МЕТОДЫ ОПТИМИЗАЦИИ ПРОГРАММ ДЛЯ СОВРЕМЕННЫХ ГРАФИЧЕСКИХ ПРОЦЕССОРОВ

А. В. Адинец, М.А. Кривов

ВВЕДЕНИЕ

Современные графические процессорные устройства (ГПУ) являются мощными специализированными вычислительными устройствами, которые могут быть использованы для ускорения решения целого ряда вычислительно сложных задач [1]. К таким задачам относятся матричные операции, сеточные методы, задачи обработки изображений, преобразование Фурье, машинного зрения, финансовых расчетов и т.д. [2]. На таких задачах удается получить значения реальной производительности ГПУ до 300 Гфлопс на одной машине, обеспечивая ускорение порядка 10 раз по сравнению с использованием ресурсов только центрального процессора. Помимо высокой производительности, ГПУ обладают рядом других преимуществ: низкое энергопотребление (около 0.5 Вт/ГФлопс), низкая стоимость (около $0.4/ГФлопс), высокая доступность.

Широкому распространению использования ГПУ для решения вычислительно сложных задач препятствует отсутствие адекватных средств программирования. Некоторое время назад единственным средством программирования ГПУ являлись интерфейсы программирования трехмерной графики, например, OpenGL [3], и шейдерные языки программирования. Ситуация изменилась с появлением средств программирования ГПУ, не связанных с программированием трехмерной графики. Такие средства можно разделить на библиотеки потокового программирования и библиотеки низкоуровневого программирования Первые используют подход метапрограммирования для написания программы на ГПУ, которые транслируются в шейдеры во время выполнения. Примером такой библиотеки является RapidMind [4]. Вторые создаются производителями ГПУ и работают только с ГПУ этого производителя. Примерами таких библиотек является CUDA (C Unified Driver Architecture) компании NVidia [5] и CAL (Computation Abstraction Layer) компании AMD [6]. Эти библиотеки содержат верхний и нижний уровень. Нижний уровень предназначен прежде всего для разработчиков компиляторов и средств программирования, и представляет собой ассемблер ГПУ, а также менеджер памяти и средства взаимодействия с ГПУ. Верхний уровень представляет собой диалект языка C, отражающий архитектурные особенности ГПУ данного производителя.

С одной стороны, низкоуровневые библиотеки позволили программировать на ГПУ в привычных программисту терминах. С другой стороны, они остаются низкоуровневыми средствами: написание с их помощью эффективных программ требует знаний архитектуры конкретного ГПУ и ручной оптимизации. При этом производительность простого и оптимизированного кодов может отличаться на порядок.

В настоящее время существует 2 основных производителя ГПУ для высокопроизводительных вычислений: AMD и NVidia. Их архитектуры [7] существенно отличаются. Поэтому если использовать низкоуровневые средства программировать, потребуется осваивать как минимум две различных системы программирования и иметь две версии программного кода для решения задачи. С появлением ГПУ Larrabee от Intel, который будет иметь архитектуру, отличную от уже существующих, потребуется иметь уже 3 версии программного кода. Как следствие, разработка на ГПУ будет затруднена, а переносимость созданного кода ограничена.

Для создания переносимого кода требуется наличие эффективных систем высокоуровневого программирования. Такими системами могли бы быть потоковые библиотеки. Однако, согласно [8], они не выполняют оптимизаций. Поэтому переносимость написанного на них кода будет без сохранения эффективности, что приведет к тем же проблемам, что и при использовании низкоуровневых библиотек.

Поэтому автоматическая оптимизация программ на ГПУ имеет критическое значение при создании высокоуровневых систем программирования для графических процессоров. Данная статья посвящена методам оптимизации программ, используемых в системе C$ [9]. Статья организована следующим образом. В разделе «архитектура ГПУ и система C$» дается краткий обзор особенностей архитектур современных ГПУ и устройства системы C$. В следующем разделе, «этапы трансляции и их особенности», более подробно рассматриваются этапы трансляции в C$, исполняемые на уровне целевого исполнителя для семейства устройств ГПУ. В следующих двух разделах, «Задача выбора отображения» и «Генерация кода на ГПУ» рассматриваются методы оптимизации, применяемые на двух этапах трансляции, наиболее критичных с точки зрения получения эффективного кода. Наконец, в разделе «Результаты и выводы» приводятся результаты работы программ в системе C$.

АРХИТЕКТУРА ГПУ И СИСТЕМА C$

Подробное описание архитектуры ГПУ можно найти, например, в [10]. Современные ГПУ являются вычислительными архитектурами SIMD-типа. Высокая пиковая производительность достигается за счет наличия

большого количества синхронно работающих *потоковых процессоров* (ПП). Число ПП в современных ГПУ может доходить до 200. Все ПП одновременно исполняют одну и ту же программу для ГПУ – *шейдер*. ПП являются RISC-процессорами и могут выполнять операции только с операндами, находящимися на их регистрах общего назначения (РОН), число которых может доходить до 128. Система команд ПП включает арифметические операции с целыми и вещественными числами, вычисление элементарных функций, операции обращения в память, а также операции переходов. Арифметические команды в ПП могут быть скалярными (у NVidia) или векторными (у AMD), т.е. работать с четверками чисел. В последнем случае использование векторных команд позволяет ускорить программу в несколько раз по сравнению с использованием скалярных команд. Схема архитектуры ГПУ семейства HD 2K приведена на рисунке 1.

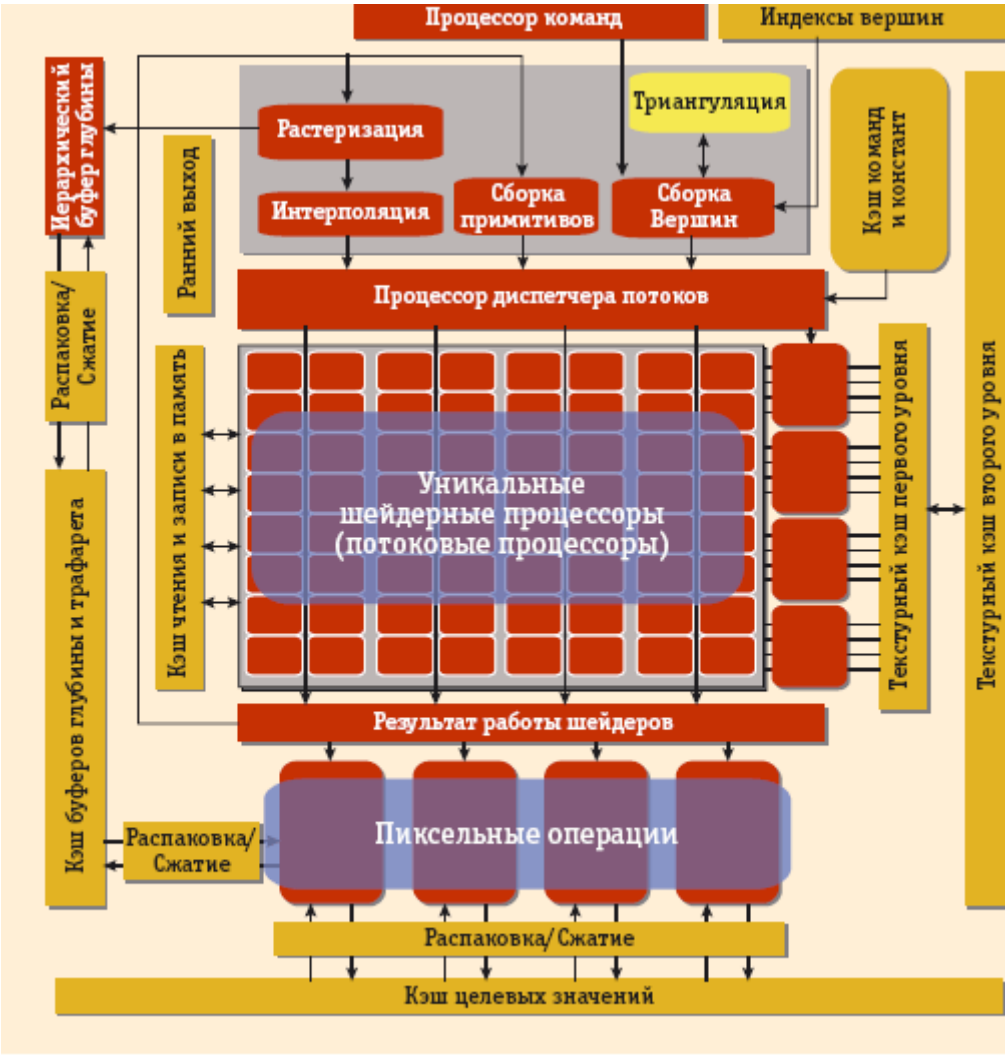

Рис. 1. Схема архитектуры графического процессора AMD Radeon HD 2K/3K.

Помимо собственно ГПУ, видеокарта содержит также интерфейс соединения с материнской платой (как правило, сейчас это PCI-Express), а также микросхемы оперативной памяти, которая называется *видеоОЗУ* (или видеопамятью). Доступ к видеоОЗУ на порядок быстрее, чем к ОЗУ ЦПУ, поэтому для эффективного использования ГПУ данные, с которыми работает шейдер, необходимо размещать в видеоОЗУ.

Дисбаланс между скоростью работы процессора и ОЗУ присутствует на большинстве вычислительных архитектур. В случае ГПУ он стоит еще более остро: за один такт один ПП может выполнить от 3 до 8 арифметических операций, в то время как задержка к доступу к видеоОЗУ может составлять до 300 тактов. Для более эффективного использования ресурсов ГПУ применяется ряд приемов. Во-первых, операции доступа к ОЗУ являются асинхронными. Во-вторых, одновременно в очереди выполнения стоит большое число *потоков ГПУ*. Их число может достигать нескольких тысяч, т.е. намного превосходить число доступных ПП. Если один из потоков блокирует-

ся по доступу к памяти, исполнение переключается на другой поток, при этом переключение контекста занимает только 1 такт. В-третьих, каждому из ПП доступен большой объем памяти для хранения временных данных. На ГПУ AMD это большое количество РОН, общий объем которых достигает 2048 байт на поток. На ГПУ NVidia потоки объединены в блоки, и одному блоку потоков доступно до 16 Кбайт разделяемой явно адресуемой *статической памяти*, которая может быть использована для хранения многократно используемых данных из видеоОЗУ. Практика показывает, что эффективное использование такой памяти для хранения временных данных позволяет на порядок повысить эффективность выполнения некоторых программ. Поэтому автоматизация оптимизации программ с использованием такой памяти является актуальной задачей.

Система C$ [9] является системой высокоуровневого программирования ГПУ. Подробное описание системы можно посмотреть, например, в [7]. Система состоит из языка программирования C$ и системы времени выполнения. Система времени выполнения обеспечивает представление программы на ГПУ в виде графа операций, его преобразование, а также трансляцию в программу для ГПУ. Кроме того, она обеспечивает управление памятью, предоставляя массив данных C$ как объект, представление которого может изменяться в зависимости от решаемой задачи, и который может находиться как в ОЗУ ЦПУ, так и в видеоОЗУ.

Язык C$ представляет собой Java/C#-подобный язык с некоторыми расширениями. В языке вводится абстрактный функциональный тип данных – базовый тип чистой функции. Конкретными типами, наследующими от него, являются функции-выражения, массивы C$, а также функции-члены классов. При помощи конструкций суперпозиции функций, связанных переменных, а также вызова функции по образцу могут определяться новые функции, которые могут вычисляться во всех точках области определения при помощи операции приведения функции к массиву. При этом такие специальные функциональные операции транслируются в код, исполняемый на графическом процессоре. Все остальные конструкции языка транслируются в промежуточный язык MSIL среды исполнения Microsoft .NET, и исполняются на ЦПУ. Использование .NET также позволяет облегчить интеграцию C$ с программами, написанными на других языках.

Общая схема системы C$ приведена на рисунке 2.

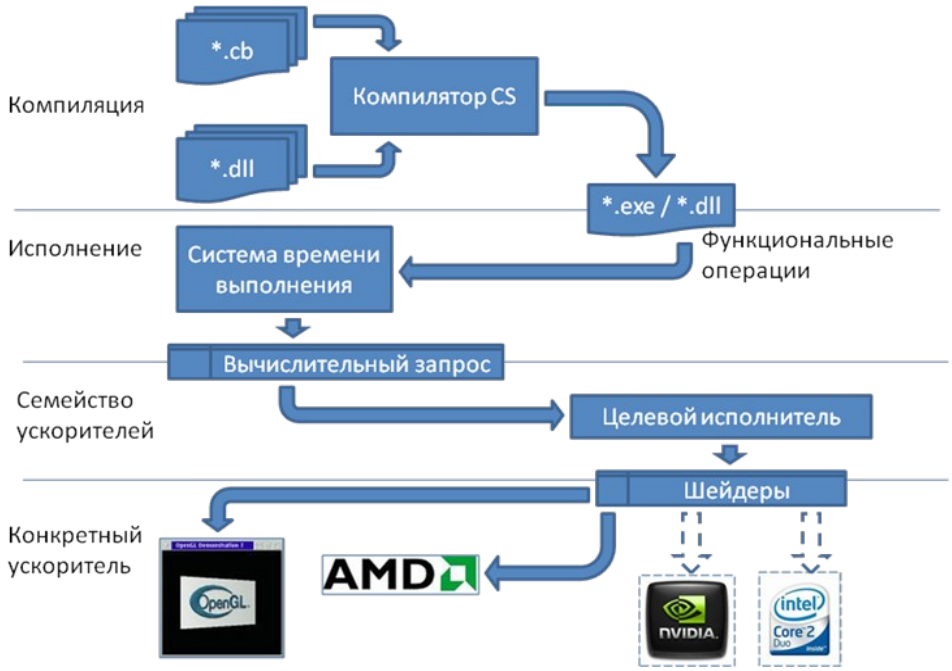

Рис. 2. Общая архитектура системы C$.

Основная работа по трансляции и оптимизации кода выполняется на этапе целевого исполнителя для семейства ускорителей («Семейство ускорителей» на рис. 2). Помимо этого, низкоуровневая оптимизация, например, распределение регистров или переупорядочивание команд, может выполняться на уровне конкретного ускорителя. Однако в данной статье мы будем касаться только высокоуровневой оптимизации.

ЭТАПЫ ТРАНСЛЯЦИИ НА ГПУ, ИХ ОСОБЕННОСТИ

Уровень целевого исполнителя для семейства ускорителей принимает на вход граф функциональных операций [7], а на выходе получает программу для ГПУ. Эта программа имеет вид сценария исполнения, в котором до-

ступны операции запуска шейдера на ГПУ с определенным набором параметров, выделение и освобождение памяти, а также передача данных из ОЗУ в видеоОЗУ и обратно. Основные этапы трансляции программы на уровне целевого исполнителя:

1. **Разбиение исходного графа на подграфы, соответствующие различным проходам.** В дальнейшем каждый подграф транслируется отдельно.
2. **Выбор отображения для данных и для кода.** Для кода выбирается и для данных выбирается их отображение на архитектуру ГПУ. При этом, например, один логический массив данных может отображаться на несколько физических буферов ГПУ, а операция редукции в исходном коде – на цикл, каждая итерация которого обрабатывает блок из 4-х элементов.
3. **Генерация кода для ГПУ.** После того, как отображения выбраны, по ним осуществляется генерация кода на ГПУ. В частности, осуществляется вычисление индексов для буферов ГПУ и ликвидация повторяющихся чтений.
4. **Преобразования сгенерированного кода.** На этом этапе производится слияние арифметических выражений в векторные команды и свертка констант.

Поскольку отдельные выражения, вычисляемые на ГПУ, относительно невелики, а проход требует большой вычислительной мощности шейдера, первый этап относительно прост. Чаще всего результатом его работы является единственный подграф, совпадающий с исходным графом. Основным критерием разбиения на проходы является повторное использование данных. Разбиение осуществляется только в том случае, если без него не удастся избежать многократного вычисления одних и тех же данных.

Самыми сложными с точки зрения реализации являются этапы 2 и 3, их описанию посвящены два следующих раздела. Этап 4 относительно прост и выполняется при помощи арифметических преобразований. Возможность слияния арифметических выражений в векторные операции обеспечивается эффективным выбором отображений на этапах 2 и 3.

ЗАДАЧА ВЫБОРА ОТОБРАЖЕНИЯ

Функциональный граф C$ определяет следующие типы вершин:

Листовые вершины:

1. **Константы**.
2. **Связанные переменные.** По сути, аналогичны параметрам цикла. Пробегают определенный диапазон и могут использоваться для вычисления индексных выражений.
3. **Функции.** Могут быть функциями-членами (в т.ч. стандартными операциями) или массивами. Если встречается функция-выражение, то она применяется к своим аргументам, таким образом, в конечном представлении ее не будет.

- **Внутренние вершины:**

5. **Применение функции к аргументам (@)**. Это может быть применение обычной функции или же обращение к элементу массива.
6. **Операция редукции (R)**. Имеет 2 аргумента, первый из которых задает редуцирующую функцию, а второй – редуцируемую. Редукция применяется по всем измерениям массива, не содержащим связанных переменных, а также по части связанных переменных, что позволяет реализовать частичную редукцию.

На рис. 3 приведен пример функционального графа для умножения матриц.

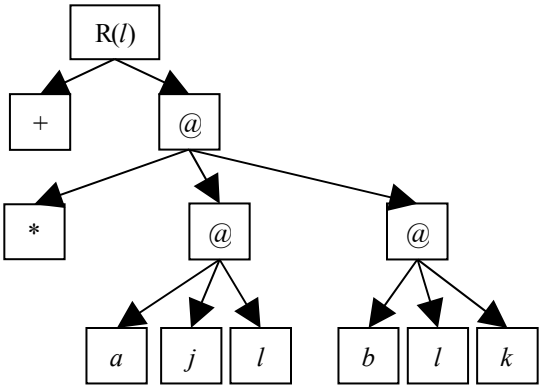

Рис. 3. Пример функционального графа для умножения матриц.

Здесь *a* и *b* – массивы (умножаемые матрицы), *j, l, k* – связанные переменные, знаки операций обозначают соответствующие функции над вещественными числами. Все связанные переменные являются целочисленными, массивы – двумерными. *R(l),* означает частичную редукцию только по переменной *l*. Соответственно, связанные переменные *j* и *k* сохраняются и попадают в результат.

Назовем *свободным измерением* в вершине измерение, которое не имеет ассоциированной связанной переменной. Если a и b – двумерные массивы, то в графе на рис. 4 в корне будет 2 свободных измерения, поскольку скалярная функция может быть применена к массиву.

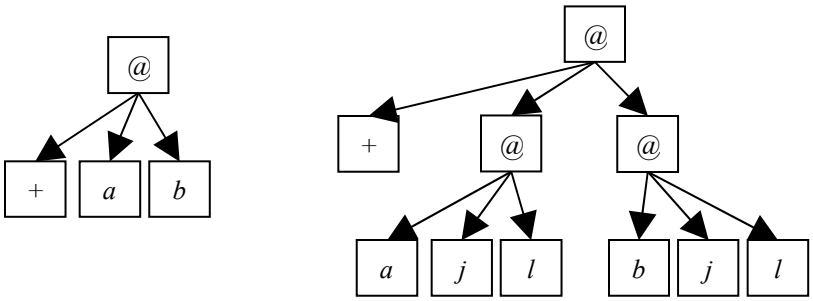

Рис. 4. Замена свободных измерений (справа) на связанные переменные (слева).

Заметим, что этот граф эквивалентен графу на рис. 4 справа со связанными переменными. С точки зрения трансляции, удобнее иметь дело с графом, содержащим только связанные переменные, без свободных измерений во внутренних вершинах. В дальнейшем только такие графы и будут рассматриваться.

Связанные переменные «распространяются» снизу вверх по графу. Оканчивать свое распространение они могут только на вершинах редукции или на корневой вершине. Также считаем, что ациклический граф устроен таким образом, что все пути распространения связанной переменной заканчиваются на одной и той же вершине. Назовем связанную переменную, оканчивающую свое распространение на корневой вершине, *целевой*, а на редукционной вершине – *редукционной.* В дальнейшем, если не оговорено иное, связанные переменные будут называться просто переменными. Все переменные, путь распространения которой заканчивается на данной вершине, назовем *собственными переменными* для данной вершины. Далее, с каждой переменной ассоциирован диапазон ее измерения, который будет называться *доменом* переменной. Совокупность связанных переменных в данном графе функциональных операций назовем *логическими измерениями кода* данного графа. Слово «логические» означает, что с точки зрения программиста конкретная реализация данного измерения не определена.

Каждый массив данных, используемый в графе, имеет определенные измерения. Например, на рис. 2 каждый из массивов *a* и *b* имеет ранг 2, соответственно, каждый имеет по 2 измерения. Кроме того, в результате работы программы, изображаемой этим графом, получается еще один массив – на рисунке ему можно сопоставить корневую вершину. Он тоже имеет 2 измерения. Назовем все различные измерения таких массивов *логическими измерениями данных* этого графа.

Пусть программа, выраженная графом функциональных операций, реализована на ГПУ. Тогда каждое логическое измерение кода реализовано на ГПУ каким-то способом (например, в виде комбинации из цикла и векторных операций или измерения прохода и векторных операций), аналогичное можно сказать и о данных. Назовем *физическим измерением кода* на ГПУ один из таких способов реализации логических измерений. Аналогично, *физическим измерением данных* на ГПУ назовем один из способов представления данных на ГПУ. Заметим, что один и тот же способ реализации может использоваться несколько раз в одном графе – например, для представления разных переменных. Соответственно, одно использование такого способа реализации будем называть *экземпляром (физического) измерения кода/данных.*

Примеры физических измерений кода:

- **Измерение вычисляемого буфера.** Поскольку ГПУ оперирует с двумерными массивами, то их всегда 2.
- **Номер вычисляемого буфера.** ГПУ может писать в некоторое количество (до 8-и) буферов.
- **Цикл.** Цикл внутри шейдера на графическом процессоре.
- **Блок.** Просто последовательность команд.
- **Векторная операция.** Обработка одной командой одновременно нескольких операций.

Примеры физических измерений данных:

*0**Размеры двумерного буфера.** В простейшем случае для представления данных используется только это измерение.

***1Несколько двумерных буферов.**

*2**Векторный элемент.** Один элемент может содержать до 4-х вещественных чисел.

Назовем *ключевыми вершинами* вершины, на которых может заканчиваться распространение связанных переменных. Очевидно, ключевой вершиной может быть или корневая вершина, или вершина редукции. Ключевые вершины примерно соответствуют операторам цикла в представлении программы на языке типа C. Подграф графа операций, у которого корнем и листьями являются ключевые вершины, и ни одна из внутренних вершин не является ключевой, назовем *подграфом вычислений* для данной ключевой вершины (аналог тела цикла). Реализацию этого подграфа для одного набора логических индексов назовем *элементом вычислений* (аналог итерации цикла). Очевидно, один элемент вычислений может быть охарактеризован *логическим индексом* – уникальным набором значений логических измерений кода. Аналогично, элемент данных также может быть охарактеризован логическим индексом. С другой стороны, в конкретной реализации программы элемент кода или данных может быть охарактеризован *физическим индексом* – уникальным набором значений экземпляров физических измерений кода или данных.

Отображением назовем взаимно однозначную функцию, которая каждому набору логических индексов ставит в соответствие набор физических индексов. Поскольку и логические, и физические индексы – целые числа, это будет функция вида:

$$p = m(\vec{i}) : \mathbb{Z}^k \to \mathbb{Z}^l$$

Где p – физический индекс, i – логический индекс, k и l – размерности физических и логических индексов, соответственно. Отображения кода отдельно определяются для каждой ключевой вершины. Для корневой вершины, если она является редукционной, отдельно могут быть определены отображение редукционных и целевых переменных. Отображения данных отдельно определяются для каждого входного массива, а также для выходного массива. В зависимости от требований архитектуры, отображения данных и кода должны быть согласованы.

Экспериментальным путем было установлено, что в случае отображений кода имеет смысл рассматривать только физические измерения цикла, блока и размера вычисляемого массива. Измерения векторных операций и множественных вычисляемых массивов появляются автоматически как представления измерений блока корневой вершины.

Таким образом, для заданного графа операций встает задача выбора отображений кода и данных. Для формализации задачи требуется, во-первых, задать допустимое множество отображений, среди которых делается выбор, во-вторых, задать оптимизируемую функцию. Поскольку наиболее критичной при выполнении программ на ГПУ является работа с памятью, то в качестве такой функции разумно выбрать общую задержку на обмены данными с памятью. При выборе отображения кода наиболее естественно разбивать циклы и вычисляемые массивы на блоки. Таким образом, целевые переменные могут отображаться на физические измерения размера вычисляемого массива и блока, а редукционные – на физические измерения цикла и блока. В любом случае, размер физического измерения цикла/размера вычисляемого массива определяется размером логического измерения. Таким образом, оптимизируемым параметром становится размер блока кода. Если размер блока равен 1, то блочное измерение в отображении не участвует. При выборе отображения данных в качестве оптимизационных параметров выбираем

количество буферов, которым будет представлено каждое измерение массива, и количество вещественных чисел в физическом элементе буфера («векторность»).

Поскольку оптимизируемая функция зависит от общего количества прочитанных из памяти данных, а данные можно читать только в вершинах чтения, нас будут интересовать только вершины чтения, конкретнее – ключевые вершины с набором вершин чтения. При рассмотрении вершин чтения имеет смысл рассматривать *набор чтений* - т.е. множества вершин чтения, которые отличаются только смещением читаемых из памяти данных. Кроме того, считаем, что для каждой вершины чтения функция, задающая зависимость логического индекса данных от логического индекса вычислений, является аффинной. Если это не так, то предварительно индексное выражение приводится к аффинному виду. Заметим, что аффинный вид функции используется только при вычислении объема прочитанных данных.

Итак, пусть G – набор чтений. Набор чтений можно представить в виде:

$G=<D, A, \{d_1, \ldots, d_n\}>$, где D – массив данных, из которого производится чтение, A – общая для набора матрица аффинного преобразования, d_i – множество различных смещений. Пусть $\boldsymbol{b}$ – вектор размеров блоков для измерений кода, $\boldsymbol{t}$ и $\boldsymbol{s}$ – векторы наборов количества буферов и количества вещественных чисел в элементе для измерений данных, $\boldsymbol{x} = <\boldsymbol{b}, \boldsymbol{t}, \boldsymbol{s}>$ - полный набор оптимизационных переменных. Далее, пусть K - некоторая ключевая вершина. Пусть $G(K)$ – множество всех наборов чтения ключевой вершины K, а $I(K)$ – множество определяющих индексов вершины, т.е. объединение всех собственных связанных переменных на пути от корня к данной вершине. Далее, пусть j – связанная переменная (или ее номер). Обозначим за *P(j) мощность связанной переменной,* т.е. размер ее диапазона. *P(K) – мощность вершины*, т.е. сколько раз вычислялся бы соответствующий подграф, если бы весь граф вычислялся бы в обычном цикле. Очевидно,

$$P(K)=\prod_{j\in I(K)} P(j)$$

Общее время чтения данных из памяти для ключевой вершины можно выразить как

$$\tilde{\tau}(K,\vec{x})=\tau_D(\vec{x})+\tau_B\rho(K,\vec{x})$$

где $\tilde{\tau}$ – общее время чтения данных из памяти (на один элемент вычислений τ_D), – задержка на чтение данных из памяти, τ_B – время чтения одного элемента данных, $p(K)$ – общий объем одного чтения данных из памяти. На современных ГПУ для компенсации задержки используют большое число потоков. Соответственно, время задержки обратно пропорционально числу потоков. В то же время число потоков обратно пропорционально числу задействованных регистров, которое, в свою очередь, выражается как сумма числа регистров, задействованных в каждой ключевой вершине. Таким образом, имеем:

$$\tau_D(\vec{x})=\frac{\tau_D N_R(\vec{x})}{N_{R0}}$$

где $N_R(x)$ – общее число регистров для заданных параметров оптимизации. Общий объем переданных данных для набора чтения считается по формуле:

$$\rho(G,\vec{x})=\pi[\hat{A}_G\vec{b}+\vec{\delta}_G-\vec{1}]$$

где $\hat{A}_G$ – матрица модулей элементов матрицы аффинного индексного выражения, $\vec{\delta}_G$ – размер области массива, покрываемый группой чтения, π – преобразование выражения, которое перемножает все элементы вектора, при этом ликвидируя повторные вхождения переменных в один и тот же множитель. Для учета того факта, что чтение возможно только всего элемента текстур сразу, функция $p(G, x)$ модифицируется, что резко усложняет ее вид. В приведенных выше терминах общая стоимость чтения данных из памяти может быть выражена следующим образом:

$$C(\vec{x}) = \sum_k \frac{\rho(K)\left(\frac{\tau_{D0} N_R(\vec{x})}{N_{R0}} + \sum_{G \in G(K)} \pi[\hat{A}_G \vec{b} + \vec{\delta}_G - \vec{1}]\right)}{\prod_{j \in I(K)} b_j}$$

Значение функции стоимости чтения из памяти минимизируется при следующий ограничениях:

- Ограничение общего размера корневого блока
- Ограничение общего числа используемых регистров
- Ограничения диапазона каждой переменной
- Ограничение общего числа входных буферов
- Ограничение на число вещественных чисел в одном элементе текстуры

В итоге получается задача целочисленной оптимизации. Условие целочисленности релаксируется, после чего исходная задача сводится к вещественной задаче геометрического программирования. Эта задача, в свою очередь, сводится к задаче выпуклой оптимизации и решается при помощи метода штрафных функций за полиномиальное время [11]. Полученные вещественные значения округляются в сторону ближайшей степени двойки, за исключением тех, для которых размер соответствующих логических измерений достаточно мал. Последние округляются либо до этого размера, либо до единицы.

Для корневых размеров блока выбирается наиболее подходящее их отображение на количество выходных буферов и количество вещественных чисел в элементе выходного буфера. Кроме этого, отображения данных могут изменяться с целью их согласованности. После этого выполняется генерация кода для графического процессора.

ГЕНЕРАЦИЯ КОДА НА ГПУ

Генерация кода для графического процессора осуществляется в соответствии с выбранным отображением. Каждой ключевой вершине ставится в соответствие некоторый набор выходных переменных, за вычисление которых она отвечает. Корневая вершина, помимо их вычисления, отвечает и за запись их в выходные буферы. Для каждого подграфа операций генерируется шаблон кода. На основании этого шаблона генерируется код для каждого набора значений экземпляров блочных измерений.

Наиболее сложной частью этапа генерации кода является генерация команд доступа в ОЗУ. Пусть R – вершина чтения, K – отвечающая за нее ключевая вершина. Далее, пусть имеются отображения:

m_D – отображение данных для массива данных, из которых происходит чтение.

m_R – отображение (не обязательно обратимое или аффинное), которое по логическому индексу элемента вычислений вычисляет логический индекс требуемого элемента данных.

m_K – отображение кода для ключевой вершины, отвечающей за данную вершину чтения.

Комбинируя эти три отображения, можно будет получить отображение (не обязательно обратимое), которое по физическим индексам элемента вычислений вычисляет физический индекс элемента данных:

$$\vec{p}_D = m_{res}(\vec{p}_C) = (m_D m_R m_K^{-1})(\vec{p}_C)$$

Далее по отображению определяется множество номеров буферов, которые могут участвовать этой операции чтения данных. Далее, для каждого из номеров буферов определяется для различных значений блочных индексов набор двумерных индексов для чтения из этого буфера. Для каждого индекса, на основании набора переменных, от которых он реально зависит, находится самая близкая к корню вершина, в которой индекс может быть вычислен. Оптимизируется вычисление индексов, которые линейны по итерациям цикла. В каждой ключевой вершине отмечается набор индексов, которые в ней требуется вычислить, что позволяет избежать повторного вычисления индексов. Операция чтения данных выполняется в той же ключевой вершине, к которой принадлежит индекс. Это позволяет избежать повторного чтения одних и тех же данных из ОЗУ, даже если они производятся разными вершинами чтения.

Для работы с индексами используются средства автоматического преобразования и упрощения целочисленных выражений. Они поддерживают преобразование выражений, содержащих арифметические операции, минимум, максимум, а также операции сравнения.

РЕЗУЛЬТАТЫ И ВЫВОДЫ

Описанные выше методы оптимизации были реализованы в системе C$. Система была протестирована на ряде вычислительных задач. Тестирование проводилось на ГПУ AMD Radeon HD3870x2. Из двух графических чи-

пов, присутствующих на карте, использовался только один, так что пиковая производительность системы составляла 488 ГФлопс (с одинарной точностью). Код вычислительного алгоритма был написан на языке высокого уровня C$ и не содержал никаких оптимизаций под архитектуру целевого процессора, все оптимизации выполнялись системой C$. Результаты исполнения программы с оптимизациями и без них, а также получаемые при этом ускорения приведены в таблице 1.

Задача	Производительность без оптимизаций, ГФлопс	Производительность с оптимизациями, ГФлопс	Ускорение
Умножение матриц	10.58	142.07	13.43
Фильтрация изображений 3*3	8.73	52.01	5.95
Фильтрация изображений 5*5	9.13	43.61	4.78
Фильтр Гаусса 7*7	2.37	30.72	12.96
Фильтр Гаусса 9*9	2.92	30.46	10.43
Опционы Блэка-Шоулза	70.71	111.11	1.57

Табл. 1. Производительность системы C$ на ряде задач.

Также проводилось сравнение сложности программного кода, написанного на C$ и на языке Brook+, который позиционируется AMD как высокоуровневая аппаратно-зависимая надстройка над CAL. Для программ, имеющих реализации на том и на другом языках, сравнивалась их сложность в терминах количества строк кода. Результаты сравнения приведены в таблице 2.

Задача	Число строк программного кода, C$	Число строк программного кода, Brook+	Преимущество C$
Умножение матриц	25	400	16
Фильтрация изображений 3*3	45	172	3.82
Фильтрация изображений 5*5	45	172	3.82
Опционы Блэка-Шоулза	104	289	2.78

Табл. 2. Сравнение C$ и Brook по размеру программы.

Для умножения матриц было выполнено сравнение кодов C$ и Brook. На максимальном размере матрицы, 4096 * 4096, код C$ давал производительность в 142 ГФлопс, оптимизированный код на Brook – 225 ГФлопс. На размерах 1024 * 1024 и ниже код на C$ и на Brook давали производительность в районе 70-80 ГФлопс.

На основании данных численных экспериментов можно сделать ряд выводов. Во-первых, использование методов оптимизации для ГПУ, описанных в статье, позволяет получить значительный прирост производительности без изменения исходного кода программы. При этом на программе с нетривиальными индексными выражениями, например, умножении матриц или фильтрации изображений, удается получить ускорение в несколько раз или даже на порядок.

Во-вторых, автоматические методы оптимизации на ряде задач дают код, по эффективности вполне сопоставимый с тем, что получается в результате длительной ручной оптимизации. При этом исходный код не привязывается к архитектуре целевой вычислительной системы, кроме того, временные затраты на его создание значительно ниже. Потенциально это позволит добиться переносимости кода между различными ГПУ с сохранением эффективности.

В-третьих, и это самое главное, удалось формально поставить задачу отображения кода на архитектуру графического процессора и решить ее с применением формальных методов оптимизации. В дальнейшем это позволит добавлять дополнительные способы оптимизации в виде физических измерений. Также можно будет организовывать отображение на другую целевую архитектуру, например, ГПУ NVidia или процессор CELL, меняя набор физических измерений и конкретный вид оптимизируемой функции, но не меняя самого подхода. Для ГПУ NVidia в качестве «блока» может выступать статическая память, разделяемая между несколькими потоками.

Дальнейшая работа будет, скорее всего, идти по нескольким направлениям. Прежде всего, будет организована трансляция кода для ГПУ NVidia. Уровень устройства и ряд оптимизаций для NVidia уже находятся в стадии

активной разработки. Во-вторых, это добавление поддержки оптимизации для более сложных типов данных и функций в язык C$. Прежде всего это относится к структурным типам данных, более сложным видам циклов (не только редукции), более сложным областям определения функций, а также к добавлению поддержки рекурсивных функций. В-третьих, это поддержка выполнения C$-программ одновременно на нескольких ГПУ, в одном или нескольких компьютерах. Наконец, это организация трансляции кода для процессора, не являющегося графическим, т.е. многоядерного ЦПУ или процессора CELL.

ЛИТЕРАТУРА:

1. Воеводин Владимир, Адинец Андрей. Графический вызов суперкомпьютерам. 4, с. 35 - 41.
2. *GPGPU.org.* http://www.gpgpu.org.
3. *OpenGL.* http://www.opengl.org/.
4. RapidMind. http://www.rapidmind.com/.
5. NVidia Corporation. NVIDIA CUDA Programming Guide Version 1.1. 11.29.2007. 143 c.
6. AMD Inc. AMD Compute Abstraction Layer Programming Guide. 2008.
7. Адинец, А. В., Сахарных, Н. А. *О системе программирования вычислений общего назначения на современных графических процессорах.* // Численные методы, параллельные вычисления и информационные технологии: Сборник научных трудов . М.: Издательство Московского Университета, 2008, , с. 25 – 52. ISBN 978-5-211-05503-3.
8. Jansen, Thomas C. GPU++: An Embedded GPU Development System for General-Purpose Computations. Munich, Germany : s.n., 2007. Ph. D. Thesis.
9. Adinetz, Andrew V. C$ Homepage. http://www.codeplex.com/cbucks.
10. *Использование ГПУ для высокопроизводительных вычислений.* http://gpu.parallel.ru/.
11. Васильев, Ф. П. *Методы оптимизации.* Москва : Факториал Пресс, 2002. 824 с. ISBN 5-88688-056-9.