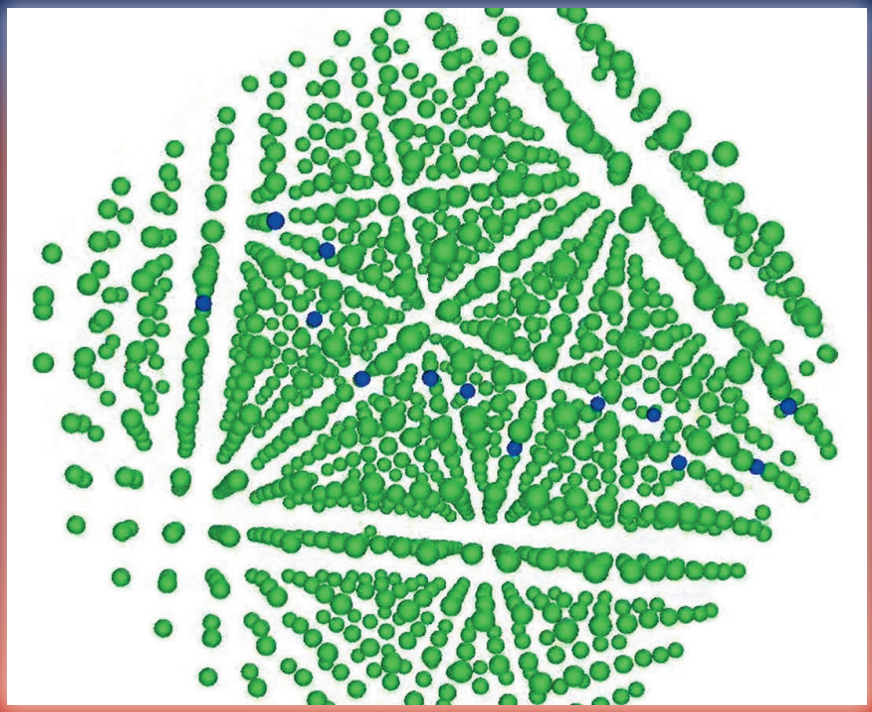
Уральский федеральный университет
имени первого Президента России Б. Н. Ельцина
Физико-технологический институт

К. А. НЕКРАСОВ
С. И. ПОТАШНИКОВ
А. С. БОЯРЧЕНКОВ
А. Я. КУПРЯЖКИН

ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ ОБЩЕГО НАЗНАЧЕНИЯ НА ГРАФИЧЕСКИХ ПРОЦЕССОРАХ

Учебное пособие



Министерство образования и науки Российской Федерации
Уральский федеральный университет
имени первого Президента России Б. Н. Ельцина

К. А. Некрасов, С. И. Поташников,
А. С. Боярченков, А. Я. Купряжкин

Параллельные вычисления общего назначения на графических процессорах

Учебное пособие

Рекомендовано методическим советом УрФУ
для студентов, обучающихся по направлениям подготовки
14.04.02 — Ядерная физика и технологии;
09.04.02 — Информационные системы и технологии;
14.04.01 — Ядерные реакторы и материалы

Екатеринбург
Издательство Уральского университета
2016

УДК 004.032.24:004.383.5(075.8)
ББК 32.97я73
П18

Авторы:
Некрасов К. А., Поташников С. И., Боярченков А. С., Купряжкин А. Я.

Рецензенты:
Институт теплофизики УрО РАН (д-р физ.-мат. наук, проф. В. Г. Байдаков); гл. науч. сотр. лаборатории математического моделирования Института промышленной экологии УрО РАН д-р физ.-мат. наук, проф. А. Н. Вадаксин

П18 **Параллельные вычисления общего назначения на графических процессорах** : учебное пособие / К. А. Некрасов, С. И. Поташников, А. С. Боярченков, А. Я. Купряжкин. — Екатеринбург : Изд-во Урал. ун-та, 2016. — 104 с.

ISBN 978-5-7996-1722-6

В учебном пособии изложены основные принципы организации высокоскоростных параллельных вычислений на графических процессорах. Рассмотрены подходы к программированию графических процессоров с использованием шейдерной модели и NVIDIA CUDA. Проанализированы примеры.

Пособие предназначено для проведения практических занятий по программированию графических процессоров для магистрантов.

Библиогр.: 22 назв. Рис. 24. Табл. 3. Прил. 1.

УДК 004.032.24:004.383.5(075.8)
ББК 32.97я73

ISBN 978-5-7996-1722-6

Введение

Для предсказания характеристик и поведения больших систем широко используется вычислительное моделирование, наиболее принципиальным методом повышения производительности которого является распараллеливание вычислений. До последнего времени наиболее доступными системами для параллельных расчетов были кластеры персональных компьютеров или близких к ним по архитектуре машин, в которых вычисления производились на центральных процессорах общего назначения (CPU). Однако такие кластеры достаточно дороги и сложны в эксплуатации. К тому же архитектура CPU персональных компьютеров не оптимизирована для интенсивных математических вычислений, поскольку основной задачей таких процессоров является исполнение последовательных программ со сложным ветвлением.

Во второй половине 1990-х годов началось быстрое развитие *графических процессоров* (*GPU*) — дополнительных вычислительных устройств для ускоренного исполнения алгоритмов визуализации трехмерных сцен [1–3]. Поскольку трехмерная визуализация допускает эффективное распараллеливание расчетов, графические процессоры разрабатывались как поточно-параллельные системы с большим количеством вычислительных блоков, конвейерной обработкой данных и памятью с максимальной пропускной способностью.

Современные графические процессоры выполняют не только стандартные алгоритмы визуализации, но и сложные пользовательские программы, что позволяет решать на них задачи общего назначения, включая физико-математическое моделирование [4–6]. При параллельных расчетах GPU может обеспечить производительность кластера из сотен обычных персональных компьютеров. По соотношению производительности и цены графические процессоры имеют большое преимущество перед другими вычислительными системами, в том числе перед специализированными суперкомпьютерами.

1. Структура и возможности вычислительной системы с графическим процессором

1.1. Задача компьютерной визуализации трехмерных сцен

Во многих областях использования компьютеров существует задача визуализации (представления на экране) трехмерных изображений (в дальнейшем — сцен). Трехмерная (3D) визуализация необходима, например, в компьютерных играх, при создании анимации и спецэффектов, а также для инженерного проектирования, наглядного представления физико-математических моделей. Поскольку графические процессоры создавались именно для решения задачи визуализации, рассмотрим ее основные составляющие.

Пусть, для определенности, трехмерная сцена представляет собой совокупность поверхностей, разбитых для дискретной компьютерной обработки на плоские треугольники. В некоторых массивах, которые обычно называют текстурами, хранится информация о цветах треугольников. Кроме того, известны положения и характеристики источников света. Задача визуализации состоит в том, чтобы сформировать изображение этой сцены на плоскости экрана, положение которой определяется точкой зрения наблюдателя.

Процесс формирования изображения включает в себя следующие основные этапы:

- проектирование треугольников, представляющих сцену, на плоскость экрана;
- разбиение полученных проекций на отдельные пиксели, для которых будут определяться цвета (стадия растеризации треугольников);

- определение видимого цвета элементов поверхности треугольников с учетом исходного цвета и отражающих свойств самой поверхности, прозрачности других поверхностей, освещенности и теней.

Обрабатываемые сцены обычно состоят из очень многих треугольников, которые разбиваются на еще большее количество пикселей, так что их визуализация требует больших вычислительных ресурсов. Вместе с тем и вершины, и пиксели можно обрабатывать почти или совсем независимо друг от друга. Соответственно задача визуализации допускает очень эффективное распараллеливание.

Именно в целях распараллеливания графических вычислений центральные процессоры персональных компьютеров стали *суперскалярными* — получили возможность одновременного (*векторного*) исполнения некоторых операций сразу над несколькими числами (расширения 3DNow! и SSE).

Большинство центральных процессоров ПК состоит из нескольких ядер, что дополнительно увеличивает их потенциал как систем для параллельных вычислений. Тем не менее исторически сложилось так, что наиболее эффективными устройствами для распараллеливания вычислений на ПК стали и до сих пор остаются специализированные графические процессоры (GPU). Эти процессоры, разрабатывавшиеся именно для обработки графики, практически полностью ориентированы на параллельную обработку данных.

1.2. Архитектура графического процессора (GPU)

1.2.1. Распараллеливание вычислений по данным

Конструктивно графический процессор представляет собой вычислительное устройство, работающее отдельно от центрального процессора, параллельно с ним. Обычно графические процессоры размещают на отдельных печатных платах с собственной системой охлаждения, которые называют **графическими ускорителями** (или **видеокартами**). Вместе с GPU на плате графического ускорителя расположена **видеопамять** — специализированная оперативная память, в которой хранятся обрабатываемые графическим процессором массивы данных.

На рис. 1.1 в качестве примера показана архитектура графического процессора G80 — одного из процессоров компании NVIDIA [4]. Главной (и общей для всех графических процессоров) характеристикой этой архитектуры является то, что GPU представляет собой систему из параллельных вычислительных устройств, каждое из которых применяет заданную, единую для всех устройств, программу (*вычислительное ядро*, англ. *kernel*) к различным элементам входных массивов данных, расположенных в общей памяти.

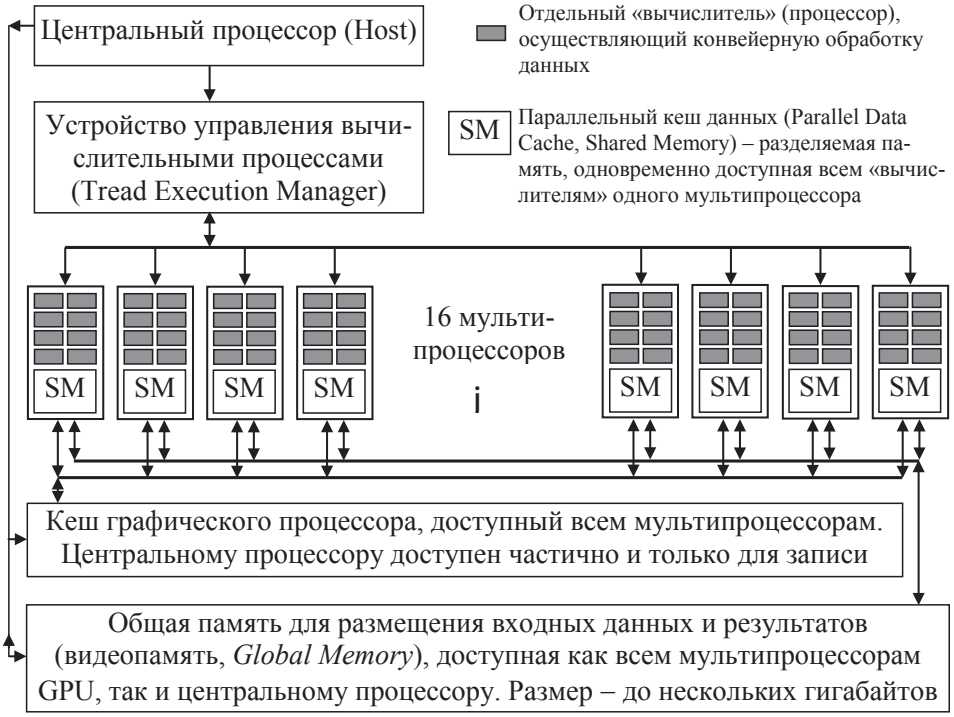


Рис. 1.1. Архитектура графического процессора NVIDIA G80 [4]

Параллельная архитектура графических процессоров ориентирована на исполнение алгоритмов, в которых элементы больших входных массивов обрабатываются одинаковым образом независимо или почти независимо друг от друга, то есть использующих *распараллеливание вычислений по данным*. Множества элементов, подвергаемых однотипной независимой обработке, называют **потоками** (данных либо результатов), так что графические процессоры осуществляют поточно-параллельную обработку данных.

Концепция программирования, заключающаяся в потоковой обработке данных, известна под аббревиатурой *SIMD* (от англ. *Single Instruction — Multiple Data* — одна инструкция для множества данных, рис. 1.2). Процессор, работающий по принципу SIMD, преобразует поток данных в поток результатов, используя программу как функцию преобразования.

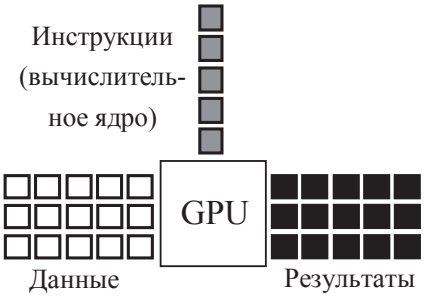


Рис. 1.2. Поточно-параллельная обработка данных SIMD

Выбор концепции SIMD для графических процессоров обусловлен тем, что она обеспечивает параллельное использование большого количества «вычислителей» без явного управления ими: распределения задач, синхронизации вычислений и коммуникации между параллельными расчетами. Разработчикам GPU это позволяет за счет упрощения архитектуры добиваться большей производительности, а при программировании сокращает работу.

С 2001 по 2006 годы графические процессоры включали в себя «вычислители» двух типов: *вершинные* и *пиксельные конвейеры* (или *шейдеры*, см. пояснения ниже) [7–8]. Первые были предназначены для проектирования на плоскость экрана вершин, задающих отображаемые поверхности, а вторые — для расчета цветов пикселей на экране. Так выпущенные в 2006 году процессоры ATI Radeon X1900–1950 имели 8 вершинных и 48 пиксельных конвейеров, которые к тому же были суперскалярными (одновременно обрабатывали по 4 компоненты вектора).

В 2007 году производители GPU перешли от различающихся вершинных и пиксельных конвейеров к унифицированным потоковым процессорам, заменяющим как вершинные, так и пиксельные конвейеры. Графические процессоры 2014–2015 годов выпуска ATI Radeon Fiji XT и NVIDIA GM200–400 включают 4096 и 3072 потоковых процес-

сора (для 32-битовых вычислений «одинарной» точности) при несколько различной внутренней архитектуре. При этом производительность графических процессоров продолжает быстро увеличиваться.

1.2.2. Взаимодействие графического и центрального процессоров

Графический процессор не имеет средств прямого взаимодействия с устройствами ввода-вывода (кроме монитора), а также доступа к оперативной памяти компьютера. Поэтому управление графическим процессором осуществляется только через центральный процессор. Схема взаимодействия центрального и графического процессоров приведена на рис. 1.3.

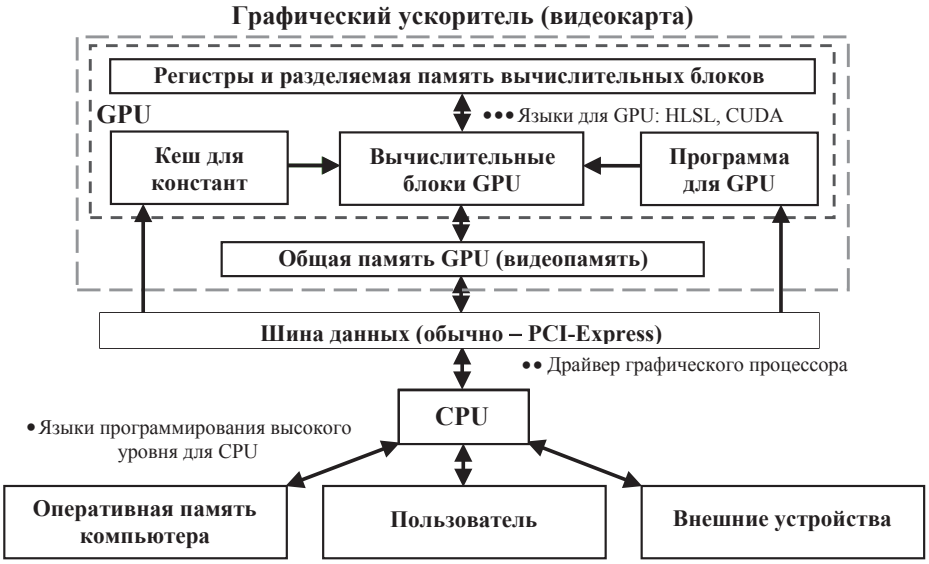


Рис. 1.3. Схема взаимодействия CPU и GPU с памятью и между собой

Графические ускорители подключаются к системной плате персонального компьютера через высокоскоростную шину данных (в настоящее время PCI-Express). Посредством этой шины центральный процессор получает доступ к видеопамяти, а также к некоторым разделам кеш-памяти, расположенной на самом графическом процессоре. Через эту же шину CPU загружает в графический процессор программу и запускает ее.

Перед запуском программы, исполняемой на GPU, центральный процессор передает графическому процессору данные двух видов:

- значения констант, используемых в программе;
- один или несколько больших массивов данных для потоковой обработки.

К константам необходим постоянный и быстрый доступ, поэтому они записываются в кеш-память (или регистры), расположенную на кристалле графического процессора. Массивы данных часто бывают настолько велики, что целиком в кеш-память не помещаются. Однако при простейшей потоковой обработке каждый из элементов массивов данных используется только один раз. Так что для хранения этих массивов предназначена видеопамять (общая память), представляющая собой отдельные микросхемы на плате графического ускорителя. Она работает медленнее кеш-памяти и регистров, зато имеет бóльший объем, до нескольких гигабайтов.

Результаты своей работы графический процессор может сразу записывать в раздел видеопамяти, называемый *буфером кадра*, откуда они передаются на монитор. Но существует также возможность вообще не отображать расчет на экране, а копировать результаты из видеопамяти в оперативную память компьютера, где они становятся доступными для дальнейшей обработки центральным процессором. На этом и основано использование графических процессоров для вычислений общего назначения, не связанных с обработкой графики.

На схеме (рис. 1.3) указаны также типы программ, которые исполняются центральным и графическим процессорами на различных этапах обработки данных. Такие программы будут далее подробно рассмотрены.

1.2.3. Иерархия памяти, доступной центральному и графическому процессорам

Как показано на схемах (рис. 1.1, рис. 1.3), в программах для графических процессоров используется память нескольких разновидностей с различными характеристиками и назначением. Это так, поскольку время, затрачиваемое центральным и графическим процессорами на операции чтения данных из памяти и записи в память, является одним из важнейших факторов, определяющих быстродействие программ для графических процессоров.

Использование памяти различных типов обусловлено необходимостью баланса между объемом памяти и скоростью доступа к данным. Разновидности памяти, имеющие наибольшую емкость, обычно характеризуются бóльшим временем доступа к данным, и наоборот. Быстродействие памяти, в свою очередь, определяется двумя характеристиками — латентностью и пропускной способностью.

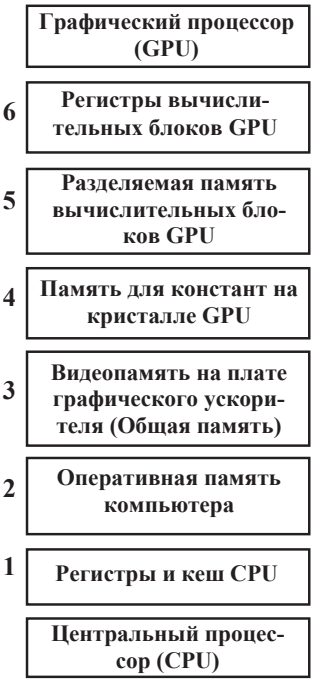
Латентность — это время доступа к памяти, точнее, время ожидания процессором данных после запроса. Латентностью определяется производительность вычислений при решении задач, требующих частого обращения к различным по расположению неупорядоченным ячейкам памяти (*произвольный доступ к памяти*). Такой обмен с памятью характерен для *интерактивных* приложений (программы, интенсивно взаимодействующие с другими приложениями и с пользователями), а также для приложений, управляющих сложными процессами. Подобные алгоритмы обычно исполняются центральным процессором.

Как можно меньшая латентность памяти необходима современным центральным процессорам, работающим на очень высоких частотах, еще и потому, что таким частотам должна соответствовать высокая скорость доступа к данным.

Пропускная способность памяти характеризует объем данных, которые могут быть переданы к процессору или от процессора за единицу времени. Высокая пропускная способность оказывается эффективнее низкой латентности в задачах, позволяющих организовать *последовательный доступ к памяти* — считывание (или запись) данных из ячеек памяти, расположенных друг за другом, непрерывным потоком.

При поточно-параллельной обработке данных предпочтителен именно последовательный доступ к памяти, поэтому видеопамять, предназначенная для обмена данными с графическим процессором, должна обладать максимальной пропускной способностью даже в ущерб латентности.

Иерархия памяти, используемой центральным и графическим процессорами, уже отчасти отражена на рис. 1.1 и рис. 1.3. На рис. 1.4 приведена еще одна схема, иллюстрирующая основные принципы использования памяти различных типов. Особенности и назначение каждого вида памяти кратко пояснены рядом и будут подробнее прокомментированы в примерах.



1. Регистры и кеш CPU – память небольшого (до нескольких мегабайтов) объема, расположенная на кристалле центрального процессора. Характеризуется минимальной латентностью. Доступна только центральному процессору. Обычно этой памятью автоматически управляет процессор, записывая в нее данные, используемые наиболее часто

2. Оперативная память компьютера – память большого (до десятков гигабайтов) объема в форме микросхем, расположенных на системной плате. Имеет среднюю латентность и пропускную способность. Доступна только центральному процессору. Используется для хранения всех обрабатываемых CPU данных, не помещающихся в кеше

3. Видеопамять – память, в которую CPU копирует данные, предназначенные для обработки графическим процессором. Сюда же GPU записывает результаты расчета. Видеопамять имеет большой объем (до нескольких гигабайтов) и максимально возможную пропускную способность для потокового доступа к данным. Расположена на плате графического ускорителя, но не на самом кристалле GPU. Доступна как графическому, так и центральному процессорам для чтения и записи. Латентность со стороны GPU средняя, а со стороны CPU очень большая (медленная передача данных)

4. Память для констант на кристалле GPU. Имеет низкую латентность со стороны GPU и предназначена для быстрого произвольного доступа GPU к константам. Доступна центральному процессору для записи констант, а графическому процессору – для их чтения

5-6. Разделяемая память и регистры вычислительных блоков GPU. Расположены на кристалле графического процессора и доступны только ему. Имеют очень низкую латентность (высокую скорость), но небольшой объем (десятки килобайтов). Предназначены для произвольного доступа к данным со стороны параллельных вычислительных процессов

Рис. 1.4. Использование памяти графическим и центральным процессорами

Отметим основные принципы разделения памяти между центральным и графическим процессорами. В персональном компьютере для хранения большей части данных и программ предназначена так называемая *оперативная память*. Микросхемы этой памяти расположены на системной плате (не на самом процессоре), они доступны центральному процессору для произвольных чтения и записи данных. Графическому процессору эта память не доступна.

На плате графического ускорителя расположена видеопамять (общая память), технологически и по объему подобная оперативной памяти компьютера. Она доступна графическому процессору для чтения и записи в последовательном и в произвольном режиме, но оптимизирована для последовательного (потокового) доступа, поскольку предназначена для хранения массивов данных, обрабатываемых GPU в поточно-параллельном режиме.

Видеопамять является основным средством обмена данными между графическим и центральным процессорами. Центральный процессор

тоже имеет к этой памяти доступ для чтения и записи. Перед началом вычислений на GPU он копирует исходные данные из оперативной памяти в видеопамять, а после окончания расчета копирует результаты обратно в оперативную память. Обмен данными между центральным процессором и видеопамятью происходит сравнительно медленно, так что проводится обычно только в начале и конце расчета.

Поскольку для размещения исходных потоков данных для GPU и результатов его работы видеопамять должна иметь сравнительно большой объем (на современных графических ускорителях до нескольких гигабайтов), она выпускается в виде отдельных микросхем и расположена не на самом кристалле графического процессора, а на плате графического ускорителя. Поэтому работает видеопамять медленно по сравнению с регистрами и разделяемой памятью GPU (ниже).

Для повышения производительности GPU оснащены небольшой быстрой памятью (регистры процессора, память для констант, разделяемая память вычислительных блоков), расположенной прямо на кристалле процессора. Доступ центрального процессора к этой памяти, наоборот, медленен и ограничен (см. следующий подпараграф). Быстрая память используется для хранения данных, к которым графическому процессору нужен произвольный, а не потоковый доступ. Это могут быть, например, константы, задаваемые перед началом расчета.

Улучшение микросхем памяти связано с уменьшением латентности и увеличением пропускной способности. В последние годы пропускная способность памяти увеличивается почти в 2 раза быстрее, чем уменьшается латентность. Быстрое возрастание пропускной способности памяти означает перспективность разработки и реализации алгоритмов, ориентированных на потоковую обработку данных, эффективность которой как раз определяется пропускной способностью памяти.

1.2.4. Конвейерная обработка данных

Уже в первых графических процессорах была реализована конвейерная обработка потоков данных. Заключается она в том, что каждый из параллельных «вычислителей» может одновременно обрабатывать

несколько элементов потока, применяя к ним различные операции, например: загружать из памяти одно число, модифицировать другое и сохранять третье. Необходимо, чтобы операции, исполняемые конвейером одновременно, не зависели друг от друга, что как раз характерно для поточно-параллельных расчетов.

Совершенствование вершинных и пиксельных шейдеров создало возможность программирования GPU для решения неграфических задач. Изначально при этом вместо координат вершин требовалось передавать исходные числовые данные, алгоритм обработки которых программировали в форме вершинного шейдера. В настоящее время графические процессоры позволяют формулировать задачи общего назначения «естественным» для них образом, однако конвейерную обработку данных сохраняют. В качестве иллюстрации принципа конвейерной обработки данных на рис. 1.5 показаны основные этапы *графического конвейера*, используемого графическим процессором визуализации трехмерных сцен [9–12].

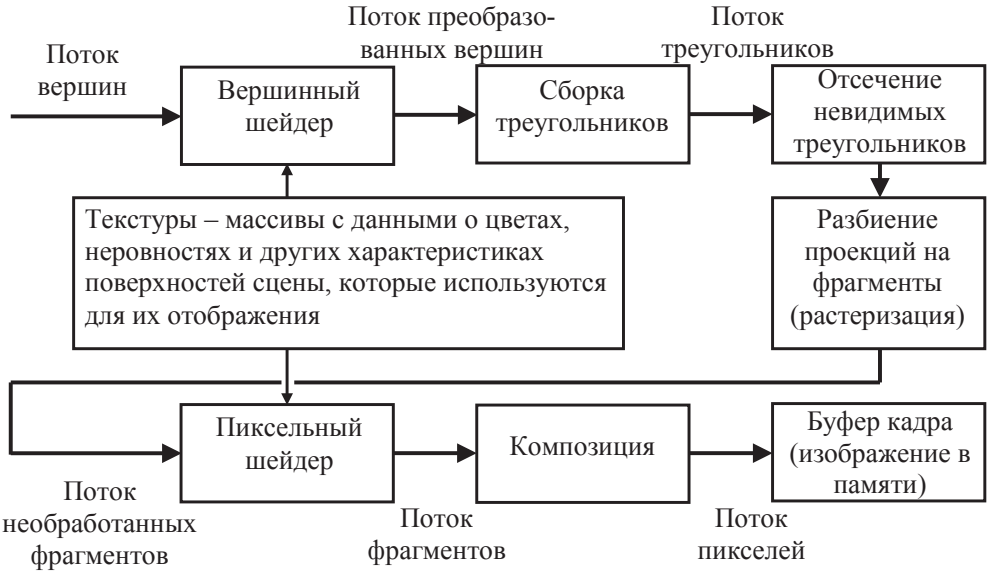


Рис. 1.5. Этапы графического конвейера, исполняемые на графическом процессоре [9]

1.3. Уровни управления графическим процессором и основные системы программирования GPU

1.3.1. Уровни управления графическим процессором

Как уже отмечено выше, графический процессор управляется пользователем не напрямую, а через центральный процессор компьютера. При этом CPU в большинстве случаев исполняет как минимум три взаимодействующие программы, которые различаются по происхождению, назначению и возможностям. Можно сказать, что эти программы работают на трех уровнях управления графическим процессором (см. рис. 1.7). На двух уровнях, «ближайших» к графическому процессору, исполняются стандартные программы, избавляющие программиста от необходимости написания рутинных процедур, таких, например, как выделение видеопамяти для данных или представление этих данных в формате, необходимом графическому процессору.

1.3.2. Драйвер графического процессора

На самом низком уровне, «ближайшем» к графическому процессору, исполняется драйвер графического процессора — программа, которая непосредственно управляет как самим GPU, так и видеопамятью. Фактически драйвер работает как часть операционной системы. Он получает от других приложений запросы — задачи для графического процессора, после чего передает их на GPU в оптимальном порядке и необходимом формате. В частности, функциями драйвера являются такие:

- распределение ресурсов графического процессора между несколькими приложениями;
- загрузка данных в видеопамять и регистры графического процессора, копирование данных из видеопамяти в оперативную память компьютера;
- загрузка в графический процессор и запуск на исполнение пользовательских программ;
- автоматическое распределение расчетов между параллельными вычислительными блоками графического процессора;

- исполнение стандартных алгоритмов обработки графики;
- регулирование частоты графического процессора для поддержания рабочей температуры.

Драйверы обычно разрабатываются самими производителями графических процессоров, поскольку это требует детального знания их технического устройства и особенностей. Они обновляются для поддержки процессоров новых моделей.

Благодаря существованию драйверов, пользовательские программы могут выполняться на различных GPU без перекомпиляции, если GPU и драйвер поддерживают функции, задействованные в программе.

1.3.3. Интерфейсы программирования приложений

К драйверу GPU можно обращаться из пользовательских приложений напрямую, однако это требует явного включения в текст программы большого количества стандартных операций управления графическим процессором. Для того чтобы автоматизировать выполнение подобных операций и унифицировать взаимодействие пользовательских приложений с различными устройствами различных производителей, существуют библиотеки специализированных процедур, известные как *программный интерфейс приложений* (*application programming interface*, или *API*).

В качестве разработчиков API для графических процессоров выступают как сами производители GPU, так и компании, специализирующиеся на программном обеспечении. Под управлением Windows наиболее широко используется пакет API DirectX от Microsoft, а для других операционных систем такие библиотеки разрабатываются в основном по спецификациям OpenCL и OpenGL.

Пакет мультимедийных библиотек DirectX и библиотека DirectCompute

Для операционной системы Windows корпорацией Microsoft уже давно выпускается собственный пакет мультимедийных библиотек API, который в целом называется DirectX. В этот пакет входит библиотека DirectCompute, предназначенная для реализации вычислений общего назначения на GPU (см., например, [13]). Начиная с Windows 98 пакет DirectX поставляется вместе с операционной системой, так что практически входит в ее состав. Интерфейс программирования при-

ложений DirectCompute поддерживается операционными системами Windows, начиная с Windows Vista.

В состав DirectX входит компилятор языка высокого уровня для пользовательского программирования графических процессоров HLSL (*High Level Shading Language*). Этот язык разработан Microsoft специально для DirectX, он совершенствуется вместе с этим пакетом библиотек.

Отметим, что программы, непосредственно предназначенные для выполнения на графических процессорах, традиционно называют *шейдерами*, от англ. *shader*. Это название относится к программам построения теней при отображении трехмерных сцен в компьютерной графике.

OpenGL и OpenCL

OpenGL (Open Graphics Library — открытая графическая библиотека) представляет собой *спецификацию*, определяющую независимый от языка программирования кросс-платформенный программный интерфейс (API) для написания приложений, использующих двумерную и трехмерную компьютерную графику. Спецификация не является конкретной библиотекой, она только описывает набор функций и точное их поведение.

На основе этой спецификации OpenGL разработчики создают библиотеки функций, называемые *реализациями*. Реализации по возможности используют аппаратные средства GPU, а другие требуемые функции эмулируют программно. Чтобы называться реализациями OpenGL, библиотеки должны удовлетворять определенным тестам на соответствие (*conformance tests*). Существует возможность написания на OpenGL *вычислительных шейдеров* (*compute shaders*), реализующих вычисления общего назначения на GPU.

Поддержкой и модернизацией OpenGL с 2006 года занимается индустриальное объединение Khronos Group. В 2008 году этой группой (при значительном участии Nvidia) была выпущена спецификация OpenGL 3.0, которая предоставляет для разработки графических приложений примерно те же возможности, что и DirectX 10.

Консорциумом Khronos Group разработана новая спецификация OpenCL (Open Computing Language — открытый язык для вычислений), включающая в себя интерфейсы программирования приложений и язык программирования для *гетерогенных систем*, имеющих в своем составе центральные и графические процессоры, как и дру-

гие процессоры для высокопроизводительных параллельных вычислений. Существуют многочисленные реализации OpenCL, в том числе от Nvidia, AMD, IBM, Apple и Intel.

Главным преимуществом OpenGL и OpenCL перед DirectX является существование их реализаций не только для Windows, но и для большинства других операционных систем, в том числе Unix-подобных систем (Linux, Mac OS), систем игровых приставок. Такие реализации разрабатываются производителями центральных и графических процессоров, поэтому реализации максимально используют аппаратное ускорение вычислений.

1.3.4. Пользовательское приложение

На «верхнем» уровне управления графическим процессором находятся пользовательские приложения — программы, разрабатываемые для решения конечных задач, таких как создание трехмерных сцен, проведение физико-математических расчетов. Структура пользовательского приложения, использующего API типа DirectCompute или OpenCL, показана на рис. 1.6.

При использовании DirectCompute или OpenCL пользовательское приложение состоит по крайней мере из двух частей, для написания которых используются различные языки программирования. Одна из этих частей — это программа для графического процессора, которую обычно называют *вычислительным ядром*, а другая часть — *центральный процессор*.

Для написания вычислительных ядер используются специальные языки программирования. При работе с DirectX это может быть либо язык высоко уровня HLSL [11], либо низкоуровневый ассемблер (*assembly shader language*). Вычислительное ядро хранится в отдельном текстовом файле, оно компилируется и загружается в память GPU средствами DirectX или OpenCL непосредственно перед запуском. Эти API используются также для загрузки в графический процессор и видеопамять данных и констант, необходимых для расчета.

Часть пользовательского приложения, которая исполняется на центральном процессоре, обеспечивает:

- взаимодействие с пользователем (*пользовательский интерфейс*);

- получение исходных данных от пользователя или из внешних источников, сохранение результатов расчета;
- подготовку и запуск расчетов на графическом процессоре;
- исполнение участков расчетного алгоритма, предназначенных для центрального процессора.

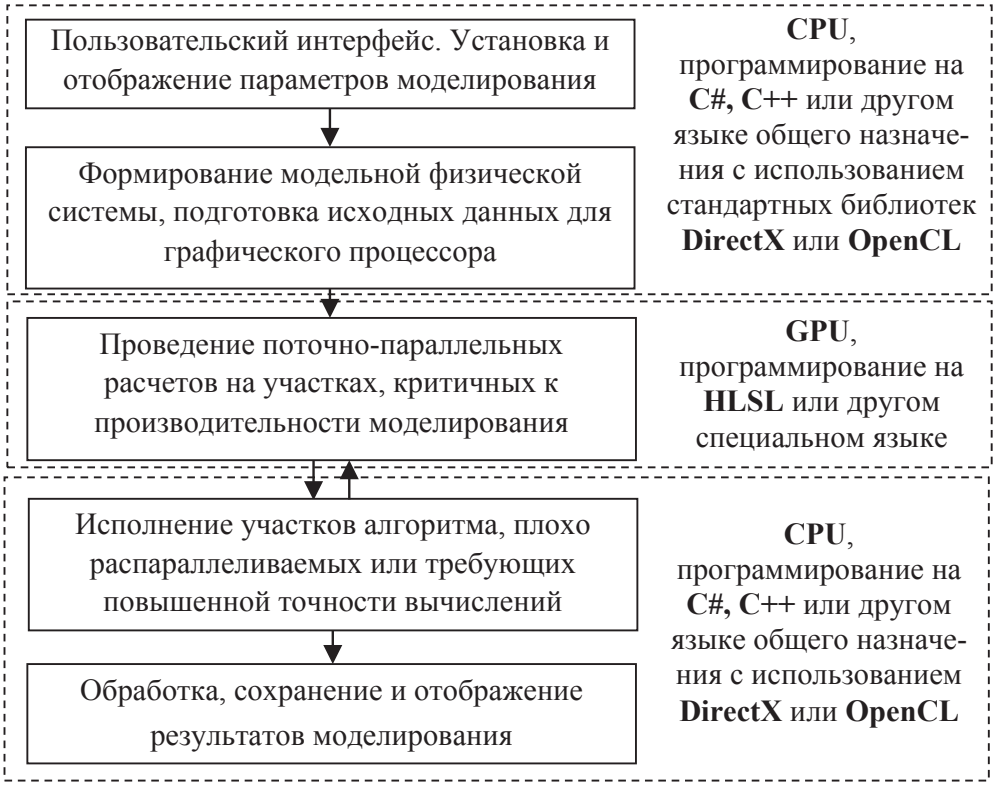


Рис. 1.6. Структура пользовательского приложения, решающего задачу общего назначения (физического моделирования) на графическом процессоре

Программы для центрального процессора обычно разрабатываются на традиционных языках высокого уровня, в частности, на C++ или C#, которые достаточно легко обращаются к процедурам из OpenCL или DirectX.

1.3.5. Программно-аппаратная платформа NVIDIA CUDA

Средства программирования GPU, описываемые выше, применимы для программирования графических процессоров всех производителей, в частности, компаний NVIDIA и AMD. Однако внутренние архитектуры GPU разных производителей и поколений существенно различаются между собой, а возможности использования особенностей каждой из архитектур универсальными системами программирования ограничены. Преимущества конкретных GPU доступны при использовании специализированных инструментов программирования, одним из которых является NVIDIA CUDA.

Платформа для параллельных вычислений NVIDIA CUDA (Compute Unified Device Architecture — архитектура единого вычислительного устройства) фактически представляет собой программно-аппаратный комплекс, в котором физическое устройство графического процессора и его программная модель соответствуют подходам к организации параллельных вычислений, разрабатываемым NVIDIA.

CUDA позволяет программировать только совместимые GPU (производства NVIDIA), однако позволяет управлять ими на более низком уровне, чем DirectX и OpenCL. В частности, преимуществами CUDA являются:

- доступ к программируемой разделяемой памяти;
- произвольная адресация при записи в память;
- значительно ускоренное взаимодействие CPU и GPU, некоторые операции выполняются асинхронно.

Интерфейс программирования приложений реализован в форме расширения языка C++. Ниже CUDA будет рассмотрена более подробно.

Общая схема, иллюстрирующая взаимодействие приложений в составе программного комплекса, для расчетов с использованием GPU приведена на рис. 1.7.

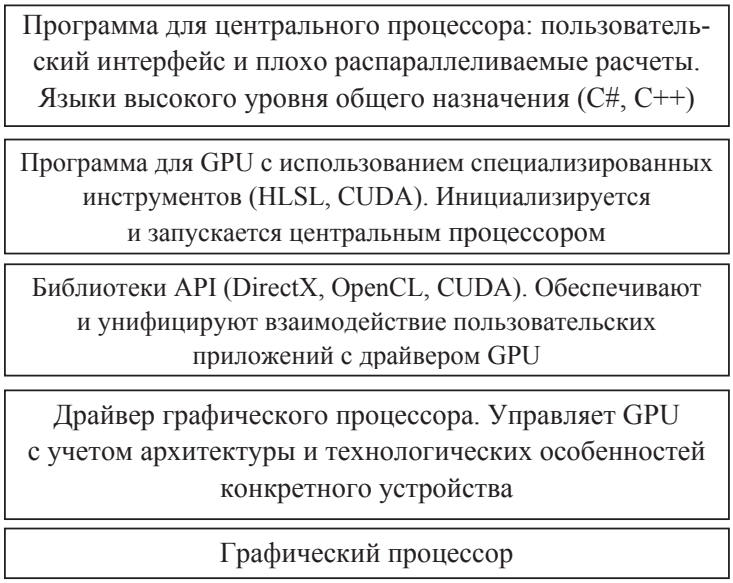


Рис. 1.7. Иерархия приложений, составляющих программный комплекс для расчетов на графических процессорах

1.3.6. Выбор платформы программирования GPU

На сегодня возможными платформами программирования GPU для вычислений общего назначения являются NVIDIA CUDA, Microsoft DirectX и OpenCL. Все платформы достаточно функциональны и динамично развиваются.

Преимуществами CUDA являются лучшая управляемость параллельных вычислений, а также использование обычного языка C++ с дополнительными библиотеками.

Достоинство платформы Microsoft DirectX (включая язык HLSL и вычислительные шейдеры) — совместимость приложений с графическими процессорами всех ведущих производителей (в частности, AMD и NVIDIA), что актуально, поскольку по теоретической производительности GPU AMD и NVIDIA примерно равны.

В ближайшие годы будет актуальной разработка приложений физико-математического моделирования как для DirectX/OpenCL, так и для CUDA.

1.4. Области применения графических процессоров

Современные графические процессоры могут эффективно решать очень многие задачи. Основное требование к таким задачам — возможность распараллеливания по данным или по задачам, а также алгоритм, в котором вычисления превалируют над условными переходами и обменом данными между параллельными процессами. Вычисления на графических процессорах уже сегодня применяются в следующий областях:

- матричные преобразования;
- молекулярная динамика;
- астрофизические расчеты;
- глобальная оптимизация;
- дискретное преобразование Фурье;
- кодирование видео;
- визуализация (томография, нанотехнологии);
- нейронные сети;
- биофизические расчеты.

Наряду с симуляцией реальных процессов, GPU используются и для визуализации (отображение результатов томографии, представление сложных молекул и больших систем).

1.5. Необходимое аппаратное и программное обеспечение

Требования к **архитектуре** и **производительности компьютера**, на который устанавливается графический процессор, состоят в следующем. Системная плата должна быть оборудована шиной данных PCI Express либо AGP (устаревший и не лучший вариант) для подключения графического ускорителя. Шина PCI Express обычна как для настольных компьютеров, так и для ноутбуков.

Наиболее производительные графические ускорители потребляют сравнительно большую мощность, так что вместо блоков питания, обычно поставляемых с корпусами настольных ПК, может потребоваться блок мощностью до 1000 Вт (как показала практика, такой блок необходим, например, для видеокарты с GPU NVIDIA GeForce GTX 780 Ti).

Скорость вычислений на GPU мало зависит от производительности центрального процессора, так что практически любой CPU может быть использован.

Объемы оперативной памяти персонального компьютера (RAM) и видеопамяти графического ускорителя должны быть достаточными для хранения обрабатываемых данных. Для моделирования больших систем может потребоваться несколько гигабайтов памяти этих типов. Связь между производительностью вычислений и объемами памяти практически полностью определяется пользовательским алгоритмом моделирования.

Для расчетов общего назначения требуется **графический ускоритель** с процессором, поддерживающим шейдерную модель 4.0 или старше. Для использования CUDA нужен графический ускоритель NVIDIA с процессором G80 или новее. Драйвер графического процессора должен поддерживать выбранную платформу программирования GPU. Для обеспечения поддержки наиболее новых версий платформы может потребоваться обновление драйвера (даже когда графический ускоритель нормально работает в других приложениях).

Для работы с платформой CUDA достаточно **операционной системы** Microsoft Windows XP Service Pack 2 (SP2). Для использования DirectCompute и OpenCL (реализующих шейдерную модель 4.0) необходимы операционные системы Microsoft Windows Vista и старше. Существуют реализации OpenCL для Unix-подобных операционных систем, включая Linux и MacOS.

Специализированные библиотеки (API) включают в себя следующие пакеты.

Пакет DirectX 10, необходимый для программирования в рамках шейдерной модели 4.0, входит в дистрибутив Windows Vista и последующих версий.

В примерах мы будем использовать процедуры из библиотеки Microsoft XNA Game Studio 2.0 (XNA 2.0), которая является надстройкой над DirectX 9.0c. Эта библиотека была разработана для упрощения работы пользователей с DirectX и для упрощения доступа к DirectX в рамках.Net Framework. Для ее использования требуется установка дистрибутивов XNA 2.0 или 3.0, доступных бесплатно.

В соответствии со стратегией, которую предлагает и осуществляет корпорация Microsoft, большинство приложений для Windows в настоящее время разрабатываются на базе **программной оболочки *Microsoft.***

Net Framework. Эта оболочка включает в себя библиотеку базовых классов (*Base Class Library*), которая содержит реализации стандартных задач, выполняемых приложениями [15], а также компонент *Common Language Runtime* [16], который исполняет пользовательские приложения. Приложения для .Net Framework компилируются не в машинные коды, а в промежуточный байт-код (*Common Intermediate Language, CIL*), который окончательно компилируется для конкретной версии операционной системы и конкретного процессора уже перед самым исполнением с помощью JIT-компилятора (*Jast In Time compiler*), тоже входящего в состав .Net Framework.

Среда .Net Framework необходима для работы с Microsoft XNA Game Studio. Компиляторы, встроенные в среду программирования Microsoft Visual Studio (см. пункт «Средства программирования» ниже), тоже по умолчанию создают именно CIL-код, а не машинные коды.

Оболочка .Net Framework автоматически устанавливается вместе с Windows, при необходимости ее новые версии (доступные бесплатно) можно устанавливать вручную. Для использования XNA 2.0 достаточно версии .Net Framework 2.0, поставляемой, например, вместе с Microsoft Visual Studio 2005.

Для выполнения **программ**, написанных на **CUDA**, требуется установка библиотек, формирующих эту платформу: CUDA Toolkit и CUDA SDK. Дистрибутивы доступны бесплатно (на сайте NVIDIA). Версии CUDA регулярно обновляются.

Средства программирования

Центральный процессор

При использовании любой платформы программирования GPU часть кода исполняется на центральном процессоре. Средой программирования CPU для Windows может быть Microsoft Visual Studio. Эта среда хорошо подходит для написания приложений с вычислениями на графических процессорах, при использовании DirectX/XNA и CUDA. Библиотеки XNA 2.0 и 3.0 автоматически встраиваются в среду Visual Studio при своей установке, как и компилятор CUDA.

Для обращения к XNA можно использовать такие языки программирования, как C++, C# и Visual Basic. Процедуры, написанные на этих языках, могут входить и в состав приложений, исполь-

зующих CUDA. Собственно CUDA представляет собой расширение C++, для компилирования процедур на CUDA существует специальный компилятор.

Вместо полных версий Visual Studio вполне возможно использование Microsoft Visual C# и C++ Express.

DirectX/XNA/HLSL

Обращение к процедурам DirectX можно осуществлять посредством Microsoft XNA. При использовании технологии .Net Framework библиотека XNA стандартным образом подключается к программам на C# или C++. В результате процедуры XNA становятся доступными аналогично процедурам из любых других классов.

Шейдеры на языке HLSL хранятся в отдельных текстовых файлах. Для их написания можно использовать любой текстовый редактор. Существуют специализированные среды программирования для написания шейдеров: ATI Render Monkey (разрабатывается AMD/ATI), FX Composer (разрабатывается NVIDIA), Shader Maker (OpenGL, OpenCL).

NVIDIA CUDA

Программу на CUDA можно редактировать в любой среде, например, в Microsoft Visual Studio. Для компилирования программы проще всего использовать компилятор CUDA (nvcc.exe [17]), который входит в состав CUDA Toolkit. Если при компиляции создать не исполняемый файл (.exe), а динамическую библиотеку (.dll), то она затем может быть подключена к проекту, разрабатываемому на любом из распространенных языков высокого уровня для CPU.

2. Поточно-параллельное программирование GPU

2.1. Распараллеливание расчетов

Пусть имеется вычислительная задача, предполагающая обработку некоторого массива из N однотипных элементов (исходных данных, моделируемых объектов, случайных событий), по одному и тому же алгоритму, включающему m операций. Может требоваться исполнение нескольких таких задач, тогда блок-схема последовательных вычислений, без распараллеливания, будет иметь общий вид, показанный на рис. 2.1. Из этой блок-схемы видно, что возможно три следующих подхода к распараллеливанию расчетов [18].

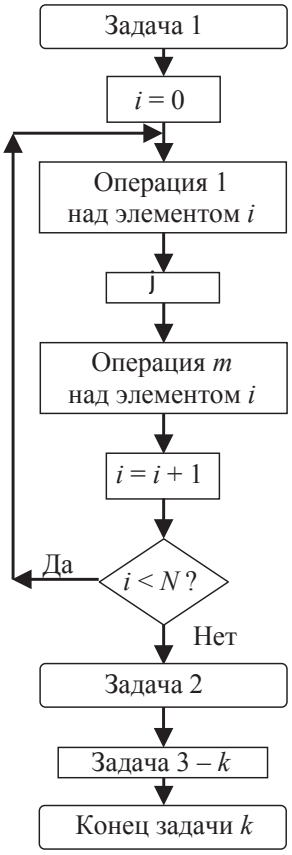


Рис. 2.1. Блок-схема последовательной обработки данных

2.1.1. Распараллеливание по задачам

Распараллеливание по задачам возможно, если задачи (1 — k на рис. 2.1) независимы друг от друга. Такое распараллеливание актуально для сетевых серверов и других вычислительных систем, выполняющих одновременно несколько функций либо обслуживающих многих пользователей. Оно может быть востребовано и в физическом моделировании: например, для исследования одной и той же системы при разных

начальных и внешних условиях. Один графический процессор, совместимый с CUDA, сможет моделировать одну и ту же систему одновременно в нескольких состояниях.

2.1.2. Распараллеливание по инструкциям

Может оказаться, что некоторые инструкции из набора операций (1 — m) (рис. 2.1) независимы друг от друга. При наличии нескольких вычислительных блоков эти инструкции могут быть исполнены параллельно. Схема распараллеливания по инструкциям показана на рис. 2.2. Такое распараллеливание аппаратно реализовано в современных центральных процессорах общего назначения, поскольку оно эффективно при исполнении программ, интенсивно обменивающихся разнородной информацией с другими программами и с пользователем компьютера.

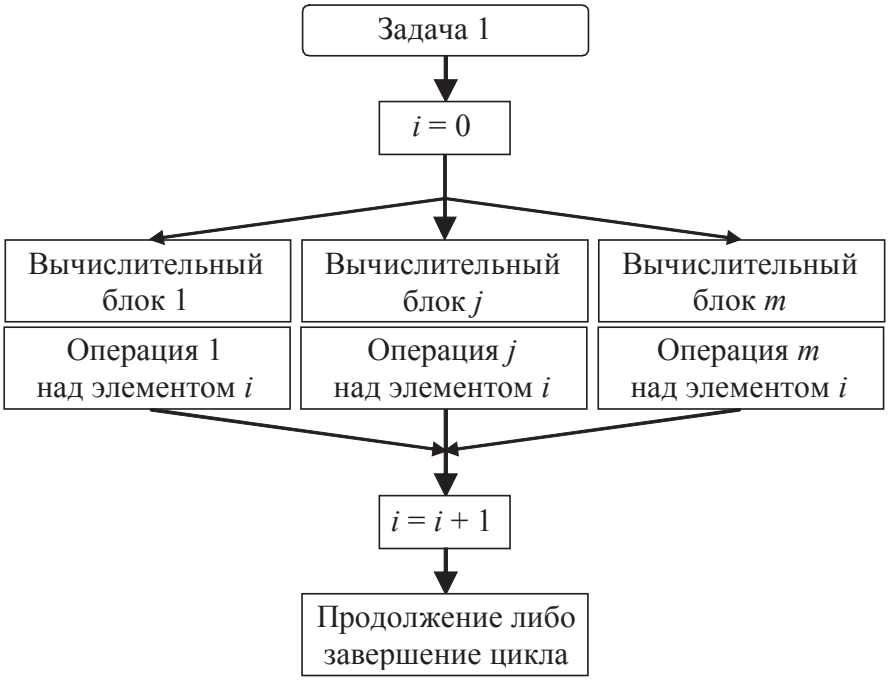


Рис. 2.2. Распараллеливание по инструкциям

2.1.3. Распараллеливание по данным

Если операции (1 — m) (рис. 2.1) над каждым i-м элементом взаимонезависимы, то эти элементы можно обрабатывать параллельно. При этом расчеты распределяются между несколькими вычислительными блоками (процессоры, конвейеры, машины), как это показано на рис. 2.3. Именно возможность распараллеливания по данным особенно характерна для задач физико-математического моделирования.

Процессор 1	:	Операции 1–m над элементами 1, M+1, j , $N - M + 1$
Процессор i	:	Операции 1–m над элементами i, $M + i$, j , $N - M + i$
Процессор M	:	Операции 1–m над элементами M, 2M, j , N

Рис. 2.3. Распараллеливание по данным

При распараллеливании по данным все элементы множества нередко обрабатываются одним и тем же алгоритмом. В таком случае вычисления могут быть организованы по принципу SIMD, который рассмотрен в параграфе 1.2.1 и проиллюстрирован на рис. 1.2. Там же показано, что этот принцип, как раз и заключающийся в независимой потоковой обработке данных по одной программе, идеально соответствует архитектуре графических процессоров, представляющих собой системы из многих «вычислителей», параллельно исполняющих одно вычислительное ядро.

Потоковая обработка данных особенно эффективна для алгоритмов, обладающих следующими свойствами, характерными для задач физического и математического моделирования:

- большая плотность вычислений — велико число арифметических операций, приходящихся на одну операцию ввода-вывода (например, обращение к памяти). Во многих современных приложениях обработки сигналов она достигает 50:1, причем со сложностью алгоритмов увеличивается;
- локальность данных по времени — каждый элемент загружается и обрабатывается за время, малое по отношению к общему времени обработки, после чего он больше не нужен. В результате в памяти потокового процессора для каждого «вычислите-

ля» можно хранить только данные, необходимые для обработки одного элемента, в отличие от центральных процессоров с моделью произвольно зависимых данных.

2.2. Преимущества графических процессоров при параллельных расчетах

Со времени своего появления в начале 1980-х годов, персональные компьютеры развивались в основном как машины для выполнения программ, сложных по внутренней структуре, содержащих большое количество ветвлений, интенсивно взаимодействующих с пользователем, но редко связанных с потоковой обработкой большого количества однотипных данных. Центральные процессоры ПК оптимизировались для решения именно таких задач, поэтому характеризовались следующим:

- большим количеством блоков для управления исполнением программы (кеширование данных, предсказание ветвлений и т. п.) и сравнительно малым количеством блоков для вычислений;
- архитектурой, оптимальной для программ со сложным потоком управления (обработка разнородных команд и данных, организация взаимодействия программ между собой и с пользователем);
- памятью с максимальной скоростью произвольного доступа к данным.

Увеличение производительности CPU в основном было связано с увеличением тактовой частоты и размеров высокоскоростной кеш-памяти (память, расположенная прямо на процессоре). Программирование CPU для ресурсоемких научных вычислений подразумевает тщательное структурирование данных и порядка инструкций для эффективного использования всех уровней кеш-памяти.

Ядра современных центральных процессоров являются суперскалярными, поддерживая векторную обработку (расширения SSE и 3DNow!), сами же CPU обычно содержат несколько ядер. Таким образом, в совокупности центральные процессоры могут реализовывать десятки параллельных вычислительных потоков. Однако графические процессоры включают в себя тысячи параллельных «вычисли-

телей». Кроме того, при поточно-параллельных расчетах графические процессоры имеют преимущество благодаря следующим особенностям архитектуры:

- память GPU оптимизирована на максимальную пропускную способность (а не на скорость произвольного доступа, как у CPU), что ускоряет загрузку потока данных;
- бóльшая часть транзисторов графического процессора предназначена для вычислений, а не для управления исполнением программы;
- при запросах к памяти, за счет конвейерной обработки данных, не происходит приостановки вычислений.

Однако обработка ветвлений (исполнение операций условного перехода) на GPU менее эффективна, поскольку каждый управляющий блок обслуживает не один, а несколько вычислителей.

Таким образом, производительность одного GPU при хорошо распараллеливаемых вычислениях аналогична кластеру из сотен обычных вычислительных машин, причем графические процессоры сейчас поддерживают практически все операции, используемые в алгоритмах общего назначения:

- распространенные математические операции и функции вещественного аргумента. В рамках SM 4.0 поддерживаются целые числа и логические операции, а в SM 4.1 и CUDA — также и вещественные числа двойной (64-битной) точности;
- организацию циклов. В SM 3 длина циклов ограничена 255 итерациями, в SM 4 длина циклов не ограничена;
- операции условного перехода (которые исполняются сравнительно медленно, поскольку в составе GPU блоков управления меньше, чем вычислительных блоков).

Графическому процессору доступна *видеопамять* — специализированная память, обычно расположенная на видеокарте, которая по объему близка к оперативной памяти персонального компьютера. Возможно последовательное и произвольное чтение данных из видеопамяти, а также последовательная запись результатов в видеопамять. На платформе CUDA возможна и произвольная запись в видеопамять. Графический процессор может использовать *регистры* — ячейки памяти, расположенные прямо на процессоре и характеризующиеся очень малой латентностью (быстрый доступ к данным):

- есть чтение из регистров для констант, которые могут хранить постоянные величины, не изменяющиеся в ходе обработки всех данных;
- есть чтение и запись во временные регистры, данные в которых не сохраняются при переходе к следующим элементам потока данных.

Длина программы для GPU в шейдерной модели 3.0 ограничена 65 536 инструкциями, чего для большинства вычислительных алгоритмов достаточно. В SM 4 и на CUDA программы уже могут иметь любую длину.

Ограничением графических процессоров в настоящее время является отсутствие динамического размещения структур данных (например, изменения размеров массивов в ходе выполнения программы). Это ограничение может приводить к неэффективному использованию памяти, однако не препятствует исполнению алгоритмов.

2.3. Принцип программирования SIMD на примере пиксельного шейдера

Поскольку шейдерные модели программирования SM 3 и SM 4 ориентированы на обработку графики, алгоритмы для них удобно рассматривать в форме задачи трехмерной визуализации (например, пиксельного шейдера). При этом исходные данные передаются графическому процессору в форме *текстуры* — двухмерного массива векторов, подобного массиву значений цвета элементов (текселей) изображения. Такую же форму имеют и массивы результатов расчета.

Представить алгоритм общего назначения как пиксельный шейдер можно в соответствии со следующими принципами.

Исходные данные, предназначенные для потоковой обработки, размещены в массивах из четырехмерных вещественных векторов (текстурах).

При работе с цветами текселей три компоненты элементов текстуры содержат интенсивности красного, зеленого и синего цветов, а четвертая компонента — степень прозрачности. Их значения лежат в диапазоне от 0 до 1, однако для исходных данных общего назначения попадание на отрезок [0, 1] необязательно.

Если пользовательской программе все четыре компоненты входных векторов не нужны, то «лишние» компоненты можно оставить нулевыми.

Массивов описанной структуры со входными параметрами может быть несколько.

К каждому из элементов входного массива при потоковой обработке должен применяться один и тот же алгоритм, не зависящий от результатов обработки других элементов (вычислительное ядро на рис. 2.1 образовано операциями (1 — m)). Этот алгоритм должен быть реализован в форме программы — шейдера (на языке HLSL).

Массивы исходных данных (текстуры) передаются графическому процессору через видеопамять, которая доступна шейдеру. Шейдер затем загружает и обрабатывает элементы текстур в поточно-параллельном режиме, как это показано на рис. 2.4.

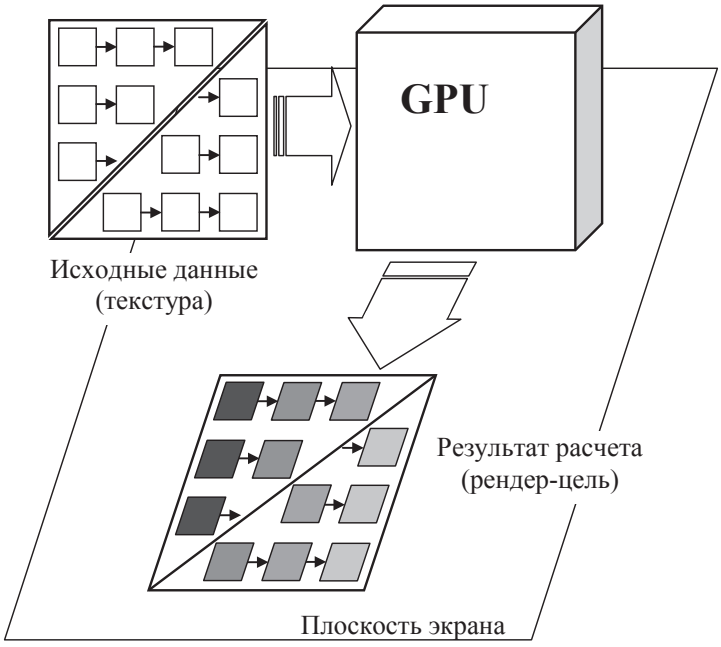


Рис. 2.4. Принцип обработки данных и создания массива результатов в форме закрашивания треугольников на экране

Результаты расчета в форме текстуры из четырехмерных векторов записываются в область видеопамяти, которую в задачах визуализации называют *рендер-целью*. Принцип формирования массива результатов показан на рис. 2.5.

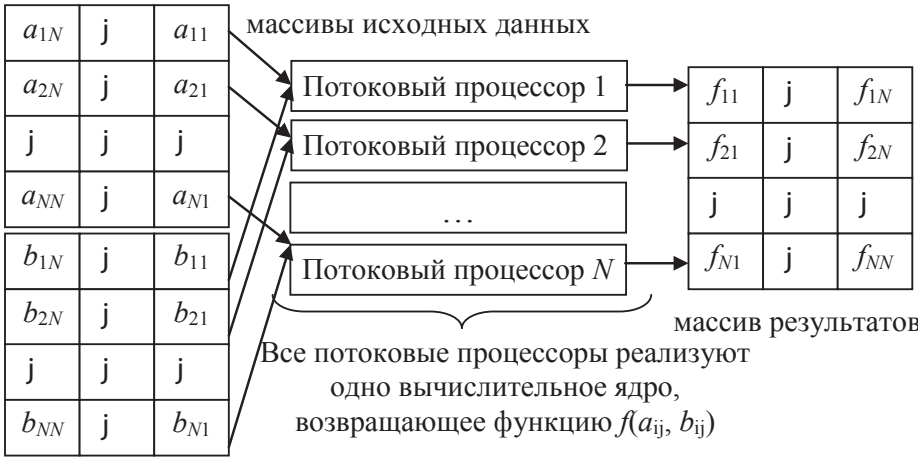


Рис. 2.5. Соответствие между массивами результатов и исходных данных

Центральный процессор компьютера имеет доступ к видеопамяти, в том числе и к рендер-цели. С использованием программы, исполняемой на центральном процессоре, рендер-цель копируют в оперативную память ПК для дальнейшей обработки.

2.4. Пример сложения матриц

2.4.1. Распараллеливание независимых вычислений

Наглядным примером распараллеливания расчетов по данным является задача сложения векторов или матриц. В этом случае однотипные данные — компоненты векторов — складываются независимо друг от друга, а результатом сложения является вектор такой же, как векторы с исходными данными,

$$\begin{pmatrix} c_1 \\ \cdots \\ c_N \end{pmatrix} = \begin{pmatrix} a_1 \\ \cdots \\ a_N \end{pmatrix} + \begin{pmatrix} b_1 \\ \cdots \\ b_N \end{pmatrix}.$$

Центральный процессор ПК решает эту задачу последовательным сложением всех компонент векторов $\vec{a}$ и $\vec{b}$

$$a_i + b_i = c_i,\ i = 1, \ldots, N\ .$$

Для упрощения примера мы не учитываем здесь то, что современные CPU являются суперскалярными, поскольку большого количества параллельных вычислительных процессов они не обеспечивают. Без распараллеливания алгоритм программы для CPU имеет вид цикла (рис. 2.6).

```
for (i = 0; i < N; i++)
    { C[i] = A[i] + B[i]; }:
```

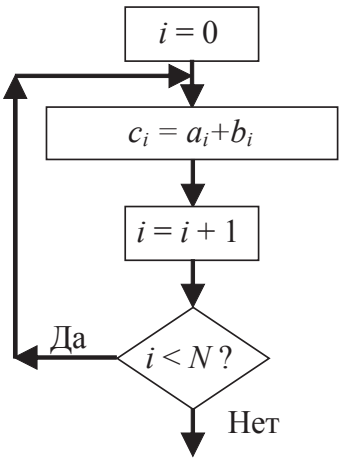


Рис. 2.6. Последовательное сложение векторов

Однако при наличии N процессоров, работающих параллельно, эту же задачу можно было бы решить в N раз быстрее, сложив на каждом из процессоров по одной из компонент векторов:

$$\begin{aligned} &\textit{Процессор}\,1:\ c_1 = a_1 + b_1; \\ &\cdots \\ &\textit{Процессор}\ N:\ c_N = a_N + b_N. \end{aligned} \tag{2.1}$$

При меньшем, чем N, количестве параллельных процессоров $m < N$, сложение векторов тоже возможно выполнить параллельно. Для этого можно разбить векторы на блоки по m чисел (что эквивалентно преобразованию векторов в двухмерные матрицы):

$$\begin{pmatrix} c_1 & c_{m+1} & c_{2m+1} \\ \cdots & \cdots & \cdots \\ c_m & c_{2m} & c_N \end{pmatrix} = \begin{pmatrix} a_1 & a_{m+1} & a_{2m+1} \\ \cdots & \cdots & \cdots \\ a_m & a_{2m} & a_N \end{pmatrix} + \begin{pmatrix} b_1 & b_{m+1} & b_{2m+1} \\ \cdots & \cdots & \cdots \\ b_m & b_{2m} & b_N \end{pmatrix}. \tag{2.2}$$

Представление (2.2) соответствует сложению каждой из строк на отдельном процессоре

$$\begin{gathered} \textit{Процессор}\,1: \\ c_1 = a_1 + b_1,\ c_{m+1} = a_{m+1} + b_{m+1}, c_{2m+1} = a_{2m+1} + b_{2m+1}; \\ \cdots \\ \textit{Процессор}\ m: \\ c_m = a_m + b_m, c_{2m} = a_{2m} + b_{2m},\ c_N = a_N + b_N. \end{gathered} \tag{2.3}$$

Принцип распараллеливания однотипных и независимых вычислений (2.1)–(2.3) хорошо реализован на графических процессорах, которые включают тысячи параллельных конвейеров, специально предназначенных для одновременного проведения одинаковых операций над числами с плавающей точкой.

В примере (2.2)–(2.3) ко всем элементам матриц применяется общая операция сложения — это пример принципа параллельного программирования SIMD. Помимо собственно распараллеливания, данный принцип вычислений позволяет избавиться от операций изменения управляющих переменных цикла (*i* в примере на рис. 2.6), проверки условия завершения цикла и выхода за границы массивов.

Частичное разворачивание циклов (например, обработка в теле цикла сразу четырех элементов) используется и для оптимизации вычислений на центральных процессорах, в частности, позволяет компилятору задействовать расширенные наборы векторных команд типа SSE и 3dNow!. Все же на графических процессорах принцип SIMD реализован в гораздо более полной мере.

2.4.2. Сложение матриц в рамках шейдерной модели 3.0

Матрицы вида (2.4) центральным процессором обычно хранятся и обрабатываются как одномерные последовательности (2.5)

$$\begin{Vmatrix} a_{11} & a_{1j} & a_{1N} \\ a_{i1} & a_{ij} & a_{iN} \\ a_{M1} & a_{Mj} & a_{MN} \end{Vmatrix}, \tag{2.4}$$

$$a_{11}, \ldots, a_{1j}, \ldots, a_{1N}, \ldots, a_{i1}, \ldots, a_{ij}, \ldots, a_{iN}, \ldots, a_{MN}. \tag{2.5}$$

В отличие от CPU, графические процессоры изначально предназначены для параллельной обработки данных, поэтому для них естественно обращаться к памяти, где данные хранятся в форме двумерных массивов и адресуются с помощью двух координат. Именно так хранятся текстуры в видеопамяти графических ускорителей. Данные в видеопамять записывает центральный процессор с помощью драйвера, который копирует их из оперативной памяти компьютера. Представить данные в нужной для копирования форме можно посредством API DirectX и OpenCL.

Шейдерные модели 3.0 и 4.0 для адресации элементов массивов (как номера) используют не целые числа, а пары чисел с плавающей точкой в диапазоне от 0 до 1 (рис. 2.7), потому что необходимо работать с графикой и потому что графические процессоры до последнего времени вообще не оперировали целочисленными типами данных. Шейдерная модель 3.0 предполагала, что адреса ячеек совпадали с координатами их центров (как на рис. 2.7), а в шейдерной модели 4.0 задаются координаты левых нижних углов.

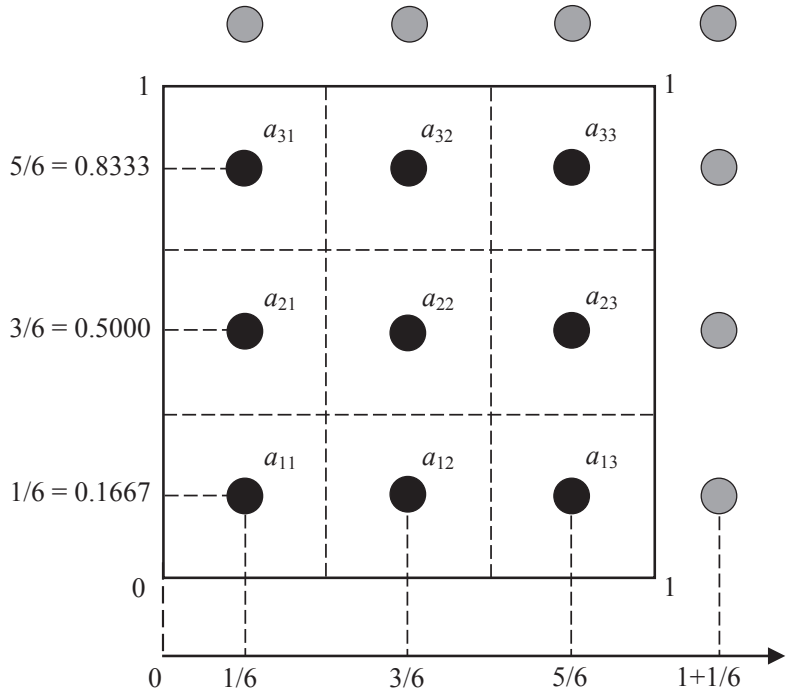


Рис. 2.7. Адресация ячеек видеопамяти при использовании чисел с плавающей точкой (координаты ячеек вдоль обеих осей изменяются от 0 до 1)

Элементы текстур (рис. 2.7) называют **текселями**, а координаты текселей (указывающие на центры ячеек) — **текстурными координатами**. Количество текселей равно количеству элементов во входных массивах. Массивы результатов (рендер-цель) также представляют собой текстуры, количество текселей в которых равно количеству элементов.

Перед началом исполнения шейдера в ячейки видеопамяти (рис. 2.4) должны быть записаны исходные данные (здесь — элементы матриц (2.2)). Затем с помощью драйвера видеокарты центральный процессор запускает расчет. В ходе расчета «вычислители» графического процессора параллельно извлекают эти данные из видеопамяти и применяют ко всем элементам один и тот же набор операций, заданный программой (рассчитывают суммы $c_{ij} = a_{ij} + b_{ij}$).

Результаты своей работы (значения c_{ij}) графический процессор записывает в видеопамять. Фактически в рассматриваемом примере результатом будет матрица скалярных чисел, однако в общем случае это могли бы быть четырехмерные векторы.

2.4.3. Структура программы для центрального процессора

Приложение, работающее с графическим процессором, состоит из нескольких логических блоков, большинство из которых исполняется на центральном процессоре. Структура программы для центрального процессора показана на рис. 2.8.

В настоящем примере программа для центрального процессора написана на языке C#, она приведена ниже. Поскольку подробное рассмотрение синтаксиса C# выходит за рамки настоящего пособия, детально прокомментированы только операции, непосредственно касающиеся графического процессора.

В качестве интерфейса (API), обеспечивающего взаимодействие пользовательского приложения с графическим процессором, мы используем библиотеку Microsoft DirectX с дополнительной библиотекой Microsoft XNA Game Studio Express 2.0, упрощающей программирование.

Подключение необходимых библиотек
Описание переменных, объектов и процедур
Выделение памяти для исходных данных и рендер-цели
Формирование массивов исходных данных
Загрузка и компилирование программы, предназначенной для графического процессора. Отправка нужного кода на GPU
Копирование исходных данных в видеопамять
Исполнение программы на GPU
Копирование результатов из рендер-цели в оперативную память компьютера
Дальнейшая обработка результатов на CPU

Рис. 2.8. Последовательность операций, выполняемых центральным процессором при взаимодействии с GPU

2.4.4. Реализация программы для центрального процессора на C#

Ниже приведен полный текст программы для CPU, организующей взаимодействие с графическим процессором. Программа написана на языке C# из состава Microsoft Visual Studio 2005. Используется библиотека Microsoft XNA Game Studio Express 2.0. Комментарии, вставленные в текст без использования символов “//” или “/*”, не соответствуют синтаксису C#, но если их удалить из текста, то программа будет работать.

Подключение необходимых библиотек

Программой используются стандартная библиотека System, а также библиотека Microsoft.Xna.Framework, которая реализует среду XNA Game Studio и осуществляет взаимодействие с DirectX. Раздел этой библиотеки Microsoft.Xna.Framework.Graphics подключаем отдельно, чтобы сократить в тексте программы описание доступа к ресурсам этого раздела.

```
using System;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
public class GPGPU: Microsoft.Xna.Framework.Game
{
//Описание переменных, используемых программой
/* Задается размер матрицы, количество элементов в строке/столбце */
int size = 3;
```

/* Описываем переменную **graphics** класса **GraphicsDeviceManager** и затем инициализируем эту переменную операцией graphics = new GraphicsDeviceManager (this). Эта переменная будет обеспечивать доступ к процедурам для управления графическим процессором из библиотеки Microsoft.Xna.Framework */

```
GraphicsDeviceManager graphics;
//Стандартные процедуры, реализующие исполнение программы на C#
```

/* Ниже описаны процедуры, которые будут последовательно исполнены сразу после запуска программы. Процедуры **Main** и **Initialize** являются стандартными для проектов на C#. Внутри процедуры Initialize находится вызов процедуры **MatrixSum ()**, которая и осуществляет суммирование матриц на графическом процессоре. */

```
static void Main (string [] args) {using (GPGPU game = new GPGPU ())
game.Run ();}
public GPGPU () {graphics = new GraphicsDeviceManager (this);}
protected override void Initialize ()
{
base.Initialize ();
MatrixSum ();
Exit ();
}
//Процедура, суммирующая матрицы с использованием GPU
private void MatrixSum ()
{
```

/* Выше была введена переменная graphics класса **GraphicsDeviceManager**. В этом классе есть нужный нам подкласс **GraphicsDevice**. Создаем переменную **gpu** этого подкласса */

```
GraphicsDevice gpu = graphics.GraphicsDevice;
```

//Выделение памяти для рендер-цели

/* Созданная в начале программы переменная **size = 3** задает размеры матриц (3×3), которые мы будем складывать. Теперь создается рендер-цель — массив 3×3, в который будет выводиться результат расчетов. Соответствующая переменная называется **tex_target**. Она является не простым массивом 3×3, а принадлежит к классу **RenderTarget2D** из библиотеки **Microsoft.Xna.Framework**. Кроме самого массива в объектах этого класса хранятся некоторые дополнительные параметры, в частности, указатель на переменную **gpu**, который задает конкретную видеокарту, для которой создается рендер-цель. */

RenderTarget2D tex_target = new RenderTarget2D (gpu, size, size, 1, SurfaceFormat.Single);

//Выделение памяти для текстур с исходными данными

/* Создаются текстуры — массивы размерами size*size, — в которых будут храниться исходные матрицы. Как и рендер-цель, соответствующие переменные не являются простыми массивами, а принадлежат к более сложному классу **Texture2D**, в объектах которого, кроме самих массивов, хранятся дополнительные параметры. Как и выше, при создании новой переменной класса, значения этих параметров устанавливаются строкой new Texture2D (gpu, size, size, 1, ResourceUsage. Dynamic, SurfaceFormat.Single, ResourceManagementMode.Manual), значениями внутри скобок. */

Texture2D tex_matrix1 = new Texture2D (gpu, size, size, 1, ResourceUsage. Dynamic, SurfaceFormat.Single, ResourceManagementMode.Manual);

Texture2D tex_matrix2 = new Texture2D (gpu, size, size, 1, ResourceUsage. Dynamic, SurfaceFormat.Single, ResourceManagementMode.Manual);

//Заполнение массивов исходных данных конкретными значениями

/* Теперь задаем конкретные значения элементов исходных матриц. Сначала эти значения записываются в одномерные массивы matrix1 и matrix2 размерами size * size. Элементы a_{ij} и b_{ij} после преобразования матриц к одномерным массивам будут иметь номера j * size + i. Затем мы копируем созданные массивы в текстуры класса **Texture2D tex_matrix1** и **tex_matrix2** с помощью операции tex_matrix1. SetData<float> (matrix1) и tex_matrix2.SetData<float> (matrix2). */

```
float [] matrix1 = new float [size * size], matrix2 = new float [size * size];
for (int j = 0; j < size; j++)
for (int i = 0; i < size; i++) {
```

```
matrix1 [j * size + i] = j * size + i;
matrix2 [j * size + i] = i * size + j;
}
tex_matrix1.SetData<float> (matrix1);
tex_matrix2.SetData<float> (matrix2);
```

/*Следующая задача — установление соответствия между целочисленными индексами элементов матриц 3×3 и текстурными координатами этих же элементов внутри квадрата (-1;-1)- (1;1), или quad'a, которыми будет оперировать графический процессор, а также внутри квадрата (0;0)-(1;1), в который отображается массив результатов (рендер-цель). Для этого достаточно задать соответствие между индексами «крайних» элементов матриц и координатами вершин квадрата. Соответствие это показано ниже. При необходимости рассмотрения матриц других размеров, знаменатель 6 нужно заменить на значение 2*size.

Соответствие между координатами вершин quad'a (-1;-1)-(1;1) и координатами элементов складываемых матриц:

(−1; 1) ..... (1/6; 1/6)
(−1; −1)..... (1/6; 1+1/6)
(1; 1)..... (1+1/6; 1/6)
(1; −1) ..... (1+1/6; 1+1/6)

Метод определения координат элементов проиллюстрирован также на рис. 2.4. Видно, что приведенные формулы действительно обеспечивают попадание элементов массива 3×3 в необходимые области. Учтен также тот факт, что ось *Y* на экране (и в рендер-цели) направлена сверху вниз.

Третья пространственная координата во входных данных у нас не используется, поэтому в конструкторах вида **Vector3 (-1, 1, 0)** третья координата равна нулю, тогда как первые два значения задают координаты вершин quad'a */

```
float dx = 0.5f/size, dy = 0.5f/size;//смещения для адресации центров текселей
VertexPositionTexture [] v = new VertexPositionTexture [] {
new VertexPositionTexture (new Vector3 (−1, 1, 0), new Vector2 (0 + dx, 0 + dy)),
new VertexPositionTexture (new Vector3 (−1, —1, 0), new Vector2 (0 + dx, 1 + dy)),
```

new VertexPositionTexture (new Vector3 (1, 1, 0), new Vector2 (1 + dx, 0 + dy)),

new VertexPositionTexture (new Vector3 (1, —1, 0), new Vector2 (1 + dx, 1 + dy))

}

/* Следующей стандартной строкой выделяется область памяти для хранения координат вешин quad'a. Этих вершин 4, чем и обусловлен размер 4 * VertexPositionTexture.SizeInBytes */

VertexBuffer quad = new VertexBuffer (gpu, 4 * VertexPositionTexture. SizeInBytes, ResourceUsage.None, ResourceManagementMode.Automatic);

quad.SetData<VertexPositionTexture> (v);

VertexDeclaration quadVertexDeclaration = new VertexDeclaration (gpu, VertexPositionTexture.VertexElements);

//Компилирование эффектов, содержащих шейдеры

/* В библиотеку Microsoft.Xna.Framework встроен компилятор текстовых файлов, содержащих шейдеры для графического процессора, таких как **sum.fx** (см. начало примера и прил. 1). Следующая строка компилирует файл **sum.fx** и записывает полученную программу (в машинных кодах) в область оперативной памяти, на которую указывает переменная **e** типа **CompiledEffect**. */

CompiledEffect e = Effect.CompileEffectFromFile ("sum.fx", null, null, CompilerOptions.None, TargetPlatform.Windows);

/* Проверка успешности компиляции */

if (! e.Success) throw new Exception (e.ErrorsAndWarnings);

//Выбор шейдера, который будет исполняться

/* Создается новая переменная, описывающая эффект, который будет исполняться на графическом процессоре. В качестве кода эффекта выбирается откомпилированная программа, на которую указывает переменная e. */

Effect fx = new Effect (gpu, e.GetEffectCode (), CompilerOptions.None, null);

/* В качестве запускаемой на видеокарте процедуры выбирается техника sum из файла sum.fx (описана выше) */

fx.CurrentTechnique = fx.Techniques ["sum"];

/* Копирование исходных данных в видеопамять, доступную графическому процессору. В качестве исходных данных передаем эффекту *fx* текстуры с исходными данными tex_matrix1 и tex_matrix2, которые мы уже создали в начале программы: */

```
fx.Parameters ["tex_matrix1"].SetValue (tex_matrix1);
fx.Parameters ["tex_matrix2"].SetValue (tex_matrix2);
/* Задаем выходные параметры: quad и рендер-цель: */
gpu.Vertices [0].SetSource (quad, 0, VertexPositionTexture.SizeInBytes);
gpu.VertexDeclaration = quadVertexDeclaration;
gpu.SetRenderTarget (0, tex_target);
//Исполнение шейдера на графическом процессоре
fx.Begin ();
fx.CurrentTechnique.Passes ["sum"].Begin ();
gpu.DrawPrimitives (PrimitiveType.TriangleStrip, 0, 2);
fx.CurrentTechnique.Passes ["sum"].End ();
fx.End ();
/* Получение результата из рендер-цели и сохранение его в текстовом файле
Копирование массива результатов из видеопамяти в оперативную память, доступную центральному процессору */
gpu.ResolveRenderTarget (0);
/* Создаем текстовый файл для записи результата */
using (System.IO. StreamWriter w = new System.IO. StreamWriter ("results.txt"))
{
/* Создаем одномерный массив, в который будут скопированы данные из рендер-цели, для дальнейшей обработки центральным процессором */
float [] matrix3 = new float [size * size];
/* Копирование данных из рендер-цели в массив matrix3. Аргумент <float> означает, что данные будут скопированы в виде скалярных чисел одинарной точности */
tex_target.GetTexture ().GetData<float> (matrix3);
/* Запись данных из массива matrix3 в текстовый файл */
for (int j = 0; j < size; j++)
{for (int i = 0; i < size; i++) w.Write ("\t{0}", matrix3 [j * size + i]);
w.WriteLine ();}}}}
```

Операции, вошедшие в рассмотренный пример, составляют только небольшую часть всех возможностей библиотеки XNA, большинство функций которой предназначены для отображения графики в играх. Продемонстрированных операций достаточно для того, чтобы инициализировать физико-математические расчеты на GPU и получить результаты из рендер-цели.

2.4.5. Программа для графического процессора

Всю совокупность операций, которые мы хотим применить к данным из входных массивов графического процессора в рамках одной программы, называют **вычислительным ядром**. Это название отражает тот факт, что вычислительное ядро не включает в себя вспомогательные операции, такие как операции управления циклами (поскольку вместо них осуществляется параллельная поточная обработка всех входных данных) или операции по подготовке данных (которые необходимы, но исполняются центральным процессором вне GPU).

Программирование графического процессора заключается в реализации вычислительного ядра в виде шейдера. Для написания шейдеров, работающих с DirectX, используется язык высокого уровня HLSL, компилятор которого входит в эту библиотеку. Программы на HLSL создаются и загружаются в виде файлов, называемых *эффектами*. Эффект содержит одну или несколько *техник* (процедур), которые в свою очередь могут состоять из одного или нескольких проходов, каждый из которых включает в себя вызов одного из шейдеров. Структура простейшего эффекта показана на рис. 2.9.

Эффект — это та часть приложения, которая исполняется непосредственно на графическом процессоре. Рассмотрим подробно написание конкретного эффекта, реализующего сложение двух матриц. Математическая задача имеет вид (2.2):

$$\begin{pmatrix} c_{11} & c_{1j} & c_{1N} \\ c_{i1} & c_{ij} & c_{iN} \\ c_{N1} & c_{Nj} & c_{NN} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{1j} & a_{1N} \\ a_{i1} & a_{ij} & a_{iN} \\ a_{N1} & a_{Nj} & a_{NN} \end{pmatrix} + \begin{pmatrix} b_{11} & b_{1j} & b_{1N} \\ b_{i1} & b_{ij} & b_{iN} \\ b_{N1} & b_{Nj} & b_{NN} \end{pmatrix},$$

вычислительным ядром здесь является операция

$$c_{ij} = a_{ij} + b_{ij}.$$

Ниже прокомментирован текст программы, складывающей матрицы на HLSL. Отметим, что синтаксис языка HLSL близок к синтаксису языка Си.

Блок описания входных данных

Первая строка программы описывает объекты tex_matrix1 и tex_matrix2 типа texture (текстура). Эти объекты представляют собой массивы входных данных, то есть содержат значения a_{ij} и b_{ij}

texture tex_matrix1, tex_matrix2;

Описание входных потоков (текстур)
Описание «избирателей» (*samplers*) — автоматически исполняемых процедур, которые передают входные данные из видеопамяти на графические конвейеры. Свой «избиратель» сопоставляется каждому из входных потоков
Текст вершинного шейдера — пользовательской подпрограммы, проектирующей «каркас» трехмерной сцены на плоскость экрана. В нашем случае просто копирует квадрат (*quad*), показанный на рис. 2.7, сам на себя
Текст пиксельного шейдера — пользовательской подпрограммы, которая обычно на основании исходных текстур рассчитывает цвета пикселей на экране, а в нашем примере будет складывать элементы двух матриц
Текст техники (*technique*) — процедуры, которая «видна» из программ, внешних по отношению к графическому процессору (в частности, из программы на C#, рассмотренной выше), и которая будет вызываться для исполнения вершинного и пиксельного шейдеров
В программе может быть несколько вершинных, пиксельных шейдеров и техник. Но каждая техника обязательно запускает только один вершинный и один пиксельный шейдер
В результате исполнения техники в видеопамяти будет сформирован массив результатов. Управление передается центральному процессору

Рис. 2.9. Структура программы для графического процессора на HLSL

Описание избирателей

Для работы с текстурами в программе необходимо описать (и затем запустить в работу) **избиратели** (*samplers*) — процедуры, которые будут передавать входные данные из видеопамяти на графические конвейеры для обработки. Сами эти процедуры встроены в графический процессор, но их описание необходимо, поскольку каждой из входных текстур сопоставляется свой избиратель, для которого требуется задать название и значения параметров, определяющих способ адресации данных в текстуре, метод интерполяции значений и т. п. Мы оставляем настройки избирателей по умолчанию: кусочно-постоянная интерполяция (*Point*) и периодическая адресация при выходе за границы (*Wrap*). В нашей программе строки

sampler matrix1 = sampler_state {Texture = <tex_matrix1>;};

sampler matrix2 = sampler_state {Texture = <tex_matrix2>;};

описывают избиратели matrix1 и matrix2 для текстур tex_matrix1 и tex_matrix2 соответственно. Операция sampler_state как раз и сопоставляет конкретному избирателю конкретную текстуру.

Вершинный шейдер

Процедур, реализующих пиксельные и вершинные шейдеры, в тексте эффекта (программы для GPU) может быть несколько, но в каждый момент исполнения программы активными могут быть только один вершинный и один пиксельный шейдер. В принципе шейдеры имеют ту же структуру, что и процедуры на языке Си:

void transform (in float4 pos_in: POSITION, in float2 tex_in: TEXCOORD0, out float4 pos_out: POSITION, out float2 tex_out: TEXCOORD0)

{tex_out = tex_in;

pos_out = pos_in;}

Приведенный текст описывает процедуру transform, которую мы будем использовать в качестве вершинного шейдера. На способ использования указывает тип void, означающий, что процедура transform не возвращает значения. Пиксельный шейдер возвращал бы цвет пикселя на экране.

В списке аргументов функции служебные слова «in» и «out» указывают на то, что следующий аргумент относится к входным данным либо к результатам. После двоеточия заданы типы использования аргументов (*usage types*). Тип POSITION означает, что массивы pos_in и pos_out используются для хранения координат (то есть позиций) вершин. Тип TEXCOORD0 относится к массивам tex_in и tex_out, которые содержат текстурные координаты. В данном случае это координаты текселей, которые соответствуют углам (то есть вершинам) квадрата на рис. 2.7.

Вершинный шейдер производит преобразование координат из трехмерного пространства вершин в пространство рендер-цели (прямоугольник на плоскости, рис. 2.7). Мы не задействуем вершинный шейдер в расчете, поэтому сводим преобразование вершин к простому копированию. Для этого координаты вершин сразу задаются в плоскости прямоугольника рендер-цели.

Входные параметры процедуры transform имеют следующие типы:

float4 — четырехкомпонентный вектор 32-битной точности, задающий координаты вершины в трехмерном пространстве сцены;

float2 — двухкомпонентный вектор 32-битной точности, задающий координаты текселя (в нашем случае представляющего собой ячейку массива исходных данных) на плоскости экрана (то есть внутри единичного квадрата, показанного на рис. 2.7).

В частности, параметрами процедуры transform являются:

pos_in — массив с координатами вершин, которые образуют исходную сцену в трехмерном пространстве;

tex_in — массив с координатами центров текселей на поверхностях треугольников, образующих трехмерную сцену;

pos_out — массив с координатами проекций вершин, представляющих сцену на плоскости экрана;

tex_out — массив с координатами центров текселей на экране.

Пиксельный шейдер

Пиксельный шейдер представляет собой функцию, которая в поточно-параллельном режиме исполняется для всех элементов входных текстур (массивов исходных данных). В обычной графической задаче элемент текстуры задает цвет текселя — участка изображения на экране, координаты которого (изменяющиеся от 0 до 1 вдоль осей x и y, рис. 2.7) совпадают с двухмерным вещественным индексом этого элемента во входном массиве. Такие координаты задаются двухмерными векторами uv типа float2, которые предаются пиксельному шейдеру в качестве параметра.

Если размер входной матрицы вдоль оси Ox — Nx, а вдоль оси Oy — Ny, то координаты элементов будут иметь значения $uv = ((u+0.5)/Nx; (v+0.5)/Ny)$ (SM 3.0) либо $uv = (u/Nx; v/Ny)$ (SM 4.0), где u и v — целые числа, принадлежащие интервалам от 0 до (Nx — 1) и от 0 до (Ny — 1). Пиксельный шейдер исполняется независимо для всех возможных пар u и v. При этом он может работать с элементами сразу нескольких текстур (входных потоков), но для обеспечения поточно-параллельного доступа они должны иметь одинаковые индексы u и v (см. схему на рис. 2.5).

Цвета текселей в текстурах задаются четырехмерными вещественными векторами — величинами типа float4 (32-битная точность, интенсивности красного, зеленого и синего цветов, а также степень прозрачности). Результат работы пиксельного шейдера в графической задаче — цвет текселя с координатами uv, полученный в результате некоторого преобразования исходного цвета либо наложения нескольких исходных текстур. Поэтому пиксельный шейдер создается как функция (в нашем примере sum), возвращающая (записывающая в рендер-цель по адресу uv) число типа float4.

После двоеточия указаны типы использования переменных. Аргумент *uv* имеет тип использования TEXCOORD0, поскольку представляет собой двухмерные координаты текселя (рис. 2.7). Сама функция характеризуется типом использования COLOR (цвет), поскольку в графических задачах возвращает цвет текселя. Тип служит также указанием на то, что функция sum — это пиксельный шейдер.

float4 sum (float2 uv: TEXCOORD0): COLOR
{return tex2D (matrix1, uv) + tex2D (matrix2, uv);}

Тело шейдера в принципе может содержать любые стандартные операторы языка C, а также все распространенные математические функции. Обязательным является оператор return, который вычисляет значение, возвращаемое функцией. Здесь оператор return возвращает сумму векторов a_{ij} и b_{ij}, расположенных в текстурах tex_matrix1 и tex_matrix2.

Выбор нужных значений $a_{ij} = a_{uv}$ и $b_{ij} = b_{uv}$ автоматически осуществляют «избиратели» matrix1 и matrix2, уже связанные с текстурами tex_matrix1 и tex_matrix2 выше. Параметр функции *uv* типа float2 представляет собой 2-вектор с текстурными координатами (TEXCOORD0) чисел (a_{uv}, b_{uv}) во входных текстурах tex_matrix1 и tex_matrix2. Функции tex2D (matrix1, uv) и tex2D (matrix2, uv) обращаются к избирателям matrix1 и matrix2, которые и берут из текстур tex_matrix1 и tex_matrix2 элементы, расположенные по адресу *uv*.

Техника, запускаемая центральным процессором

В тексте эффекта может быть описано несколько вершинных и несколько пиксельных шейдеров. Однако центральный процессор не может обращаться к ним непосредственно, поскольку эти шейдеры компилируются как локальные процедуры, «невидимые» из-за пределов эффекта. Для запуска на исполнение вершинный и пиксельный шейдеры должны быть объединены в технику (*technique*). **Техники** — это глобальные процедуры, которые входят в текст эффекта и исполняются графическим процессором, но запускаются на исполнение центральным процессором (посредством Direct3D и драйвера GPU).

Текст эффекта может включать в себя несколько техник, но при каждом обращении к GPU исполняется только одна из них. Каждая техника в свою очередь обязательно включает в себя вызов одного (только одного) вершинного и одного (только одного) пиксельного шейдеров из числа описанных в эффекте. Кроме исполнения шейде-

ров техника может устанавливать значения системных переменных (в примере ниже — строки ZEnable = false и CullMode = none)

В нашем примере нужна только одна техника (sum):

```
/* Декларация для внешних вызовов и параметры обработки */
technique sum {pass sum {
ZEnable = false;
CullMode = none;
PixelShader = compile ps_3_0 sum ();
VertexShader = compile vs_3_0 transform ();}}
```

Здесь слова «pass sum» означают, что данная процедура будет запускаться внешними по отношению к откомпилированному эффекту программами (см. ниже) под именем sum.

Из примера видно, что в рамках техники можно задать значения некоторых параметров, определяющих режим работы графического процессора. В частности, строка ZEnable = false означает, что будет отключен Z-буфер, хранящий информацию о том, насколько каждая из вершин трехмерной сцены удалена от плоскости экрана, для отсечения невидимых объектов. Строка CullMode = none отключает режим отсечения невидимых элементов изображения, который здесь не нужен, поскольку в нашем расчете нет разделения объектов на видимые и невидимые. Эти настройки используются во всех наших программах физического моделирования.

Наконец, последние две строки запускают рассмотренные выше процедуры sum () и transform () в качестве пиксельного и вершинного шейдеров.

Компилирование и выполнение эффекта

Текст эффекта должен храниться на каком-либо носителе информации (например, на жестком диске компьютера) в виде обычного текстового файла того же формата, что файлы с расширением.txt. Традиционно файлы с эффектами имеют особое расширение.fx. Перед исполнением эффекта этот файл загружается и компилируется средствами DirectX/XNA, после чего полученный код передается графическому процессору. В нашем примере файл с эффектом назван **sum.fx**. Его текст полностью прокомментирован выше.

2.4.6. Вычислительные шейдеры модели 5.0

В очередной (2009 год) шейдерной модели 5.0 и соответствующей версии DirectX 11 появилась возможность написания *вычислительных шейдеров*, отличающихся следующими особенностями:

- произвольной записью в память;
- наличием разделяемой памяти (в виде разделяемых регистров);
- обработкой параллельной записи по одному адресу;
- адресацией памяти оператором [] (аналогично массивам на CPU);
- поддержкой двойной точности для вычислений (64-битные целые и вещественные числа);
- отсутствием необходимости применения вершинных шейдеров («рисования квада», рис. 2.7);
- задаваемой программно иерархией потоков обработки (сетка, связка, поток) с 1D, 2D или 3D индексной адресацией (потоков в связке, связки в сетке).

Использование перечисленных нововведений характерно для программирования на CUDA, рассматриваемого ниже.

3. Программирование графических процессоров на CUDA

3.1. Модель программирования графических процессоров как универсальных вычислительных систем

3.1.1. Взаимодействие параллельных вычислительных процессов

Как показано в предыдущих главах, шейдерные модели 3.0 и 4.0 хорошо реализуют поточно-параллельную обработку массивов данных без обмена данными между отдельными «вычислителями». Параллельные потоковые процессоры (аналогично графическому конвейеру) применяют ко всем векторам из входных массивов один и тот же алгоритм обработки полностью независимо друг от друга. Для наглядности этот принцип проиллюстрирован на рис. 3.1.

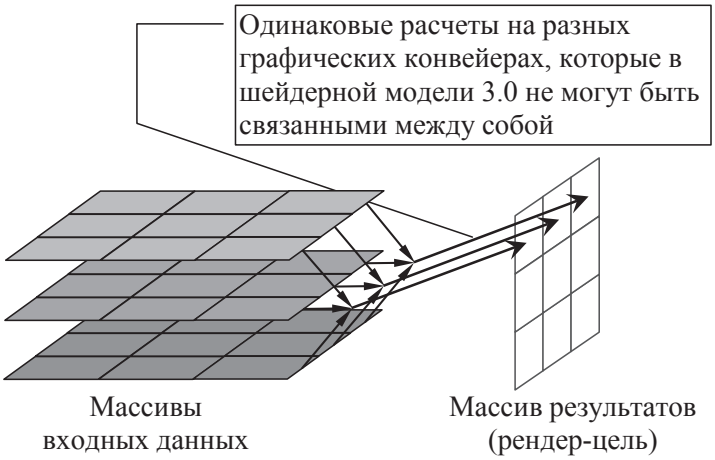


Рис. 3.1. Поточно-параллельный расчет без связи между процедурами обработки отдельных элементов входных массивов

На рис. 3.1 каждой тройке одинаковых по индексам 4-векторов из входных массивов (таких массивов не обязательно именно три) ставится в соответствие одна ячейка массива результатов (рендер-цели). При этом связи между графическими конвейерами, обрабатывающими различные тройки входных векторов, организовать нельзя.

Алгоритмы, в которых нет связи между расчетами на параллельных графических конвейерах, имеют то преимущество, что не требуют синхронизации работы конвейеров, которая замедляет вычисления, а также может осложнить программирование.

Однако существуют такие алгоритмы, в которых обмен данными между параллельными вычислительными потоками неизбежен, но не отнимает много времени либо даже обеспечивает ускорение по сравнению с «чистым» поточно-параллельным вариантом.

В рамках шейдерных моделей 3 и 4 алгоритмы с обменом данными можно было осуществлять либо в несколько проходов (с копированием рендер-цели во входные массивы и повторным запуском GPU), либо переносом части вычислений на центральный процессор. Оба варианта сильно понижали производительность вычислений.

3.1.2. Концепция универсального вычислительного устройства CUDA

Выпускаемые с 2007 года графические процессоры очередного поколения (NVIDIA GeForce G80, GTX 280 и новее, AMD/ATI Radeon HD) характеризуются многими нововведениями, важными для неграфического программирования, среди которых:

- замена пиксельных и вершинных конвейеров (имевших уже практически одинаковые возможности) универсальными потоковыми процессорами;
- поддержка целочисленных типов данных и побитовые операции с целыми числами;
- работа с вещественными числами двойной точности.

Указанные нововведения значительно увеличили спектр алгоритмов, реализуемых на графических процессорах. Дополнительные возможности появились в рамках концепции универсального вычислительного устройства компании NVIDIA, известной как CUDA. На аппаратном уровне эта концепция выразилась в следующем:

- появилась возможность идентификации в ходе выполнения вычислительного ядра конкретных вычислительных процессов (англ. термин *treads*), исполняемых данным потоковым процессором в данный момент;
- стало реализуемым варьирование того, какие именно элементы потоков данных загружаются и обрабатываются конкретными потоковыми процессорами и конкретными вычислительными процессами;
- первые две возможности, задействованные вместе, дают некоторую свободу ветвления алгоритмов, исполняемых различными вычислительными процессами;
- вместо автоматической записи результатов расчета в рендерцель появилась возможность произвольной записи в видеопамять, что обеспечивает гибкое формирование массивов результатов;
- потоковые процессоры, входящие в один и тот же вычислительный блок, получили доступ к общей разделяемой кеш-памяти, что позволяет вычислительным процессам обмениваться результатами работы на промежуточных этапах алгоритма (для оптимизации вычислительных схем и обмена данными между видеопамятью и кешем).

Вычислительные блоки с общей разделяемой памятью названы компанией NVIDIA потоковыми мультипроцессорами (*Stream Multiprocessors*). На рис. 3.2 [6, 19] показана структура такого мультипроцессора в составе графического процессора GTX 200. Каждый мультипроцессор состоит из 8 скалярных процессоров (*Scalar Processors*), каждому из которых доступны для чтения и записи, наряду с разделяемой памятью, еще и собственные регистры.

В мультипроцессоре есть еще два типа кеш-памяти, доступной всем его скалярным процессорам, но только для чтения, — это память для констант (*Constant Cash*) и память для текстур (*Texture Cash*). Данные в память таких разновидностей загружает только центральный процессор. Первая используется для хранения констант, одинаковых во всех вычислительных потоках, а вторая — для хранения текстур в графических приложениях. Всем мультипроцессорам целиком доступна общая память (видеопамять, *Device Memory* на рис. 3.2) большого объема (порядка 1 Gb), расположенная на плате графического ускорения в форме отдельных микросхем.

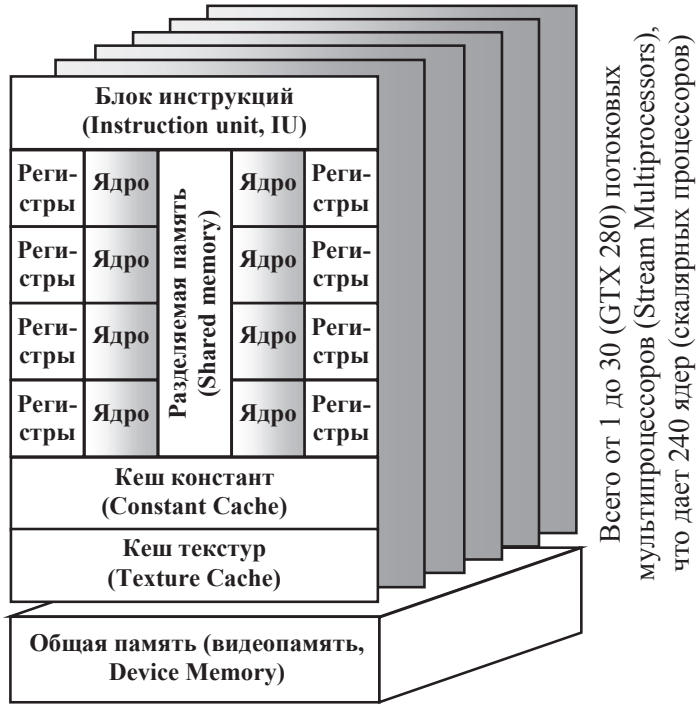


Рис. 3.2. Потоковые мультипроцессоры (Stream Multiprocesors) графических процессоров серии NVIDIA GTX 200

Мультипроцессоры с разделяемой памятью (впервые появившиеся в процессоре NVIDIA G80, рис. 1.1) явились новыми вычислительными единицами, модель программирования которых вышла за рамки шейдерных моделей 3 и 4 в силу того, что потоковые процессоры в составе одного мультипроцессора могут совместно использовать разделяемую кеш-память.

3.1.3. Иерархия вычислительных процессов и памяти CUDA

Как и GPU предыдущих поколений, графические процессоры архитектуры CUDA представляют собой системы из параллельных «вычислителей» — *потоковых процессоров* (они же названы ядрами и *скалярными процессорами* (рис. 3.2), каждый из них применяет заданное вычислительное ядро к некоторым элементам входных массивов данных.

По иерархическому положению потоковые процессоры эквивалентны отдельным графическим конвейерам GPU предыдущих поколений, но могут быть идентифицированы и по-разному управляемы в ходе исполнения алгоритма.

Потоковые процессоры универсальны, они не подразделяются больше вершинные и пиксельные конвейеры. Вместо этого потоковые процессоры физически и логически объединены в вычислительные блоки — мультипроцессоры, внутри которых они имеют общий доступ к разделяемой кеш-памяти (рис. 3.2), через которую могут обмениваться данными в ходе параллельных вычислений.

Все мультипроцессоры исполняют одно и то же вычислительное ядро, но могут использовать различные исходные данные, что в принципе позволяет реализовать параллелизм не только по данным, но и по задачам.

Всем мультипроцессорам доступна общая память (*Device Memory*, она же *видеопамять*, рис. 3.2), в которой центральный процессор размещает исходные данные, а графический процессор — результаты расчетов, которые становятся доступными центральному процессору.

Общая память представляет собой отдельные микросхемы на плате графического ускорителя, таким образом, расположена вне самого графического процессора. Поэтому она работает сравнительно медленно, зато имеет большой объем (порядка гигабайт), который обычно достаточен для хранения всех нужных данных.

Вычислительные процессы (*threads*) в принципе могут в ходе вычислений обмениваться результатами через видеопамять, однако это сравнительно медленный вариант.

Кеш-память для хранения констант (*Constant Memory* на рис. 3.2) — «быстрая» память, доступная одновременно всем мультипроцессорам (и следовательно, всем вычислительным процессам), но только для чтения. Эта память имеет небольшой размер (64 KB), расположена прямо на кристалле графического процессора и работает со скоростью регистров (намного быстрее, чем видеопамять). Используется для хранения констант, необходимых при выполнении программы. Константы загружаются центральным процессором перед началом параллельного расчета.

Каждому из мультипроцессоров доступен *параллельный кеш данных* (или *разделяемая память Parallel Data Cache* на рис. 1.1, *Shared Memory* на рис. 3.2 (16 KB на мультипроцессор)). Размещена

эта память на кристалле GPU и работает со скоростью регистров процессора. Предназначена для того, чтобы вычислительные процессы могли модифицировать общие данные и обмениваться информацией в ходе параллельного расчета.

Каждый блок разделяемой памяти доступен одновременно всем «вычислителям» в составе одного мультипроцессора (см. рис. 3.2) для чтения и для записи. Конструкцией GPU предусмотрена автоматическая синхронизация доступа «вычислителей» к разделяемой памяти. «Вычислители», принадлежащие к разным мультипроцессорам, общей кеш-памяти не имеют.

3.1.4. Возможности и ограничения процессоров архитектуры CUDA

Начиная с 2007 года компанией NVIDIA предложен целый ряд графических процессоров и вычислительных систем на основе архитектуры CUDA. При этом их вычислительные возможности постепенно увеличивались, а версия CUDA 1.0 сменилась версией 7.0 (2015 год). NVIDIA характеризует внутреннюю архитектуру своих процессоров вычислительной способностью (*Compute Capability* [6]), современная версия которой имеет номер 5.0. Некоторые параметры этой версии приведены ниже:

количество вычислительных потоков (*threads*) в одной логической «связке» (*block*) 1024
размерность сетки потоков в логической «связке» 3
размерность сетки логических «связок» 3
максимальное количество 32-битных регистров, доступных одному потоку 255
максимальное количество 32-битных регистров, доступных одной логической «связке» ..... 2^{16}
объем разделяемой памяти, доступный одной логической «связке», байт ..... $48 \cdot 2^{10}$
максимальный объем параметров, передаваемых графическому процессору при запуске вычислительного ядра, байт 256
максимальное количество инструкций в вычислительном ядре ..... 2^{29}
поддержка целочисленных типов и операций 64-битной (двойной) точности ..... есть
поддержка вещественных чисел и операций 64-битной (двойной) точности ..... есть

Полная спецификация имеется в руководстве [6].

Конкретные модели GPU, поддерживающие CUDA, могут различаться вычислительной способностью, количеством мультипроцессоров, шириной шины данных, частотами памяти и ядра.

3.1.5. Конвейерная обработка данных в архитектуре CUDA

Потоковые процессоры имеют конвейерную архитектуру, то есть могут одновременно исполнять несколько вычислительных процессов, находящихся на разных стадиях алгоритма (например, когда один процесс записывает данные в глобальную видеопамять, другой может вести вычисления). Каждый мультипроцессор в составе GPU одновременно исполняет до 2048 вычислительных процессов в одной или двух «связках», где «связка» — логическое объединение вычислительных процессов с общей разделяемой памятью в программах на CUDA.

Применение «связок» (англ. *blocks*) вычислительных потоков обусловлено тем, что в программах на CUDA должны быть логически разделены потоки, исполняемые на различных мультипроцессорах, поскольку только потоки, исполняемые на одном и том же мультипроцессоре, имеют общий доступ к разделяемой памяти.

Каждая «связка» исполняется на одном мультипроцессоре, а один процессор может исполнять две связки, что дополнительно сокращает время его простаивания при конвейерных вычислениях. В «связке» может быть до 1024 потоков. Оптимальное количество «связок» и вычислительных потоков должно быть таким, чтобы максимально задействовать ресурсы мультипроцессора (регистры и разделяемую память), но так, чтобы данные всех потоков размещались в регистрах и памяти одновременно. Предельно допустимое количество вычислительных потоков на один мультипроцессор в двух «связках» не более 2048.

3.2. Особенности программирования на CUDA

3.2.1. Идентификация вычислительного потока

CUDA дает программисту возможность управлять распределением обрабатываемых данных и задач по «связкам» и по конкретным вычислительным процессам. Для этого существуют специальные регистры,

которые внутри каждого конкретного вычислительного процесса содержат индексы именно этого процесса, а также индексы той «связки», к которой он принадлежит.

Программист имеет возможность задавать способ индексации связок и вычислительных процессов. Эти индексы могут быть одно-, двух- или трехмерными координатами внутри логической сетки, в которую связки и процессы могут быть организованы по требованиям алгоритма (см. пример перемножения матриц).

Переменные, которые возвращают значения индексов, имеют тип dim3, соответствующий в CUDA 3-компонентным вещественным векторам одинарной точности. Имена и способ использования этих переменных приведены на рис. 3.3, 3.4 и проиллюстрированы в примере перемножения матриц. Подробно принципы и способ индексации «связок» и вычислительных потоков рассмотрены в руководстве [6].

«Связки» вдоль оси y: **gridDim.y = 3**

blockIdx.x = 0 **blockIdx.y = 2**	**blockIdx.x = 1** **blockIdx.y = 2**	**blockIdx.x = 2** **blockIdx.y = 2**
blockIdx.x = 0 **blockIdx.y = 1**	**blockIdx.x = 1** **blockIdx.y = 1**	**blockIdx.x = 2** **blockIdx.y = 1**
blockIdx.x = 0 **blockIdx.y = 0**	**blockIdx.x = 1** **blockIdx.y = 0**	**blockIdx.x = 2** **blockIdx.y = 0**

Количество «связок» вдоль оси x:
gridDim.x = 3

Рис. 3.3. Индексация «связок» потоков, двухмерная сетка

«Связки» вдоль оси y: **blockDim.y = 3**

threadIdx.x = 0 **threadIdx.y = 2**	**threadIdx.x = 1** **threadIdx.y = 2**	**threadIdx.x = 2** **threadIdx.y = 2**
threadIdx.x = 0 **threadIdx.y = 1**	**threadIdx.x = 1** **threadIdx.y = 1**	**threadIdx.x = 2** **threadIdx.y = 1**
threadIdx.x = 0 **threadIdx.y = 0**	**threadIdx.x = 1** **threadIdx.y = 0**	**threadIdx.x = 2** **threadIdx.y = 0**

Количество «связок» вдоль оси x:
blockDim.x = 3

Рис. 3.4. Индексация вычислительных потоков внутри связки, двухмерная сетка

3.2.2. Совместимость с шейдерными моделями

Графические процессоры архитектуры CUDA могут исполнять программы, написанные в рамках шейдерных моделей 3–5 без изменения алгоритмов, поскольку возможность поточно-параллельной обработки данных сохранена, а изменение внутренней архитектуры процессоров учтено в новых драйверах. Такими средствами программирования, как язык HLSL, библиотеки DirectX, XNA и OpenGL, по-прежнему можно пользоваться для написания новых программ. В отличие от написанных на CUDA, такие программы будут исполняться не только на GPU от NVIDIA, но и на последних GPU других производителей (в частности, AMD/ATI).

3.2.3. Язык программирования CUDA

Система программирования CUDA основана на расширении языка C путем добавления в него новых типов данных и процедур для управления графическим процессором. CUDA включает в себя возможности языков, предназначенных для написания вычислительного ядра (HLSL), и графических библиотек (DirectX, XNA, OpenCL), обеспечивающих взаимодействие между центральным и графическим процессорами, в том числе такие:

- операции выделения памяти GPU для внесения данных, операции записи данных в память и удаления из памяти (выполняются центральным процессором);
- операторы описания типов переменных, используемых в программе;
- функции для математических вычислений на GPU;
- средства для запуска параллельных расчетов на исполнение.

Практически программирование на CUDA сводится к использованию новых операций, процедур и типов данных в «обычных» программах на языках C, C++ и C#. Для этого требуются специальные библиотеки CUDA, разработанные компанией NVIDIA, распространяемые свободно [20].

3.2.4. Структура программы на CUDA

Благодаря универсальности архитектуры CUDA, исполняемые этой платформой алгоритмы могут иметь различные, достаточно сложные структуры. Простейшая схема, отражающая оптимальную последовательность загрузки вычислительными потоками данных из памяти, состоит в следующем.

1. CPU загружает необходимые данные в видеопамять, регистры графического процессора и в память для констант.

2. Центральный процессор запускает вычислительное ядро, исполняемое на GPU. Размеры сеток, в которые будут организованы «связки» и вычислительные потоки, указывает программист.

3. В начале исполнения вычислительного ядра каждый поток определяет свой локальный и (или) глобальный номер, в соответствии с которым будет обрабатывать свою часть исходных данных, предписанную программистом. Номеру могут соответствовать константы и условия, задающие ветвления алгоритма.

4. Вычислительные потоки в параллельном режиме копируют данные из видеопамяти в разделяемую память и в регистры, чтобы затем эти данные быстрее обрабатывались.

5. Если данные, копируемые в разделяемую память, будут затем совместно использоваться различными вычислительными потоками, то требуется синхронизация (оператор __syncthreads ()). GPU ждет, пока все потоки не закончат копирование данных.

6. В параллельном режиме производятся расчеты, предусмотренные алгоритмом, запись результатов в разделяемую память или сразу в видеопамять, если результаты окончательные.

7. Бывает необходимо, чтобы все вычислительные потоки закончили текущий этап расчета до перехода любого из них к следующему этапу (например, к загрузке новых данных в разделяемую память). Для этого производят синхронизацию вычислительных потоков оператором __syncthreads ().

8. Далее происходит переход к следующему этапу алгоритма либо завершение работы вычислительного ядра.

Распараллеливание этапов, приведенных выше, демонстрирует схема, показанная на рис. 3.5. Она иллюстрирует один из возможных методов распараллеливания вычислений на CUDA, а также использование разделяемой памяти мультипроцессоров для ускорения обме-

на данными между вычислительными потоками и сравнительно медленной видеопамятью.

Как и при использовании шейдерных моделей, задача центрального процессора сводится к тому, чтобы скопировать в видеопамять исходные данные, а после выполнения расчета получить обратно результат.

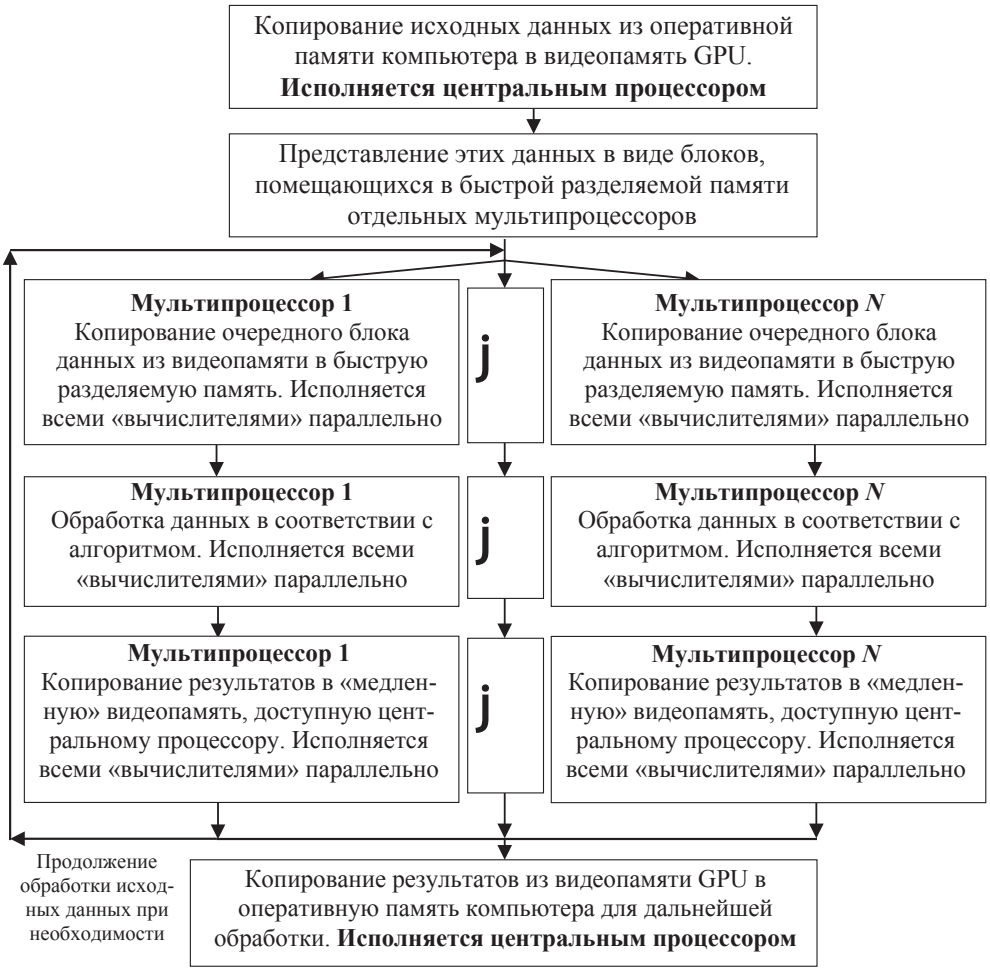


Рис. 3.5. Примерная схема программы для процессора архитектуры CUDA

Вычислительные ядра, исполняемые графическим процессором, в тексте на CUDA имеют вид обычных функций языка C. По сравнению с функциями, предназначенными для CPU, они имеют следующие ограничения:

- могут обращаться только к памяти GPU, но не к оперативной памяти компьютера;
- не возвращают никакого значения (то есть являются процедурами, а не функциями своих аргументов в математическом смысле этого слова; на C это означает, что тип возвращаемого значения только void);
- допускают только фиксированное количество аргументов;
- не допускают рекурсивного вызова (обращения функции к самой себе);
- не могут включать статических переменных (переменных, сохраняющих свои значения после завершения функции).

Аргументы этих функций автоматически копируются из оперативной памяти компьютера в общую память GPU (видеопамять). Копирование осуществляется центральным процессором.

При написании процедур (функций) на CUDA необходимо указывать, должны они исполняться графическим или центральным процессором. Для этого используются следующие спецификаторы:

- **__global__** — функция, вызываемая центральным процессором, но исполняемая на графическом процессоре, то есть вычислительное ядро;
- **__device__** — функция, исполняемая на GPU и вызываемая только другими функциями, исполняемыми на GPU, то есть из вычислительного ядра (не может вызываться центральным процессором);
- **__host__** — функция, вызываемая и исполняемая центральным процессором, без использования GPU;
- **__host__ and __device__** — функция, которая может исполняться либо на центральном, либо на графическом процессорах (применяется, например, при «перегрузке» операторов).

Проект с кодом CUDA редактируется и компилируется с использованием Microsoft Visual Studio аналогично «обычному» проекту. При сборке проекта возможен выбор конфигураций: Release, Debug, EmuRelease, EmuDebug. Две последние конфигурации эмулируют GPU на центральном процессоре. Конфигурации типа Debug компилируют проект в отладочном режиме.

3.3. Анализ алгоритма параллельного перемножения матриц

3.3.1. Алгоритм перемножения матриц

Одним из главных преимуществ CUDA является возможность программируемого управления отдельными блоками «вычислителей», что во многих задачах позволяет оптимизировать использование ресурсов графического процессора и минимизировать время, затрачиваемое на взаимодействие с памятью. Хорошим примером этого является реализация алгоритма перемножения матриц.

Математически задача перемножения двух матриц A и B формулируется следующим образом:

$$\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix} \times \begin{pmatrix} b_{11} & \cdots & b_{1k} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{nk} \end{pmatrix} = \begin{pmatrix} c_{11} & \cdots & c_{1k} \\ \vdots & \ddots & \vdots \\ c_{m1} & \cdots & c_{mk} \end{pmatrix} \Rightarrow c_k^i = \sum_{j=1}^{n} a_j^i b_k^j .$$

Для того чтобы умножение было возможным, необходимо, чтобы ширина матрицы A (равная n) совпадала с высотой матрицы B (тоже равной n). При этом получится, что у результирующей матрицы C высота совпадает с высотой матрицы A, а ширина — с шириной матрицы B. Этот принцип очень наглядно иллюстрируется умножением матрицы на вектор:

$$\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix} \times \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix} = \begin{pmatrix} c_1 \\ \vdots \\ c_m \end{pmatrix} .$$

Алгоритм параллельного перемножения матриц может быть следующим:

- исходные матрицы A и B разбиваются на блоки с тем, чтобы каждый из мультипроцессоров вычислял произведение одного из блоков матрицы A на один из блоков матрицы B. Пусть для простоты эти блоки будут кубическими;
- размер блоков (*Block_Size*) выбирается таким образом, чтобы два перемножаемых блока целиком помещались в разделяемую память мультипроцессора (*Shared memory* на рис. 3.2);

- в ходе исполнения программы на каждом мультипроцессоре исполняются одна или две «связки» потоков, а каждый поток исполняется на одном конкретном «вычислителе». Потокам внутри «связки» нужно поставить в соответствие двухмерные номера — значения индексов *i* и *k* в диапазоне от 1 до *Block_Size*;
- каждый вычислительный поток рассчитывает сумму

$$c_{Sub}[i,k] = \sum_{j=1}^{Block\_Size} a_j^i b_k^j$$

при фиксированных индексах (*i*, *k*), как это показано на рис. 3.6.

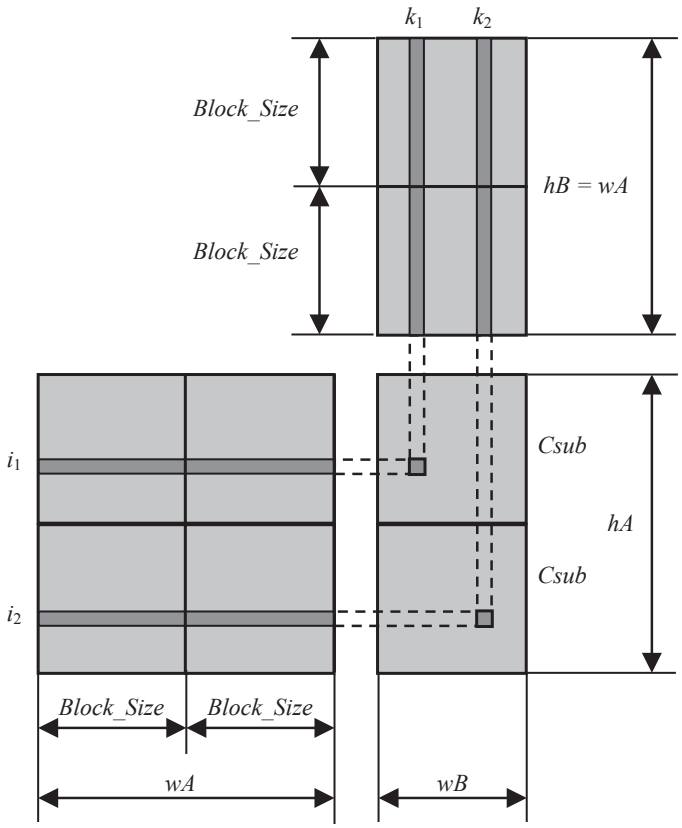


Рис. 3.6. Схема перемножения матриц на GPU архитектуры CUDA [6]

На рис. 3.6 каждая матрица *Csub* равна произведению двух прямоугольных блоков: блока матрицы *A* размерами (*wA*, *Block_Size*), индексы строк которого совпадают с индексами строк матрицы *Csub*, и бло-

ка матрицы *B* размерами (*Block_Size*, *wA*), индексы столбцов которого совпадают с индексами столбцов матрицы *Csub*. Окончательно матрица *Csub* вычисляется как сумма произведений квадратных блоков, показанных на рис. 3.6,

$$c_k^i = \sum_{j=1}^{Block\_Size} a_j^i b_k^j + \sum_{j=Block\_Size+1}^{2\cdot Block\_Size} a_j^i b_k^j + \ldots + \sum_{j=(n-1)\cdot Block\_Size+1}^{wA} a_j^i b_k^j ,$$

где *n* — количество блоков, приходящееся на ширину матрицы *A* и равную ей высоту матрицы *B*, $n = wA/Block\_Size$.

Для расчета каждого произведения сначала в разделяемую память загружается два соответствующих блока из глобальной памяти, а затем каждый поток «связки» вычисляет один элемент произведения. При этом происходит накопление суммы результатов $a_j^i b_k^j$, которая затем сохраняется в глобальной памяти.

Рассмотренный алгоритм перемножения матриц на CUDA иллюстрирует принципы управления вычислительными потоками и оптимизации использования памяти при программировании GPU на CUDA. В частности, разбиение матриц на блоки позволило задействовать быструю разделяемую память, сокращая число выборок из глобальной памяти до $n = wA/Block\_Size$ раз.

3.3.2. Процедура перемножения матриц на CUDA

Рассмотрим программу, реализующую перемножение матриц на CUDA (см. приложение). Эта программа может быть откомпилирована любыми средствами, поддерживающими библиотеки CUDA. Детали реализации алгоритма даны в комментариях.

```
//Файл matrixMul_kernel.cu
#ifndef _MATRIXMUL_KERNEL_H_
#define _MATRIXMUL_KERNEL_H_
#include <stdio.h>
#include «matrixMul.h»
```

/* Следующая процедура имеет модификатор **__global__**, то есть является вычислительным ядром. Это ядро исполняется одновременно на каждом мультипроцессоре и каждым потоком. Но все-таки конкретные вычислительные процессы, исполняемые в различных «связках» и потоках, различаются значениями таких системных переменных, как

blockIdx и **threadIdx**. Это позволяет нам «привязать» конкретные ячейки матриц к конкретным потокам.

Параметрами процедуры являются указатели (float*) на области оперативной памяти компьютера, в которых расположены матрицы *A*, *B* и *C*, а также целочисленные (int) значения *wA* и *wB*, задающие ширину входных матриц в единицах размера блока *Block_Size* */

__global__ void

matrixMul (float* C, float* A, float* B, int wA, int wB)

{

/* Присвоение локальным переменным значений координат текущей «связки» потоков */

int block_x = blockIdx.x;

int block_y = blockIdx.y;

/* Определение индексов (*i*, *k*), соответствующих конкретному потоку в составе «связки» */

int i = threadIdx.y;

int k = threadIdx.x;

/* Для понимания дальнейшего необходимо учесть, что в глобальной памяти графического процессора (Device Memory на рис. 3.2) элементы исходных матриц *A* и *B* хранятся последовательно, в форме линейного массива, как это здесь показано

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \Rightarrow \left(a_{11}, a_{12}, a_{13}, \ldots, a_{31}, a_{32}, a_{33}\right).$$

Как можно увидеть из рис. 3.6, элементы матрицы *A*, обрабатываемые одной «связкой», расположены в линейном массиве последовательно. Если нумеровать все элементы линейного массива подряд с нуля, то номер первого элемента последовательности, с которой работает заданная «связка», —

$$a\_First = Block\_y \cdot \left(wA \cdot Block\_Size\right).$$

Вводим целочисленную константу, равную этому начальному номеру: */

int a_First = wA * Block_Size * block_y;

/* Как видно из рис. 3.6, последовательность элементов матрицы *A*, обрабатываемая выделенной «связкой», проходит через несколько бло-

ков размера *Block_Size*, которые загружаются в разделяемую память поочередно. Для организации цикла, внутри которого будут загружаться в память и обрабатываться эти блоки, нам потребуется номер 1-го элемента в последнем из блоков, который вычисляется по формуле

$$a\_Last = a\_First + (wA - 1) \cdot Block\_Size .$$

Определяем соответствующую целочисленную константу: */

int a_Last = a_First + (wA — 1) * BLOCK_SIZE;

/* Для организации этого же цикла потребуется шаг, задающий изменение 1-го номера (то есть «левого верхнего» элемента в блоке) при переходе от текущего блока к следующему */

int a_Step = Block_Size;

/* Индекс первого блока матрицы *B*, обрабатываемого текущей связкой

int b_First = Block_Size * block_x;

/* Шаг итераций «сверху вниз» (рис. 3.7) по блокам матрицы *B* */

int b_Step = Block_Size * wB;

/* Величины шагов при переходах от одних блоков к другим в матрицах *A* и *B* показаны на рис. 3.7. Вводим переменную, в которую будет записываться значение искомого элемента матрицы *Csub* (рис. 3.6, 3.7) */

float Csub = 0;

//Внешний цикл по всем блокам матриц *A* и *B*

for (int a = a_First, b = b_First;
a <= a_Last;
a += a_Step, b += b_Step) {

/*

/* Создаем в разделяемой памяти массив, куда из глобальной видеопамяти будет скопирован очередной блок матрицы *A*. Модификатор __shared__ как раз означает, что массив As будет создан в «быстрой» разделяемой памяти */

__shared__ float As [Block_Size] [Block_Size];

/* Такой же массив создается для хранения блока матрицы *B*: */

__shared__ float Bs [Block_Size] [Block_Size];

/* Копируем блоки матриц в разделяемую память. Каждый поток загружает 1 элемент матрицы *A* и один элемент матрицы *B* */

As [i] [k] = A [a + wA * i + k];

Bs [i] [k] = B [b + wB * i + k];

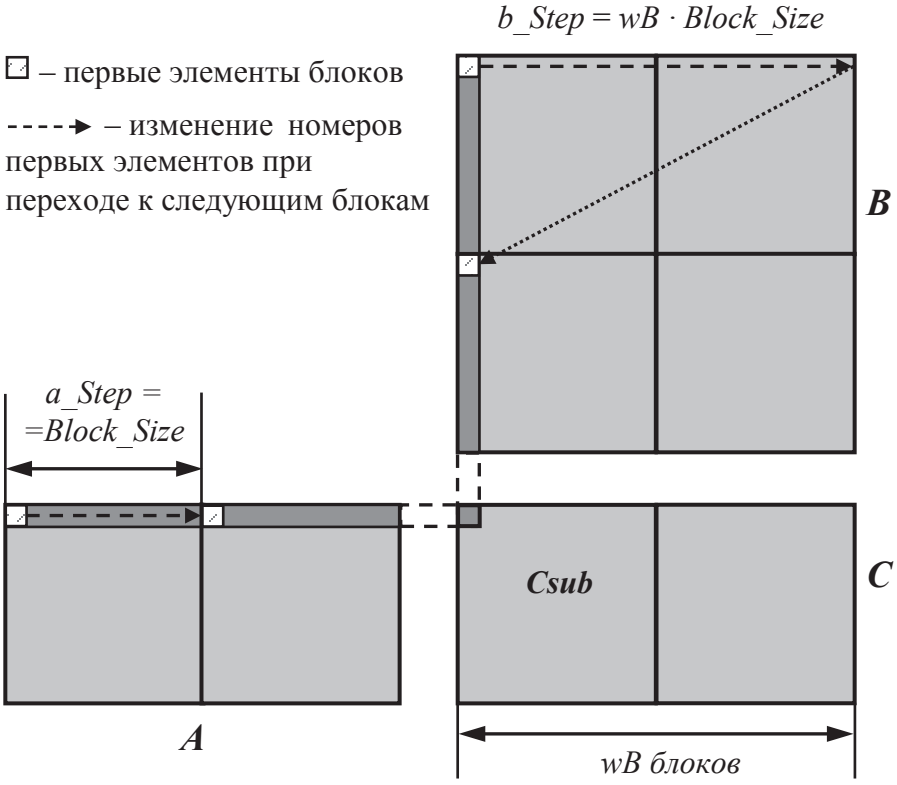


Рис. 3.7. Переходы к следующим блокам при умножении матриц */

/* Принцип определения номера загружаемого элемента в блоке показан на рис. 3.8 */

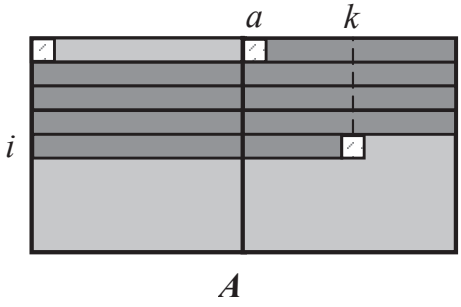


Рис. 3.8. Принцип расчета линейного номера элемента с индексами (i, j) в текущем блоке (a — номер 1-го элемента в данном блоке; (i, j) — индексы вычислительного потока, копирующего данный элемент (эти индексы отсчитываются не от начала матрицы, а от начала блока)

/* Синхронизируем потоки, чтобы гарантировать то, что блоки матриц загружены полностью */

__syncthreads ();

/* На данном этапе все элементы обрабатываемых блоков загружены в разделяемую память. Создавать цикл для их загрузки не требуется, так как этот код будет параллельно исполняться стольким числом потоков, равным количеству элементов в блоке матрицы *Block_Size*· *Block_Size*. */

/* Перемножение блоков матриц *A* и *B*, загруженных в «быструю» память GPU. Каждый поток вычисляет один элемент результирующей матрицы *Csub*

$$c_{Sub}[i,k] = \sum_{j=1}^{Block\_Size} a_j^i b_k^j .$$

Использование оператора += обеспечивает добавление рассчитываемого слагаемого ко всей сумме

$$c_k^i = \sum_{j=1}^{Block\_Size} a_j^i b_k^j + \sum_{j=Block\_Size+1}^{2\cdot Block\_Size} a_j^i b_k^j + \ldots + \sum_{j=(n-1)\cdot Block\_Size+1}^{wA} a_j^i b_k^j \ . */$$

for (int j = 0; j < Block_Size; ++j) Csub += As [i] [j] * Bs [j] [k];

/* Отметим, что при суммировании в последней формуле конкретный поток работает не только с теми элементами матриц, которые были загружены в разделяемую память именно этим потоком, но и элементами, которые загружались другими потоками. Это показывает преимущество общего доступа потоков к разделяемой памяти. */

/* Синхронизируем потоки. Это гарантирует, что все вычисления будут закончены до того, как начнется загрузка других блоков на следующей итерации внешнего цикла */

__syncthreads ();

}//Возвращение к началу внешнего цикла

/* Обратное копирование вычисленного блока результирующей матрицы *C* в глобальную память GPU, где данные будут доступны центральному процессору. Каждый поток сохраняет 1 элемент */

int c = wB * Block_Size * block_y + Block_Size * block_x;

C [c + wB * i + k] = Csub;

}

#endif//#ifndef _MATRIXMUL_KERNEL_H_

3.3.3. Оптимизация доступа к памяти при умножении матриц

Разделяемая память позволяет ускорить доступ к данным разными потоками связки за счет сокращения числа обращений к более медленной глобальной памяти. Кроме того, доступ к глобальной памяти по шаблону, препятствующему объединению транзакций, можно заменить объединенным доступом с последующим переупорядочиванием данных в разделяемой памяти. В отличие от глобальной памяти, разделяемая память не замедляется при непоследовательном или невыровненном доступе потоками, принадлежащими одному и тому же *полуварпу* (16 потоков, обрабатываемых на одном и том же участке конвейера), пока не возникает конфликт банков памяти.

Рассмотрим программу для умножения матриц $C = AB$, где A — размером $M \times 16$, B — размером $16 \times N$, а C — размером $M \times N$. Для простоты будем считать, что M и N делятся на 16. Простейшая декомпозиция задачи на подзадачи — использование связок размером 16×16 и подматриц такого же размера. В представлении на основе подматриц 16×16 матрица A будет столбцом, B — строкой, а C — их векторным произведением. Сетка связок для вычислительного ядра будет состоять из $N/16$ на $M/16$ связок, где каждая связка будет вычислять элементы различных подматриц матрицы C на основе одной подматрицы из A и одной — из B. Простейшее ядро выглядит следующим образом:

```
__global__ void simpleMultiply (float *A, float* B, float *C, int N) {
int row = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;
float sum = 0.0f;
for (int i = 0; i < TILE_DIM; i++)
sum += A [row * TILE_DIM + i] * B [i * N + col];
C [row * N + col] = sum;}
```

Здесь *blockDim.x*, *blockDim.y* и TILE_DIM равны 16. Каждый поток связки размером 16x16 вычисляет один элемент матрицы C, расположенный в строке *row* и столбце *col*, путем перемножения элементов строки матрицы A и элементов столбца матрицы B. Эта реализация имеет небольшую эффективную пропускную способность: 8.7 ГБ/с на GeForce GTX 280 и 0.7 ГБ/с на GeForce 8800 GTX (здесь и далее в этом параграфе эффективная пропускная способность вычисляется как отношение суммарного числа прочитанных и записанных ядром байтов ко времени выполнения ядра).

Для анализа производительности мы рассмотрим тип доступа потоков полуварпа к глобальной памяти. Каждый полуварп вычисляет одну строку подматрицы матрицы *C* на основе строки матрицы *A* и целой подматрицы из *B*. На *i*-й итерации цикла все потоки полуварпа читают одно и то же значение из глобальной памяти (индекс row*TILE_DIM+i одинаков для всех потоков полуварпа), что приводит к генерации шестнадцати транзакций на устройствах с вычислительной способностью 1.0 или 1.1 и одной транзакции на устройствах с вычислительной способностью 1.2 и выше. Но даже в случае одной транзакции пропускная способность расходуется неэффективно, так как используется лишь 4 байта из 32-байтной транзакции. Чтение строки подматрицы из *B* является последовательным и объединенным на устройствах любой вычислительной способности.

Таким образом, производительность можно увеличить чтением подматрицы из *A* в разделяемую память, как показано в следующем ядре:

```
__global__ void coalescedMultiply (float *A, float* A, float *C, int N) {
int row = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;
__shared__ float aTile [TILE_DIM] [TILE_DIM];
aTile [threadIdx.y] [threadIdx.x] = A [row*TILE_DIM+threadIdx.x];
float sum = 0.0f;
for (int i = 0; i < TILE_DIM; i++)
sum += aTile [threadIdx.y] [i] * B [i * N + col];
C [row * N + col] = sum;}
```

Здесь каждый элемент подматрицы *A* читается из глобальной памяти однократно, причем этот доступ является объединенным и ни один байт транзакций не пропадает зря. В данном случае синхронизация потоков после чтения подматрицы *A* не требуется, так как данные, записанные каждым потоком, читаются только потоками из этого же полуварпа. Эффективная пропускная способность данного ядра составляет 14.3 ГБ/с на GeForce GTX 280 и 8.2 ГБ/с на GeForce 8800 GTX.

Теперь обратимся к матрице *B*. При расчете строки подматрицы матрицы *C* производится чтение всей подматрицы из *B*. Многократное ее чтение из глобальной памяти заменяется доступом к разделяемой памяти:

```
__global__ void sharedABMultiply (float *A, float* B, float *C, int N) {
int row = blockIdx.y * blockDim.y + threadIdx.y;
```

```
int col = blockIdx.x * blockDim.x + threadIdx.x;
__shared__ float aTile [TILE_DIM] [TILE_DIM],
bTile [TILE_DIM] [TILE_DIM];
aTile [threadIdx.y] [threadIdx.x] = A [row*TILE_DIM+threadIdx.x];
bTile [threadIdx.y] [threadIdx.x] = B [threadIdx.y*N+col];
__syncthreads ();
float sum = 0.0f;
for (int i = 0; i < TILE_DIM; i++)
sum += aTile [threadIdx.y] [i] * bTile [i] [threadIdx.x];
C [row * N + col] = sum;}
```

Отметим, что после чтения подматрицы из B в разделяемую память требуется синхронизация потоков, так как каждый ее элемент читается потоками разных варпов. В данном случае ускорение произошло не за счет объединения транзакций, как в процедуре coalesced Multiply выше, а за счет меньшего числа обращений к глобальной памяти. Скорость для всех трех реализаций сведена в табл. 3.1.

Таблица 3.1

Результаты оптимизации умножения матриц $C = AB$, ГБ/с

Версия	GeForce 280	GeForce 8800
Без оптимизаций	8.7	0.7
Объединенный доступ к подматрице из A с ее сохранением в разделяемой памяти	14.3	8.2
Использование разделяемой памяти для уменьшения числа обращений к матрице B в глобальной памяти	29.7	15.7

Теперь рассмотрим вариант умножения матрицы A на транспонированную матрицу A^{T} ($C = AA^{\mathrm{T}}$) для иллюстрации доступа к глобальной памяти с неединичным шагом и конфликты банков разделяемой памяти. Простейшее ядро имеет следующий вид:

```
__global__ void simpleMultiply (float *A, float *C, int M)
{
int row = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;
float sum = 0.0f;
for (int i = 0; i < TILE_DIM; i++)
sum += A [row * TILE_DIM + i] * A [col * TILE_DIM + i];
C [row * M + col] = sum;}
```

Здесь элемент из строки *row* и столбца *col* матрицы *C* получается как скалярное произведение строк с индексами *row* и *col* матрицы *A*. Эффективная пропускная способность данного ядра составляет всего 1.1 ГБ/с для GeForce GTX 280 и 0.5 ГБ/с для GeForce 8800 GTX, то есть даже меньше пропускной способности простейшего ядра для умножения $C = AB$. Различие заключается в доступе потоков полуварпа к глобальной памяти во втором множителе *A* [*col* × *TILE_DIM*+*i*], где *col* соответствует последовательным столбцам транспонированной матрицы *A*. Поэтому *col* × *TILE_DIM* означает доступ к глобальной памяти с шагом 16, что приводит к невозможности объединенного доступа на GeForce 8800 GTX и неэффективному расходованию пропускной способности GeForce GTX 280. Для устранения неэффективного шаблона доступа к памяти мы снова используем разделяемую память, но сейчас будем читать потоками полуварпа строку матрицы *A*, сохраняя ее в столбец массива в разделяемой памяти:

```
__global__ void coalescedMultiply (float *A, float *C, int M) {
__shared__ float tile [TILE_DIM] [TILE_DIM],
tile_T [TILE_DIM] [TILE_DIM];
int row = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;
float sum = 0.0f;
tile [threadIdx.y] [threadIdx.x] = A [row*TILE_DIM+threadIdx.x];
tile_T [threadIdx.x] [threadIdx.y] =
A [(blockIdx.x * blockDim.x + threadIdx.y) * TILE_DIM +
threadIdx.x];
__syncthreads ();
for (int i = 0; i < TILE_DIM; i++)
sum += tile [threadIdx.y] [i] * tile_T [i] [threadIdx.x];
}
C [row * M + col] = sum;}
```

Здесь в разделяемую память (в массив tile_T) сохраняется также подматрица транспонированной матрицы для устранения необъединенного доступа во втором множителе слагаемых скалярного произведения, а кеширование первого множителя аналогично примеру с умножением $C = AB$. В данном случае пропускная способность составляет 24.9 и 13.2 ГБ/с соответственно, что чуть медленнее оптимальной версии умножения $C = AB$ из-за конфликтов банков разделяемой памяти.

Чтение элементов массива tile_T в цикле не содержит конфликтов, так как потоки каждого полуварпа читают по строкам подматрицы, то есть с единичным шагом по банкам. Однако конфликты возникают при копировании подматрицы из глобальной памяти в разделяемую. В целях получения объединенного доступа к глобальной памяти данные читаются последовательно. Однако это требует сохранения в столбцах tile_T, а использование подматриц размером 16x16 приводит к шагу 16 по банкам разделяемой памяти, и пропускная способность снижается в 16 раз. Простейшим приемом для преодоления этих конфликтов является добавление в массив tile_T еще одного столбца:

__shared__ float tile_T [TILE_DIM] [TILE_DIM + 1];

В результате конфликты банков полностью устраняются, так как запись производится с шагом 17 по банкам разделяемой памяти (табл. 3.2).

Как видно из табл. 3.1 и 3.2, разумное использование разделяемой памяти может значительно повысить производительность.

Таблица 3.2

Результаты оптимизации умножения матриц $C = AA^T$, ГБ/с

Версия	GeForce 280	GeForce 8800
Без оптимизаций	1.1	0.5
Объединенный доступ за счет использования разделяемой памяти	24.9	13.2
Устранение конфликтов банков	30.4	15.6

3.4. Динамика N тел на CUDA. Пример ускорения программы за счет скорости GPU

Рассмотрим задачу моделирования динамики гравитационно-взаимодействующих *N* тел (*N*-body simulation) методом прямого суммирования. Структура таких вычислений является фундаментальным строительным блоком в более сложных задачах и прямо используется в приближенных иерархических методах: Treecode и FMM (Fast Multipole Method). Опыт, полученный при реализации вычислительных ядер для этой физической задачи, полезен и в других областях при статистическом анализе информации, ее поиске и извлечении.

Может казаться, что при решении этой задачи легко извлечь производительность, близкую к теоретическому максимуму на разных аппаратных платформах. Ключевой и наиболее затратной частью *гравитационной динамики* является расчет парных взаимодействий в системе из N частиц. Ядро очень простое и требовательное к скорости вычислений процессора и потому хорошо подходит для ускорителей, таких как GPU и PowerXCell/8i. Тем не менее даже такое простое ядро требует немало усилий для его оптимизации.

Для простоты остановимся на рассмотрении случая достаточно большого числа частиц N. Случай малых N тоже важен, к примеру, при использовании в иерархических алгоритмах, но требует учитывать большее число низкоуровневых деталей архитектуры. Хотя доступ к памяти достаточно регулярный, но архитектуры рассмотренных ниже ускорителей потребовали тщательного управления быстрой локальной памятью для эффективного использования ресурсов, что привело к необходимости рассмотрения нетривиальных компромиссов.

В 2009 году было проведено исследование по сравнению реализаций гравитационной динамики на следующих платформах [1]:

- 8-ядерная система на базе 4-ядерных процессоров Intel Xeon X5550 с архитектурой Nehalem (за счет технологии Hyper-Threading получается 16 эффективных параллельных потоков обработки);
- 16-ядерная система на базе 4-ядерных процессоров AMD Opteron 8350 с архитектурой Barcelona;
- блэйд-сервер IBM QS22 на базе процессора Sony-Toshiba-IBM PowerXCell/8i;
- профессиональный графический адаптер NVIDIA Tesla C870 с унифицированной архитектурой первого поколения G80;
- профессиональный графический адаптер NVIDIA Tesla C1060 с унифицированной архитектурой второго поколения архитектурой G200.

Интегрирование уравнений движения на каждом шаге требует расчета и суммирования парных сил для каждой i-й частицы

$$\vec{F}_i = -Gm_i \sum_{1 \le j \le N,\, j \ne i} m_j \frac{\vec{r}_i - \vec{r}_j}{\left\| \vec{r}_i - \vec{r}_j \right\|^3},$$

где i-я частица имеет массу m_i и расположена в позиции $\vec{r}_i$ трехмерного пространства; G — универсальная гравитационная постоянная, которую для простоты нормализуем к единице.

На практике вместо расчета силы, пропорциональной массе i-й частицы, и применения второго закона Ньютона $\vec{F}_i = m_i \cdot \vec{a}_i$, рассчитывают сразу ускорение $\vec{a}_i$, а для улучшения стабильности алгоритма добавляют в знаменатель сглаживающий параметр, не допускающий деления на ноль или очень близкие к нулю числа,

$$\vec{a}_i \approx \sum_{1 \le j \le n} m_j \frac{\vec{r}_i - \vec{r}_i}{(\|\vec{r}_i - \vec{r}_i\|^2 + \varepsilon^2)^{3/2}}.$$

Расчет ускорения доминирует в расчете полного шага, поэтому при сравнении производительностей различных вычислительных платформ мы можем проигнорировать затраты на интегрирование уравнений движения. Скалярный псевдокод для расчета ускорения можно записать следующим образом:

1. **for** all bodies i **do**
2. *//Загрузить координаты (x_i, y_i, z_i)*
3. $(a_x, a_y, a_z) = (0,0,0)$
4. **for** all bodies j ≠ i **do**
5. *//Загрузить координаты (x_j, y_j, z_j)*
6. $(\Delta x, \Delta y, \Delta z) = (x_i - x_j, y_i - y_j, z_i - z_j)$
7. $\gamma = (\Delta x)^2 + (\Delta y)^2 + (\Delta z)^2 + \varepsilon^2$
8. $s = m_j/(\gamma * \sqrt{\gamma})$
9. $(a_x, a_y, a_z) = (a_x + s\Delta x, a_y + s\Delta y, a_z + s\Delta z)$
10. **end for**
11. *//Сохранить ускорение (a_x, a_y, a_z)*
12. **end for**

Здесь записано 18 операций плавающей арифметики: 3 вычитания (строка 6), 3 умножения и 3 сложения (строка 7), 1 умножение, 1 квадратный корень и 1 деление (строка 8), 3 умножения и 3 сложения (строка 9). Будем считать стоимость расчета парного взаимодействия равной 20 FLOP, а в сумме по всем частицам получим $20N(N-1)$ FLOP.

Здесь также $(4N^2+6N)$ обращений к массивам в общей памяти, в основном это загрузка трех координат в строках 5–6 и массы в строке 8. Однако повторное использование загруженных величин при помощи подходящего кеширования позволит приблизиться к минимальному

числу промахов кеша $4n$, поэтому весь алгоритм будет ограничен вычислительными возможностями процессоров: его вычислительная интенсивность составит $20n^2/4n = 5n$ FLOP на доступ к 32-битному слову. А внутри j-цикла доступ будет происходить к внутричиповой памяти (кеши и разделяемая память GPU).

На всех платформах применялось грубое распараллеливание внешнего i-цикла и более тонкое распараллеливание по данным внутреннего j-цикла. При простейшем распараллеливании только внешнего i-цикла не требуется синхронизации операций записи, поскольку каждый поток будет записывать ускорение только для своего блока частиц (которые не пересекаются). В табл. 3.3 приведена количественная оценка сложности реализации для каждой платформы.

Таблица 3.3

Сравнение сложности реализаций гравитационной динамики для разных платформ [21]

Критерий	Tesla C870	Tesla C1060	QS22	Nehalem	Barcelona
Число строк в коде	380	390	850	500	500
Число дней	2	3	20	5	5

Отметим, что версии для двойной точности уступают версиям одинарной по скорости более чем в два раза, что связано с наличием операций извлечения квадратного корня и деления, для которых ни на одной платформе не предусмотрена быстрая аппаратная реализация с двойной точностью. В результате сравнения [21] GPU и CUDA оказались самыми быстрыми, самыми эффективными по цене и энергопотреблению, и при этом соответствующая программа оказалась наиболее простой.

3.5. Распараллеливание алгоритмов сортировки. Пример ускорения программы за счет скорости GPU

Сортировка является фундаментальным строительным блоком для многих компьютерных программ, для чего были разработаны эффективные алгоритмы для множества параллельных архитектур. Некоторые алгоритмы основаны на процедурах сортировки как на базисе их собственной эффективности, другие используют очень близкий к со-

ртировке математический аппарат. В базах данных широко применяются операции сортировки. Конструирование пространственных структур данных в графических приложениях и геоинформационных системах также опирается на сортировку. Сортировка используется при реализации алгоритмов умножения разреженных матриц и каркасах параллельного программирования, таких как MapReduce. Важно иметь эффективные процедуры сортировки на каждой программной платформе, причем все время сохраняется необходимость исследования методов сортировки на новых архитектурах.

Для простоты изложения рассмотрим только два класса алгоритмов: *поразрядную сортировку* (*radix sort*), основанную на бинарном представлении ключей, и *сортировку слиянием* (*merge sort*), которая требует только функцию сравнения ключей.

GPU поддерживает несколько тысяч параллельных потоков обработки, и предоставление программисту тонкого параллелизма (с малыми затратами на переключение потоков) критично для разработки эффективных алгоритмов на таких архитектурах. В поразрядной сортировке такой параллелизм используется при построении алгоритма, основанного на операциях параллельного сканирования (*parallel scan*), а в сортировке слиянием при построении алгоритма попарного параллельного слияния адаптивные схемы основаны на параллельном разделении (*parallel splitting*) и бинарном поиске. Блочная структура алгоритмов сортировки позволяет использовать быструю внутричиповую память GPU. Она используется для локального упорядочения данных, значительно улучшая эффективность использования пропускной способности для этапов рассеяния (*scatter*) в поразрядной сортировке.

В 2009 году было проведено исследование по сравнению реализаций сортировки на следующих платформах [22]:

- 8-ядерная система на базе двух 4-ядерных процессоров Intel Xeon E5345 (Clovertown) с частотой 2.33 ГГц:
  - реализация Intel “tbb: parallel_sort” в библиотеке Threading Building Blocks (TBB);
  - реализация авторов сравнения (Сатиш и др.) на базе библиотеки TBB;
  - реализация с ручным SSE2-распараллеливанием на базе Intel Pthreads;
- игровой графический адаптер NVIDIA GeForce GTX 280 с унифицированной архитектурой второго поколения G200.

Сравнение скорости работы алгоритмов сортировки выполнялось для последовательностей пар ключ–значение, где ключи и значения были 32-битными числами. Хотя этот случай и не включает все практические задачи, он прямо относится к сортировке точек в пространстве и построению нерегулярных структур данных. Для генерации ключей использовался генератор псевдослучайных чисел с равномерным распределением. Сравниваемые времена счета не включали в себя время копирования данных по шине PCI-Express, так как сортировка наиболее важна как часть более сложного вычислительного алгоритма, в этом случае сортируемые данные сгенерированы одним GPU-ядром, а после сортировки будут поданы на вход другому GPU-ядру.

Вычислительная сложность рассматриваемых алгоритмов при достаточно больших N: $O(N)$ для поразрядной сортировки и $O(N/\log(N))$ для сортировки слиянием. При сортировке небольших последовательностей большая часть времени уходит на фиксированные издержки (не меняющиеся с числом элементов последовательности).

Реализованные на GPU алгоритмы сортировки достаточно хорошо справляются со своей задачей. Поразрядная сортировка обладает наибольшей скоростью при $N > 8000$, где в среднем в 4.4 раза превосходит реализацию tbb: parallel_sort и в 3 раза — TBB-реализацию авторов сравнения. На последовательности из 10 миллионов элементов скорость сортировки достигает 150 миллионов пар в секунду (что, если не считать промежуточные операции, соответствует пропускной способности в 1.2 ГБ/с).

Заключение

В настоящем пособии рассмотрены основные принципы проведения вычислений общего назначения на графических процессорах. Показано, что графические процессоры могут быть очень эффективными как многопроцессорные системы, реализующие распараллеливание вычислений по данным и по задачам. Современное развитие графических процессоров обеспечивает исполнение и оптимизацию все более сложных алгоритмов. Таким образом, разработка приложений для физико-математического моделирования и других ресурсоемких расчетов на графических процессорах становится все более актуальной и перспективной.

Приложение

Перемножение матриц на CUDA
Программа, исполняемая центральным процессором

/* Текст вычислительного ядра, исполняемого на GPU, с подробными комментариями приведен в параграфе 3.3.2. Здесь рассматриваются те части приложения, которые предназначены для центрального процессора */

//Файл matrixMul.cu [6]

/* Умножение матриц: $C = A \cdot B$. Вызывающий код на CPU. В этом примере реализовано умножение матриц из документации CUDA Programming Guide [6]. Поскольку данная реализация предназначена для изучения принципов программирования на CUDA, в целях улучшения ясности изложения использован не самый эффективный из существующих ходов. Высокопроизводительная версия умножения матриц реализована в библиотеке CUBLAS [21] */

//Подключение библиотек со стандартными процедурами и функциями

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <cutil.h>
```

/* Подключение файла, содержащего текст вычислительного ядра, исполняемого на CPU (п. 3.3.2) */

```
#include <matrixMul_kernel.cu>
```

//предварительное объявление функций

```
void runTest (int argc, char** argv);
void randomInit (float*, int);
void printDiff (float*, float*, int, int);
```

/* Процедура, которая для сравнения перемножает матрицы на центральном процессоре: */

```
extern "C" void computeGold (float*, const float*, const float*, unsigned int, unsigned int, unsigned int);
```

/* Процедура, запускающая перемножение матриц на GPU. Само перемножение исполняется процедурой runTest */

```
int main (int argc, char** argv)
{
runTest (argc, argv);
CUT_EXIT (argc, argv);//Завершение работы с GPU
}
```

//Текст процедуры runTest, перемножающей матрицы на GPU

```
void runTest (int argc, char** argv)
{
CUT_DEVICE_INIT ();
```

/* инициализация генератора псевдослучайных чисел rand () для заполнения матриц случайными числами */

```
srand (2006);
//выделение памяти для матриц A и B
unsigned int size_A = WA * HA;
unsigned int mem_size_A = sizeof (float) * size_A;
float* h_A = (float*) malloc (mem_size_A);
unsigned int size_B = WB * HB;
unsigned int mem_size_B = sizeof (float) * size_B;
float* h_B = (float*) malloc (mem_size_B);
```

//заполнение исходных матриц псевдослучайными числами

```
randomInit (h_A, size_A);
randomInit (h_B, size_B);
```

//выделение памяти на GPU

```
float* d_A;
CUDA_SAFE_CALL (cudaMalloc ((void**) &d_A, mem_size_A));
float* d_B;
CUDA_SAFE_CALL (cudaMalloc ((void**) &d_B, mem_size_B));
```

//копирование матриц на GPU

```
    CUDA_SAFE_CALL (cudaMemcpy (d_A, h_A, mem_size_A,
cudaMemcpyHostToDevice));
    CUDA_SAFE_CALL (cudaMemcpy (d_B, h_B, mem_size_B,
cudaMemcpyHostToDevice));
    //выделение памяти на GPU для матрицы-результата
    unsigned int size_C = WC * HC;
    unsigned int mem_size_C = sizeof (float) * size_C;
    float* d_C;
    CUDA_SAFE_CALL (cudaMalloc ((void**) &d_C, mem_size_C));
    //создание и запуск таймера
    unsigned int timer = 0;
    CUT_SAFE_CALL (cutCreateTimer (&timer));
    CUT_SAFE_CALL (cutStartTimer (timer));
    //установка параметров выполнения ядра
    dim3 threads (BLOCK_SIZE, BLOCK_SIZE);
    dim3 grid (WC/threads.x, HC/threads.y);
    //запуск ядра
    matrixMul<<< grid, threads >>> (d_C, d_A, d_B, WA, WB);
    //проверяем на наличие ошибок выполнения ядра
    CUT_CHECK_ERROR («Ошибка выполнения ядра»);
    //выделяем память на CPU для матрицы-результата
    float* h_C = (float*) malloc (mem_size_C);
    //копируем результат на CPU
    CUDA_SAFE_CALL (cudaMemcpy (h_C, d_C, mem_size_C,
    cudaMemcpyDeviceToHost));
    //останавливаем таймер и освобождаем ресурсы
    CUT_SAFE_CALL (cutStopTimer (timer));
    printf (“Processing time: %f (ms) n”, cutGetTimerValue (timer));
    CUT_SAFE_CALL (cutDeleteTimer (timer));
    //вычисляем произведение A*B на CPU для сравнения точности
    float* reference = (float*) malloc (mem_size_C);
    computeGold (reference, h_A, h_B, HA, WA, WB);
    //проверяем результат
    CUTBoolean res = cutCompareL2fe (reference, h_C, size_C, 1e-6f);
    printf (“Тест %s n”, (1 == res)? “ПРОЙДЕН”: “ПРОВАЛЕН”);
    if (res!=1) printDiff (reference, h_C, WC, HC);
    //освобождаем выделенную память
    free (h_A);
```

```
free (h_B);
free (h_C);
free (reference);
CUDA_SAFE_CALL (cudaFree (d_A));
CUDA_SAFE_CALL (cudaFree (d_B));
CUDA_SAFE_CALL (cudaFree (d_C));
}
```

/* Процедура, заполняющая перемножаемые матрицы псевдослучайными числами */

```
void randomInit (float* data, int size)
{
for (int i = 0; i < size; ++i)
data [i] = rand ()/(float)RAND_MAX;
}

void printDiff (float *data1, float *data2, int width, int height)
{
int i, j, k;
int error_count=0;
for (j=0; j<height; j++) {
for (i=0; i<width; i++) {
k = j*width+i;
if (data1 [k]!= data2 [k]) {
printf ("diff (%d, %d) CPU=%4.4f, GPU=%4.4f n", i, j, data1 [k], data2
[k]);
error_count++;
}}}
printf (" Количество ошибок =%d n", error_count);
}

//Файл matrixMul.h [6]
```

//Copyright 1993–2007 NVIDIA Corporation. All rights reserved.

```
#ifndef _MATRIXMUL_H_
#define _MATRIXMUL_H_
```

//Размер связки потоков

```
#define BLOCK_SIZE 16
```

/* Размеры матрицы (для простоты кода выбраны кратными размеру связки потоков) */

#define WA (3 * BLOCK_SIZE)//Matrix A width
#define HA (5 * BLOCK_SIZE)//Matrix A height
#define WB (8 * BLOCK_SIZE)//Matrix B width
#define HB WA//Matrix B height
#define WC WB//Matrix C width
#define HC HA//Matrix C height
#endif//_MATRIXMUL_H_
#ifndef _MATRIXMUL_H_
#define _MATRIXMUL_H_
//Размер связки потоков
#define BLOCK_SIZE 16
//Размеры матрицы (для простоты кода выбраны кратными размеру связки потоков)
#define WA (3 * BLOCK_SIZE)//Matrix A width
#define HA (5 * BLOCK_SIZE)//Matrix A height
#define WB (8 * BLOCK_SIZE)//Matrix B width
#define HB WA//Matrix B height
#define WC WB//Matrix C width
#define HC HA//Matrix C height
#endif//_MATRIXMUL_H_

Вычисление скалярного произведения векторов на CUDA

Листинг 1. Код центрального процессора для вычисления скалярного произведения двух векторов

//Число обрабатываемых пар векторов
const int VECTOR_N = 256;
//Число элементов в каждом векторе
//Наилучшая производительность достигается тогда, когда число элементов кратно размеру варпа
const int ELEMENT_N = 4096;
//Полное число элементов
const int DATA_N = VECTOR_N * ELEMENT_N;
//Размер одной половины исходных векторов в байтах

```
const int DATA_SZ = DATA_N * sizeof (float);
//Размер результирующего вектора в байтах
const int RESULT_SZ = VECTOR_N * sizeof (float);

void main (int argc, char *argv [])
{
float *cpuA, *cpuB, *cpuC1, *cpuC2;
float *gpuA, *gpuB, *gpuC;
unsigned int hTimer;

//Выделение памяти на CPU
cpuA = (float *)malloc (DATA_SZ);
cpuB = (float *)malloc (DATA_SZ);
cpuC1 = (float *)malloc (RESULT_SZ);
cpuC2 = (float *)malloc (RESULT_SZ);

//Генерация входных данных на CPU
srand (123);
for (int i = 0; i < DATA_N; i++)
{
cpuA [i] = (float)rand ()/(float)RAND_MAX;
cpuB [i] = (float)rand ()/(float)RAND_MAX;
}

//Инициализация CUDA
cudaSetDevice (0);
cutCreateTimer (&hTimer);

//Выделение памяти на GPU
cudaMalloc ((void **)&gpuA, DATA_SZ);
cudaMalloc ((void **)&gpuB, DATA_SZ);
cudaMalloc ((void **)&gpuC, RESULT_SZ);

//Копирование исходных векторов на GPU
cudaMemcpy (gpuA, cpuA, DATA_SZ, cudaMemcpyHostToDevice);
cudaMemcpy (gpuB, cpuB, DATA_SZ, cudaMemcpyHostToDevice);
```

```
    //Запуск обработки на GPU и вывод времени работы ядра
    cudaThreadSynchronize ();
    cutResetTimer (hTimer);
    cutStartTimer (hTimer);
    scalarProdGPU<<<128, 256>>> (
    gpuC, gpuA, gpuB, VECTOR_N, ELEMENT_N);
    cudaThreadSynchronize ();
    cutStopTimer (hTimer);
    printf ("GPU time: %f msecs.\n", cutGetTimerValue (hTimer));

    //Копирование результата на CPU для проверки правильности
    cudaMemcpy (cpuC1, gpuC, RESULT_SZ, cudaMemcpyDeviceTo-
Host);

    //Запуск обработки на CPU
    scalarProdCPU (cpuC2, cpuA, cpuB, VECTOR_N, ELEMENT_N);

    //Расчет нормы L1 разности результатов GPU и CPU
    double delta = 0, ref = 0;
    for (int i = 0; i < VECTOR_N; i++)
    {
    delta += fabs (cpuC1 [i] — cpuC2 [i]);
    ref += cpuC2 [i];
    }
    printf ("L1 error: %E\n", delta/ref);
    printf ((delta/ref < 1e-6)? "PASSED\n": "FAILED\n");

    //Освобождение ресурсов перед завершением программы
    cudaFree (gpuC); cudaFree (gpuB); cudaFree (gpuA);
    free (cpuC2); free (cpuC1); free (cpuB); free (cpuA);
    cutDeleteTimer (hTimer);
    cudaThreadExit ();
    cutilExit (argc, argv);
    }

    //Расчет скалярного произведения на CPU
    void scalarProdCPU (
```

```
float *C, float *A, float *B, int vectorN, int elementN)
{
for (int v = 0; v < vectorN; v++) {
int v_from = elementN * v, v_to = v_from + elementN;
double sum = 0;
for (int i = v_from; i < v_to; i++) sum += A [i] * B [i];
C [v] = (float)sum;
}
}
```

Листинг 2. GPU-код для вычисления скалярного произведения двух векторов

```
//Число аккумуляторов в кеше
#define A_COUNT 1024

//Ядро для расчета скалярного произведения на GPU
//Ограничения на параметры:
//1) ElementN по возможности должно быть кратно размеру варпа;
//2) A_COUNT должно быть равно целой степени двойки
__global__ void scalarProdGPU (
float *C, float *A, float *B, int vectorN, int elementN)
{
//Кеш для накопления суммы (аккумуляторы)
__shared__ float accumulator [A_COUNT];

//Цикл по всем парам векторов, учитывающий возможное
//различие числа векторов и числа связок потоков
for (int v = blockIdx.x; v < vectorN; v += gridDim.x)
{
int v_from = elementN * v, v_to = v_from + elementN;

//Цикл накопления суммы, пробегающий по векторам
//с шагом равным числу аккумуляторов A_COUNT
//На этом шаге требуется только, чтобы число A_COUNT
//было кратно размеру варпа, чтобы мог выполняться
```

```
//объединенный доступ к памяти (memory coalescing)
for (int a = threadIdx.x; a < A_COUNT; a += blockDim.x)
{
float sum = 0;
for (int i = a + v_from; i < v_to; i += A_COUNT)
sum += A [i] * B [i];
accumulator [a] = sum;
}
//Иерархическая редукция накопленных результатов
//Здесь необходимо, чтобы A_COUNT было степенью двойки
for (int stride = A_COUNT/2; stride > 0; stride >>= 1)
{
__syncthreads ();
for (int a = threadIdx.x; a < stride; a += blockDim.x)
accumulator [a] += accumulator [stride + a];
}
if (threadIdx.x == 0) C [v] = accumulator [0];
}
}
```

Компиляция программ на CUDA

Опции компилятора nvcc

Nvcc — инструмент для управления всеми этапами компиляции, называемый также драйвером компиляции (CUDA compiler driver). Компиляция при помощи nvcc происходит в несколько этапов (phases). Выбор этих фаз производится на основе опций, передаваемых через командную строку, и расширений входных файлов, причем на выполнение этих фаз можно влиять при помощи других опций командной строки.

В процессе своей работы nvcc вызывает следующие инструменты:

- CUDAFE (CUDA Language Front End);
- NVOPENCC (CUDA Open64 Compiler);
- PTXAS (PTX Optimizing Assembler);
- хостовые компилятор и линковщик (Microsoft Visual C++ для ОС Windows или gcc для ОС Linux).

Компилятор поддерживает следующие расширения файлов:

- .cu — исходные файлы CUDA, содержащие код для хоста и для устройств;
- .cup — исходные файлы CUDA после предварительной обработки (preprocessed);
- .cc,.cxx,.cpp — исходные файлы C++;
- .gpu — промежуточный файл для GPU;
- .ptx — файл с промежуточным ассемблерным PTX-кодом;
- .o,.obj — объектный файл;
- .a,.lib — библиотечный файл;
- .res — файл с ресурсами;
- .so — разделяемый объектный файл.

Nvcc не различает объектные, библиотечные и ресурсные файлы, а сразу пропускает их к линковщику на соответствующей фазе. В отличие от gcc, nvcc при получении на входе файла с незнакомым расширением (то есть не перечисленном выше) генерирует ошибку.

Nvcc различает 3 варианта опций командной строки: булевы (флаги) — без аргументов, опции с одним аргументом и опции с несколькими аргументами (списки). Аргументы отделяются от названия опции символом равенства либо одним или более пробелами. В некоторых случаях совместимости с компилятором gcc (опции –I, — l, — L) аргумент может не отделяться от названия опции. Несколько аргументов одной опции могут разделяться запятыми, допускается повтор таких опций, возможна также комбинация этих двух вариантов.

Каждая опция имеет длинное и сокращенное имя, которые отличаются количеством дефисов: два для длинного имени и один для короткого. Длинные имена, как правило, используются в сценариях автоматической компиляции, где длина менее важна, чем ценность самоописания.

Список опций, определяющих фазу компиляции

Имя		Описание
длинное	короткое	
—cuda	-cuda	Компиляция.cu →.cu.c/.cu.cpp
—cubin	-cubin	Компиляция.cu/.gpu/.ptx →.cubin. Хостовый код отбрасывается
—ptc	-ptx	Компиляция.cu/.gpu →.ptx. Хостовый код отбрасывается

Имя		Описание
длинное	короткое	
—gpu	-gpu	Компиляция.cu →.gpu Хостовый код отбрасывается
—preprocess	-E	Предварительная обработка.c/.cc/.cpp/.cxx/.cu.
—generate-dependencies	-M	Генерация для одного.c/.cc/.cpp/.cxx/.cu файла зависимостей, который может быть включен в make-файл
—compile	-c	Компиляция.c/.cc/.cpp/.cxx/.cu →.obj
—link	-link	Поведение по умолчанию: компиляция и линковка всех файлов
—lib	-lib	Компиляция всех входных файлов в объектные и объединение результатов в один библиотечный файл
—run	-run	Компиляция, линковка и запуск. При этом путь к необходимым динамическим библиотекам CUDA берется из файла nvcc.profile

Список опций, указывающих файлы и пути к файлам

Имя		Описание
длинное	короткое	
—output-file <file>	-o	Выходной файл. Более одного аргумента допускается только в режиме линковки/архивации
—pre-incude <file>, ...	-include	Заголовочные файлы
—library <file>, ...	-l	Библиотечные файлы. Путь для их поиска указывается опцией ‘-L’
—define-macro <def>, ...	-D	Определение макросов
—undefine-macro <def>, ...	-U	Отмена уже определенных макросов
—include-path <path>, ...	-I	Пути поиска заголовочных файлов
—system-include <path>, ...	-isystem	Пути поиска системных заголовочных файлов
—library-path <path>, ...	-L	Пути поиска библиотечных файлов
—output-directory <dir>	-odir	Путь для сохранения выходного файла. Полезна для генерации зависимостей через ‘—generate-dependencies’
—compiler-bindir <dir>	-ccbin	Путь к хостовому компилятору: Microsoft Visual Studio cl или gcc

Список опций для изменения поведения компилятора/линковщика

Имя		Описание
длинное	короткое	
—profile	-pg	Инструментирование кода для профилировщика gprof (только в ОС Linux)
—debug <level>	-g	Генерация отлаживаемого кода
—device-debug	-G	Генерация отлаживаемого кода устройства
—optimize <level>	-O	Генерация оптимизированного кода
—shared	-shared	Генерация разделяемой библиотеки во время линковки. Если требуются другие опции линковщика, используйте опцию '-Xlinker'
—machine	-m	Выбор 32-битной или 64-битной архитектуры. В данный момент она нужна только на платформе linux64 при компиляции с опцией '-cubin'

Список опций, управляющих вызываемыми nvcc инструментами

Имя		Описание
длинное	короткое	
—compiler-options <opts>,..	-Xcompiler	Передача опций компилятору/препроцессору
—linker-options <opts>, ...	-Xlinker	Передача опций линковщику
—cudafe-options <opts>, ...	-Xcudafe	Передача опций cudafe
—opencc-options <opts>, ...	-Xopencc	Передача опций nvopencc, обычно для управления оптимизацией. К примеру, '-LIST: source=on' приводит к включению в ptx-код комментариев с исходным кодом
—ptxas-options <opts>, ...	-Xptxas	Передача опций оптимизирующему PTX-ассемблеру. К примеру, '-v' приводит к выводу статистики генерации кода

Список опций, управляющих поведением nvcc

Имя		Описание
длинное	короткое	
—dryrun	-dryrun	Перечислить команды компиляции, генерируемые nvcc, вместо их выполнения
—verbose	-v	Перечислить команды компиляции, генерируемые nvcc, вместе с выполнением

Имя		Описание
длинное	короткое	
—keep	-keep	Сохранить все промежуточные файлы
—save-temps	-save-temps	То же, что и '— keep'
—dont-use-profile	-noprof	Не использовать файл настроек nvcc.profile
—clean-targets	-clean	Вместо каждой фазы компиляции удалить файлы, которые бы остались после нее
—run-args <arg>, ...	-run-args	Используется вместе с опцией '-R' для передачи аргументов исполняемому файлу
—input-drive-prefix <prefix>	-idp	В ОС Windows: сконвертировать все аргументы, включающие имена файлов, в родной формат Windows. Это относится к представлению абсолютных путей к файлам в «текущей» среде разработки
—dependency-drive-prefix <prefix>	-ddp	В ОС Windows при генерации файлов с зависимостями (опция '-M') все имена файлов конвертируются в такие, которые поддерживаются используемой утилитой make. К примеру, некоторые варианты make не поддерживают символ двоеточия
—drive-prefix <prefix>	-dp	Задает префикс логического диска для обеих опций: input-drive-prefix и dependency-drive-prefix

Список опций, управляющих компиляцией CUDA

Имя		Описание
длинное	короткое	
—device-emulation	-deviceemu	Сгенерировать код для библиотеки эмуляции GPGPU. В релизе CUDA 3.0 режим эмуляции стал считаться устаревшим
—use_fast_math	-use_fast_math	Использование библиотеки быстрой арифметики
—ftz <boolean>	-ftz	Управление денормализацией чисел с одинарной точностью (false означает поддержку денормализованных чисел, true — их обнуление)
—prec-div <boolean>	-prec-div	Управление точностью 32-битного деления (true — для соответствия стандарту IEEE, false — использование аппроксимации)
—prec-sqrt <boolean>	-prec-sqrt	Управление точностью 32-битного квадратного корня (true — для соответствия стандарту IEEE, false — использование аппроксимации)
—entries <entry>, ...	-e	В случае компиляции.ptx/.gpu →.cubin: определить функции, являющиеся глобальными точками входа, для которых будет сгенерирован код. По умолчанию код генерируется для всех

Список опций, управляющих генерацией GPU-кода

Имя		Описание
длинное	короткое	
— gpu-architecture <gpuarch>	-arch	Задает архитектуру GPU — реальную или виртуальную (PTX) — для всех этапов компиляции вплоть до генерации PTX-кода. Поддерживаются виртуальные архитектуры: compute_10, compute_11, compute_12, compute_13; и реальные: sm_10, sm_11, sm_12, sm_13, — которые их реализуют
— gpu-code <gpuarch>, ...	-code	Влияет на генерацию бинарного кода, который задействуется только в случае его совместимости с используемым GPU, иначе происходит компиляция встроенного PTX-кода. Указанная архитектура должна быть совместима с указанной в опции '-arch'. Если опция '-arch' указана, то здесь должна быть выбрана виртуальная PTX-архитектура. Если опция '-code' не указана, то берется значение из опции '-arch'
— generate-code'-arch=<arch> -code=<code>, ...'	-gencode	Позволяет многократно вызывать компилятор nvopencc, генерируя код для нескольких виртуальных архитектур. Аргументы 'arch' и 'code' соответствуют двум предыдущим опциям
— export-dir <file>	-dir	Указывает директорию или zip-файл, в которые будут скопированы все сгенерированные образы кода. Этот файл будет играть роль репозитория кода, инспектируемого драйвером CUDA во время выполнения программы. В обоих случаях nvcc сохранит структуру директорий для помощи драйверу CUDA при поиске. Если эта опция не указана, то все внешние образы будут отброшены. Если указано имя несуществующего файла, то будет создана директория с таким именем
— extern-mode <all|none|real|virtual>	-ext	Определяет, какие образы кода будут скопированы в директорию, указанную в опции 'export-dir'. Если эта опция и 'intern-mode' не указаны, то все образы кода будут считаться внутренними

Имя		Описание
длинное	короткое	
— intern-mode <all|none|real|virtual>	-int	Определяет, какие образы кода будут встроены в выходной (например, исполняемый) файл. Если эта опция и 'extern-mode' не указаны, то все образы кода будут считаться внутренними
— maxrregcount <amount>	-maxrregcount	Определяет максимальное число регистров, которые может использовать один поток GPU. Из-за ограниченных ресурсов мультипроцессора GPU слишком большое значение приведет к ограничению числа потоков в связке. Однако при недостаточно большом для какого-то вычислительного ядра значении его производительность будет ниже в том числе за счет использования вместо регистров медленной локальной памяти GPU. Указанное значение округляется в большую сторону до кратного четырем, вплоть до максимума в 128 регистров. Если опция не указана, то ограничение не применяется (только физические ограничения)

Профайлер

В инструментарий CUDA Toolkit входит профайлер, который во время выполнения вычислительного ядра и операций копирования собирает информацию об их длительности. Это позволяет находить узкие места в программах и численно оценить выгоду от оптимизации отдельного ядра. Собирание этой информации контролируется следующими переменными среды:

- CUDA_PROFILE — значения 1 или 0 включают и отключают сбор информации;
- CUDA_PROFILE_LOG — указание имени файла и пути к нему, в который будет произведена запись собранной информации. Если путь не указан, то запись производится в файл "cuda_profile.log" в текущей директории. При запуске на нескольких GPU можно записывать данные для каждого GPU в отдельный файл,

если использовать при указании имени файла переменную %d, обозначающую номер устройства;
- CUDA_PROFILE_CSV — включение и отключение формата записанной информации с разделением столбцов запятыми (Comma-Separated Values);
- CUDA_PROFILE_CONFIG — использование файла с конфигурацией счетчиков производительности GPU.

В GPU встроены следующие счетчики производительности: timestamp — моменты времени запуска ядер и операций копирования;
- gpustarttimestamp — время начала выполнения ядра на GPU;
- gpuendtimestamp — время завершения выполнения ядра на GPU;
- gridsize — число связок потоков в сетке по всем измерениям *X* и *Y*;
- threadblocksize — число потоков в связке по всем измерениям *X*, *Y* и *Z*;
- dynsmemperblock — размер динамически выделяемой разделяемой памяти в байтах у каждой связки потоков;
- statsmemperblock — размер статически выделяемой разделяемой памяти в байтах у каждой связки потоков;
- regperthread — число регистров на один поток;
- memtransferdir — направление копирования памяти — 0 — для операций хост→устройство, 1 — для операций устройство→хост;
- memtransfersize — объем копируемого блока данных;
- memtransferhostmemtype — тип памяти хоста, участвующий в операции копирования: со страничной организацией (pageable) или с ее блокировкой (page-locked);
- streamid — идентификатор потока при запуске ядра.

Для детализации процесса выполнения ядра поддерживаются следующие счетчики:
- local_load — число инструкций чтения из локальной памяти на 1 варп мультипроцессора;
- local_store — число инструкций записи в локальную память на 1 варп мультипроцессора;
- gld_request — число инструкций чтения из глобальной памяти на 1 варп мультипроцессора;
- gst_request — число инструкций чтения из глобальной памяти на 1 варп мультипроцессора;
- divergent_branch — число уникальных ветвлений, которые проявляются;

- branch — число уникальных инструкций ветвления в программе;
- sm_cta_launched — число связок потоков, выполняемых на мультипроцессоре.

На устройствах с вычислительной способностью 1.x имеются следующие счетчики для процесса выполнения ядра:

- gld_incoherent — число необъединенных (некогерентных) операций чтения из глобальной памяти;
- gld_coherent — число объединенных операций чтения из глобальной памяти;
- gld_32b — число 32-байтных транзакций чтения из глобальной памяти;
- gld_64b — то же, 64-байтных;
- gld_128b — то же, 128-байтных;
- gst_incoherent — число необъединенных (некогерентных) операций записи в глобальную память;
- gst_coherent — число объединенных операций записи в глобальную память;
- gst_32b — число 32-байтных транзакций записи в глобальную память;
- gst_64b — то же, 64-байтных;
- gst_128b — то же, 128-байтных;
- instructions — число выполненных инструкций;
- warp_serialize — число потоков варпа, которым приходится сериализовать операции доступа к разделяемой или константной памяти;
- cta_launched — число выполненных связок потоков;
- prof_trigger_00, ..., prof_trigger_07 — триггеры профайлера (до 8 штук), которые можно включать в пользовательском коде ядра через вызов встроенной функции __prof_trigger (int N), где N — индекс триггера;
- tex_cache_hit — число попаданий текстурного кеша;
- tex_cache_miss — число промахов текстурного кеша.

На устройствах с вычислительной способностью 2.0 и более высокой возможен одновременный сбор информации по большему числу счетчиков производительности в зависимости от типа выбранных счетчиков, кроме того, добавлены следующие счетчики:

- shared_load — число инструкций чтения из разделяемой памяти на 1 варп мультипроцессора;
- shared_store — число инструкций записи в разделяемую память на 1 варп мультипроцессора;
- inst_issued — число выполненных инструкций, включая реплеи (replays);
- inst_executed — число выполненных инструкций, исключая реплеи;
- warps_launched — число запущенных варпов на мультипроцессоре;
- threads_launched — число запущенных потоков на мультипроцессоре;
- l1_global_load_hit — число попаданий кеша L1 при чтении;
- l1_global_load_miss — число промахов кеша L1 при чтении.

Библиографический список

1. ATI Technologies [Электронный ресурс]. — Режим доступа: http://en.wikipedia.org/wiki/ATI_Technologies. — Заглавие с экрана.
2. Nvidia [Электронный ресурс]. — Режим доступа: http://en.wikipedia.org/wiki/Nvidia.– Заглавие с экрана.
3. S3 Graphics [Электронный ресурс]. — Режим доступа: http://en.wikipedia.org/wiki/S3_Graphics. — Заглавие с экрана.
4. NVIDIA Corp. NVIDIA CUDA Compute Unified Device Architecture. Programming Guide Version 1.0 [Электронный ресурс]/NVIDIA Corp. 2701 San Tomas Expressway, Santa Clara, CA 95050. — Режим доступа: http://www.nvidia.com. — Заглавие с экрана.
5. NVIDIA Corp. CUDA Technical Training [Электронный ресурс]/NVIDIA Corp. 2701 San Tomas Expressway, Santa Clara, CA 95050. — Режим доступа: http://www.nvidia.com. — Заглавие с экрана.
6. CUDA C Programming Guide [Электронный ресурс]. — Режим доступа: http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz3k3HvOyd7. — Заглавие с экрана.
7. Comparison of ATI graphics processing units [Электронный ресурс]. — Режим доступа: http://en.wikipedia.org/wiki/Comparison_of_ATI_graphics_processing_units. — Заглавие с экрана.
8. Comparison of Nvidia graphics processing units [Электронный ресурс]. — Режим доступа: http://en.wikipedia.org/wiki/Comparison_of_Nvidia_graphics_processing_units. — Заглавие с экрана.
9. Графический конвейер [Электронный ресурс]. — Режим доступа:/http://ru.wikipedia.org/wiki. — Заглавие с экрана.
10. Shaders With Ogre [Электронный ресурс]. — Режим доступа: http://www.ogre3d.ru/wik/pmwiki.php?n=Main.ShadersWithOgre. — Заглавие с экрана.

11. Reference for HLSL [Электронный ресурс]. — Режим доступа: https://msdn.microsoft.com/ru-ru/library/windows/desktop/bb509638 (v=vs.85).aspx. — Заглавие с экрана.
12. Медведев, А. Современная терминология 3D графики [Электронный ресурс]/А. Медведев. Режим доступа: http://www.ixbt.com/video2/terms2k5.shtml.
13. DirectCompute [Электронный ресурс]. — Режим доступа: https://en.wikipedia.org/wiki/DirectCompute. — Заглавие с экрана.
14. OpenGL 3 против DirectX 11: война закончена [Электронный ресурс]. — Режим доступа: http://www.thg.ru/graphic/open_gl_3_directx_11/index.html. — Заглавие с экрана.
15. Net Framework [Электронный ресурс]. — Режим доступа: http://ru.wikipedia.org/wiki/.NET_Framework. — Заглавие с экрана.
16. Common Language Runtime [Электронный ресурс]. — Режим доступа: http://ru.wikipedia.org/wiki/Common_Language_Runtime. — Заглавие с экрана.
17. NVIDIA CUDA Compiler Driver NVCC [Электронный ресурс]. — Режим доступа: http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/#axzz3k3HvOyd7. — Заглавие с экрана.
18. Немнюгин, С. А. Параллельное программирование для многопроцессорных вычислительных систем/С. А. Немнюгин, О. Л. Стесик. — СПб. : БХВ-Петербург, 2002. — 400 с.
19. Обзор видеокарты NVIDIA GeForce GTX 280 — очень быстро и жутко горячо [Электронный ресурс]. — Режим доступа: http://tfile.ru/forum/viewtopic.php?t=191087. — Заглавие с экрана.
20. GPU-accelerated libraries [Электронный ресурс]. — Режим доступа: https://developer.nvidia.com/gpu-accelerated-libraries. — Заглавие с экрана.
21. Arora, N. Direct N-body Kernels for Multicore Platforms/N. Arora, A. Shringarpure, R. W. Vuduc//Proceedings of the International Conference on Parallel Processing. — 2009. — P. 379–387.
22. Satish, N. Designing efficient sorting algorithms for manycore GPUs/N. Satish, M. Harris, M. Garland//Proceeding IPDPS ‘09 Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing. — 2009. — P. 1–10.

Оглавление

Учебное издание

Некрасов Кирилл Александрович,
Поташников Святослав Игоревич,
Боярченков Антон Сергеевич,
Купряжкин Анатолий Яковлевич

ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ ОБЩЕГО НАЗНАЧЕНИЯ НА ГРАФИЧЕСКИХ ПРОЦЕССОРАХ

Редактор И. В. Меркурьева
Верстка О. П. Игнатьевой

Подписано в печать 06.05.2016. Формат 70×100/16.
Бумага писчая. Печать цифровая. Гарнитура Newton.
Уч.-изд. л. 5,2. Усл. печ. л. 8,4. Тираж 50 экз.
Заказ 123

Издательство Уральского университета
Редакционно-издательский отдел ИПЦ УрФУ
620049, Екатеринбург, ул. С. Ковалевской, 5
Тел.: 8(343)375-48-25, 375-46-85, 374-19-41
E-mail: rio@urfu.ru

Отпечатано в Издательско-полиграфическом центре УрФУ
620075, Екатеринбург, ул. Тургенева, 4
Тел.: 8(343) 350-56-64, 350-90-13
Факс: 8(343) 358-93-06
E-mail: press-urfu@mail.ru


ИЗДАТЕЛЬСТВО УНИВЕРСИТЕТСКОГО КНИГОИЗДАНИЯ
УНИВЕРСИТЕТСКАЯ КНИГА
2015


9 785799 617226