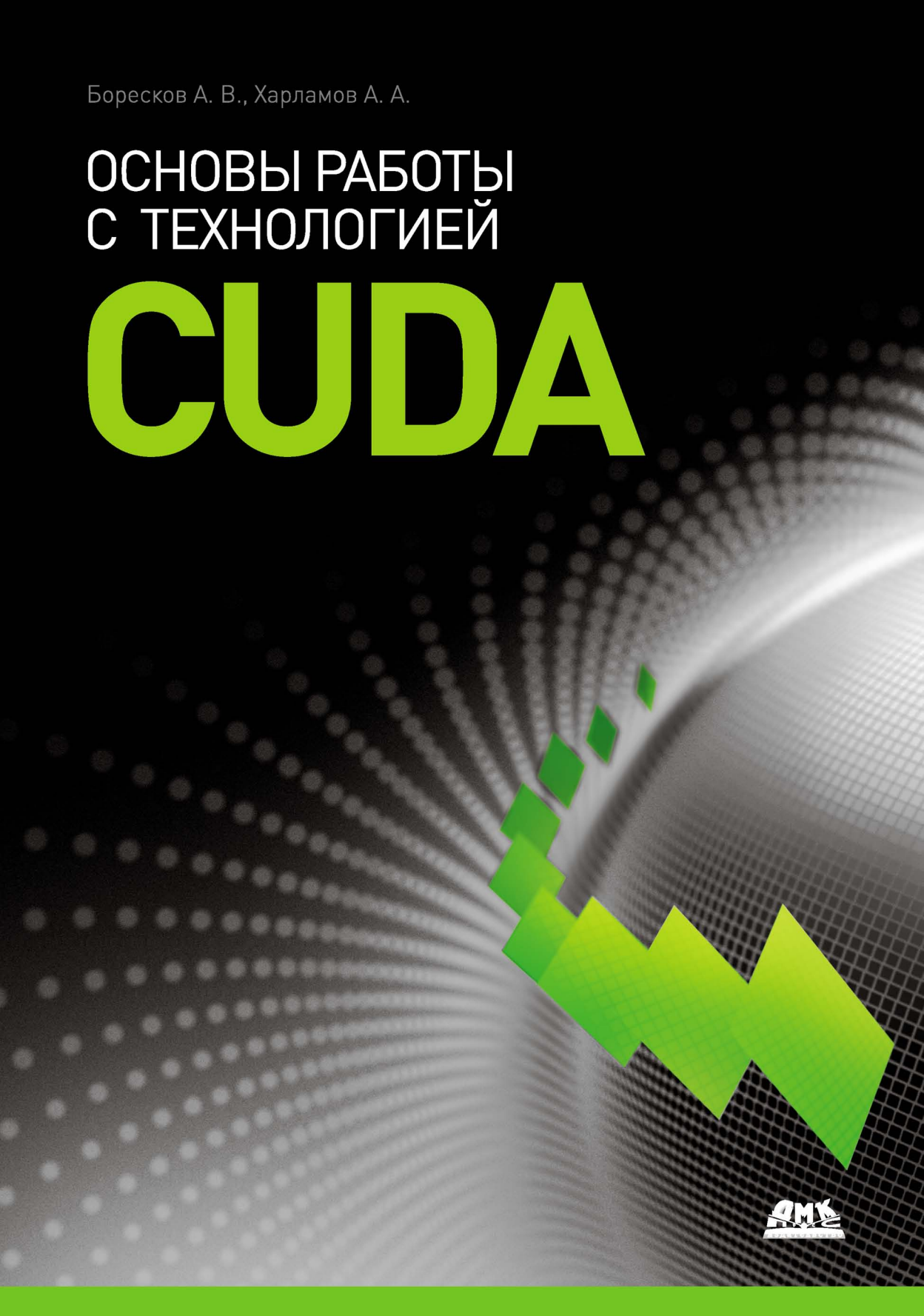
Боресков А. В., Харламов А. А.

ОСНОВЫ РАБОТЫ С ТЕХНОЛОГИЕЙ CUDA



Боресков А. В., Харламов А. А.

Основы работы с технологией CUDA

ДМК Пресс
издательство
Москва, 2010

УДК 32.973.26-018.2
ББК 004.4
Б82

Б82 **Боресков А. В., Харламов А. А.**
Основы работы с технологией CUDA. – М.: ДМК Пресс, 2010. – 232 с.: ил.
ISBN 978-5-94074-578-5

Данная книга посвящена программированию современных графических процессоров (GPU) на основе технологии CUDA от компании NVIDIA. В книге разбираются как сама технология CUDA, так и архитектура поддерживаемых GPU и вопросы оптимизации, включающие использование .PTX.

Рассматривается реализация целого класса алгоритмов и последовательностей на CUDA.

УДК 32.973.26-018.2
ББК 004.4

ISBN 978-5-94074-578-5

Содержание

Глава 1
Существующие многоядерные системы. Эволюция GPU. GPGPU

Одной из важнейших характеристик любого вычислительного устройства является его быстродействие. Для математических расчетов быстродействие обычно измеряется в количестве floating-point операций в секунду (*Flops*). При этом довольно часто рассматривается так называемое пиковое быстродействие, то есть максимально возможное число операций с вещественными (*floating-point*) величинами в секунду (когда вообще нет других операций, обращений к памяти и т. п.).

Реальные цифры по загрузке (пиковая vs реальная) по CPU и GPU

На самом деле реальное быстродействие всегда оказывается заметно ниже пикового, поскольку необходимо выполнять другие операции, осуществлять доступ к памяти и т. п.

Для персонального компьютера быстродействие обычно напрямую связано с тактовой частотой центрального процессора (CPU). Процессоры архитектуры x86 за время с момента своего появления в июне 1978 года увеличили свою тактовую частоту почти в 700 раз (с 4,77 МГц у Intel 8086 до 3,33 ГГц у Intel Core i7).

Таблица 1.1. Динамика роста тактовых частот

Год	Тактовая частота	Процессор
1978	4.77 MHz	Intel 8086
2004	3.46 GHz	Intel Pentium 4
2005	3.8 GHz	Intel Pentium 4
2006	2.333 GHz	Intel Core Duo T2700
2007	2.66 GHz	Intel Core 2 Duo E6700
2007	3 GHz	Intel Core 2 Duo E6800
2008	3.33 GHz	Intel Core 2 Duo E8600
2009	3.06 GHz	Intel Core i7 950

Однако если внимательно посмотреть на динамику роста частоты CPU, то становится заметно, что в последние годы рост частоты заметно замедлился, но зато появилась новая тенденция – создание многоядерных процессоров и систем и увеличение числа ядер в процессоре.

Это связано как с ограничениями технологии производства микросхем, так и с тем фактом, что энергопотребление (а значит, и выделение тепла) пропорцио-

нально четвертой степени частоты. Таким образом, увеличивая тактовую частоту всего в 2 раза, мы сразу увеличиваем тепловыделение в 16 раз. До сих пор с этим удавалось справляться за счет уменьшения размеров отдельных элементов микросхем (так, сейчас корпорация Intel переходит на использование технологического процесса в 32 нанометра).

Однако существуют серьезные ограничения на дальнейшую миниатюризацию, поэтому сейчас рост быстродействия идет в значительной степени на счет увеличения числа параллельно работающих ядер, то есть через параллелизм. Скорее всего, эта тенденция сохранится в ближайшее время, и появление 8- и 12-ядерных процессоров не заставит себя долго ждать.

Максимальное ускорение, которое можно получить от распараллеливания программы на N процессоров (ядер), дается законом Амдала (Amdahl Law):

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

В этой формуле P – это часть времени выполнения программы, которая может быть распараллелена на N процессоров. Как легко видно, при увеличении числа процессоров N максимальный выигрыш стремится к $\frac{1}{1-P}$. Таким образом, если вы можем распараллелить 3/4 всей программы, то максимальный выигрыш составит 4 раза.

Именно поэтому крайне важно использование хорошо распараллеливаемых алгоритмов и методов.

1.1. Многоядерные системы

Рассмотрим в качестве иллюстрации несколько существующих многоядерных систем и начнем наше рассмотрение с процессоров Intel Core 2 Duo и Intel Core i7.

1.1.1. Intel Core 2 Duo и Intel Core i7

Процессор Intel Core 2 Duo содержит два ядра (P0 и P1), каждое из которых фактически является процессором Pentium M, со своим L1-кешем команд и L1-кешем данных (по 32 Кбайта каждое). Также имеется общий L2-кеш (размером 2 или 4 Мб), совместно используемый обоими ядрами (см. рис. 1.1).

Процессор Core i7 содержит уже четыре ядра (P0, P1, P2 и P3). Каждое из этих ядер обладает своими L1-кешем для данных и L1-кешем для команд (по

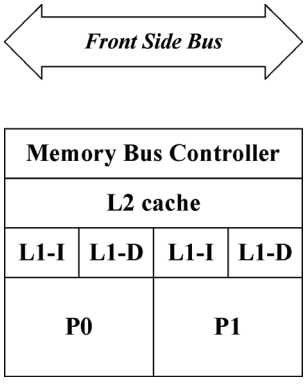


Рис. 1.1. Схема процессора Intel Core 2 Duo

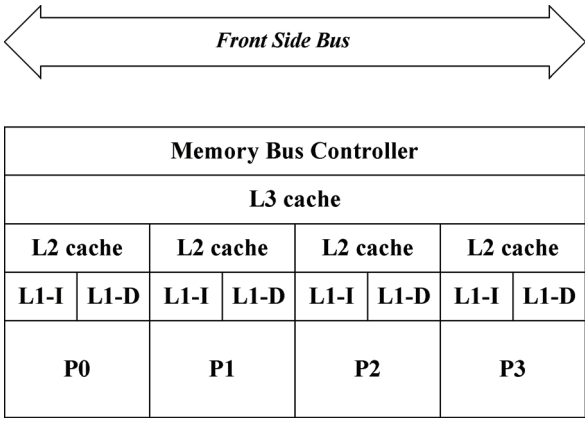


Рис. 1.2. Схема процессора Intel Core i7

32 Кбайта каждое) и L2-кешем (256 Кбайт). Кроме того, имеется общий для всех ядер L3-кеш размером 8 Мбайт.

1.1.2. Архитектура SMP

Кроме этих процессоров, есть также и другие многопроцессорные архитектуры. Одной из таких систем является система на основе симметричной мультипроцессорной архитектуры (Simmetric MultiProcessor Architecture, SMP).

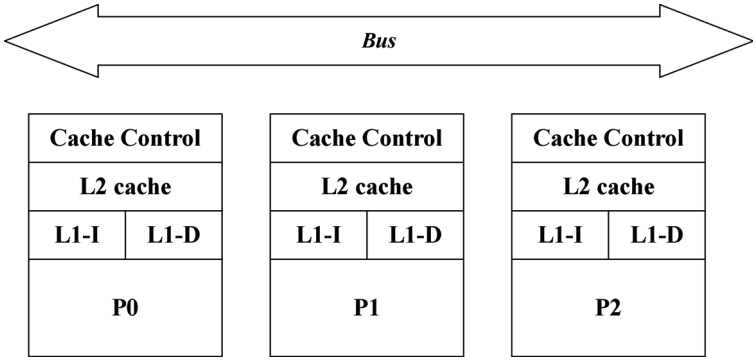


Рис. 1.3. Архитектура SMP-систем

В таких системах каждое ядро содержит свои кеши L1 и L2, и все ядра подсоединены к общей шине. Блок Cache Control отслеживает изменения памяти другими процессорами и обновляет соответствующим образом содержимое кешей – ведь если один и тот же участок памяти содержится в кеше сразу нескольких ядер, то любое изменение этого участка памяти одним из ядер должно быть немедленно передано в кеши других ядер, работающих с данным участком памяти.

Это типичная проблема архитектур с набором процессоров, подключенных к общей шине, – принципиальным моментом для всех многоядерных систем является то, что каждый процессор должен «видеть» целый и корректный образ памяти, что неизбежно ведет к необходимости для каждого процессора отслеживать обращения к памяти всех остальных процессоров для поддержания актуальности своих кешей. Подобная задача имеет квадратичную сложность от числа процессоров.

1.1.3. BlueGene/L

Еще одним примером многопроцессорной архитектуры является суперкомпьютер BlueGene/L. Он состоит из 65 536 двухъядерных узлов (*nodes*). Каждый узел содержит два 770 МГц процессора PowerPC. Каждый из них имеет свои кеши первого и второго уровней, специализированный процессор для *floating-point* вычислений *(Double Hammer FPU*). Оба процессора подключены к общему кешу третьего уровня (L3) размером 4 Мб и имеют собственный блок памяти размером 512 Мб (см. рис. 1.4).

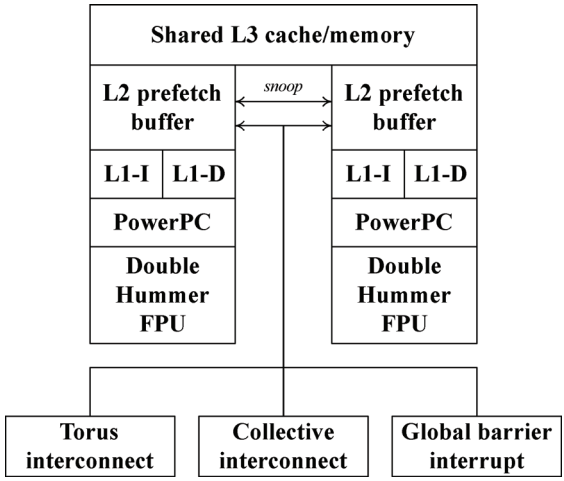


Рис. 1.4. Устройство одного узла в архитектуре BlueGene/L

Отдельные узлы могут соединяться между собой различными способами при помощи набора портов. Таким образом, каждый узел может непосредственно обратиться всего к небольшому числу других узлов, но за счет соединения всех узлов в сеть сообщение может быть передано любому другому узлу за небольшое количество шагов.

Для минимизации числа шагов, необходимых для передачи сообщения от одного узла другому, очень хорошо подходит топология тороидального куба. В ней все узлы образуют куб и у каждого узла есть ровно восемь соседей. При этом если узел лежит на одной из граней куба, то недостающих соседей он берет с противоположной грани (рис. 1.5).

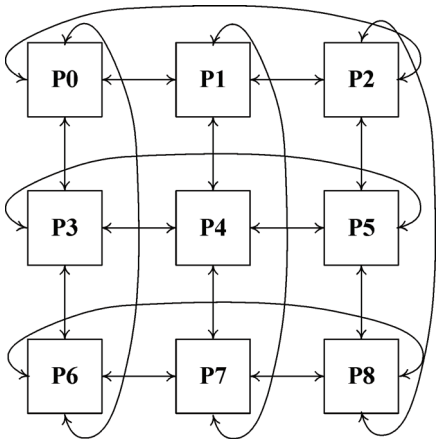


Рис. 1.5. Двухмерный тороидальный куб 3×3

В данной архитектуре именно за счет ограничения на соединения узлов между собой удалось объединить такое большое количество узлов в одном компьютере.

1.1.4. Архитектура GPU

Графические процессоры (GPU) также являются параллельными архитектурами, и на рис. 1.6 изображена сильно упрощенная архитектура GPU серии G80 (архитектура GPU будет подробно рассмотрена в последующих главах). Как видно по рисунку, GPU обладает своей памятью (DRAM), объем которой уже достигает 1 Гбайта для некоторых моделей. Также GPU содержит ряд потоковых мультипроцессоров (SM, Streaming Multiprocessor), каждый из которых способен одновременно выполнять 768 (1024 – для более поздних моделей) нитей. При этом количество потоковых мультипроцессоров зависит от модели GPU. Так, GTX 280 содержит 30 потоковых мультипроцессоров. Каждый мультипроцессор работает независимо от остальных.

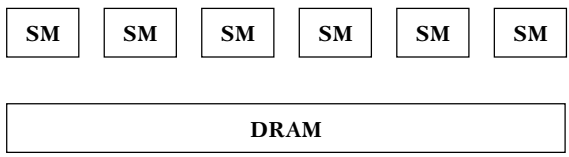


Рис. 1.6. Упрощенная архитектура G80

1.2. Эволюция GPU

Сам термин GPU (Graphics Processing Unit) был впервые использован корпорацией Nvidia для обозначения того, что графический ускоритель, первоначально используемый только для ускорения трехмерной графики, стал мощным про-

граммируемым устройством (процессором), пригодным для решения широкого класса задач, никак не связанных с графикой.

Сейчас современные GPU представляют из себя массивно-параллельные вычислительные устройства с очень высоким быстродействием (свыше одного терафлопа) и большим объемом собственной памяти (DRAM).

Однако начиналось все более чем скромно – первые графические ускорители Voodoo компании 3DFx представляли из себя фактически просто растеризаторы (переводящие треугольники в массивы пикселов) с поддержкой буфера глубины, наложения текстур и альфа-блендинга. При этом вся обработка вершин проводилась центральным процессором, и ускоритель получал на вход уже отображенные на экран (то есть спроектированные) вершины.

Однако именно эту очень простую задачу Voodoo умел делать достаточно быстро, легко обгоняя универсальный центральный процессор, что, собственно, и привело к широкому распространению графических ускорителей 3D-графики.

Причина этого заключалась в значительной степени в том, что графический ускоритель мог одновременно обрабатывать сразу много отдельных пикселов, пусть и выполняя для них очень простые операции.

Вообще, традиционные задачи рендеринга очень хорошо подходят для параллельной обработки – все вершины можно обрабатывать независимо друг от друга, точно так же отдельные фрагменты, получающиеся при растеризации треугольников, тоже могут быть обработаны совершенно независимо друг от друга.

После своего появления ускорители трехмерной графики быстро эволюционировали, при этом, помимо увеличения быстродействия, также росла и их функциональность. Так, графические ускорители следующего поколения (например, Riva TNT) уже могли самостоятельно обрабатывать вершины, разгружая тем самым CPU (так называемый в то время T&L), и одновременно накладывать несколько текстур.

Следующим шагом было увеличение гибкости при обработке отдельных фрагментов (пикселов) с целью реализации ряда эффектов, например попиксельного освещения. На том этапе развития сделать полноценно программируемый обработчик фрагментов было нереально, однако довольно большую функциональность можно было реализовать при помощи появившихся в GeForce256 *register combiner*'ов.

Это были блоки, способные реализовывать довольно простые операции (например, вычисление скалярного произведения). При этом эти блоки можно быстро настраивать и соединять между собой их входы и выходы. Проведя необходимую конфигурацию *register combiner*'ов, можно было реализовывать основные операции попиксельного освещения. В настоящее графический ускоритель, используемый в iPhone и iPod Touch, также поддерживает *register combiner*'ы.

Следующим шагом было появление вершинных программ (GeForce 2) – можно было вместо фиксированных шагов по обработке вершин задать программу, написанную на специальном ассемблере (см. листинг 1). Данная программа выполнялась параллельно для каждой вершины, и вся работа шла над числами типа float (размером 32 бита).

```
!!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
```

Следующим принципиальным шагом стало появление подобной функциональности уже на уровне отдельных фрагментов – возможности задания обработки отдельных фрагментов при помощи специальных программ. Подобная возможность появилась на ускорителях серии GeForce FX. Используемый для задания программ обработки отдельных фрагментов ассемблер был очень близок к ассемблеру для задания вершинных программ и также проводил все операции при помощи чисел типа float.

При этом как вершинные, так и фрагментные программы выполнялись параллельно (графический ускоритель содержал отдельно вершинные процессоры и отдельно – фрагментные процессоры). Поскольку количество обрабатываемых в секунду пикселов было очень велико, то получаемое в результате быстродействие в Flop'ах также было очень большим.

Фактически графические ускорители на тот момент стали представлять себой мощные SIMD-процессоры. Термин SIMD (Single Instruction Multiple Data) обозначает параллельный процессор, способный одновременно выполнять одну и ту же операцию над многими данными. Фактически SIMD-процессор получает на вход поток однородных данных и параллельно обрабатывает их, порождая тем самым выходной поток (рис. 1.7).

Рис. 1.7. Работа SIMD-архитектуры

Сам модуль, осуществляющий подобное преобразование входных потоков в выходные, принято называть ядром (kernel). Отдельные ядра могут соединяться между собой, приводя к довольно сложным схемам обработки входных потоков.

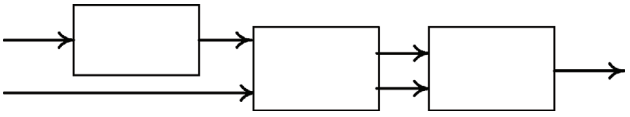


Рис. 1.8. Соединение нескольких ядер для задания сложной обработки данных

Добавление поддержки текстур со значениями в форматах с плавающей точкой (16- и 32-битовые типы float вместо используемых ранее 8-битовых беззнаковых целых чисел) позволило применять фрагментные процессоры для обработки

больших массивов данных. При этом сами такие массивы передавались GPU как текстуры, со значениями компонент типа float, и результат обработки также сохранялся в текстурах такого же типа.

Фактически мы получили мощный параллельный процессор, на вход которому можно передать большой массив данных и программу для их обработки и на выходе получить массив результатов. Появление высокоуровневых языков для написания программ для GPU, таких как Cg, GLSL и HLSL, заметно облегчило создание подобных программ обработки данных.

Ниже приводится пример программы (шейдера) на языке GLSL.

```
varying vec3 lt;
varying vec3 ht;

uniform sampler2D tangentMap;
uniform sampler2D decalMap;
uniform sampler2D anisoTable;

void main (void)
{
  const vec4  specColor = vec4 ( 0, 0, 1, 0 );
  vec3  tang  = normalize ( 2.0*texture2D ( tangentMap, gl_TexCoord [0].xy ).xyz - 1.0);
  float dot1  = dot  ( normalize ( lt ), tang );
  float dot2  = dot  ( normalize ( ht ), tang );
  vec2  arg   = vec2 ( dot1, dot2 );
  vec2  ds    = texture2D ( anisoTable, arg*arg ).rg;
  vec4  color = texture2D ( decalMap, gl_TexCoord [0].xy );

  gl_FragColor   = color * ds.x + specColor * ds.y;
  gl_FragColor.a = 1.0;
}
```

В результате возникло такое направление, как GPGPU (*General-Purpose computing on Graphics Processing Units*) – использование графических процессоров для решения неграфических задач. За счет использования высокой степени параллелизма довольно быстро удалось получить очень высокую производительность – ускорение по сравнению с центральным процессором часто достигало 10 раз.

Оказалось, что многие ресурсоемкие вычислительные задачи достаточно хорошо ложатся на архитектуру GPU, позволяя заметно ускорить их численное решение. Так, многие игры используют довольно сложные модели для расчета волн в воде, решая при этом дифференциальные уравнения на GPU в реальном времени.

Всего за несколько лет за счет использования GPU удалось заметно ускорить решение ряда сложных вычислительных задач, достигая ускорения в 10 и более раз (рис. 1.9).

Ряд процедур обработки изображения и видео с переносом на GPU стали работать в реальном времени (вместо нескольких секунд на один кадр).

При этом разработчики использовали один из распространенных графических API (OpenGL и Direct3D) для доступа к графическому процессору. Используя такой API, подготавливались текстуры, содержащие необходимые входные данные, и через операцию рендеринга (обычно просто прямоугольника) на графическом процессоре запускалась программа для обработки этих данных. Результат

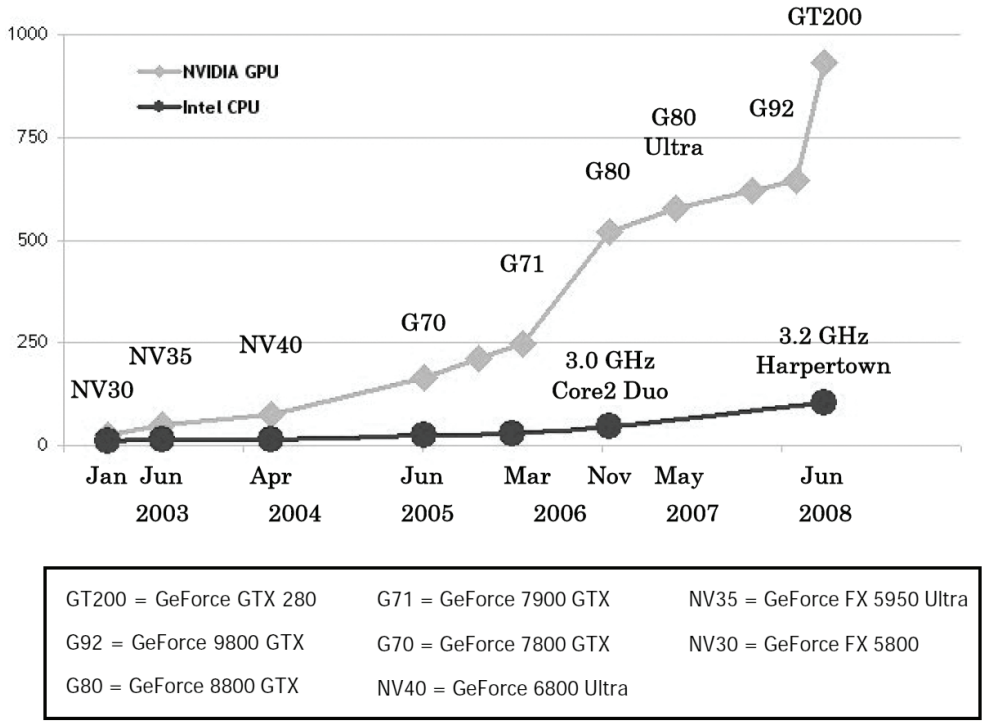


Рис. 1.9. Динамика роста производительности для CPU и GPU

получался также в виде текстур, которые потом считывались в память центрального процессора.

Фактически программа писалась сразу на двух языках – на традиционном языке программирования, например C++, и на языке для написания шейдеров. Часть программы (написанная на традиционном языке программирования) отвечала за подготовку и передачу данных, а также за запуск на GPU программ, написанных на шейдерных языках.

Однако традиционный GPGPU обладает и рядом недостатков, затрудняющих его распространение. Все эти ограничения непосредственно связаны с тем, что использование возможностей GPU происходит через API, ориентированный на работу с графикой (OpenGL или Direct3D).

И в результате все ограничения, изначально присущие данным API (и вполне естественные с точки зрения графики), влияют на реализацию расчетных задач (где подобные ограничения явно избыточны).

Так, в графических API полностью отсутствует возможность какого-либо взаимодействия между параллельно обрабатываемыми пикселами, что в графике действительно не нужно, но для вычислительных задач оказывается довольно желательным.

Еще одним ограничением, свойственным графическим API, является отсутствие поддержки операции типа scatter. Простейшим примером такой операции служит построение гистограмм по входным данным, когда очередной элемент входных данных приводит к изменению заранее неизвестного элемента (или элементов) гистограммы.

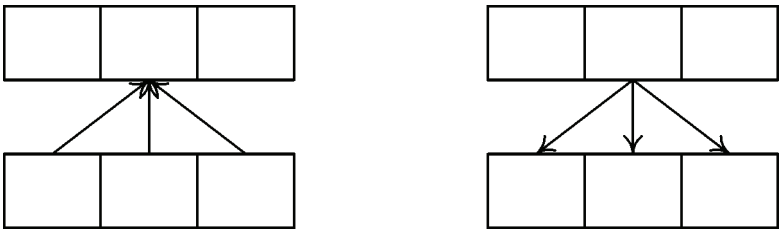


Рис. 1.10. Операции *scatter* и *gather*

Это связано с тем, что в графических API шейдер может осуществлять запись лишь в заранее определенное место, поскольку для фрагментного шейдера заранее определяется, какой фрагмент он будет обрабатывать, и он может записать только значение для данного фрагмента.

Еще одним унаследованным недостатком является то, что разработка ведется сразу на двух языках программирования: один – традиционный, соответствующий коду, выполняемому на CPU, и другой – шейдерный, соответствующий коду, выполняемому на GPU.

Все эти обстоятельства усложняют использование GPGPU и накладывают серьезные ограничения на используемые алгоритмы. Поэтому вполне естественно возникла потребность в средствах разработки GPGPU-приложений, свободных от этих ограничений и ориентированных на решение сложных вычислительных задач.

В качестве таких средств выступают CUDA, OpenCL и DX11 Compute Shaders.

Глава 2
Модель программирования в CUDA. Программно-аппаратный стек CUDA

2.1. Основные понятия

Предложенная компанией Nvidia технология CUDA (Compute Unified Device Architecture) заметно облегчает написание GPGPU-приложений. Она не использует графических API и свободна от ограничений, свойственных этим API.

Данная технология предназначена для разработки приложений для массивно-параллельных вычислительных устройств. На сегодняшний момент поддерживаемыми устройствами являются все GPU компании Nvidia, начиная с серии GeForce8, а также специализированные для решения расчетных задач GPU семейства Tesla.

Основными преимуществами технологии CUDA являются ее простота – все программы пишутся на «расширенном» языке C, наличие хорошей документации, набор готовых инструментов, включающих профайлер, набор готовых библиотек, кроссплатформенность (поддерживаются Microsoft Windows, Linux и Mac OS X).

CUDA является полностью бесплатной, SDK, документацию и примеры можно скачать с сайта developer.nvidia.com. На данный момент последней версией является CUDA 2.3.

CUDA строится на концепции, что GPU (называемый устройством, *device*) выступает в роли массивно-параллельного сопроцессора к CPU (называемому *host*). Программа на CUDA задействует как CPU, так и GPU. При этом обычный (последовательный, то есть непараллельный) код выполняется на CPU, а для массивно-параллельных вычислений соответствующий код выполняется на GPU как набор одновременно выполняющихся нитей (потоков, *threads*).

Таким образом, GPU рассматривается как специализированное вычислительное устройство, которое:

- является сопроцессором к CPU;
- обладает собственной памятью;
- обладает возможностью параллельного выполнения огромного количества отдельных нитей.

При этом очень важно понимать, что между нитями на CPU и нитями на GPU есть принципиальные различия:

- нити на GPU обладают крайне небольшой стоимостью создания, управления и уничтожения (контекст нити минимален, все регистры распределены заранее);
- для эффективной загрузки GPU необходимо использовать много тысяч отдельных нитей, в то время как для CPU обычно достаточно 10–20 нитей.

За счет того, что программы в CUDA пишутся фактически на обычном языке C (на самом деле для частей, выполняющихся на CPU, можно использовать язык C++), в который добавлено небольшое число новых конструкций (спецификаторы типа, встроенные переменные и типы, директива запуска ядра), написание программ с использованием технологии CUDA оказывается заметно проще, чем при использовании традиционного GPGPU (то есть использующего графические API для доступа GPU). Кроме того, в распоряжении программиста оказывается гораздо больше контроля и возможностей по работе с GPU.

В следующем листинге приводится простейший пример – фрагмент кода, использующий GPU для поэлементного сложения двух одномерных массивов.

```
      // Ядро, выполняется параллельно на большом числе нитей.
__global__ void sumKernel ( float * a, float * b, float * c )
{
      // Глобальный индекс нити.
  int idx = threadIdx.x + blockIdx.x * blockDim.x;

      // Выполнить обработку соответствующих данной нити данных.
  c [idx] = a [idx] + b [idx];
}

void sum ( float * a, float * b, float * c, int n )

{
  int numBytes = n * sizeof ( float );
  float * aDev = NULL;
  float * bDev = NULL;
  float * cDev = NULL;

      // Выделить память на GPU.
  cudaMalloc ( (void**)&aDev, numBytes );
  cudaMalloc ( (void**)&bDev, numBytes );
  cudaMalloc ( (void**)&cDev, numBytes );

      // Задать конфигурацию запуска n нитей.
  dim3 threads = dim3(512, 1);
  dim3 blocks  = dim3(n / threads.x, 1);

      // Скопировать входные данные из памяти CPU в память GPU.
  cudaMemcpy ( aDev, a, numBytes, cudaMemcpyHostToDevice );
  cudaMemcpy ( bDev, b, numBytes, cudaMemcpyHostToDevice );

      // Вызвать ядро с заданной конфигурацией для обработки данных.
  sumKernel<<<blocks, threads>>> (aDev, bDev, cDev);

      // Скопировать результаты в память CPU.
  cudaMemcpy ( c, cDev, numBytes, cudaMemcpyDeviceToHost );

      // Освободить выделенную память GPU.
  cudaFree ( aDev );
  cudaFree ( bDev );
  cudaFree ( cDev );
}
```

В приведенном выше листинге первая функция (`sumKernel`) является ядром – она будет параллельно выполняться для каждого набора элементов `a[i]`, `b[i]` и `c[i]`. Спецификатор `__global__` используется для обозначения того, что это ядро, то есть функция, которая работает на GPU и которая может быть вызвана (точнее, запущена сразу на большом количестве нитей) только с CPU.

Вначале функция `sumKernel` при помощи встроенных переменных вычисляет соответствующий данной нити глобальный индекс, для которого необходимо произвести сложение соответствующих элементов и записать результат.

Выполняемая на CPU функция `sum` осуществляет выделение памяти на GPU (поскольку GPU может непосредственно работать только со своей памятью), копирует входные данные из памяти CPU в выделенную память GPU, осуществляет запуск ядра (функции `sumKernel`), после чего копирует результат обратно в память CPU и освобождает выделенную память GPU.

Хотя для массивов, расположенных в памяти GPU, мы и используем обычные указатели, так же как и для данных, расположенных в памяти CPU, важно помнить о том, что CPU не может напрямую обращаться к памяти GPU по таким указателям. Вся работа с памятью GPU ведется CPU при помощи специальных функций.

Рассмотренный выше пример очень хорошо иллюстрирует использование CUDA:

- выделяем память на GPU;
- копируем данные из памяти CPU в выделенную память GPU;
- осуществляем запуск ядра (или последовательно запускаем несколько ядер);
- копируем результаты вычислений обратно в память CPU;
- освобождаем выделенную память GPU.

Обратите внимание, что мы фактически для каждого допустимого индекса входных массивов запускаем отдельную нить для осуществления нужных вычислений. Все эти нити выполняются параллельно, и каждая нить может получить информацию о себе через встроенные переменные.

Важным моментом является то, что хотя подобный подход очень похож на работу с SIMD-моделью, есть и принципиальные отличия (компания Nvidia использует термин *SIMT – Single Instruction, Multiple Thread*). Нити разбиваются на группы по 32 нити, называемые *warp*'ами. Только нити в пределах одного *warp*'а выполняются физически одновременно. Нити из разных *warp*'ов могут находиться на разных стадиях выполнения программы. При этом управление *warp*'ми прозрачно осуществляет сам GPU.

Для решения задач CUDA использует очень большое количество параллельно выполняемых нитей, при этом обычно каждой нити соответствует один элемент вычисляемых данных. Все запущенные на выполнение нити организованы в следующую иерархию (рис. 2.1).

Верхний уровень иерархии – сетка (*grid*) – соответствует всем нитям, выполняющим данное ядро. Верхний уровень представляет из себя одномерный или двухмерный массив блоков (*block*). Каждый блок – это одномерный, двухмерный

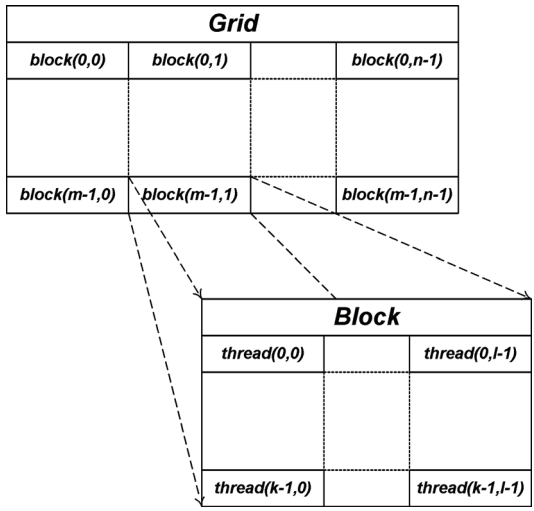


Рис. 2.1. Иерархия нитей в CUDA

или трехмерный массив нитей (*thread*). При этом все блоки, образующие сетку, имеют одинаковую размерность и размер.

Каждый блок в сетке имеет свой адрес, состоящий из одного или двух неотрицательных целых чисел (индекс блока в сетке). Аналогично каждая нить внутри блока также имеет свой адрес – одно, два или три неотрицательных целых числа, задающих индекс нити внутри блока.

Поскольку одно и то же ядро выполняется одновременно очень большим числом нитей, то для того, чтобы ядро могло однозначно определить номер нити (а значит, и элемент данных, который нужно обрабатывать), используются встроенные переменные `threadIdx` и `blockIdx`. Каждая из этих переменных является трехмерным целочисленным вектором. Обратите внимание, что они доступны только для функций, выполняемых на GPU, – для функций, выполняющихся на CPU, они не имеют смысла.

Также ядро может получить размеры сетки и блока через встроенные переменные `gridDim` и `blockDim`.

Подобное разделение всех нитей является еще одним общим приемом использования CUDA – исходная задача разбивается на набор отдельных подзадач, решаемых независимо друг от друга (рис. 2.2). Каждой такой подзадаче соответствует свой блок нитей.

При этом каждая подзадача совместно решается всеми нитями своего блока. Разбиение нитей на warp'ы происходит отдельно для каждого блока; таким образом, все нити одного *warp*'а всегда принадлежат одному блоку. При этом нити могут взаимодействовать между собой только в пределах блока. Нити разных блоков взаимодействовать между собой не могут.

Подобный подход является удачным компромиссом между необходимостью обеспечить взаимодействие нитей между собой и стоимостью обеспечения подоб-

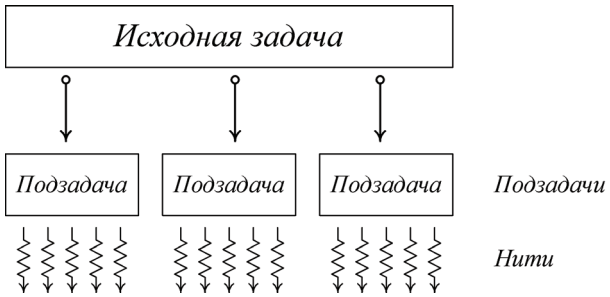


Рис. 2.2. Разбиение исходной задачи на набор независимо решаемых подзадач

ного взаимодействия – обеспечить возможность взаимодействия каждой нити с каждой было бы слишком сложно и дорого.

Существуют всего два механизма, при помощи которых нити внутри блока могут взаимодействовать друг с другом:

- разделяемая (*shared*) память;
- барьерная синхронизация.

Каждый блок получает в свое распоряжение определенный объем быстрой разделяемой памяти, которую все нити блока могут совместно использовать. Поскольку нити блока не обязательно выполняются физически параллельно (то есть мы имеем дело не с «чистой» SIMD-архитектурой, а имеет место прозрачное управление нитями), то для того, чтобы не возникало проблем с одновременной работой с shared-памятью, необходим некоторый механизм синхронизации нитей блока.

CUDA предлагает довольно простой способ синхронизации – так называемая барьерная синхронизация. Для ее осуществления используется вызов встроенной функции `__syncthreads()`, которая блокирует вызывающие нити блока до тех пор, пока все нити блока не войдут в эту функцию. Таким образом, при помощи `__syncthreads()` мы можем организовать «барьеры» внутри ядра, гарантирующие, что если хотя бы одна нить прошла такой барьер, то не осталось ни одной за барьером (не прошедшей его)(рис. 2.3).

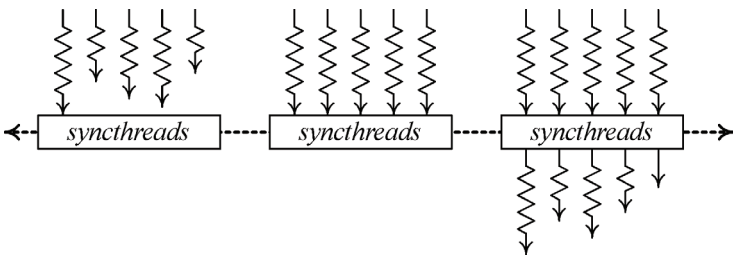


Рис. 2.3. Барьерная синхронизация

2.2. 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;
- спецификаторов переменных, задающих тип памяти, используемый для данных переменных;
- директивы, служащей для запуска ядра, задающей как данные, так и иерархию нитей;
- встроенных переменных, содержащих информацию о текущей нити;
- *runtime*, включающей в себя дополнительные типы данных.

2.2.1. Спецификаторы функций и переменных

В CUDA используется следующие спецификаторы функций (табл. 2.1).

Таблица 2.1. Спецификаторы функций в CUDA

Спецификатор	Функция выполняется на	Функция может вызываться из
__device__	device (GPU)	device (GPU)
__global__	device (GPU)	host (CPU)
__host__	host (CPU)	host (CPU)

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`.

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`;