементов в матрице.

Работа с памятью

При объявлении переменной для ее размещения в глобальной памяти используется ключевое слово __device__, для размещения в разделяемой памяти – __shared__, а для размещения в константной – __constant__.

Объявленные внутри ядра переменные без использования этих трех ключевых слов автоматически отображаются на регистры или локальную память (область из глобальной памяти, область видимости – поток).

Текстуры объявляются с помощью шаблона texture< >. Размер и область глобальной памяти, к которой будет привязана текстура, определяются с помощью вызова cudaBindTexture().

Функция cudaGetSymbolAddress() через первый аргумент возвращает адрес переменной или массива, объявленного с помощью __device__ или __constant__:

```
texture< float, 1, cudaReadModeElementType > tex;
__device__ int dev_c;
__device__ float dev_a[ 1024*256 ];
__constant__ float dev_g[ 1024*256 ];
__global__ void MyKernel( float * c ){
 int i = threadIdx.x + blockDim.x * blockIdx.x;
  __shared__ int sb[ 256 ];
  sb[ threadIdx.x ] = dev_g[ i ] + dev_c;
  __syncthreads();
  c[i] = sb[threadIdx.x] + tex1Dfetch(tex, i) + dev_g[i];
}
int main(){
  float *p_dev_a;
  cudaGetSymbolAddress( &p_dev_a, dev_a );
  cudaBindTexture( 0, tex, p_dev_a, tex.channelDesc,
1024*256*sizeof(float) );
  MyKernel <<< 1024, 256 >>>( dev_c );
}
```

Функция cudaMalloc позволяет выделить область в глобальной памяти графического ускорителя указанного размера и возвращает указатель на эту область через первый аргумент:

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

При этом значение самого указателя будет храниться в оперативной памяти компьютера:

```
int main(){
  float *dev_a;
  cudaMalloc( &dev_a, sizeof(int)*1024*256 );
  cudaBindTexture( 0, tex, dev_a, tex.channelDesc,
1024*256*sizeof(float) );
  MyKernel <<< 1024, 256 >>>( dev_c );
  cudaFree( dev_a );
```

» **Пропустили номер?** Узнайте на с. 108, как получить его прямо сейчас.

Электронная копия журнала Linux Format. Нелегальное распространение преследуется по закону РФ. Заказ LC0. Владелец копии: Балдин Евгений М. , email: E.m.Baldin@inp.nsk.su

```
}
```

Функция cudaMallocPitch выделяет область памяти для расположения в ней двумерных массивов. При этом увеличивается размер строки в байтах до ближайшего числа, кратного 128, чтобы каждая строка двумерного массива начиналась со 128-байтного сегмента памяти. Это связано с оптимизацией доступа в память:

```
cudaError_t cudaMallocPitch( void** p_dev, size_t* pitch size_t
width, size_t height );
```

где p_dev – возвращаемый адрес выделенной памяти, pitch – возвращаемый новый размер строки в байтах, width – размер исходной строки в байтах, а height – количество строк.

Копирование памяти осуществляется с помощью семейства функций cudaMemcpy:

```
cudaError_t cudaMemcpy(void* dst, const void* src, size_t
count,enum cudaMemcpyKind kind);
cudaError_t cudaMemcpyToSymbol(const char* dst, const void*
src, size_t count, size_t offset=0);
cudaError_t cudaMemcpyFromSymbol(void* dst, const char* src,
size_t count, size_t offset=0);
cudaError_t cudaMemcpy2D(void* dst, size_t dpitch, const
void* src, size_t spitch, size_t width, size_t height, enum
cudaMemcpyKind kind);
```

где dst определяет область памяти, куда копируется сегмент памяти, определяемый переменной src. В функции cudaMemcpy сегменты памяти определяется через указатели. В функциях же cudaMemcpyToSymbol и cudaMemcpyFromSymbol одна из областей является переменной или массивом, определенным с помощью ключевых слов __device__ или __constant__. Функция cudaMemcpy2D копирует двумерный массив, память под который была выделена с помощью вызова cudaMallocPitch.

Через аргумент count указывается количество копируемых байт, а через offset – смещение относительно начала. Последний аргумент kind указывает направление копирования, возможные значения которого:

- cudaMemcpyDeviceToHost
- cudaMemcpyHostToDevice
- cudaMemcpyDeviceToDevice

где Device – память графического ускорителя, а Host – память компьютера.

Ожидание завершения

Запуск ядер осуществляется в асинхронном режиме. Это означает, что после постановки ядра драйвером в очередь на запуск управление тут же возвращается в программу. То есть функция cudaMalloc, вызванная сразу после вызова ядра, скопирует не те данные, которые ожидаются, поскольку ядро не завершило работу. Чтобы дождаться исполнения всех запущенных ядер, необходимо воспользоваться функцией cudaDeviceSynchronize:

```
int main(){
  float *dev_a, *dev_b;
  float *host_a;
  host_a = (float*)malloc(sizeof(float) * 1024*256);
  cudaMalloc( &dev_a, sizeof(float)*1024*256 );
  cudaMalloc( &dev_b, sizeof(float)*1024*256 );
  cudaMemcpy( dev_a, host_a, sizeof(float)*1024*256,
cudaMemcpyHostToDevice );
  MyKernel <<< 1024, 256 >>>( dev_a, dev_b );
  cudaDeviceSynchronize();
  cudaMemcpy( host_a, dev_a, sizeof(float)*1024*256,
cudaMemcpyDeviceToHost );
  cudaFree( dev_a );
  cudaFree( dev_b );
  free( host_a );
}
```

Проверка ошибок

Любая функция CUDA Runtime возвращает значение типа cudaError_t, сигнализирующее об успешности выполнения. Функция cudaGetLastError возвращает результат последней вызванной функции CUDA Runtime или ядра. Функция cudaGetErrorString возвращает строку, расшифровывающую ошибку.

Для удобства можно определить следующий макрос проверки ошибок:

```
#define cudaCheck {
  cudaError_t err = cudaGetLastError();
  if ( err != cudaSuccess ){
    printf(« cudaError = '%s' \n in '%s' %d\n»,\
cudaGetErrorString( err ),\ __FILE__, __LINE__ );
    exit(0);\
  }
}
```

Этот макрос проверяет, была ли ошибка, а если была, то выводит на печать описание, имя файла и номер строки, где возникла ошибка. За счет проверки каждого вызова функций CUDA Runtime и ядер удается найти большую часть ошибок на ранних стадиях разработки программ.

Время работы?

Измерение времени работы ядра с помощью системных функций не точно, поскольку оно включает время, затрачиваемое на взаимодействие программы с драйвером. Для точного измерения времени работы ядер определены специальный тип события cudaEvent_t и функции работы с ним:

```
cudaError_t cudaEventCreate (cudaEvent_t *event)
cudaError_t cudaEventRecord (cudaEvent_t event)
cudaError_t cudaEventSynchronize (cudaEvent_t event)
cudaError_t cudaEventElapsedTime (float *ms, cudaEvent_t start,
cudaEvent_t end)
```

Здесь cudaEventCreate инициализирует переменную типа cudaEvent_t, cudaEventRecord устанавливает событие, cudaEventSynchronize дожидается завершения события, а cudaEventElapsedTime измеряет время между двумя событиями в миллисекундах. Функция cudaEventElapsedTime выдает ошибку, если события start или end не завершились.

Пример программы с измерением времени работы ядра:

```
cudaEvent_t start, end;
float time;
cudaEventCreate( &start );
cudaEventCreate( &end );
cudaEventRecord( start );
MyKernel<<< 1024, 256 >>>();
cudaEventRecord( end );
cudaEventSynchronize( end );
cudaEventElapsedTime( &time, start, end );
printf(«Time: %.2f\n», time/1000.0 );
```

Заключение

Это лишь малая часть технической информации, которая потребуется на пути изучения CUDA. С другой стороны, пользовательский интерфейс – это не ракетные технологии, и его вполне можно освоить за разумный промежуток времени. LXF

Обратная связь

Приглашаем высказаться потенциальных авторов статей по параллельным вычислениям – ценные предложения, критику и советы присылайте по электронной почте: kalgin@ssd.sscc.ru, E.M.Baldin@inp.nsk.su.

Электронная копия журнала Linux Format. Нелегальное распространение преследуется по закону РФ. Заказ LC0. Владелец копии: Балдин Евгений М. , email: E.m.Baldin@inp.nsk.su