О системе программирования вычислений общего назачения на графических процессорах

А. В. Адинец*, Н. А. Сахарных◇

Статья посвящена обсуждению вопросов программирования современных графических процессоров для решения с их помощью общих вычислительных задач. Статья начинается с общего понятия графических процессоров и вычислений на них. Далее приводится обзор типичных архитектур и основных ее особенностей, которые определяют как класс решаемых задач, так и возможности программирования. После этого дается краткий обзор существующих средств программирования для графических процессоров и описание системы, созданной авторами данной работы.

Система C$ состоит из двух частей: языка C$ и среды исполнения, выполняющей трансляцию на графический процессор. Язык C$ представляет собой Java (C#)-подобный язык, расширенный конструкциями для операций с функциями и массивами. На этапе компиляции параллельный код отделяется от непараллельного, после чего транслируется в вызовы к среде исполнения. Среда исполнения работает по принципу ленивых вычислений, строя изначально граф массивных операций и исполняя его на графическом процессоре; подобный алгоритм обусловлен особенностями целевой архитектуры. В заключение даются результаты работы алгоритма, анализ и возможные направления дальнейшего развития разработанной системы.

*Научно-исследовательский вычислительный центр МГУ
◇Факультет ВМК МГУ имени М.В.Ломоносова

1. Введение

Начиная с 2003 года, активные исследования ведутся в области использования современных графических процессорных устройств (ГПУ) для решения вычислительных задач [1]. В зарубежной литературе это направление получило название GPGPU (сокращение от General Purpose Graphics Processor Unit, графическое процессорное устройство общего назначения), в настоящей статье в качестве его перевода на русский язык будет использоваться ОВГПУ (общие вычисления на графических процессорах). В настоящее время ГПУ используются для решения ряда вычислительных задач, для которых традиционно использовались суперкомпьютерные архитектуры, например, для выполнения матричных операций [2], решения уравнений в частных производных при помощи сеточных методов [3, 4], решения задач машинного зрения [5], выполнения операций обработки изображений и звука, в т.ч. преобразования Фурье [6], трассировки лучей [7] и многих других. При грамотном использовании ресурсов графического процессора удается добиться прироста реальной производительности во много раз по сравнению с использованием ресурсов только центрального процессора (речь идет о сравнении наиболее передовых образцов графических и центральных процессоров в определенный период времени). На некоторых задачах достигается реальная производительность в сотни ГФлопс [8], которая до недавнего времени была доступна лишь на компьютерных кластерах и суперкомпьютерах. Кроме того, графические процессоры обладают меньшей стоимостью в расчете на ГФлопс пиковой производительности (порядка 1–2$/ГФлопс), а также, что особенно становится актуальным, меньшим энергопотреблением (порядка 0.5 Вт/ГФлопс) по сравнению с кластерными системами. Поэтому написание эффективных программ для графических процессоров является актуальной задачей.

В настоящее время наблюдается существенный дисбаланс между возможностями графических процессоров и их реальным использованием, вызванный прежде всего отсутствием адекватных средств программирования для данной архитектуры. В то время как для кластеров и суперкомпьютеров существует огромное количество

языков программирования (прежде всего Си и ФОРТРАН [9]) и библиотек, до недавнего времени программы на ГПУ писались исключительно с использованием интерфейсов для работы с трехмерной полигональной графикой — таких как OpenGL [10]. Поскольку эти интерфейсы предназначены для эффективной работы с графикой, писать с их помощью программы для графических процессоров неудобно. В дополнение к этому программы получаются громоздкими, сложными в отладке, и непереносимыми на архитектуры, отличные от графических процессоров. В таких условиях программирование графических процессоров требует не только понимания модели программирования, но и знания всех тонкостей интерфейса для работы с трехмерной графикой и изучения языка шейдеров, например GLSL [11], что препятствует распространению ОВГПУ среди специалистов в области параллельного программирования.

В подобных условиях становится важной задача разработки инструментария для написания программ для графических процессоров. В первую очередь это именно средства программирования, затем — средства отладки, профилировки и т.д. При этом создаваемое средство должно быть сравнительно простым в освоении для специалистов в области параллельного программирования и позволять создавать эффективные и переносимые программы.

В настоящее время направление ОВГПУ получило поддержку не только в академических кругах, но и среди производителей графических процессоров. Ведущие производители графических процессоров, AMD [12] и NVIDIA [13] уже выпустили как программные средства для взаимодействия с графическими процессорами в обход интерфейсов для работы с графикой (соответственно, Data Parallel Virtual Machine (DPVM) [14] и NVIDIA CUDA [15]), так и аппаратные ускорители, ориентированные на общие вычисления. Последние представляют собой те же графические процессоры (соответственно, ATI Radeon X1900 и NVIDIA GeForce 8800), но продаваемые как решения для ускорения вычислений общего назначения.

Данная статья организована следующим образом. В разделе «Особенности архитектуры графических процессоров» дается обзор архитектуры графических процессоров и их основных возмож-

Рис. 1. Программируемый графический конвейер

ностей. В разделе «Существующие подходы программирования» рассматриваются уже существующие подходы программирования ГПУ, дается их анализ. В разделе «Система C$: идеи и архитектура» рассматриваются идеи, положенные в основу системы C$ и их влияние на архитектуру создаваемой системы. В разделе «Трансляция операций над массивами в системе C$» рассказывается, как по дереву вычислений строится программа для ГПУ, учитывающая особенности архитектуры конкретного ГПУ. В разделе «Результаты и направления дальнейшей работы» даются результаты работы и направления, в котором в дальнейшем будет развиваться система.

2. Особенности архитектуры графических процессоров

Архитектура современных графических процессоров определяется их основной задачей — возможностью создавать реалистичные изображения в реальном масштабе времени. Традиционно архитектура графических процессоров изображается в виде графического конвейера, изображенного на Рис. 1.

Примитивы для обработки (в виде вершин, индексных буферов, задающих принадлежность вершин, а также дополнительных атрибутов вершин, таких как текстурные координаты или нормали) задаются при помощи интерфейса трехмерного программирования (например, OpenGL). После этого они передаются на графический

процессор, где отдельные вершины обрабатываются при помощи вершинных шейдеров. Шейдер — это программа для графического процессора. Вершинные шейдеры получают на вход атрибуты вершины и выдают новые атрибуты вершины; чаще всего они преобразовывают координаты и нормаль в соответствии с видовой матрицей. На этапе сборки примитивов из вершин собираются примитивы (на более современных ГПУ этот этап заменен геометрическим шейдером — который позволяет осуществлять более гибкий набор операций с примитивами, например, разбиение). Растеризация преобразует примитив (чаще всего это треугольник) в набор фрагментов — проекций этого треугольника на пиксели экрана. Для каждого фрагмента выполняется фрагментный шейдер. Основная его задача — это вычисление цвета фрагмента, и, возможно, его глубины. После этого фрагмент подается на заключительный этап — пиксельных операций. Наиболее важные из пиксельных операций — это тест глубины (выполняет аппаратное удаление невидимых поверхностей; очень эффективно реализован в большинстве современных ГПУ) и тест трафарета. Следует заметить, что если фрагментный шейдер не вычисляет значение глубины, то отсечение фрагмента по глубине происходит до работы фрагментного шейдера для этого фрагмента, экономя тем самым вычислительные ресурсы ГПУ и предоставляя довольно эффективный механизм для маскирования неактивных фрагментов в ОВГПУ.

Более подробно о графическом конвейере см., например, в [16]. Здесь же отметим только, что наибольшее значение с точки зрения ОВГПУ имеют фрагментные (или пиксельные) шейдеры. Это обусловлено рядом особенностей архитектуры графического процессора:

1. Фрагментные шейдеры позволяют осуществлять произвольный доступ к определенным областям памяти (при помощи текстур), в то время как вершинные шейдеры этого не могут.

2. Результат работы фрагментных шейдеров доступен непосредственно, в то время как результат работы вершинных шейдеров доступен только после интерполяции.

3. Фрагментные шейдеры являются узким местом работы большинства современных приложений трехмерной визуализации, поэтому архитектура графических процессоров разрабатывается прежде всего для оптимальной работы фрагментных шейдеров. Как следствие, для них обычно выделяется больше шейдерных функциональных устройств (ФУ), и они обладают более гибкими возможностями, чем вершинные шейдеры. В последних графических процессорах с унифицированной шейдерной архитектурой функциональные устройства разделяются между различными видами шейдеров, поэтому их возможности примерно одинаковы.

В настоящее время на рынке производства дискретных, т.е. не интегрированных в системную плату, ГПУ доминируют две компании — AMD (ATI) и NVIDIA. И если архитектура ГПУ предыдущего поколения (ATI X1K и NVIDIA GeForce 7) у них во многом была похожей, то текущее поколение их архитектур (ATI HD 2K и NVIDIA GeForce 8) значительно отличаются. И хотя графические процессоры обоих производителей поддерживают работу через интерфейсы программирования трехмерной графики, внутренние реализации их отличаются настолько, что для каждой приходится писать свою эффективную реализацию одного и того же фрагментного шейдера. Ниже сначала дается описание общих характеристик современных ГПУ, а затем приводится описание особенностей различных архитектур графических процессоров.

2.1. Общие характеристики современных ГПУ. Для эффективной работы фрагментных шейдеров современные графические процессоры имеют большое количество шейдерных ФУ. Их число лежит в диапазоне от 4 до 320, причем сами ФУ могут быть скалярными или работающими с 4-ками вещественных чисел. Каждое шейдерное ФУ имеет не менее 32 регистров и работает с вещественными числами одинарной точности; самые новые ГПУ поддерживают также работу с целыми числами. Поток управления либо один на все шейдерные ФУ, либо один на группу из достаточно большого (до 8) числа шейдерных ФУ (в этом случае программа у всех одна и та же). Таким образом, современные ГПУ являются параллельной

архитектурой типа ОКМД.

Все ФУ выполняют один и тот же шейдер для всех целочисленных пар координат заданного прямоугольника на плоскости со сторонами, параллельными осям координат, и результат исполнения для каждого элемента записывается в буфер в видеопамяти. Современные ГПУ позволяют записывать эти данные в промежуточный буфер без вывода на экран, равно как и поддерживать вывод в 8 буферов одновременно. Шейдер может быть сменен только после того, как текущий шейдер проработает для всех элементов прямоугольника; смена шейдера считается дорогой операцией. Для каждого исполнения шейдера координаты его выхода жестко заданы и не могут быть изменены, что позволяет оптимизировать запись результатов в видеопамять. Исполнения шейдера для различных пикселей полностью логически независимы, могут выполняться параллельно, и не существует никаких специальных средств синхронизации между исполнениями шейдера для различных пикселей. Современные ГПУ предоставляют альтернативный режим работы, в котором запись может быть осуществлена в произвольное место выходного буфера, однако не предоставляют никаких средств синхронизации подобной записи. В этом случае, однако, кэширование записи не производится, и для достижения высокой производительности приходится моделировать режим регулярной записи, только с большим количеством записываемых элементов.

Каждое ФУ может выдавать команды на чтение данных из оперативной памяти. Чтения полностью асинхронны и исполняются на специальных устройствах, именуемых текстурными устройствами. В зависимости от типа и режима работы, доступна либо вся память, либо набор из не более чем 32 сегментов (буферов). В последнем случае загрузка сегментных регистров производится до выполнения шейдера, и во время его выполнения они изменены быть не могут. Адресация может быть либо линейной, либо двумерной. Чтение может быть кэшированным или некэшированным (см. описания ГПУ от конкретных производителей).

Области памяти, адресуемые сегментными регистрами, могут располагаться как в собственной памяти графического процессо-

ра (видеоОЗУ), так и в ОЗУ. ВидеоОЗУ современных ГПУ имеет размер от 128 до 1024 МБ. Доступ к видеоОЗУ осуществляется со скоростью от 25 до 80 ГБ/сек, доступ к ОЗУ ЦПУ на порядок медленнее. Выгрузка данных обратно из видеоОЗУ в ОЗУ ЦПУ еще медленнее и осуществляется на скорости до 1 ГБ/сек. Таким образом, обработку данных следует организовывать таким образом, чтобы программы работали только с данными в видеоОЗУ.

При программировании ГПУ операции доступа к памяти считаются медленными операциями: обмен данными с памятью может занимать до 400 тактов, в то время как за 1 такт одно ФУ может выполнить до 8 арифметических операций. Для сокрытия расходов на взаимодействие с памятью используется аппаратная многопоточность. Потоки ГПУ намного менее гибки, чем традиционные потоки; каждый из них состоит из определенного числа одновременно обрабатываемых фрагментов (от 4 до 48), и хранит весь контекст внутри ГПУ, что позволяет переключать потоки без потери времени (реально — за 1 такт). Как следствие, количество потоков ограничено размером массива регистров общего назначения (РОН) внутри ГПУ и обратно пропорционально числу используемых в шейдере регистров. Еще одним следствием такого механизма является отсутствие программного контроля переключения потоков, так как все операции переключения осуществляются аппаратно с целью повышения быстродействия. Аппаратура выполняет переключение потоков лишь в том случае, если один из них заблокировался на операции с памятью.

За счет полной независимости исполнений шейдера для различных фрагментов ГПУ достигают достаточно большой степени параллелизма и значительной производительности в терминах вещественных вычислений. Например, графический процессор X1950 XTX с 48 шейдерными ФУ имеет пиковую производительность 300 ГФлопс при вещественных вычислениях с одинарной точностью, что сравнимо с производительностью современных многомашинных комплексов и на порядок превосходит производительность самого мощного современного центрального процессора.

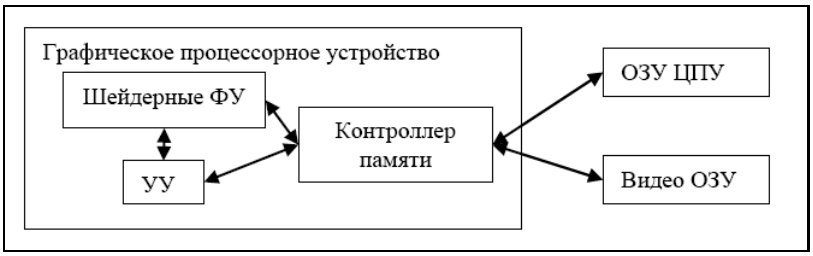

Рис. 2. Архитектура графического процессора компании AMD

2.2. ГПУ компании AMD (ATI). Общая схема архитектуры графического процессора компании AMD приведена на рисунке 2. Данная схема описывает как ГПУ серии X1K, так и ГПУ серии HD 2K. Хотя HD 2K на уровне шейдерных ФУ имеет иную архитектуру, сходств между семействами значительно больше, чем различий.

ГПУ семейства X1K имеют от 4 до 48 шейдерных ФУ, каждое из которых обрабатывает четверки вещественных чисел (ОКМД ФУ). ФУ способны выполнять вещественные арифметические операции, а также вычисление элементарных функций (только для скалярных величин) и команды управления; кроме того, они могут выдавать запросы на чтение/запись контроллеру памяти. Каждому исполнению шейдера доступны до 128 128-битных вещественных регистров. Чтение данных может производиться из 16 двумерных буферов размера до 4096×4096 элементов, расположенных в видеоОЗУ или ОЗУ. Каждый элемент может быть либо вещественным числом, либо четверкой вещественных чисел; в качестве буфера глубины поддерживаются буферы из 16-битных беззнаковых целых чисел. Запись возможна в один из 4-х целевых буферов, также поддерживается режим произвольной записи. Чтение и запись кэшируются; однако даже в случае попадания в кэш стоимость одного чтения может составлять до 4-х тактов, что делает понижение общего количества доступов в память важной задачей оптимизации.

ГПУ семейства HD 2K [17] имеют от 4 до 64 шейдерных ФУ, построенных по архитектуре с длинным командным словом. Каждое

шейдерное ФУ имеет в себе 5 подустройств, каждое из которых за такт способно выполнить 2 арифметические команды с вещественным числом. Помимо этого, 5-ое ФУ также может вычислять элементарные функции и выполнять специальные команды. Поддерживаются команды для работы с целыми числами, а также команды управления. Количество регистров осталось прежним, однако появилась возможность косвенной адресации как самих регистров, так и номера текущего буфера чтения. Размеры буферов увеличились до 8192×8192, число входных буферов — до 32, а число выходных буферов — до 8. Появились новые возможности, поддерживающие интерфейс программирования трехмерной графики DirectX 10. Однако в остальном архитектура изменилась мало, и длинное командное слово, хотя и является более гибким, чаще используется для работы с теми же четверками вещественных чисел.

2.3. ГПУ компании NVIDIA. В отличие от AMD, шейдерные ФУ в ГПУ серии GeForce 8 от NVIDIA являются суперскалярными, а не ОКМД, и работают они с отдельными вещественными и целыми числами, а не с четверками чисел. С одной стороны, такое аппаратное решение позволяет повысить тактовую частоту ФУ. С другой стороны, это облегчает написание шейдеров для работы на таких процессорах, и как следствие — упрощает программирование ФУ. Еще одним плюсом, не относящимся напрямую к архитектуре, является возможность использовать диалект C CUDA для программирования ГПУ NVIDIA. И хотя программирование ведется по старой схеме загрузки данных и исполнения шейдеров, написание шейдеров на языке C значительно облегчает задачу.

Второй особенностью ГПУ NVIDIA является замена кэш-памяти на явно адресуемую пользователем статическую память, скорость обращения к которой примерно соответствует скорости обращения к регистру. Подобное решение также применяется, например, в процессоре CELL [18]. Работа с видеоОЗУ, таким образом, становится некэшируемой. С одной стороны, такое решение дает дополнительные возможности по оптимизации, но с другой — усложняет создание эффективных программ на подобных архитектурах.

Подводя итог по архитектурам, можно сказать, что задача со-

здания эффективных средств построения высокоуровневых и переносимых программ на ГПУ является, с одной стороны, актуальной, а с другой — достаточно сложной и интересной. Ведь при создании эффективного кода требуется учитывать различные особенности различных архитектур ГПУ — такие как статическую память, ОКМД ФУ и другие, при этом эти детали желательно по возможности скрыть от конечного пользователя, или предоставить в понятных ему терминах. Кроме того, возникает вопрос о переносимости подобных программ на другие архитектуры — например, на многоядерные процессоры, процессоры типа CELL и т.д., тем более что во многом эти архитектуры похожи и во многих случаях они используются для решения одних и тех же классов задач (например, обработка изображений).

3. Существующие подходы программирования

При рассмотрении ОВГПУ шейдер можно рассматривать как элемент программы графического процессора, а текстуры и буфера, с которыми он работает — просто как двумерные массивы данных. Таким образом, шейдер можно рассматривать как:

- как функцию («ядро»), которая перерабатывает порцию элементов входного массива, а на выходе дает элемент другого массива,
- как функцию, которая применяется к входным массивам и в результате дает выходной массив,

причем в обоих случаях эта функция не имеет побочного эффекта.

Традиционно для описания ГПУ применялся первый подход, который получил название «потоковое программирование» (streaming programming) [19]. Идея этого подхода состоит в следующем. Программа разбивается на ядра, т.е. небольшие программные элементы со строго определенными входами и выходами. Ядра обрабатывают потоки входных элементов. Одно ядро может иметь один или несколько потоков входных элементов. На каждый входной элемент

ядро генерирует столько-то выходных элементов. При отображении на графический процессор потоки отображаются на текстуры, а ядра — на шейдеры. Для ядер с условной генерацией выходных элементов используются возможности маскирования ГПУ за счет буфера глубины. Доступ к нескольким элементам одного и того же потока осуществляется при помощи операций типа scatter и gather; при этом операция gather отображается в чтение соседних элементов текстуры, а операция scatter — в дополнительный проход, в котором результаты работы фрагментного шейдера подаются на вход вершинному шейдеру (который может изменять позицию записи).

Этот подход хорошо работал, когда размеры шейдеров были ограничены (не более 20 команд), и накладывались ограничения на чтение из текстуры. Это подходило для простых алгоритмов; для более сложных алгоритмов и для более сложных шейдеров (например, с циклами), подход не работает. Например, для того, чтобы сформулировать в нем алгоритм трассировки лучей, его приходится разбивать на элементарные операции [20]. Непонятно в том числе и то, как в терминах таких операций выразить, например, операцию умножения матриц.

Поэтому со временем более популярным стал второй подход — шейдер есть функция, которая применяется к набору входных массивов и выдает выходной массив. Эта функция не имеет побочного эффекта и может быть вычислена независимо (как следствие, параллельно) для различных элементов выходного массива.

Но как определить такую функцию? Самый простой способ заключается в следующем: есть элементарные операции с массивами (например, сложение, умножение и другие), они комбинируются, и полученное арифметическое выражение рассматривается как функция. Поскольку смена шейдера влечет за собой накладные расходы, появляется стремление поместить как можно больше операций обработки в один шейдер. Вычисления массивных операций в таком случае осуществляется ленивым образом: пока результат явно не нужен, для него строится дерево, а когда результат реально востребован (например, он приводится к типу языкового массива из специального класса, реализующего массивы на ГПУ), то по дере-

ву строится шейдер, он компилируется, исполняется, а результат загружается обратно.

По такому принципу устроены библиотеки Accelerator [21], Peak Stream [22] и Rapid Mind [23]. По сути, они предлагают некоторую библиотеку массивных операций над вещественными числами, которая использует ленивые вычисления и динамическую компиляцию. Кроме того, подобные библиотеки можно делать переносимыми между различными системами — например, Rapid Mind работает также на CELL [18] и на многоядерных системах.

4. Система C$: идеи и архитектура

Система C$ [24] заимствует идеи ленивого исполнения и динамической трансляции у уже существующих систем, и одновременно предлагает целый ряд нововведений.

Во-первых, для программирования ГПУ предлагается использовать не библиотеку, а язык программирования. При этом, хотя на архитектуру языка и оказала влияние архитектура современных ГПУ, сам язык является машинно-независимым и в принципе может быть перенесен на другие архитектуры, в том числе CELL и многоядерные системы. Синтаксис нового языка и большинство конструкций заимствованы из языка C# (и Java), что должно облегчить освоение. А разработка языка с использованием среды исполнения .NET облегчает интеграцию его в уже существующие системы. Специфические объекты языка (такие как функции или массивы) в других .NET-языках, таких как C#, будут видны просто как классы, а специфические языковые конструкции — как вызовы методов.

Процесс вычисления на графическом процессоре можно представить как *параллельное независимое вычисление функции без побочных эффектов на ее области определения*. Ведь один вызов шейдера является вычислением функции (шейдера) на его области определения (экранном прямоугольнике). Функция может быть достаточно простой, как в случае сложения двух массивов, или, наоборот, достаточно сложной, как в случае трассировки лучей [7]. Кроме того, для вычисления одного элемента выходного массива может использоваться много элементов входного массива — например, целая

строка или столбец, как в случае умножения матриц.

Поэтому в язык C$ введено как понятие функции без побочного эффекта, так и понятие типа функции без побочного эффекта, или функционального типа. Сам по себе функциональный тип является абстрактным классом. Поэтому значением переменной функционального типа может быть ссылка на объект одного из производных классов: массив, метод класса, элементарная операция или уже сконструированное, но еще не вычисленное выражение. Любое такое значение называется *функциональным объектом*. Таким образом, массив в языке C$ представляет собой частный случай функции.

Программа на языке C$ состоит из этапов, на каждом из которых выполняется построение такой функции. Параллельное вычисление осуществляется за счет вычисления функции на всей ее области определения.

Основной операцией, используемой для конструирования новых функций в C$, является операция суперпозиции функций. В отличие от других языков, где для суперпозиции используется специальная функция или обозначение, в C$ суперпозиция осуществляется за счет конструкции функционального продвижения. Она работает следующим образом: пусть имеется функция h типа T5(T1, T3), т.е. функция h принимает на вход одно значение типа T1 и одно значение типа T3, и возвращает значение типа T5. Пусть также имеются функции g типа T3(T4) и f типа T1(T2). Тогда запись h(f, g) в языке C$ задает функция типа T5(T2, T4) (если T2 и T4 - разные типы) или T5(T2) (если это — один и тот же тип). При этом значение этой функции может быть вычислено следующим образом: функция передает свои аргументы в функции f и g, результаты их работы, в свою очередь, передает в функцию h, после чего возвращает значение, полученное из h.

Функциональное продвижение используется как дополнительное средство разрешения перегрузки функции — когда прочие средства, такие как приведение типов, не могут найти подходящую функцию.

Таким образом, любая функция может применяться не только к скалярным значениям, но и к другим функциям — становясь функцией более высокого порядка (в терминах функционального про-

граммирования).

Поскольку массивы в языке C$ являются частным случаем функций, за счет суперпозиции к массивам можно применять арифметические операции и скалярные функции. Таким образом, в языке C$ появляются возможности массивно-ориентированного программирования.

Пример использования функционального продвижения в языке C$ приведен в листинге 1.

```
float sin(float x) { ... } // функция синус
float[] g, l; // целочисленный массив
float(float) m; // функция float -> float; может быть как методом, так и ассоциативным массивом
...
var h = g + sin; // var выполняет автоматический вывод типа переменной; тип float(int, float)
var t = l * g; // тип float (int)
var w = m + sin; // тип float (float)
```

Листинг 1. *Пример функционального продвижения в C$*

Элементы объектно-ориентированного и функционального программирования в языке C$ сосуществуют. При этом обращения к полям или к свойствам трактуются как функции, которые принимают один параметр (объект класса) и возвращают одно значение (объект типа поля класса). Как следствие, операция «.» участвует в суперпозиции функций и может применяться к функциям. Пример этого приведен в листинге 2.

```
final class float3 {float x, y, z; } // класс трехмерного вектора
float3 [] vs; // массив вершин
var tt = vs.x; // функция, которая принимает на вход целое число, а возвращает элементы x векторов
var ll = tt * vs.y + g; // см. выше; тип float (int)
```

Листинг 2. *Пример функционального продвижения с использованием доступа к члену в C$*

В языке C$ вводится понятие **простого неизменяемого класса** — это класс, от которого нельзя наследовать, поля которого нельзя изменять после создания объекта класса и ссылка на который не может быть **null**. Также требуется, чтобы поля объекта этого класса не могли ссылаться на объекты этого же класса, прямо или косвенно. Такие классы могут использоваться совместно с функциями без побочных эффектов. Кроме того, массивы объектов таких классов можно хранить не как массивы ссылок, а как массивы самих объектов. Для объявления такого класса перед ключевым словом **class** нужно поставить модификатор **final.** Если при этом требуется, чтобы класс был обычным ненаследуемым, а не простым неизменяемым, то перед ним нужно поставить модификатор **mutable**.

Идея введения подобного класса не нова — в .NET, например, уже существуют типы-структуры (в противовес типам-классам), хотя они не являются неизменяемыми. Идея неизменяемых классов, состоящих только из простых типов, также используется в языках X10 [25] и Titanium [26], и для тех же целей — обеспечить возможность хранения массивов объектов как массивов самих объектов, а не массивов ссылок.

В языке также есть возможность вводить классы, которые наследуют от функциональных типов. Путем переопределения метода вычисления функции можно, например, связать такой класс с файлом — тогда собственно чтение из файла не произойдет до того момента, как эта функция будет достигнута при ленивом вычислении. Если операция чтения дополнительно переопределена, например, для случая, когда на вход подается массив подряд идущих индексов, это позволит выполнять ее более эффективно и прочитать только ту часть файла, которая действительно необходима.

Операция редукции реализована как применение двуместной функции с типом T (T, T) к функции, которая возвращает тип T. Если такая функция помечена как коммутативная или ассоциативная, то при редукции может изменяться порядок вычислений. В дальнейшем возможно введение в язык операции обобщенной редукции — любой операции, которая берет на вход массив и возвращает скаляр, при этом может быть вычислена в цикле за время O(n) (n — размер

массива) с O(1) памяти.

С точки зрения компилятора, редукция, как и суперпозиция, является способом разрешения перегрузки функций.

В языке C$ любой функциональный объект имеет ряд атрибутов — одним из них является домен, или область определения. В качестве домена может выступать любая функция (хотя чаще всего это декартово произведение целочисленных промежутков). Операция редукции может быть выполнена над функцией только в том случае, функция имеет конечный домен — в противном случае генерируется ошибка. Проверка конечности производится на этапе выполнения.

Для того, чтобы показать, что при вычислении 2 измерения массива должны пробегаться параллельно, а не независимо, может быть использована конструкция, называемая *связанной переменной*. Связанная переменная — это специальная переменная, которая имеет тип, но не имеет определенного значения; она связывает измерения (параметры) различных функций. Область, пробегаемая переменной, вычисляется исходя из доменов функций, в которых она применяется, и явных ограничений на нее (вида i:f, где i — связанная переменная, а f — функция). Связанные переменные не обязательно объявлять явно — если они не объявлены, то их тип выводится из их первого применения. Если связанная переменная присутствует только внутри редукционной конструкции, то по ней осуществляется редукция. Пример использования операции редукции и связанных переменных для умножения матриц приведен в листинге 3.

```
type matrix = float(int, int); // псевдоним типа
matrix mul(matrix a, matrix b) {
  var c(j, k) = + (a(j, l) * b(l, k));
  return c;
}
```

Листинг 3. *Пример умножения матриц в языке C$*

В данном случае унарный + выполняет редукцию (в C$ нет стандартного унарного оператора +) по связанной переменной l, а пере-

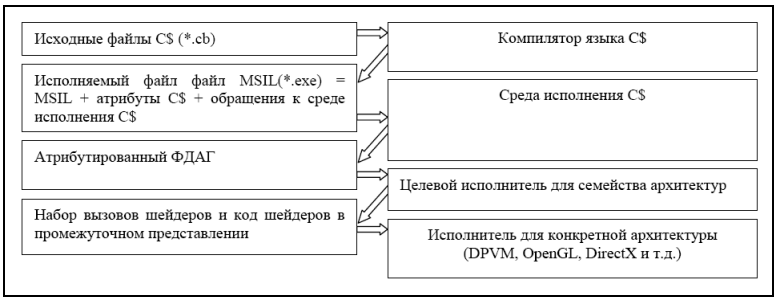

Рис. 3. Архитектура графического процессора компании AMD

менные j и k остаются, т.к. они объявлены вне оператора редукции. Для введения какой-либо переменной в выражение (функцию) может использоваться конструкция типа **let**. Для удобства введены операции **sum**() вместо редукционного + и **prod**() вместо редукционного *. Кроме того, в языке в качестве операций представлены **min** и **max**, которые также используются как редукционные.

В терминах императивного программирования связанные переменные представляют собой цикл без заголовка; заголовок им не нужен, поскольку область определения связанной переменной вычисляется автоматически. В терминах функционального программирования связанные переменные аналогичны comprehension-конструкциям — с той разницей, что в C$ нет специальных списочных типов.

Использование связанных переменных позволяет программировать задачи с большей вычислительной мощностью, а значит, более полно использовать возможности графического процессора. Например, операция сложения двух больших массивов, скорее всего, не позволить достигнуть более 2% пиковой производительности ГПУ, т.к. она имеет малую вычислительную мощность. С другой стороны, операция умножения матриц позволяет выполнить $O(n^3)$ вычислительных операций на $O(n^2)$ входных данных, что позволит достигать до 50–60% пиковой производительности ГПУ.

Общая архитектура системы C$ изображена на рисунке 3.

Компилятор языка преобразует языковые конструкции в вызовы к среде исполнения C$. При этом исходный код делится на обычный и параллельный в терминах C$. Код считается параллельным в терминах C$, если он содержит использование функционального продвижения, редукции или связанных переменных. Таким образом, решается задача выделения параллельных элементов программы, которые можно отображать в код для ГПУ. Операция вычисления функции на всем домене выражается как приведение функции к массивному типу (т.е. типу с квадратными скобками). Исходный код на языке C$ транслируется в промежуточное представление — в настоящее время это специальное разработанное нами представление, однако планируется для этой цели использовать Microsoft Intermediate Language (MSIL) — представление промежуточного кода в .NET.

Обращения к среде выполнения C$ создают направленный ациклический граф (ДАГ) функциональных операций. Он также называется функциональным ДАГ (ФДАГ). В настоящее время этот граф имеет 3 типа листовых вершин: скаляры, функции, связанные переменные (в т.ч. и пустые), и 3 типа внутренних вершин: применения функции, редукции и введения новый связанных переменных.

После машинно-независимых оптимизирующих преобразований ФДАГ передается целевому исполнителю для семейства архитектур, например, для ГПУ. На этом этапе выбираются способы хранения данных и отображения кода на целевую архитектуру, а также производится основной набор машинно-зависимых оптимизаций. Сама программа из ФДАГ переводится в набор шейдеров на промежуточном языке. И хотя этот набор шейдеров еще не является программой, исполняемой на аппаратуре, он настолько близок к ней, что работа, выполняемая на следующем этапе (исполнителя для конкретной архитектуры) является скорее механической.

Наконец, самым нижним уровнем нашей системы является исполнитель для конкретной архитектуры. Основной его задачей является предоставление стандартного интерфейса уровню семейства архитектур. Этот интерфейс включает в себя механизм управления памятью, механизм трансляции кода из промежуточного представления в машинный, и механизм сообщения о возможностях устрой-

ства. К последним относится, например, максимальное количество доступных для чтения или записи буферов, максимальное количество доступных регистров и другие особенности.

Работа с памятью тоже распределена по уровням системы, как и работа с трансляцией программы. Уровень системы времени выполнения работает с функциями и прямоугольными массивами — например, с массивами вещественных чисел. Он знает ранг и домен массива, но ничего не знает о том, как он хранится в памяти. Наиболее важным уровнем является уровень исполнителя для семейства архитектур. Его задача состоит в том, чтобы эффективно представить исходный массив произвольной размерности в памяти устройства для организации в дальнейшем его эффективной обработки. На этом уровне, например, решаются вопросы отображения исходных массивов в массивы размерности, поддерживаемой аппаратурой, например, размерности 2, как на большинстве ГПУ. Кроме того, решается задача синхронизации данных; это позволяет минимизировать количество дорогостоящих операций обмена между ОЗУ и видеоОЗУ. Наконец, уровень исполнителя для конкретной архитектуры решает задачу управления физической памятью.

5. Трансляция операций над массивами в системе C$

Трансляция массивных операций осуществляется в 3 этапа: средой времени выполнения, целевым исполнителем для семейства и целевым исполнителем для конкретной архитектуры. При этом среда времени выполнения осуществляет машинно-независимую оптимизацию — такую как удаление повторяющихся выражений и свертку констант. Также она определяет списки измерений в каждой вершине ФДАГ. Самое важное, на этом этапе для каждой вершины графа вычисляется ее домен. Впоследствии он используется для выделения память под целевую функцию и определения области ее вычисления.

Размеченный таким образом граф поступает на вход целевому исполнителю для семейства архитектур ГПУ, который преобразует

его в последовательность шейдеров. Его работа состоит из нескольких этапов:

1. Деление исходного графа на проходы. Каждый проход соответствует исполнению одного шейдера.

2. Выбор отображения данных и кода на целевую архитектуру. Здесь выбирается схема представления данных в физической памяти устройства, а способ физической реализации массивных операций.

3. Генерация индексов для операций чтения из памяти. Поскольку при этом должны учитываться отображения исходного и целевого массива, а также промежуточных операций, эта генерация вынесена в отельный этап.

4. Генерация шейдерного кода в промежуточном представлении по ФДАГ.

При этом массивные операции отображаются на арифметические операции в шейдере, а операции редукции — либо на редукцию вдоль измерений буфера памяти, если ее результатом является скаляр, либо на цикл внутри шейдера, если результатом ее является другая функция.

Сгенерированный шейдерный код подается на вход уровню исполнителя для конкретной архитектуры. Его задача — преобразовать код из этого представления в код, пригодный для исполнения на целевом ГПУ.

На рис. 4 приведено дерево, которое строится по коду умножения матриц, приведенному в листинге 3. Используемые типы вершин: применения функции @, редукции R. При этом в скобках у вершин редукции указаны связанные переменные, по которым идет редукция. Связанные переменные и массивы помечены буквами. На рис. 5 приведен код, полученный из этого графа выражений для умножения матриц размером 1024 на 1024. Обратите внимание, что код был сгенерирован с учетом того, что ГПУ работает с четверками вещественных чисел. Операция редукции была автоматически

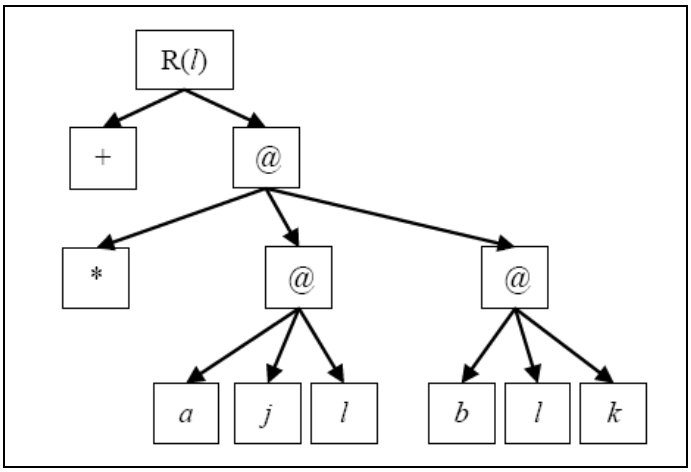

Рис. 4. ДАГ операций, построенный по коду перемножения матриц

преобразована в двойной цикл, т.к. максимальное число итераций цикла на ГПУ (255) меньше, чем половина размера матрицы. Матрица представлена буфером из 512×512 четверок вещественных чисел, при этом каждая из четверок — это подматрица размером 2×2. Следует сказать, что более эффективный (в 2 раза) код, генерируемый нашей системой, еще в два раза длиннее, и не приведен здесь по причине экономии места.

6. Результаты и направления дальнейшей работы

Часть системы C$ была реализована на языке C# 2.0 для работы в среде выполнения .NET 2.0. Доступ к графическому процессору осуществлялся средствами ATI DPVM [14]. Текущая реализация системы имеет следующие ограничения по сравнению с описанными выше идеями:

- Поддерживаются только простые функции (т.е. операции с простыми типами — float, int) и массивы.

```
ps_3_0
  dcl vPos.xy
  dcl_2d s0
  dcl_2d s1
  mov r1.x, vPos.x
  mov r1.y, c1.x
  mov r2.x, c1.x
  mov r2.y, vPos.y
  mov r0, c0.zzzz
  rep i1
  rep i0
    texld r3, r1, s0
    texld r4, r2, s1
    add r1.y, r1.y, c0.y
    add r2.x, r2.x, c0.y
    mul r5, r3.xxzz, r4.xyxy
    mad r6, r3.yyww, r4.zwzw, r5
    add r0, r0, r6
  endrep
  endrep
  mov oC0, r0
```

Рис. 5. Шейдерный код, сгенерированный по ДАГ на рис. 4

- Поддерживаются только домены, представляющие собой декартово произведение целочисленных промежутков, либо полные домены.

- Трансляция осуществляется в специфическое промежуточное представление, а не в код MSIL. Это, однако, планируется исправить.

- Не поддерживаются вершины введения дополнительных переменных (let-вершины). Однако поддерживаются редукционные вершины и вершины суперпозиции.

В остальном же функциональность текущей системы реализована в соответствии с высказанными выше идеями.

Система была протестирована на коде умножения матриц, приведенном в листинге 3. Тестирование проводилось на матрицах различных размеров и с различными настройками оптимизации. В таблице 1 приведены результаты (производительность в Гфлопс) для различных настроек оптимизации. При этом матрица представлялась в виде простого двумерного массива (1×1), в виде массива подматриц 2×2 и массива подматриц 4×4. Кроме того, для сравнения приводится производительность ручной реализации процедуры sgemm на ассемблере из примеров AMD для DPVM.

Размер матрицы Настройки оптимизации	128×128	256×256	512×512	1024×1024
float, $1 * 1$	3.94	5.00	5.16	4.61
float4, $2 * 2$	8.10	14.20	15.66	15.81
float4, $4 * 4$	9.03	24.13	29.16	29.35
ручная оптимизация	—	—	30	31

Табл. 1. Производительность программы умножения матриц на матрицах различных размеров при различных настройках оптимизации

На основании результатов, приведенных в таблице 1, можно сделать ряд выводов. Во-первых, с увеличением размера матрицы производительность вычислений растет. Это можно объяснить двумя причинами. С одной стороны, запуск шейдера на ГПУ требует фиксированных накладных расходов, таких как коммуникации между ЦПУ и ГПУ, установки регистров данных и команд, и других. С другой стороны, и это более важно, в самом шейдере есть участок начальной инициализации (инициализация индексов, переменных редукции) и циклический участок. С ростом размера матрицы растет

доля циклического участка в общем числе команд, а значит, возрастает производительность.

Во-вторых, оптимизации, связанные с повышением вычислительной мощности сгенерированного кода, приносят значительный выигрыш. Трехкратный рост производительности наблюдается при переходе от тривиального алгоритма к алгоритму с блоками 2×2. Двукратный рост наблюдается при переходе к блокам размера 4×4. Следует заметить, что максимальный размер блока ограничен общим числом выходных элементов, поэтому увеличивать размер блока дальше нет возможности. При этом рост производительности при переходе к блокам 2×2 обусловлен как повышением отношения количества арифметических операций к количеству чтений, так и повышением эффективности использования арифметических операций. Рост же при переходе от 2×2 к 4×4 обусловлен только повышением отношения количества арифметических операций к чтению из памяти, потому он и меньше.

Результаты, приведенные в таблице 1, уже сами по себе неплохи. В лучшем случае достигнута примерно половина пиковой производительности, и, что самое важное - почти 100% производительности, достигнутой при помощи ручной оптимизации. Дальнейшее развитие системы может идти по нескольким направлениям.

Во-первых, это расширение гибкости системы C$. Прежде всего это возможность использовать определенные пользователем функции, а также простые неизменяемые классы. Это позволит применять систему для решения более широкого класса] задач.

Во-вторых, это расширение набора целевых архитектур, на которых система C$ сможет работать. Помимо поддержки OpenGL и DirectX, которые хороши прежде всего возможностью работать с практически любыми ГПУ, большой интерес представляют процессоры семейства NVIDIA GeForce 8. Для их эффективного программирования необходимо разработать методы работы с промежуточной статической памятью. В более далекой перспективе интересны многоядерные системы и процессоры CELL. Кроме этого, потребуется более широкий набор оптимизаций для нового поколения ГПУ AMD.

В-третьих, это расширение системы C$ на целевые архитектуры с несколькими графическими процессорами. Эта задача становится актуальной с возрастанием распространенности подобных систем, созданных на основе ГПУ от AMD и NVIDIA. Кроме того, специализированные решения для высокопроизводительных вычислений от этих компаний чаще всего оснащаются ГПУ от AMD и NVIDIA.

Список литературы

[1] GPGPU.org. http://www.gpgpu.org/.

[2] Fatahalian, K., Sugerman, J. и Hanrahan, P. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication// ACM Press, 2004. P. 133–137. ISBN ÎSSN:1727–3471, 3-905673-15-0.

[3] Bolz, Jeff, etc. Sparse matrix solvers on the GPU: conjugate gradients and multigrid// ACM Press, 2005. ACM SIGGRAPH 2005.

[4] Goodnight, Nolan, etc. A multigrid solver for boundary value problems using programmable graphics hardware// Proceedings of SIGGRAPH/EUROGRAPHICS Workshop On Graphics Hardware. P.102–111. ISBN ÎSSN:1727–3471, 1-58113-739-7.

[5] Farrugia, J.P., etc. GPUCV: A framework for image processing acceleration with graphics processors // Proceedings of IEEE International Conference on Multimedia & Expo (ICME 2006).

[6] Mitchel, Jason L., Ansari, Marvan Y., Hart, Evan. Advanced Image Processing with DirectX 9 Pixel Shaders. ATI Technologies Inc. Shader X2. 2004.

[7] Adinetz, Andrew V., Berezin, Sergey B. Implementing Classical Ray Tracing on GPU - a Case Study of GPU Programming// Proceedings of Graphicon'06. P.1–7. ISBN 5-89407-262-X.

[8] Belleman, Robert G., Bedorf, Jeroen, Portegies Zwart, Simon F. High Performance Direct Gravitational N-Body Simulations on Graphics Processing Units. Elsevier Preprint. 16 июля 2007 г.

[9] Wikipedia. Fortran. http://en.wikipedia.org/wiki/Fortran.

[10] OpenGL.org. http://www.opengl.org.

[11] Kessenich, John, Baldwin, Dave, Rost, Randi. The OpenGL Shading Language. 7 September 2006.

[12] AMD Inc. ATI Web Site. http://ati.amd.com/.

[13] NVIDIA Corporation. http://www.nvidia.com/page/home.html.

[14] Peercy, Mark, Segal, Mark, Gerstmann, Derek. A performance-oriented data parallel virtual machine for GPUs// Proceedings of ACM SIGGRAPH 2006 Sketches. ISBN: 1-59593-364-6.

[15] NVIDIA Corporation. NVIDIA CUDA Complete Unified Device Architecture. 12 February 2007.

[16] Segal, Mark, Akeley, Kurt. The OpenGL (R) Graphics System: A Specification (Version 2.1). Edited by Pat Brown. 1 December 2006.

[17] Persson, Emil. ATI Radeon TM HD 2000 Programming Guide. June 2007.

[18] IBM Corporation. Cell Broadband Engine Architecture. 3 October 2006.

[19] Buck, Ian, etc. Brook for GPUs: stream computing on graphics hardware// ACM Press, 2004. P.777–786.

[20] Foley, Tim, Sugerman, Jeremy. KD-tree acceleration structures for a GPU raytracer// Proceedings of SIGGRAPH/EUROGRAPHICS Workshop On Graphics Hardware. P.15–22. ISBN:1-59593-086-8.

[21] Tarditi, David, Puri, Sidd, Oglesby, Jose. Accelerator: using data parallelism to program GPUs for general-purpose uses// Proceedings of the 12th international conference on Architectural

support for programming languages and operating systems. P.325–335. SESSION: Embedded and special-purpose systems.

[22] PeakStream Inc. http://www.peakstreaminc.com/.

[23] Rapid Mind Inc. http://www.rapidmind.net/.

[24] Adinetz, Andrew V. C$ Project Web Site. http://www.codeplex.com/cbucks.

[25] Saraswat, Vijay. Report on the Experimental Language X10. 8 декабря 2006.

[26] Bonachea, Dan, etc. Titanium Language Reference Manual. Berkeley, California, USA. Август 2006 г.