О ПРОГРАММИРОВАНИИ ВЫЧИСЛЕНИЙ ОБЩЕГО НАЗНАЧЕНИЯ НА ГРАФИЧЕСКИХ ПРОЦЕССОРАХ

Адинец А. В., Сахарных Н. А.

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 процессорных устройствах). 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 т.д. При грамотном использовании ресурсов ГПУ удается добиться прироста в 10 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 250 – 300 ГФлопс, которая до недавнего времени была доступна лишь на компьютерных кластерах и суперкомпьютерах.

В настоящее время наблюдается существенный дисбаланс между возможностями ГПУ 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 ФОРТРАН [8]) и библиотек, до недавнего времени программы на ГПУ писались исключительно с использованием интерфейсов для работы с трехмерной полигональной графикой – таких как OpenGL [9]. Поскольку эти интерфейсы предназначены прежде всего 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 ГПУ требует не только понимания модели программирования, но и знания всех тонкостей интерфейса для работы с трехмерной графикой и изучения языка шейдеров, например GLSL [10], что препятствует распространению ОВГПУ среди специалистов в области параллельного программирования.

В подобных условиях важной становится задача разработки инструментария для написания программ для ГПУ. 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. Высокая производительность по относительно небольшой цене, которая может быть достигнута за счет использования ГПУ для решения реальных задач, делает задачу создания подобных средств разработки актуальной.

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

ОСОБЕННОСТИ АРХИТЕКТУРЫ ГРАФИЧЕСКИХ ПРОЦЕССОРОВ

Архитектура современных ГПУ обусловлена их основной задачей – возможностью создавать реалистичные изображения в реальном масштабе времени. Об архитектуре можно более подробно прочитать, например, в [15]. Здесь же отметим, что программирование ГПУ возможно при помощи шейдеров - программ обработки элементов трехмерных данных. Вершины исходной геометрии обрабатываются вершинными шейдерами, а фрагменты

(проекции исходного треугольника на элементы экранной решетки) – фрагментными (пиксельными) шейдерами. По ряду причин, наибольшее значение с точки зрения ОВГПУ имеют фрагментные шейдеры:

1. Фрагментные шейдеры позволяют осуществлять произвольный доступ к определенным областям памяти на чтение (при помощи текстур).
2. Результат работы фрагментных шейдеров доступен непосредственно.
3. Работа именно фрагментных шейдеров максимально оптимизирована на современных архитектурах ГПУ, так что их использование позволяет получить более высокую производительность.

Для эффективной работы фрагментных шейдеров современные графические процессоры имеют большое количество шейдерных ФУ – их число может доходить до 64. Каждое шейдерное ФУ работает с четверками вещественных чисел и имеет до 128 собственных 128-битных регистров. Каждое ФУ способно выполнять по 2 арифметических операции за 1 такт. Помимо арифметических, в систему команд входят команды доступа к памяти и команды управления (ветвления и циклы). Поток управления один на все шейдерные ФУ; таким образом, ГПУ являются параллельной архитектурой типа ОКМД. Размер одного шейдера на большинстве современных ГПУ ограничен числом от 512 до 1024 команд.

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 от 4 до 8 буферов одновременно. Шейдер может быть сменен только после того, как текущий шейдер проработает для всех элементов прямоугольника; соответственно, 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, и исполнять шейдеры для различных пар координат независимо.

Каждое ФУ может выполнять произвольные чтения из памяти при помощи одного из 16 сегментных регистров (с точки зрения графического интерфейса доступных как текстурные устройства). Каждый регистр может адресовать область памяти размером до 4096*4096 элементов; в качестве элементов обычно могут выступать целые (8 или 16 бит) или вещественные (32 бит) числа, а также пары и четверки таких чисел. Логически эта область представляется как двумерный массив, и потому всегда адресуется в двумерных координатах. Иногда накладывается дополнительное ограничение – такая область (в двумерном представлении) должна быть квадратной и иметь размер, равный степени двойки (это упрощает трансляцию адресов). Сегменты памяти (с точки зрения интерфейса графического программирования – текстуры) могут перекрываться. Загрузка адресов сегментов (равно как и адреса выходного буфера) в сегментные регистры выполняется до исполнения шейдера; внутри шейдера эти адреса не могут быть изменены.

Области памяти, адресуемые сегментными регистрами, могут располагаться как в собственной памяти графического процессора (видеоОЗУ), так и ОЗУ. ВидеоОЗУ современных ГПУ имеет размер от 128 до 512 МБ. Доступ к видеоОЗУ осуществляется на скорости от 25 до 50 ГБ/сек; доступ к ОЗУ ЦПУ на порядок медленнее. Выгрузка данных обратно из видеоОЗУ в ОЗУ ЦПУ еще медленнее и осуществляется на скорости до 100 МБ/сек, что препятствует обработке на ГПУ массивов данных, которые не помещаются в ГПУ.

Каждое из шейдерных ФУ может выдать команду доступа к памяти через заданный сегментный регистр и двумерный адрес; при этом ФУ не блокируется, если у следующей команды не установлен флаг ожидания данных, что позволяет совмещать арифметические операции и обмен с памятью по времени. Команда доступа к памяти передается на одно из устройств доступа к памяти, где и обрабатывается. Имеется кэш данных первого уровня, его размер обычно 16 – 32 КБ. При программировании ГПУ операции доступа к памяти считаются медленными операциями; для их ускорения используется аппаратная многопоточность. Потоки ГПУ намного менее гибки, чем традиционные потоки; каждый из них состоит из определенного числа одновременно обрабатываемых фрагментов (обычно его размер совпадает с числом шейдерных ФУ), и хранит весь контекст внутри ГПУ, что позволяет переключать потоки без потери времени (реально – за 1 такт). Как следствие, количество потоков ограничено размером массива РОН внутри ГПУ и обратно пропорционально числу используемых в шейдере регистров; еще одним следствием является отсутствие программного контроля переключения потоков. Переключение потоков происходит лишь в том случае, если один из них заблокировался на операции с памятью.

Устройство управления (УУ ГПУ) управляет планированием потоков и назначением фрагментов на шейдерные ФУ. Оно является программируемым и имеет свой небольшой набор команд, который позволяет загружать

сегментные регистры, устанавливать текущий шейдер, запускать и ожидать завершения его работы, а также управлять размещением константных и работой кэша. Данное описание архитектуры приведено на основании устройства ATI DPVM [(13)] - прочие современные ГПУ имеют сходную архитектуру, хотя количественные характеристики могут различаться. Самые новые ГПУ являются скалярными (т.е. ориентированы на работы не с четверками чисел, а с отдельными вещественными числами) – однако каждое шейдерное ФУ имеет 4 независимых ФУ для работы с вещественными числами, и для их загрузки все равно придется либо использовать команды для работки с четверками чисел, либо программную конвейеризацию обработки [16].

СУЩЕСТВУЮЩИЕ ПОДХОДЫ ПРОГРАММИРОВАНИЯ

При рассмотрении ОВГПУ шейдер можно рассматривать как элемент программы графического процессора, а буфера, с которыми он работает – как двумерные массивы данных. Таким образом, шейдер можно рассматривать как (в обоих случаях функция не имеет побочного эффекта):

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

Традиционно для работы с ГПУ применялся первый подход – «потоковое программирование» (streaming programming) [17]. В этом подходе программа разбивается на ядра – небольшие программные элементы со строго определенными входами и выходами. Ядра обрабатывают потоки входных элементов. На каждый входной элемент ядро генерирует один или более (может и ни одного) выходной элемент. 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 scatter отображается в чтение соседних элементов текстуры, а операция scatter – в дополнительный проход, в котором результаты работы фрагментного шейдера подаются на вход вершинному (который может изменять позицию записи).

Этот подход хорошо работал, когда размеры шейдеров были ограничены (не более 20 команд), и налагались ограничения на чтение из текстуры. Он хорошо работал для простых алгоритмов; для более сложных алгоритмов и для более сложных шейдеров (например, с циклами), он не работает. Например, для того, чтобы сформулировать в нем алгоритм трассировки лучей, его приходится разбивать на элементарные операции [18]. Непонятно также, как в терминах таких операций выразить, например, операцию умножения матриц.

Поэтому со временем более популярен стал второй подход – шейдер есть функция, которая применяется к набору входных массивов и выдает выходной массив; при этом 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 – есть элементарные операции с массивами (например, сложение, умножение и т.д.), 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. Таким образом, вычисления массивных операций в таком случае осуществляется ленивым образом – пока результат явно не требуется, для него строится дерево, а когда результат реально требуется (например, он приводится к типу языкового массива из специального класса, реализующего массивы на ГПУ), по дереву строится шейдер, он компилируется, исполняется, а результат загружается обратно.

По такому принципу устроены библиотеки Accelerator [19], Peak Stream [20] и Rapid Mind [21]. 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 [22] и на многоядерных системах.

СИСТЕМА C$: ИДЕИ И АРХИТЕКТУРА

Система C$ [23] заимствует идеи ленивого исполнения и динамической трансляции у уже существующих систем. С другой стороны, она предлагает ряд нововведений.

Во-первых, и это важный момент, 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 (по крайней мере, применение одного шейдера) можно представить как *параллельное независимое вычисление функции без побочных эффектов на ее области определения* (которой в данном случае служит экранный прямоугольник). Соответственно, язык C$ вводит понятие функции и функционального типа без побочного эффекта. Функциональный тип является абстрактным классом – значением переменной функционального типа может быть массив (который есть частный случай функции в языке), функция – член класса, элементарная операция или уже сконструированное, но еще не вычисленное выражение. Программа на языке (или ее этап) выполняет построение такой функции, а параллельное вычисление осуществляется за счет вычисления функции на всей ее области определения, например, путем приведения функции к массиву.

Основной операцией, используемой для конструирования новой функции в C$, является операция *суперпозиции.* В C$ суперпозиция осуществляется за счет конструкции *функционального продвижения* – которая позволяет применять функцию к другим функциям, типы возвращаемых значений которых соответствуют типам параметров функции. С точки зрения транслятора это работает при разрешении перегрузки – если ее не удалось разрешить стандартных средств, компилятор выполняет функциональное продвижение. Суперпозиция покрывает, как следствие, и операции над массивами данных, а ленивая семантика вычисления оставляет необходимый простор для оптимизации.

В языке возможны, например, следующие конструкции:

```
float sin(float x) { … } // функция синус
float[] g = …, l = …; // целочисленный массив
float(float) m = …;  // функция-отображение
var h = g + sin; // автовывод типов + функциональное продвижение; 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$.

Объектная ориентированность и функциональная ориентированность в языке интегрируются. С одной стороны, операция . (точка) обращения к члену может участвовать в суперпозиции, например:

```
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$.

Язык также вводит понятие простого неизменяемого класса – это финальный класс, который не может содержать ссылок на самого себя (т.е. имеет поля либо простых типов, либо таких же финальных классов, причем без возможности рекурсии) – и не имеет модификатора mutable (объекты классов без модификаторов mutable не могут иметь побочных эффектов). Такие классы могут использоваться совместно с функциями без побочных эффектов (например, класс вектора выше).

С другой стороны, язык может 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, например, по идущему подряд массиву значений, переопределена, то это позволит выполнять ее более эффективно и прочитать только ту часть файла, которая действительно необходима.

Другими применениями суперпозиции является карринг, пустые параметры (вместо параметра передается пустое место; в этом случае считается, что по этому измерению может быть произведено функциональное продвижение), продвижение приведений типов и арифметических операций.

Операция редукции реализована как применение двуместной функции с прототипом вида T (T, T) к функции, которая возвращает тип T. Если такая функция помечена как коммутативная или ассоциативная, то при редукции может изменяться порядок вычислений. С точки зрения компилятора редукция, как и суперпозиция, является еще одним средством разрешения перегрузки, которое имеет более высокий приоритет, чем карринг, но более низкий, чем пользовательские операции. 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.

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, если она производится по конечному домену – 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, используется конструкция, называемая *связанной переменной*. Связанная переменная – это специальная переменная, которая имеет тип, но не имеет определенного значения; она связывает измерения (параметры) различных функций, и аналогична параметру цикла, только без заголовка – 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. Например, операция умножения двух матриц записывается следующим образом:

```
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 по связанной переменной *l*, а переменные *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.

Использование связанных переменных позволяет программировать задачи большей вычислительной мощности, 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 (за счет использования различных оптимизаций) до 25 – 60% пиковой производительности ГПУ.

Общая архитектура системы изображена на рисунке 1.

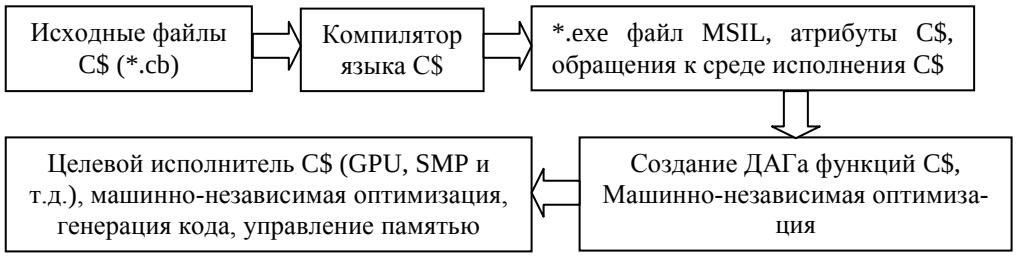

Рис. 1. Общая архитектура системы C$

Компилятор языка преобразует языковые конструкции в вызовы к среде исполнения C$. При этом исходный код делится на обычный и «параллельный». Код считается «параллельным», 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). Обычный код транслируется в некоторое промежуточное представление – в настоящее время это просто промежуточный код, 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. 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 (apply), редукции (reduce) и введения новый связанных переменных (let).

После машинно-независимых оптимизирующих преобразований ДАГ передается целевому исполнителю, который преобразует его в машинный код (код ГПУ), исполняет и возвращает результат.

Следует заметить, что выделением физической памяти занимается целевой исполнитель; он же занимается отображением логического массива на физические адреса памяти графического процессора. Однако целевой исполнитель должен обеспечивать стандартный интерфейс работы с областью памяти – так называемым представлением – который позволяет загружать в это представление и считывать из него обратно данные, обеспечивая при этом независимость работы с памятью от целевой архитектуры.

ТРАНСЛЯЦИЯ ОПЕРАЦИЙ НАД МАССИВАМИ В СИСТЕМЕ C$

Трансляция массивный операций осуществляется в 2 этапа: средой времени выполнения и целевым исполнителем. При этом среда времени выполнения осуществляет машинно-независимую оптимизацию – такую как удаление повторяющихся выражений и свертку констант. Также система времени исполнения определяет списки свободных измерений и связанных переменных. Самый важный момент – на этом этапе для каждой вершины графа вычисляется ее домен (т.е. домен функции, которой соответствует эта вершина) – он используется в дальнейшем для

Размеченный таким образом граф подается на целевой исполнитель для графического процессора, который выполняет генерацию кода. Она состоит из нескольких этапов:

1. На первом этапе исходный ДАГ разбивается на участки, соответствующие отдельным шейдерам. В настоящее время разбиение идет по вершинам, которые соответствуют операциям редукции, в результате которых получаются скалярные значения.
2. Для каждого участка строится ДАГ шейдерных операций.
3. По ДАГ шейдерных операций осуществляется генерация кода шейдера. При генерации учитываются параметры, заданные в файле конфигурации. Например, можно задать генерацию операций для четверок вещественных чисел, повышая тем самым производительность, или задать параметры развертывания цикла.

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.

При этом сразу же генерируется код на шейдерном ассемблере, выполняется выделение регистров. После этого создается буфер команд для исполнения шейдера, после чего командный буфер подается на исполнение графическому процессору. После исполнения командного буфера данные, полученные в результате, будут доступны в языке как функция.

На рис. 2 приведено дерево, которое строится по коду умножения матриц, приведенному в листинге 3. Типы вершин – @ (apply), R (reduce; в скобках указаны связанные переменные, по которым производится редукция). Связанные переменные и массивы помечены буквами. На рис. 3 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 преобразована в двойной цикл, т.к. максимальное число итераций цикла на ГПУ – 255 – меньше, чем половина размера матрицы.

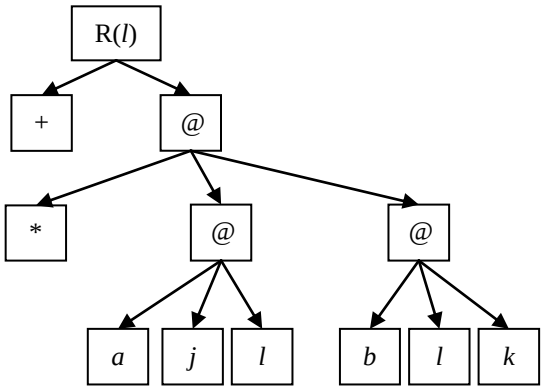

Рис. 2. ДАГ операций, построенный по коду перемножения матриц из листинга 3

```
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. 3. Шейдерный код, сгенерированный под ДАГ на рис. 2.

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 [13]. 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
- 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 настроек оптимизации, отличающихся по 2-м параметрам: использованию возможностей по работе с четверками вещественных чисел – они используются (float4) или нет (float), а также по конвейеризации цикла – цикл конвейеризируется (pipeline) или нет (simple). Замеры проводились на гра-

фическом процессоре ATI Radeon X1800 XL с 16 шейдерными ФУ, работающими на частоте 550 МГц, и 256 МБ памяти.

	128 x 128	256 x 256	512 x 512	1024 x 1024
float, simple	3.94	5.00	5.16	4.61
float, pipeline	3.76	4.72	4.79	4.66
float4, simple	8.10	14.20	15.66	15.81
float4, pipeline	7.53	13.78	14.79	15.22

Табл. 1. Производительность (Гфлопс) 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 (за исключением первых двух строк последнего столбца) производительность вычислений растет. Быстрый рост при переходе с малых размеров на большие может быть объяснен тем, что на запуск программы на графическом процессоре требуются некоторые накладные расходы (см. выше) – но с увеличением размера матрицы их вклад быстро затухает, т.к. время собственно обработки матрицы растет примерно как куб ее размера по одному из измерений.

Во-вторых, программная конвейеризация цикла почти всегда дает небольшое снижение производительности (за исключением первой половины последнего столбца). Скорее всего, это связано с многопоточным механизмом современных ГПУ (см. выше) – конвейеризация цикла требует дополнительных регистров, соответственно, это снижает количество потоков и возможность ГПУ скрывать за счет многопоточной работы задержки при обращениях к памяти. Как следствие – небольшое падение производительности.

В-третьих, заметно, что использование обработки четверок вещественных чисел дает примерно трехкратный прирост производительности, что вызвано не столько тем, что арифметические команды выполняются быстрее, сколько тем, что повышается вычислительная мощность выполняемой программы – удвоение линейного размера умножаемых подматриц (т.е. переход от 1x1 к 2x2) повышает количество обращений к памяти в 4 раза (на итерацию), а количество арифметических операций – в 8 раз, что приводит к росту вычислительной мощности в 2 раза.

Результаты, приведенные в таблице 1, уже сами по себе неплохи – достигнута примерно четверть пиковой производительности в наилучшем случае. С другой стороны, остается еще простор для оптимизации – производительность полностью оптимизированного кода для перемножения матриц, поставляемого вместе с DPVM, в 2 – 2.5 раза больше. Дальнейшая работа будет идти по нескольким направлениям.

Во-первых, в ходе работы выяснилось, что методы оптимизации для графического процессора зависят в основном от параметров самого графического процессора (числа шейдерных ФУ, наличия или отсутствия в нем геометрических шейдеров, числа буферов, в которые одновременно возможна запись и т.д.), чем от интерфейса доступа к нему. Как следствие, имеет смысл ввести дополнительный уровень абстракции между целевым исполнителем и аппаратурой – он будет заворачивать интерфейс доступа к железу (DirectX, OpenGL и т.д.) и предоставлять его целевому исполнителю в единообразном виде.

Во-вторых, на ГПУ можно выделить несколько разнородных измерений как при хранении данных (четверки вещественных чисел, измерения буфера, несколько буферов), так и при исполнении (четверки вещественных чисел, циклы, развертки и конвейеризации циклов, измерения выходного буфера, множественность выходных буферов и т.д.). Некоторые из них соответствуют реальному параллелизму ГПУ (например, операции с четверками чисел), некоторые – нет, но могут увеличить производительность за счет повышения вычислительной мощности (например, использование множества выходных буферов). В любом случае размеры этих физических измерений на различных ГПУ различны, некоторые есть не на всех ГПУ (например, GeForce 8800 работает не с четверками вещественных чисел, а со скалярами), так что для разных ГПУ задачу отображения логических измерений хранения и выполнения данных необходимо решать по-разному. Приведение этой задачи к единому виду и возможность параметризировать генерацию конечного кода шейдера различным отображением на измерения с оптимизацией по различным метрикам (например, по вычислительной мощности) могло бы позволить ставить проблему оптимизации для такого рода отображения и генерировать более высокоэффективный код.

В-третьих, следует рассматривать не только целочисленные домены, но и домены других типов, а также целочисленные домены, которые не представляются в виде декартова произведения. Особый интерес представляют

аффинные индексные выражения, так как в этот класс попадает до 99% индексных выражений в реальных программах. В этом случае интересно рассматривать домены как наборы линейных ограничений, для которых определены операции пересечения, проверки на непустоту, проекции на набор измерений, а также операция выделения наибольшего подмножества, представимого в виде декартова произведения целочисленных промежутков – это позволило бы расширить класс задач, поддерживаемых нашей системой.

ЛИТЕРАТУРА:

1. *GPGPU.org*. - основной ресурс, посвященный ОВГПУ. http://www.gpgpu.org/.
2. *Understanding the efficiency of GPU algorithms for matrix-matrix multiplication.* Fatahalian, K., Sugerman, J. и Hanrahan, P. Гренобль // ACM Press, 2004. стр. 133 - 137. ISBN ~ ISSN:1727-3471 , 3-905673-15-0.
3. *Sparse matrix solvers on the GPU: conjugate gradients and multigrid.* Bolz, Jeff, и др. Лос-Анджелес // ACM Press, 2005. ACM SIGGRAPH 2005.
4. *A multigrid solver for boundary value problems using programmable graphics hardware.* Goodnight, Nolan, и др. Сан-Диего, Калифорния // Ассоциация Eurographics, 2003. SIGGRAPH/EUROGRAPHICS Семинар по графическому аппаратному обеспечению, стр. 102 - 111. ISBN ~ ISSN:1727-3471 , 1-58113-739-7.
5. *GPUCV: A framework for image processing acceleration with graphics processors.* Farrugia, J. P., и др. Торонто, Онтарио, Канада // IEEE, 2006. IEEE International Conference on Multimedia & Expo (ICME 2006).
6. Mitchel, Jason L., Ansari, Marvan Y. и Hart, Evan. Advanced Image Processing with DirectX 9 Pixel Shaders. ATI Technologies Inc // *Shader X2.* 2004.
7. *Implementing Classical Ray Tracing on GPU - a Case Study of GPU Programming.* Adinetz, Andrew V. и Berezin, Sergey B. Новосибирск, 2006 // труды конференции Graphicon'06. стр. 1 - 7. ISBN 5-89407-262-X.
8. Wikipedia - Fortran. http://en.wikipedia.org/wiki/Fortran.
9. OpenGL.org. основной сайт OpenGL. http://www.opengl.org.
10. Kessenich, John, Baldwin, Dave и Rost, Randi. *The OpenGL Shading Language.* Спецификация языка шейдеров GLSL. 7 сентября 2006 г.
11. AMD Inc. *ATI Web Site.* Сайт подразделения ATI компании AMD Inc. http://ati.amd.com/.
12. NVIDIA Corporation. Сайт компании NVIDIA. http://www.nvidia.com/page/home.html.
13. *A performance-oriented data parallel virtual machine for GPUs.* Peercy, Mark, Segal, Mark и Gerstmann, Derek. Бостон, Массачуссетст // ACM Press, 2006 . ACM SIGGRAPH 2006 Sketches. ISBN:1-59593-364-6.
14. NVIDIA Corporation. NVIDIA CUDA Complete Unified Device Architecture - спецификация архитектуры прямого доступа к ГПУ производства NVIDIA 12 февраля 2007 г.
15. Segal, Mark и Akeley, Kurt. The OpenGL (R) Graphics System: A Specification (Version 2.1). - спецификация интерфейса программирования трехмерной графики OpenGL, версия 2.1. [ред.] Pat Brown. 1 December 2006 г.
16. Persson, Emil. ATI Radeon TM HD 2000 Programming Guide. Руководство по программированию ГПУ ATI Radeon HD 2000. Июнь 2007 г.
17. *Brook for GPUs: stream computing on graphics hardware.* Buck, Ian, и др. Лос-Анджелес, Калифорния // ACM Press, 2004. стр. 777 - 786.
18. *KD-tree acceleration structures for a GPU raytracer.* Foley, Tim и Sugerman, Jeremy. Лос-Анджелес, Калифорния // ACM Press, 2005 . SIGGRAPH/EUROGRAPHICS Семинар по графическому аппаратному обеспечнию. стр. 15 - 22. ISBN:1-59593-086-8.
19. *Accelerator: using data parallelism to program GPUs for general-purpose uses.* Tarditi, David, Puri, Sidd и Oglesby, Jose. Сан-Хосе, Калифорния, США // ACM Press, 2006. Труды 12-ой международной конференции по архитектурной поддержке языков программирования и операционных систем. стр. 325 - 335. Секция: Встраиваемые системы и системы специального назначения.
20. PeakStream Inc. Сайт компании PeakStream Inc. http://www.peakstreaminc.com/.
21. Rapid Mind Inc. Сайт компании Rapid Mind Inc. http://www.rapidmind.net/.
22. IBM Corporation. *Cell Broadband Engine Architecture.* Архитектура процессора CELL // Корпорация IBM, 3 октября 2006 г.
23. Адинец Андрей. Сайт проекта C$. http://www.codeplex.com/cbucks.