Математические структуры и моделирование
2010, вып. 21, с. 96–103

УДК 004.925

ОЦЕНКА ПРОИЗВОДИТЕЛЬНОСТИ СВЯЗЫВАНИЯ CUDA С ГРАФИЧЕСКИМИ API НА ПРИМЕРЕ ЗАДАЧИ SAXPY

А.Ю. Суравикин, В.В. Коробицын

В статье исследована CUDA как одна из технологий GPGPU. Создана программа решения задачи SAXPY на CUDA, описаны способы организации взаимодействия с графическими API. Предоставлены листинги базовой программы и функций обмена данных с OpenGL. Приведено сравнение времени расчетов с аналогичными программами, которые используют другие технологии GPGPU.

Введение

На сегодняшний день существует множество задач, которые требуют высоких вычислительных затрат и имеют параллельные алгоритмы решения. При этом современные графические процессоры (GPU — Graphics Processing Units) обладают параллельной архитектурой и высокой производительностью, поэтому могут эффективно применяться для решения подобных задач. Однако достигнуть высокой скорости решения задач можно лишь после значительной оптимизации алгоритма под параллельную архитектуру и конкретный GPU. Научное направление по созданию, реализации и оптимизации различных алгоритмов, напрямую не связанных с компьютерной графикой, получило названием GPGPU (General Purpose computations on Graphics Processing Units — вычисления общего назначения на графических процессорах).

Компания NVIDIA предложила свое решение, предназначенное для разработки приложений для массивно-параллельных вычислительных устройств, и назвала его CUDA (Compute Unified Device Architecture — унифицированная архитектура компьютерных вычислений). CUDA ориентирована на графические процессоры NVIDIA GeForce 8-й серии и новее, а также специализированные процессоры NVIDIA Tesla. Технология активно развивается и поддерживается разработчиками ПО. Для CUDA разработаны вспомогательные библиотеки: CUBLAS, CUDA Performance Primitives, thrust. В состав пакета CUDA Toolkit входит компилятор PTX (Parallel Thread eXecution), позволяющий работать с ассемблерным кодом программ, выполняемым на графическом процессоре.

Омский государственный университет им. Ф.М. Достоевского.
E-mail: tizona_del_cid@rambler.ru

Целью настоящей статьи является описание подхода создания несложной программы на CUDA и способов обмена данными с графическими API. Для этого реализовано решение простейшей задачи линейной алгебры SAXPY (Scalar Alpha X Plus Y) — задачи скалярного умножения и векторного сложения [1]. Она заключается в вычислении результата двух векторных операций: умножения на скаляр и сложения векторов. Необходимо вычислить новое значение вектора $\mathbf{y}$ по формуле

$$\mathbf{y} = \alpha \cdot \mathbf{x} + \mathbf{y},$$

где α — число, $\mathbf{x}$ и $\mathbf{y}$ — векторы большой размерности.

1. Описание технологии CUDA

Программы для CUDA (соответствующие файлы обычно имеют расширение .cu) пишутся на расширении языка C и компилируются при помощи компилятора nvcc, предоставляемого бесплатно компанией NVIDIA через Интернет.

Перечислим основные расширения языка C, которые введены в CUDA:

1) спецификаторы функций, показывающих, где будет выполняться функция (host или device) и откуда она может быть вызвана;

2) спецификаторы переменных, задающие тип памяти, используемый для данной переменной;

3) директива, предназначенная для запуска ядра программы на GPU, задающая как данные, так и иерархию потоков выполнения;

4) встроенные переменные, содержащие информацию о выполняемом потоке;

5) библиотека, включающая в себя дополнительные типы данных и функции работы с памятью, управления устройством и другое.

Описание архитектуры CUDA и пример использования можно посмотреть в [2], [3].

Функция решения задачи SAXPY на CUDA входит в библиотеку CUBLAS [4], [5]. Однако мы представляем собственную реализацию для оценки возможностей разработки с использованием инструмента CUDA.

2. Пример программы CUDA

Ниже представлен код программы CUDA, который осуществляет инициализацию и загрузку данных на вычислительное устройство (device). Такой код выполняется в модуле, компилируемом программой nvcc, однако некоторыми функциями CUDA можно воспользоваться и в модуле C++ с подключенными заголовочными файлами и библиотеками CUDA.

```
   cudaSetDevice(cutGetMaxGflopsDeviceId());
   // параметры данных
   vectorSize = n;
   dataSize = n * sizeof(float);
   // выделим память для векторов на стороне GPU
   cudaMalloc(&vectorX, dataSize);
   cudaMalloc(&vectorY, dataSize);
   // ... и на CPU
   cudaMallocHost(&vectorXHost, dataSize);
   cudaMallocHost(&resultHost, dataSize);
   // инициализация входных данных:
   // заполним массив памяти CPU (Host) псеводслучайными числами
   // в интервале [0; 1]
   srand(timeGetTime());
   for (int i = 0; i < vectorSize; i++)
   {
      vectorXHost[i] = (float)rand() / RAND_MAX;
   }
   // скопируем данные массива в вектор X на GPU (Device)
   cudaMemcpy(vectorX, vectorXHost, dataSize,
              cudaMemcpyHostToDevice);
   // вектор Y обнулим
   cudaMemset(vectorY, 0, dataSize);
```

Далее представлена функция обработки данных, в которой устанавливаются размеры блока выполнения ядер (kernel) [7]), запускается ядро и копируется результат вычислений в память CPU.

```
void    ProcessData(int iterations)
{
   // размер блока - некоторая константа
   dim3 block(blockSize);
   // количество блоков зависит от общего числа данных
   dim3 grid(vectorSize / block.x);
   // производим несколько итераций запуска ядра
   for (int i = 0; i < iterations; i++)
   {
      // передаем параметр \alpha и указатели на векторы X и Y
      kernel_SAXPY<<<grid, block>>>(0.5f, vectorX, vectorY);
   }
   // копируем обработанные данные в host-память
   cudaMemcpy(resultHost, vectorY, dataSize, cudaMemcpyDeviceToHost);
}
```

Обратим внимание на размеры блоков выполнения. Их значения необходимо подбирать для каждого ядра таким образом, чтобы максимально загрузить

устройство (графический процессор). Для этого оцениваем число занимаемых ядром регистров. Чем меньше регистров используется в ядре, тем больше блоков можно запустить одновременно. Теперь рассмотрим программу-ядро, выполняемую на GPU:

```
__global__ void kernel_SAXPY(float alpha, float* X, float* Y)
{
   uint addr = __umul24(blockIdx.x, blockDim.x) + threadIdx.x;
   Y[addr] = alpha * X[addr] + Y[addr];
}
```

В приведенном листинге ключевое слово __global__ определяет функцию как ядро. В самой функции сначала рассчитываем адрес обрабатываемого элемента, исходя из номера потока, номера и размера блока. Для целочисленного умножения небольших чисел используется функция __umul24(), которая выполняется значительно быстрее, чем полноценное умножение 32-битных чисел. Далее совершаются собственно операции SAXPY: каждый поток вычисляет один элемент вектора, происходит чтение из элементов векторов X, Y и запись результата в элемент вектора Y.

После обработки и копирования данных в память CPU их можно вывести любым способом, доступным для языка C++. Удаление указателей в памяти устройства осуществляется функцией cudaFree(), в памяти хоста — функцией cudaFreeHost().

3. Взаимодействие CUDA с графическими API

Во многих случаях результаты расчетов требуется вывести в графическом виде. Для этого используются специальные функции взаимодействия с графическим API.

3.1. Взаимодействие с OpenGL

В CUDA версии 2.3 взаимодействие с OpenGL ограничено обменом данных с OpenGL Buffer Objects, т.е. с объектами-буферами [7]). Рассмотрим процесс обмена данных между памятью CUDA и буферами OpenGL.

Для работы с OpenGL требуется, чтобы устройство CUDA было указано функцией cudaGLSetGLDevice() перед всеми вызовами функций CUDA. Прежде чем проецировать буфер для CUDA, необходимо его зарегистрировать следующей функцией:

```
GLuint bufferObj;
cudaGLRegisterBufferObject(bufferObj);
```

Когда буфер зарегистрирован, ядро может считывать из него данные или записывать в него. Для обращения необходимо использовать адрес, возвращаемый функцией cudaGLMapBufferObject().

```
GLuint bufferObj;
float* vectorYPtr;
cudaGLMapBufferObject((void**)&vectorYPtr, bufferObj);
```

Например, можно выполнить следующий код:

```
kernel_SAXPY<<<grid, block>>>(0.5f, vectorX, vectorYPtr);
```

После выполнения функции ядра в буфере bufferObj будет содержаться вектор Y, который можно использовать для рендеринга, например в качестве координат точек, трансформируемых далее в вершинном шейдере.

Отключение проекции и регистрации осуществляется при помощи функций cudaGLUnmapBufferObject()и cudaGLUnregisterBufferObject(). При обработке изображений или других двумерных массивов данных имеет смысл передавать не вершинную информацию, а пиксельную. Для этого используется объект Pixel Buffer Object (PBO), который создается следующим образом:

```
glGenBuffers(1, &pboId);
glBindBuffer(GL_PIXEL_UNPACK_BUFFER, pboId);
glBufferData(GL_PIXEL_UNPACK_BUFFER, dataSize, 0, GL_DYNAMIC_DRAW);
glBindBuffer(GL_PIXEL_UNPACK_BUFFER, 0);
```

Здесь pboId – идентификатор буфера-PBO, dataSize – его размер. От создания вершинного буфера он отличается константой GL_PIXEL_UNPACK_BUFFER, обозначающей чтение (распаковку — unpack) командами вида glDrawPixels и glTexImage2D пиксельных данных из буфера. О расширении библиотеки OpenGL ARB_pixel_buffer_object подробнее изложено в [8]. О буферах PBO как части стандарта OpenGL 3.2 в [9]. Буфер pboId один раз регистрируется в CUDA, затем каждый кадр отрисовки проецируется для запуска ядра, далее ядро выполняет расчеты, после чего проекция закрывается. Для рендера полученного изображения копируем данные из PBO в текстуру texId. Следующий код иллюстрирует процесс копирования:

```
glBindBuffer(GL_PIXEL_UNPACK_BUFFER, pboId);
glBindTexture(GL_TEXTURE_2D, texId);
glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, width, height, GL_RGBA,
                GL_FLOAT, 0);
glBindBuffer(GL_PIXEL_UNPACK_BUFFER, 0);
```

Теперь текстуру texId можно наложить на прямоугольник и произвести ее рендер любым подходящим способом. Отметим также, что в бета-версии CUDA 3.0 стало доступным взаимодействие с текстурами OpenGL напрямую.

3.2. Взаимодействие с Direct3D

Взаимодействие CUDA с Direct3D описано в [7]. Приведем лишь основные моменты реализации.

Прежде чем использовать вызовы библиотеки, необходимо передать устройство Direct3D 9 в функцию cudaD3D9SetDirect3DDevice() (для Direct3D 10, соответственно, cudaD3D10SetDirect3DDevice()). Ресурсы Direct3D 9 (вершинные буферы IDirect3DVertexBuffer9, поверхности IDirect3DSurface9 и т.д.) и Direct3D 10 (буферы ID3D10Buffer, и текстуры ID3D10Texture2D и т.д.) ассоциируются функциями cudaD3D9RegisterResource(), cudaD3D10RegisterResource(). Для работы с ресурсами нужно спроецировать их память. Проекция массива ресурсов открывается функцией cudaD3D9MapResources() и закрывается функцией cudaD3D9UnmapResources().

В бета-версии CUDA 3.0 добавлена возможность взаимодействия с Direct3D версии 11.

4. Компьютерный эксперимент

Для оценки производительности реализации на CUDA был проведен компьютерный эксперимент. Определялось время выполнения задачи SAXPY на CUDA и других технологиях GPGPU: DirectCompute и OpenCL. Измерения производились следующим образом: сначала подготавливаются все данные для ядра, затем запускается таймер, после чего выполняется функция-ядро. Далее выполняется синхронизация с CPU любым доступным способом, и таймером замеряется результат. Конфигурация тестовой системы: процессор – Core 2 Duo 2.26 ГГц, оперативная память объемом 4ГБ, видеокарта на основе видеопроцессора GeForce 9800M GTS с 1ГБ видеопамяти.

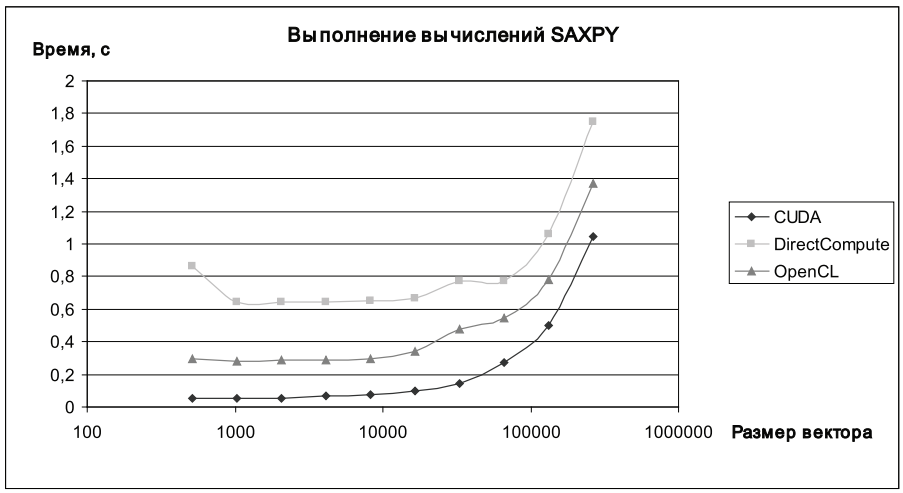

Рис. 1. Сравнение времени вычисления SAXPY на различных API

Графики на рисунке 1 показывают время выполнения операции SAXPY различными API на векторах размером от 512 до 262144 элементов. Видно, что CUDA демонстрирует самую высокую скорость выполнения. OpenCL и DirectCompute отстают по скорости. Возможно, это связано с лишними затратами времени на установку состояния API и синхронизацию с GPU при запус-

ке каждой итерации расчета. Вызовы функций OpenCL и DirectCompute осуществляются через CUDA, поэтому на выполнение требуется дополнительное время.

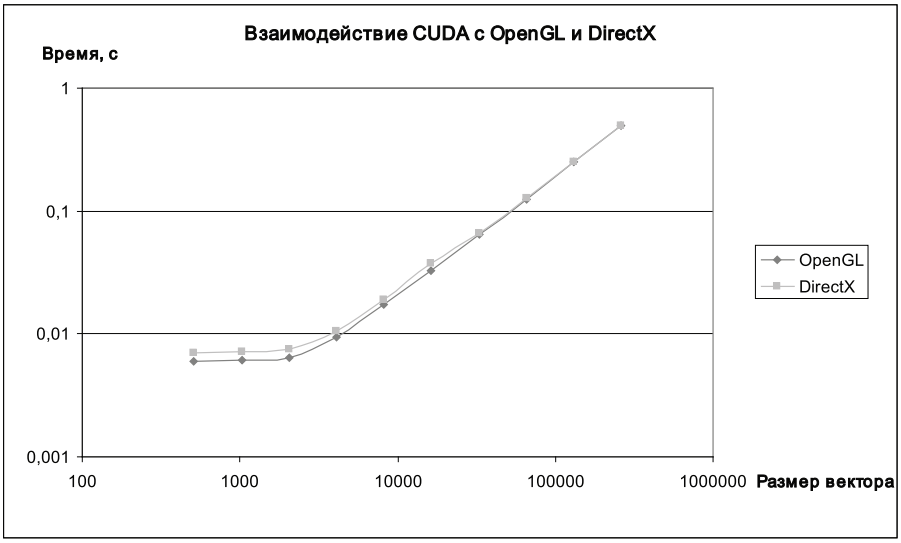

Рис. 2. Время расчета SAXPY на CUDA и вывода результатов с помощью графических API

Второй эксперимент заключался в расчете SAXPY и выводе результатов на экран с помощью средств графических API, т.е. в данном случае проводилось сравнение производительности работы CUDA с OpenGL 3.0 и DirectX 11. Обмен данными производился с помощью копирования данных из буфера. График на рисунке 2 показывает, что при малых объемах данных OpenGL работает быстрее, но при увеличении размера вектора различия становятся минимальными.

5. Выводы

С использованием технологии CUDA было реализовано решение задачи линейной алгебры SAXPY. Также получена программа, которая пересылает исходные данные на GPU, обрабатывает их и возвращает результат. После этого рассмотрено взаимодействие CUDA с графическими API. В случае OpenGL был описан способ обмена данными через Pixel Buffer Object. Для проведения экспериментов также были реализованы программы расчета SAXPY на CUDA, OpenCL, DirectCompute и вывод результатов в графическом виде с использованием OpenGL 3.0 и DirectX 11. Мы не ставили своей задачей освоить работу с разделяемой памятью, хотя оптимизация вычислений под разделяемую, локальную, константную память позволяет значительно повысить производительность приложений CUDA. Это является серьезным преимуществом по сравнению с использованием графического API, так как позволяет оптимизировать программу под конкретную архитектуру оборудования.

Сравнение времени выполнения реализаций алгоритма на CUDA, OpenCL и DirectCompute показало, что взаимодействие с дополнительными API (т.е.

выполнение функций OpenCL и DirectCompute через CUDA) добавляет значительные задержки, особенно при большом числе итераций и незначительной вычислительной нагрузке. Следовательно, для ядер с большими вычислительными затратами API не будет иметь значения, а для большого количества ядер с меньшим временем выполнения CUDA будет выполняться быстрее. Эксперимент с выводом результатов вычислений из CUDA на экран с помощью графических API показал, что производительность одинакова при использовании как OpenGL, так и DirectX.

Литература

1. Описание алгебраической функции SAXPY // Википедия. URL: http://en.wikipedia.org/wiki/SAXPY (дата обращения: 28.02.2010).
2. Боресков А. Основы CUDA. URL: http://steps3d.narod.ru/tutorials/cuda-tutorial.html (дата обращения: 28.02.2010).
3. Zibula A. General Purpose Computation on Graphics Processing Units (GPGPU) using CUDA // Parallel Programming and Parallel Algorithms Seminar. Munster: Westfalische Wilhelms-Universitat, 2009.
4. Библиотека CUBLAS для CUDA. URL: http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/ docs/CUBLAS_Library_2.3.pdf (дата обращения: 28.02.2010).
5. Страницы загрузок средств разработки CUDA 2.3. URL: http://developer.nvidia.com/object/cuda_2_3_downloads.html (дата обращения: 28.02.2010).
6. CUDA 3.0 beta. URL: http://forums.nvidia.com/index.php?showtopic=149959 (дата обращения: 28.02.2010).
7. NVIDIA CUDA Programming Guide. URL: http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/ NVIDIA_CUDA_Programming_Guide_2.3.pdf (дата обращения: 28.02.2010).
8. Спецификация расширения OpenGL ARB_pixel_buffer_object. URL: http://www.opengl.org/registry/specs/ARB/pixel_buffer_object.txt (дата обращения: 28.02.2010).
9. Спецификация OpenGL 3.2. URL: http://www.opengl.org/registry/doc/glspec32.core.20091207.pdf (дата обращения: 28.02.2010).