УДК 004.383

ПРЕИМУЩЕСТВА СОВМЕСТНОГО ИСПОЛЬЗОВАНИЯ CPU И CUDA-УСТРОЙСТВА

Круглов В.Н., Папуловская Н.В., Чирышев А.В.

ФГАОУ ВПО «Уральский федеральный университет имени первого Президента России Б.Н. Ельцина», Екатеринбург, e-mail: v.krouglov@mail.ru

Статья посвящена современному направлению развития информационных технологий – параллельным вычислениям с использованием программно-аппаратной архитектуры CUDA. В работе описана возможность потоковых вычислений на графических процессорах. Приведена модель параллельного программирования на низкоуровневой платформе CUDA. Акцентируется внимание на различиях в организации потоков, обработке данных и доступа к памяти на CPU и GPU. Как показывает проведенный анализ, эффективное программирование массивно-параллельных процессоров требует детального понимания принципов параллельного программирования, а также моделей параллелизма, обмена данными и знания различных архитектурных ограничений этих процессоров. В статье доказывается, что прирост производительности от использования технологии CUDA определяется возможностью распараллеливания кода. Рассмотрен многопоточный обмен информации между центральным и графическим процессором и произведён анализ его эффективности при различных аспектах обработки данных. Перечислены инновационные, технологические особенности современной процессорной архитектуры NVIDIA Kepler.

Ключевые слова: графический процессор, параллельные вычисления, производительность

ADVANTAGES OF COMBINED CPU AND CUDA DEVICES USE

Kruglov V.N., Papulovskaya N.V., Chiryshev A.V.

Ural Federal University n.a. the first President of Russia B.N. Yeltsin, Ekaterinburg, e-mail: v.krouglov@mail.ru

The article is devoted to the modern direction of information technologies – parallel computing platform CUDA invented by NVIDIA. The work describes the possibility of data-flow computing on graphics processors. A model of parallel programming based on low-level platform named CUDA is presented. The differences in the organization of data processing threads and memory access using CPU or GPU are highlighted. Effective GPU programming requires thorough understanding of the principles of parallel programming as well as models of overlapping, data exchange and knowledge of various architectural restrictions of these processors. It is showed that the effectiveness of CUDA technology on performance increasing is relies on the possibility of parallelizing code. Multithreaded exchange of information between the CPU and GPU is inspected and its effectiveness in various aspects of data processing is analyzed. The novelty technological features of state-of-the-art processor named NVIDIA Kepler are listed.

Keywords: graphic engine, parallel calculation, productivity

Применение графических процессоров (graphics processing units – GPU) начиналось с программирования динамической трёхмерной графики. Главным отличием архитектуры графического процессора (GPU), позволившим добиться выдающихся результатов в решении подобных задач, является реальное распараллеливание данных, т.е. способность к одновременной обработке большого числа независимых элементов (пикселей). До 2007 года программирование GPU было возможным только при помощи графических библиотек OpenGL и Microsoft DirectX, а выполняемые на GPU программы, называемые шейдерами (от слова shading – закрашивание, затенение), писались на соответствующих шейдерных языках: GLSL, HLSL, Cg. Изначально программируемые вершинные и пиксельные шейдеры были добавлены в графический конвейер для того, чтобы разработчики игр могли реализовывать более реалистичные визуальные эффекты. Чтобы понять, какие преимущества приносит перенос расчётов на GPU, приведём усреднённые цифры, полученные исследователями по всему миру. В среднем при переносе вычислений на GPU во многих задачах достигается ускорение в 5–30 раз, по сравнению с быстрыми универсальными процессорами.

В конце 2006 года были разработаны низкоуровневые платформы для вычислений общего назначения: NVIDIA CUDA (Compute Unified Device Architecture) и ATI Stream (ранее называлась Close-To-Metal). Главным преимуществом Direct3D и OpenGL по сравнению с низкоуровневыми платформами была их независимость от производителя GPU. В свою очередь низкоуровневые платформы, ориентированные на конкретное изделие, позволяют более гибко программировать GPU и не ограничивают объем массивов данных и количеством инструкций.

В 2003 году Марк Харрис [4] в своей докторской диссертации на тему «Моделирование и рендеринг облаков в реальном времени» ввёл термин GPGPU. GPGPU (General Purpose computing on Graphics Processing Units) – технология использования графических адаптеров для любых

вычислительных задач, решаемых на центральных процессорах. В 2006 году Ян Бак [1] защитил докторскую диссертацию в Стэнфордском университете на тему «Потоковые вычисления на графических процессорах». Такое использование GPU стало возможным после появления аппаратной поддержки плавающей арифметики двойной точности. Реализация арифметики двойной точности, необходимой при решении большого числа научных и технических задач, появилась в 2007–2008 годах в сериях GPU NVIDIA GeForce 200 (чип GT200) и AMD Radeon HD3800 (чип R600).

В конце 2008 года были выпущены специализированные GPGPU-платформы: DirectCompute (в рамках DirectX 11) от Microsoft и OpenCL (Open Compute Language) от группы компаний Khronos, которые независимы от производителя GPU и предоставляют возможности гибкого программирования [2]. Вычисления на GPU развивались очень быстро. В дальнейшем два основных производителя видеочипов, NVIDIA и AMD, разработали и анонсировали соответствующие платформы под названием CUDA (Compute Unified Device Architecture) и CTM (Close To Metal или AMD Stream Computing) соответственно. В отличие от предыдущих моделей программирования GPU эти были выполнены с учётом прямого доступа к аппаратным возможностям видеокарт. Названные платформы не совместимы между собой, CUDA – это расширение языка программирования C, а CTM – виртуальная машина, исполняющая ассемблерный код. Зато обе платформы ликвидировали некоторые из важных ограничений предыдущих моделей GPGPU, использующих традиционный графический конвейер и соответствующие интерфейсы Direct3D или OpenGL.

Основная часть

NVIDIA CUDA (Compute Unified Device Architecture) – это универсальная архитектура параллельных вычислений [5]. Она включает набор инструкций архитектуры ISA (Instruction Set Architecture) и реализацию параллельных вычислений на GPU. Для программирования разработчик может использовать наиболее распространенные языки высокого уровня C и C++.

Аппаратная и программная часть архитектуры CUDA разрабатывались с учетом двух основных целей.

Первая цель – предоставить небольшой набор расширений стандартных языков программирования высокого уровня, позволяющих реализовывать только параллельные алгоритмы.

Вторая цель – предоставить поддержку для гетерогенных вычислений с одновременным использованием CPU и GPU. Последовательные части алгоритма выполняются на CPU, а параллельные – на GPU. Благодаря этому CUDA можно добавлять в имеющиеся приложения постепенно. CPU и GPU считаются отдельными устройствами с независимыми пространствами памяти.

В основе модели параллельного программирования CUDA, описанного в [3], лежат три ключевые абстракции – иерархия потоков обработки, разделяемая память (shared memory) и барьерная синхронизация, которые доступны для программиста через небольшой набор расширений языка C. Эти абстракции предоставляют мелкозернистый параллелизм по данным и по потокам, вложенные в крупнозернистый параллелизм по данным и по задачам. Они приводят программиста к разбиению задачи на подзадачи, которые можно решать независимо и одновременно связками потоков. Каждая подзадача в свою очередь разбивается на более мелкие части, которые решаются совместно потоками из одной связки. Эта декомпозиция позволяет потокам взаимодействовать при решении каждой подзадачи и при этом обеспечивает автоматическую масштабируемость программы. Таким образом, каждая связка потоков может быть направлена на выполнение любому из имеющихся мультипроцессоров, в любом порядке, последовательно или параллельно. Поэтому скомпилированная CUDA-программа может выполняться на GPU с любым количеством мультипроцессоров, и только система времени выполнения должна знать это количество.

Масштабируемая программная модель позволяет архитектуре CUDA распространиться на очень широкий диапазон видеоадаптеров с различным числом мультипроцессоров и объемом памяти: от высокопроизводительных дорогих видеокарт серии GeForce, профессиональных адаптеров серий Quadro и Tesla до дешевых графических чипов, входящих в состав материнских плат [8].

Программирование на CUDA включает подготовку кода для запуска на двух аппаратных платформах одновременно: хост, состоящий из одного или более CPU, и одно или более устройств GPU, как правило, видеоадаптеры совместимые с CUDA.

Видеоадаптеры традиционно ассоциируют с рендерингом трехмерной графики, однако они являются также высокопроизводительными арифметическими устройствами, способными выполнять параллельно тысячи потоков обработки, поэтому применяются и для разнородных вычислений,

поддающихся распараллеливанию. Однако архитектура GPU отличается от архитектуры CPU. Для эффективного использования технологии CUDA важно знать эти различия и понимать, как они определяют производительность программ, выполняемых на графических процессорах.

Главные отличия GPU касаются системы потоков обработки и модели доступа к памяти.

Ресурсы для многопоточности. Конвейеры выполнения на хосте поддерживают небольшое число параллельных потоков обработки. К примеру, сервер с четырьмя 4-ядерными процессорами способен выполнять одновременно не более 16 потоков (не считая технологии HyperThreading, удваивающей это число). Современные GPU от NVIDIA поддерживают до 2048 одновременно активных потоков на мультипроцессор (multiprocessor). На устройствах с 15-ю мультипроцессорами (таких, как NVIDIA GeForce GTX TITAN) в сумме получаем 30720 активных потоков.

Потоки. Потоки на CPU, как правило, «тяжеловесны». Операционная система для обеспечения многозадачности должна управлять динамическим назначением потоков на выполнение ядрам CPU и их последующим снятием с выполнения. Вот почему переключение контекста, возникающее при динамическом снятии с выполнения одного потока и назначении другого, происходит медленно. Потоки на GPU очень «легковесны». В типичной задаче могут выполняться тысячи потоков. До начала вычислений всем потокам распределяются ресурсы памяти и регистров видеоадаптера, что обеспечивает «легкость» переключения между потоками в процессе их исполнения. Обработка потоков осуществляется варпами. Варп (warp) – минимальная единица параллелизма на CUDA-устройстве – содержит 32 потока. Мультипроцессор на GPU выполняет произвольный готовый к исполнению варп из очереди.

Оперативная память. Хост и устройство GPU имеют каждый свою оперативную память. На хосте оперативная память целиком доступна любому выполняющемуся коду (в рамках ограничений, задаваемых операционной системой). Напротив, на графическом устройстве оперативная память разделена между потоками.

Любое CUDA-приложение должно включать следующие этапы:

- выбор и инициализация GPU и выделение массивов данных в его памяти;
- копирование данных из памяти CPU в память GPU;
- запуск вычислительного ядра, выполняемого на GPU заданной иерархией потоков обработки;
- копирование результатов вычислений в память CPU.

Язык программирования CUDA C расширяет стандартный язык C, позволяя программисту определять CUDA-функции (выполняемые на CUDA-устройстве), которые выполняются N раз параллельно N потоками обработки (threads), в отличие от обычных функций C, выполняемых последовательно N раз.

CUDA-функция определяется как функция с дополнительным ключевым словом _global_, а число потоков обработки, выполняющих ядро после конкретного вызова, задается при помощи нового синтаксиса: <<<…>>>. Каждый выполняемый поток имеет уникальный индекс threadID, который доступен внутри ядра через встроенную переменную threadIdx.

Для примера приведем код ядра и его вызов, реализующие сложение двух векторов A и B длины N и сохраняющие результат в вектор C:

```
// Определение ядра
 _global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
int main()
{
    ...
    // Вызов CUDA-функции с созданием на GPU N потоков
    VecAdd<<<1, N>>>(A, B, C);
}
```

В этом примере каждый из N потоков обработки, создаваемых для выполнения ядра VecAdd, производит одно скалярное сложение.

Структура потоков обработки. Для удобства переменная threadIdx представляет собой вектор из трех элементов, поэтому индекс потока может быть одно-

мерным, двухмерным или трехмерным, а в сумме эти потоки образуют одномерную, двухмерную или трехмерную связку. Такая организация устанавливает путь запуска вычисления по всем элементам связки. Индекс потока и его ID связаны следующим простым соотношением: для одномерной связки потоков они совпадают; для двухмерной связки размера (Dx, Dy) и индекса (x, y) идентификатор потока – ($x + yDx$); для трехмерной связки размера (Dx, Dy, Dz) и индекса (x, y, z) идентификатор потока – ($x + yDx + zDxDy$).

В качестве примера теперь приведем CUDA-функцию для сложения матриц размера NxN:

```
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = threadIdx.x, j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}
...
{
    dim3 threadsPerBlock(N, N);
    MatAdd<<<1, threadsPerBlock>>>(A, B, C);
}
```

Имеется ограничение на число потоков в связке, так как все потоки одной связки обязательно будут выполняться на одном мультипроцессоре и должны разделять его ограниченные ресурсы памяти. Так, на первом поколении CUDA-GPU связка могла содержать до 512 потоков. CUDA-устройство может выполнять множество связок одинакового размера, так что суммарное число потоков будет равно числу потоков в связке, умноженному на число связок.

Связки потоков образуют одномерную, двухмерную или трехмерную сетку связок (grid of thread blocks). Число связок в сетке обычно диктуется объемом обрабатываемых данных или числом процессорных ядер, причем число связок может многократно превосходить число процессорных ядер.

Число потоков в связке и число связок в сетке, которые указываются при помощи синтаксиса <<<…>>>, могут иметь скалярный тип int или векторный dim3. Каждая связка в сетке имеет уникальный одномерный или двухмерных индекс, доступный в коде через встроенную переменную blockIdx, а размеры связки – через переменную blockDim.

Покажем, как можно расширить пример сложения матриц использованием нескольких связок потоков:

```
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N) C[i][j] = A[i][j] + B[i][j];
}
...
{
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x,
                   N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);}
```

Размер связки потоков 16×16 (т.е. 256 потоков), хотя и произвольный размер, но используется довольно часто.

Необходимо учитывать, что связки потоков должны выполняться независимо: их выполнение может происходить в любом порядке, как параллельно, так и последовательно. Это требование позволяет выполнять код на CUDA-устройстве с любым числом мультипроцессоров, что и приводит к масштабируемости программы.

Потоки внутри одной связки могут обмениваться данными посредством разделяемой памяти, при этом они должны синхронизировать свое выполнение для координации доступа к этой памяти. Точки синхронизации в ядре задаются вызовом встроенной функции __syncthreads(),

которая работает как барьер, у которого каждый поток дожидается остальных.

В связи с существенными различиями между хостом и CUDA-устройством важно разделять задачу на подзадачи так, чтобы каждая аппаратная часть занималась тем, что у нее лучше всего получается. Для этого необходимо учитывать следующие аспекты.

Устройство идеально подходит для вычислений, которые могут проводиться на многочисленных элементах данных параллельно. Обычно это арифметические операции на данных больших объемов (к примеру, матриц), где одинаковые операции осуществляются над тысячами или даже миллионами элементов одновременно. Это приводит нас к необходимому условию для высокой производительности на CUDA: программа должна использовать большое число потоков обработки. Поддержка большого числа параллельно выполняемых потоков происходит от использования архитектурой CUDA модели легковесных потоков.

Доступы к памяти в программе устройства должны быть когерентными. Определенные шаблоны доступа к памяти позволяют аппаратной части объединять группы операций чтения или записи множества элементов данных в одну операцию. Если данные нельзя структурировать для такого объединенного доступа, и они не обладают достаточным уровнем локальности для эффективного использования кэша, то выигрыш в производительности от вычислений на CUDA большим не будет.

Для использования CUDA данные должны быть перенесены с хоста в память устройства по шине PCI-Express. Эти перемещения относительно дорогие с точки зрения производительности и их частоту необходимо минимизировать. Стоимость перемещения имеет несколько составляющих:

■ Сложность операций по обработке данных должна соответствовать цене перемещения данных на устройство и обратно. Код, который переносит данные для кратковременного использования небольшим числом потоков, не позволит получить выигрыш производительности, или этот выигрыш будет небольшим. В идеальном случае большое число потоков должно произвести большой объем обработки. К примеру, копирование двух матриц на устройство с целью выполнить их сложение не принесет выгоды. Важно учитывать число выполняемых операций, приходящихся на единицу переносимых данных. Если в примере со сложением матриц считать, что они имеют размеры NxN, то получим N^2 скалярных операций сложения на $3N^2$ переносимых числа, то есть соотношение числа операций к объему данных равно 1:3 или O(1), т.е. не зависит от размера задачи. Выигрыш в производительности будет тем больше, чем больше это соотношение. К примеру, умножение тех же матриц потребует уже $2N^3$ операций, и соотношение увеличится до $2N/3 = O(N)$, поэтому выигрыш будет расти пропорционально размерам матриц. Типы операций – дополнительный фактор, так как время выполнения сложения мало по сравнению с вычислением тригонометрических функций. Таким образом, важно учитывать затраты на перенос данных на устройство и обратно при выборе процессора, на котором должны быть произведены операции: CPU или GPU.

■ Данные необходимо сохранять на устройстве как можно дольше. Так как следует минимизировать переносы данных, то программы, запускающие на устройстве несколько вычислительных ядер, должны предпочитать сохранение данных на устройстве между вызовами ядер повторному копированию промежуточных результатов на хост и обратно в цикле. Так, в примере с матрицами, если эти матрицы уже расположены в памяти устройства как результат предыдущих вычислений, или они потребуются в некоторых последующих вычислениях, то их необходимо оставить на устройстве. Такой подход необходимо применять, даже если какой-то один этап из серии расчетов быстрее выполняется на хосте. Даже относительно медленное вычислительное ядро, выполняемое на устройстве, может быть предпочтительно при достаточном уменьшении объема данных, копируемых по шине PCI-Express.

Многопоточный обмен данными позволяет минимизировать затраты времени, вызванные переносом данных между центральным процессором и GPU. Понятие «потоки» (streams) можно рассмотреть как независимые последовательности операций, выполняемые в строго определенном порядке. Операции, присвоенные одному и тому же потоку, выполняются последовательно. При этом порядок выполнения операций между разными потоками не является строго определенным и может изменяться [4, 5].

CUDA позволяет параллельно выполнять операции копирования памяти и загрузки ядра в разных потоках. Таким образом, задача переноса данных между CPU и GPU может быть разбита на несколько частей (потоков). Каждый поток выполнит копирование памяти и загрузку ядра для своей части данных.

Рассмотрим, за счёт чего получается экономия при использовании многопоточного копирования. Предположим, что

существует глобальная задача, которая разбита на две независимые части (задача А и задача Б). Обе задачи могут выполняться параллельно, но каждая должна ждать пока скопируются данные в память GPU. Предположим, что время передачи данных и время выполнение ядра для каждой задачи одинаковое. На рис. 1 представлена временная диаграмма однопоточного выполнения глобальной задачи.

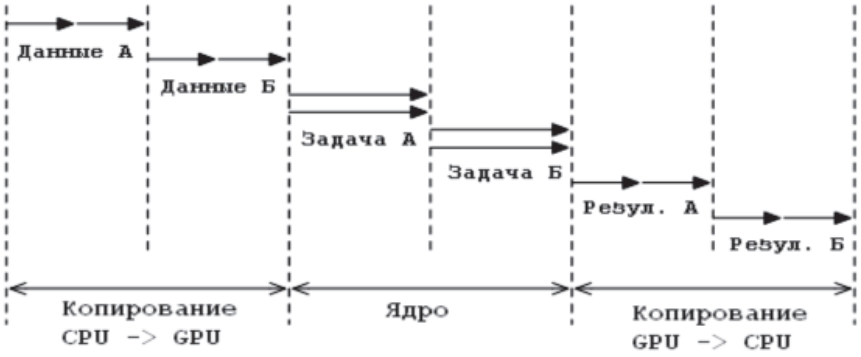

Рис. 1. Диаграмма однопоточного исполнения

Видно, что ядро ждет, пока данные обеих задач копируются из CPU в память GPU. Затем происходит последовательная обработка этих данных и передача их обратно в память CPU. Время выполнения одного потока данных можно рассматривать как уравнение вида

$$t = t_{CPU–GPU} + t_{\text{ядро}} + t_{GPU–CPU}.$$

Теперь рассмотрим многопоточное выполнение глобальной задачи. Так как задачи А и Б независимые, то операции копирования и обработки данных могут выполняться одновременно. Как уже было сказано, поток копирования памяти может перекрываться с потоком исполнения ядра. За счет этого общее время выполнения глобальной задачи может быть уменьшено. Данная стратегия представлена на временных диаграммах (рис. 2 и 3).

На временной диаграмме рис. 2 видно, что время копирования данных задачи Б перекрывается с временем исполнения ядра задачи А. Время выполнения глобальной задачи в таком случае можно рассматривать как уравнение вида

$$t = \frac{t_{CPU-GPU}}{2} + t_{\text{ядро}} + t_{GPU-CPU}.$$

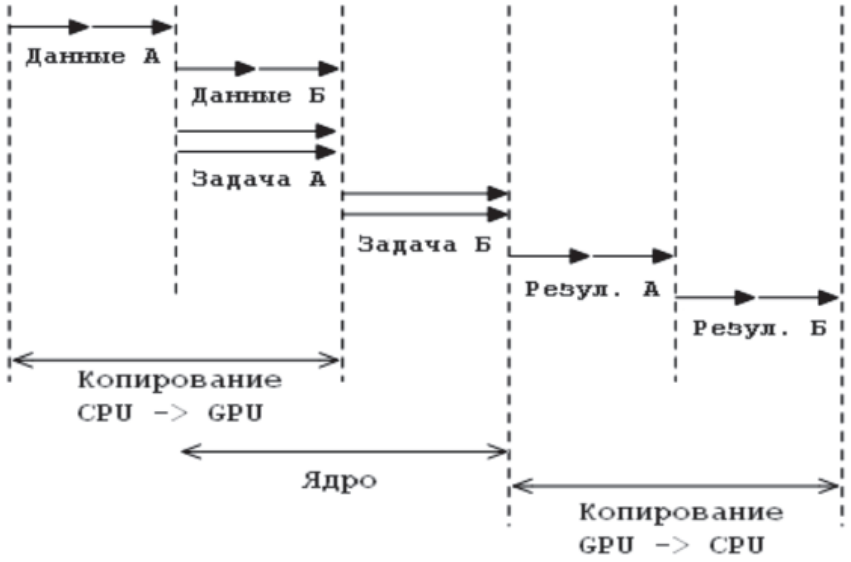

Рис. 2. Диаграмма многопоточного исполнения с перекрытием загрузки данных

Диаграмма, показанная на рис. 3, описывает возможность перекрытия потока исполнения ядра и копирования памяти после того, как один из потоков завершил обработку данных.

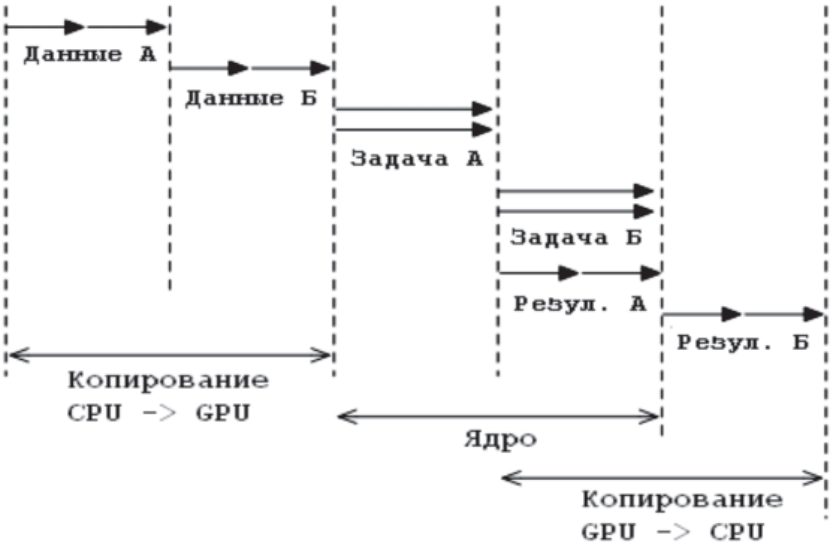

Рис. 3. Диаграмма многопоточного исполнения с перекрытием выгрузки данных

Данные обеих задач последовательно копируются на устройство. Ядро задачи А и задачи Б обрабатывает данные последовательно. После того как ядро задачи А заканчивает обработку данных, результат этой обработки копируется обратно на CPU параллельно с обработкой данных задачи Б. Общее время выполнения глобальной задачи можно рассматривать как уравнение:

$$t = t_{CPU-GPU} + t_{\text{ядро}} + \frac{t_{GPU-CPU}}{2}.$$

Рассмотрим еще один возможный вариант выполнения глобальной задачи, представленный на рис. 4. Из диаграммы видно, что время копирования данных в обоих направлениях перекрывается с временем обработки задачи А и задачи Б. В этом случае общее время исполнения можно рассматривать как уравнение вида:

$$t = \frac{t_{CPU-GPU}}{2} + t_{\text{ядро}} + \frac{t_{GPU-CPU}}{2}.$$

Рис. 4. Диаграмма многопоточного исполнения с перекрытием копирования данных в обоих направлениях

Таким образом, многопоточное исполнение программы с перекрытием передачи данных в обоих направлениях позволяет максимально эффективно организовать вычислительный процесс.

Достижимый прирост производительности от использования технологии CUDA практически целиком определяется возможностью распараллеливания кода. Как уже упоминалось выше, код, который невозможно распараллелить на достаточно большое число потоков, необходимо выполнять на хосте, если только это не приведет к дополнительному копированию данных между хостом и устройством.

Закон Амдала определяет максимальное ускорение, которое можно достичь от распараллеливания какой-то части последовательной программы. Он утверждает, что ускорение программы ограничено величиной

$$S = \frac{1}{(1-P)+\frac{P}{N}},$$

где P – доля времени при последовательном выполнении программы, занимаемая той частью кода, которая будет распараллелена, а N – число процессоров, выполняющих эту часть кода параллельно. С увеличением числа процессоров, отношение P/N будет уменьшаться. Для оценки удобно, чтобы N стремилось к бесконечности, тогда P/N будет стремиться к нулю, а уравнение упростится до формы $S = 1/(1 – P)$. *К* примеру, если распараллелить 3/4 программы, то ускорение относительно последовательной программы будет составлять $1/(1 – 3/4) = 4$ раза.

В большинстве случаев важно помнить только, что с увеличением доли P ускорение будет расти. Кроме того, если P мало, то даже значительное увеличение числа процессоров не сильно увеличит производительность. А если учитывать и затраты на взаимодействие между ними (когда оно необходимо), то производительность может даже уменьшиться. Так что для достижения наилучшего результата необходимо увеличивать объем кода, который будет распараллелен.

Разработанная компанией NVIDIA программно-аппаратная архитектура CUDA хорошо подходит для решения широкого круга задач с высоким параллелизмом. Она постоянно развивается и совершенствуется благодаря инновационным вычислительным технологиям и возможностям. Современные графические ускорители NVIDIA строятся на базе новой архитектуры Kepler. На сегодняшний день это самая быстрая и энергоэффективная архитектура для широкого спектра высокопроизводительных вычислений [6]. Выдающаяся производительность Kepler стала возможна благодаря:

■ *SMX* потоковый мультипроцессор. Архитектура обеспечивает большую энергоэффективность и производительность обработки данных благодаря новому инновационному строению потоковых мультипроцессоров, которое позволяет использовать большую площадь для размещения ядер по сравнению с управляющей логикой.

■ Динамический параллелизм. Функция позволяет потокам GPU динамически генерировать новые потоки, чтобы динамически адаптироваться к данным. Новая технология существенно упрощает параллельное программирование за счет применения GPU-ускорения к широкому спектру распространенных алгоритмов, таких как адаптивное уточнение сеток, быстрые мультипольные и мультисеточные методы [7].

■ *Hyper-Q*. Функция сокращает время ожидания центрального процессора, позволяя многочисленным ядрам CPU одновременно использовать один графический процессор на базе архитектуры Kepler, и значительно увеличивает возможности программирования и энергоэффективность [6].

Заключение

Модель параллельного программирования CUDA позволяет разрабатывать программное обеспечение, независящее от числа используемых процессорных ядер, и при этом имеет легкость в обучении программистов, знакомых со стандартным языком программирования С. Безусловно, написание оптимального кода – задача не из лёгких и нуждается в длительной ручной работе, но CUDA даёт программисту контроль над аппаратными возможностями GPU.

Список литературы

1. Buck I., Purcell T. GPU Gems: Programming techniques, tips and tricks for real-time graphics. URL: http://developer.nvidia.com/object/gpu_gems_home.html (дата обращения 11.10.2013).

2. GPGPU. General-Purpose Computation on Graphics Hardware. URL: http://gpgpu.org (дата обращения 01.10.2013).

3. GPGPU. Использование видеокарт для вычислений. URL: http://www.gpgpu.ru (дата обращения 10.10.2013).

4. Halfhill Tom R. Parallel processing with CUDA// Micriprocessor report, 2008. URL: http://www.nvidia.com/docs/IO/55972/220401_Reprint.pdf . (дата обращения 15.10.2013).

5. Harris M. Mapping computational concepts to GPUs. In GPU Gems 2, Pharr M., (Ed.). AddisonWesley, Mar. 2005, ch. 31, P. 493–508.

6. NVIDIA. Kepler Compute Architecture. URL: http://www.nvidia.com/object/nvidia-kepler.html (дата обращения 03.03.2014).

7. NVIDIA. Dynamic Parallelism in CUDA. URL: http://developer.download.nvidia.com/assets/cuda/files/CUDADownloads/TechBrief_Dynamic_Parallelism_in_CUDA.pdf (дата обращения 03.03.2014).

8. Stratton J.A., Stone S.S., Wen-mei W., Hwu W.W. MCUDA: an Efficient Implementation of CUDA Kernels on Multi-cores // University of Illinois at Urbana-Champaign, 2008. URL: http://www.crhc.uiuc.edu/IMPACT/ftp/report/impact-08-01-mcuda.pdf. (дата обращения 10.10.2013).

References

1. Buck I., Purcell T. GPU Gems: Programming techniques, tips and tricks for real-time graphics. URL: http://developer.nvidia.com/object/gpu_gems_home.html (accessed: 11 October 2013).

2. GPGPU. General-Purpose Computation on Graphics Hardware. URL: http://gpgpu.org (accessed: 01 October 2013).

3. GPGPU. Ispolzovanie videokart dlja vichisleniiy. URL: http://www.gpgpu.ru (accessed: 10 October 2013).

4. Halfhill Tom R. Parallel processing with CUDA// Micriprocessor report, 2008. URL: http://www.nvidia.com/docs/IO/55972/220401_Reprint.pdf. (accessed: 15 October 2013).

5. Harris M. Mapping computational concepts to GPUs. In GPU Gems 2, Pharr M., (Ed.). AddisonWesley, Mar. 2005, ch. 31, pp. 493–508.

6. NVIDIA. Kepler Compute Architecture. URL: http://www.nvidia.com/object/nvidia-kepler.html (accessed: 03 March 2014).

7. NVIDIA. Dynamic Parallelism in CUDA. URL: http://developer.download.nvidia.com/assets/cuda/files/CUDADownloads/TechBrief_Dynamic_Parallelism_in_CUDA.pdf (accessed: 03 March 2014).

8. Stratton J.A., Stone S.S., Wen-mei W., Hwu W.W. MCUDA: an Efficient Implementation of CUDA Kernels on Multi-cores // University of Illinois at Urbana-Champaign, 2008. URL: http://www.crhc.uiuc.edu/IMPACT/ftp/report/impact-08-01-mcuda.pdf. (accessed: 10 October 2013).

Рецензенты:

Доросинский Л.Г., д.т.н., профессор, заведующий кафедрой «Информационные технологии», УрФУ, г. Екатеринбург;

Поршнев С.В., д.т.н., профессор, заведующий кафедрой «Радиоэлектроника информационных систем», УрФУ, г. Екатеринбург.

Работа поступила в редакцию 21.05.2014.