ТЕХНОЛОГІЇ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

УДК 681.3

Институт программных систем НАН Украины
А.Е. Дорошенко, К.А. Жереб

Техника и инструментарий переписывающих правил для инженерии программного обеспечения графических ускорителей

Графические ускорители позволяют достичь высокой производительности за счет большого количества вычислительных ядер, однако их программирование является сложным и трудоемким процессом. В работе предложен подход к созданию эффективных программ для графических ускорителей с использованием техники переписывающих правил. Предложенный подход и разработанный инструментарий позволяют достичь высокой производительности параллельных программ, а также повысить производительность разработчиков.

Графічні прискорювачі дозволяють досягти високої продуктивності за рахунок великої кількості обчислювальних ядер, проте їх програмування є складним і трудомістким процесом. У роботі запропонований підхід до створення ефективних програм для графічних прискорювачів з використанням техніки переписуючих правил. Запропонований підхід і розроблений інструментарій дозволяють досягти високої продуктивності паралельних програм, а також підвищити продуктивність розробників.

The graphic accelerating allow to attain a high yield due to plenty of calculable kernels, however their programming is a difficult and labour intensive process. In-process offered approach near creation of the effective programs for the graphic accelerating with the use of technique of rewritting rules. Offered approach and the worked out tool allow to attain the high yield of the parallel programs, and also promote the productivity of developers.

Ключевые слова: graphical processing units, rewriting rules technique, automated program transformations, formal design methods, development tools, Microsoft .NET framework.

Введение

В последнее время активно развивается новая аппаратная и программная платформа для высокоэффективных параллельных вычислений – графические ускорители (graphical processing units, GPUs [1]). Эти специализированные устройства, изначально предназначенные только для обработки графики, оказались весьма эффективными для решения вычислительных задач, не связанных с графикой (general purpose computations on GPUs, GPGPU [2]). Их преимуществом является очень высокая степень возможного параллелизма, за счет большого количества вычислительных ядер и специального планировщика, позволяющего эффективно управлять исполнением миллионов потоков. Кроме того, графические ускорители поддерживают иерархию памяти, от быстрой, но ограниченной по объему, до большой по объему и медленной. Это позволяет использовать разные виды памяти для разных задач, что также повышает эффективность параллельных программ.

Однако для графических ускорителей характерен и весьма существенный недостаток, а именно сложность разработки приложений для этих платформ. Первоначально для программирования вычислений на графических ускорителях использовались средства разработки графических приложений – шейдеры (shaders). Вычисления представлялись в виде действий с графическими объектами, такими как текстуры [3]. В последнее время интерес к использованию графических ускорителей в качестве средств высокопроизводительных вычислений поддерживается усилиями ведущих разработчиков аппаратуры. Так, компания NVidia представляет платформу для вычислений на графическом ускорителе CUDA [4]. Аналогично, компания AMD выступила с инициативой

Stream [5]. Тем не менее, эти платформы остаются сложными для программирования и изучения, поскольку они используют непривычную для разработчиков модель программирования, требуют знания деталей аппаратной и программной платформы, недостаточно поддержаны инструментальными средствами.

Таким образом, актуальной является задача создания средств программирования эффективных параллельных вычислений для графических ускорителей. При этом ставятся две цели: повышение производительности параллельных программ, а также повышение производительности труда разработчика. В существующих платформах разработки эти цели часто находятся в противоречии. Так, платформа NVidia CUDA позволяет создавать программы для графических ускорителей на языке C for CUDA путем минимального изменения кода последовательных программ на языке C. Однако такие программы в большинстве случаев являются недостаточно производительными, иногда работают даже медленнее, чем исходные последовательные программы. Платформа CUDA также предоставляет средства для оптимизации программ, однако эти средства являются весьма сложными и требуют от разработчика знания деталей функционирования графического ускорителя. Например, авторы [6] описывают сложности, возникающие в процессе изучения разделяемой памяти (shared memory), которая является необходимым условием достижения высокой производительности.

В данной работе предложен подход к проектированию и разработке эффективных параллельных приложений для графических ускорителей, направленный как на достижение высокой производительности созданных программ, так и на повышение эффективности разработчика. Предложенный подход состоит в автоматизированном преобразовании моделей программ с использованием техники переписывающих правил [7]. Описан разработанный инструментарий для инженерии программного кода, основанный на системе переписывающих правил Termware [7, 8]. Предложено три направления использования техники переписывающих правил для инженерии кода приложений для графических ускорителей, а именно переход между низкоуровневыми и высокоуровневыми моделями программ, автоматизация распараллеливающих и оптимизирующих преобразований программ, генерация тестовых программ. Предложенный подход и разработанный инструментарий опробованы на вычислительной задаче, при этом достигнуто значительное повышение производительности по сравнению с последовательной программой.

Материал данной работы организован следующим образом. В разделе 1 проведен обзор текущего состояния задачи программирования графических ускорителей для платформы CUDA. В разделе 2 описан разработанный инструментарий для инженерии программного кода на основе техники переписывающих правил. Раздел 3 содержит направления использования техники переписывающих правил, а также пример реализации вычислительной задачи. Работу завершают выводы и направления дальнейшей работы.

1. Программирование графических ускорителей

Современные графические ускорители поддерживают высокую степень параллелизма, специально приспособленную для выполнения графических задач. Использование возможностей графических ускорителей для общецелевых вычислений требует от разработчика понимания особенностей аппаратной платформы и модели программирования CUDA [4].

Общая схема устройства графического ускорителя показана на рис. 1. Графическое устройство (device) содержит несколько мультипроцессоров, а также общую для них графическую память. Каждый мультипроцессор содержит несколько вычислительных ядер (скалярных процессоров), а также один управляющий блок, поддерживающий многопоточное исполнение. В результате количество вычислительных ядер (а значит, и степень возможного параллелизма) оказывается существенно выше, чем у общецелевых многоядерных процессоров.

Графические ускорители поддерживают несколько разных видов памяти, отличающихся по объему, скорости доступа и особенностям реализации. Самая быстрая память – регистры вычислительных ядер; однако их количество в каждом ядре ограничено.

Кеш данных, или разделяемая память (shared memory) поддерживает произвольный доступ из любого вычислительного блока, однако имеет ограниченный размер и требует явной синхронизации доступа. Есть также два специфических вида памяти – кеш констант и кеш текстур. Они поддерживают только чтение, но имеют больший объем по сравнению с разделяемой памятью. Разница между текстурной и константной памятью заключается в том, что текстурная память поддерживает специальные режимы доступа, полезные для графических задач. Всем мультипроцессорам доступна так-

же графическая память устройства. Она является самой большой по объему и самой медленной памятью, доступной графическому ускорителю.

Параллельная программа для графических ускорителей состоит из большого количества потоков. Особенности аппаратного обеспечение, а именно большое количество вычислительных ядер, позволяют использовать очень мелкозернистый параллелизм, вплоть до выделения отдельного потока для каждого элемента данных. Для исполнения на графическом ускорителе потоки объединяются в блоки. Каждый блок выполняется на одном мультипроцессоре. Различные блоки, соответствующие одной или нескольким программам, по возможности распределяются равномерно по доступным мультипроцессорам.

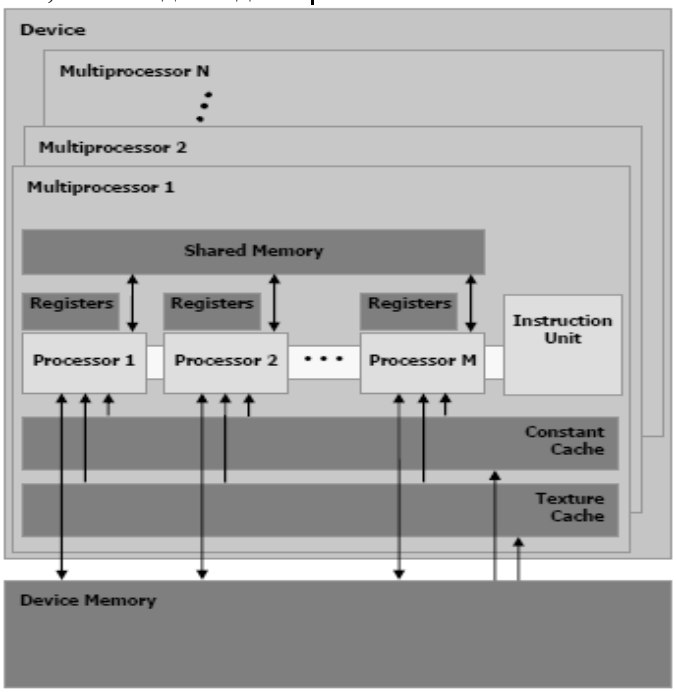

Рис. 1. Основные аппаратные компоненты графического ускорителя.

При исполнении на мультипроцессоре, потоки объединяются в так называемые warp'ы по 32 потока. На каждом шаге многопоточный планировщик выбирает один из доступных warp'ов. Далее очередная инструкция для потоков этого warp'а выполняется одновременно на вычислительных ядрах. Следует отметить, что наличие ветвлений в потоках понижает производительность, поскольку каждый из вариантов исполнения в таких случаях выполняется последовательно, а остальные потоки warp'а ожидают выполнения.

Программа для графического ускорителя, написанная на CUDA, разделяется на две части: код для исполнения на графическом устройстве (device code) и код для исполнения на обычном процессоре (host code). Код для исполнения на устройстве представлен в виде ядер (kernel). Ядра могут быть написаны как на специальном низкоуровневом языке (PTX), так и на расширении языка C (C for CUDA). В последнем случае ядро имеет вид функции языка C, описывающей поведение одного потока.

Платформа CUDA предлагает два разных способа вызова ядер. Более простой способ, C for CUDA, является расширением языка C. Вызов ядра при этом похож на вызов функции C, но с передачей дополнительных параметров. Эти параметры определяют размер блока, а также количество одновременно исполняемых блоков в сетке (grid). Каждый поток имеет доступ к своему номеру в блоке и номеру блока в сетке, что позволяет различным потокам работать с разными данными.

Более сложным способом вызова ядер является driver API. В этом случае ядра необходимо предварительно скомпилировать в бинарный формат, после чего используются функции для загрузки и выполнения ядер. Driver API является более сложным в разработке и может привести к появлению ошибок, так как требует ручного выполнения многих действий, которые автоматизированы в C for CUDA. Пример таких действий – передача параметров ядра. С другой стороны, driver API дает разработчику больший контроль над выполнением программы.

Оба способа вызова CUDA предоставляют функции для работы с графической памятью. Обычно работа программы состоит из таких шагов: инициализация структур данных в графической памяти, копирование из обычной памяти в графическую входных данных, вызов ядра (или нескольких ядер, или многократный вызов одного ядра), копирование результатов из графической памяти, освобождение графической памяти.

Платформа CUDA на данный момент поддерживает только язык C, поэтому для использования возможностей графического ускорителя из .NET программ необходимы дополнительные компоненты. К подобным компонентам относится библиотека CUDA .NET [9]. Эта библиотека позволяет .NET коду выполнять ядра CUDA, используя возможности, близкие к driver API.

Таким образом, для использования CUDA .NET необходимо сначала реализовать ядро CUDA, что делается средствами C for CUDA. После этого необходимо написать C# код для вызова ядра. Так как CUDA .NET использует более сложный driver API, от разработчика требуется корректная реализация многих низкоуровневых деталей. В частности, при передаче параметров необходимо иметь представление об их размещении в памяти и корректно передать размеры и относительные смещения параметров. Необходимость выполнения этих, довольно сложных, операций вручную затрудняет разработку и повышает вероятность ошибки.

Область исследований, связанная с автоматизацией разработки приложений для графических процессоров, в настоящее время активно развивается. При этом рассматриваются как задачи перехода от последовательных к параллельным программам, так и задачи оптимизации существующих параллельных программ с использованием возможностей графических ускорителей. Так, в работе [10] рассмотрен автоматический переход от многопоточной программы, реализованной с использованием технологии OpenMP [11], к реализации данной программы на платформе CUDA. Работа [12] описывает платформу для оптимизации циклов в программах для графических ускорителей. Разработаны системы для автоматического распараллеливания и оптимизации программ из конкретной предметной области, например, data mining [13] или обработка изображений [14]. Авторы работы [15] описывают библиотеку высокоуровневых структур данных для графических ускорителей. Также разрабатываются платформы программирования графических ускорителей, предоставляющие более высокоуровневые средства по сравнению с CUDA, такие как hiCUDA [16] и BSGP [17].

В отличие от существующих работ по данной тематике, в данной работе рассмотрена автоматизация перехода на платформу CUDA с высокоуровневого языка C#. Это позволяет использовать возможности платформы Microsoft .NET, которая в настоящее время широко используется для разработки приложений в различных областях. Кроме того, использование переписывающих правил для описания распараллеливающих и оптимизирующих преобразований позволяет легко добавлять новые преобразования.

2. Инструментарий на основе переписывающих правил

2.1. Общее описание подхода. В работе используется следующий подход для разработки эффективных программ для графических ускорителей. В качестве входных данных используется исходный код последовательной программы, описывающий алгоритм решения задачи. Программный код представляется в виде высокоуровневой модели с использованием алгебры алгоритмов Глушкова [18]. Далее к этому высокоуровневому представлению применяются преобразования, представленные в виде переписывающих правил. Преобразования могут быть направлены на переход от последовательной к параллельной версии программы (*распараллеливающие преобразования*), а также на повышение производительности параллельной программы (*оптимизирующие преобразования*). Кроме того, возможно применение преобразований для создания вспомогательных артефактов, таких как автоматизированные тесты. После применения преобразований к высокоуровневой модели программы, используется генератор кода для создания исходного кода преобразованной программы.

Многие этапы описанного процесса выполняются в автоматическом или автоматизированном режиме. Так, переход от исходного кода программы к ее высокоуровневой алгебраической модели, а также переход в обратном направлении (генерация кода) выполняется автоматически, для заданной предметной области и языка программирования (этот процесс подробно описан в п. 3.1). Применение преобразований программ, представленных в виде переписывающих правил, также является автоматизированным: от пользователя требуется лишь указать, какие преобразования следует применять и на каком участке кода они должны действовать. Наиболее сложной частью описанного процесса, выполняемой вручную, является

создание переписывающих правил, описывающих преобразование. Тем не менее, многие из таких правил применимы для различных программ, что способствует их повторному использованию.

2.2. Система Termware. Для автоматизации преобразований программ в данной работе используется техника переписывающих правил, реализованная системой переписывающих правил Termware [7, 8]. Система Termware направлена на разработку высокодинамичных прикладных систем, к которым предъявляются повышенные требования, как к встроенному интеллекту для обеспечения интерактивности разрабатываемых систем, так и удешевлению разработки, сокращению сроков проектирования и улучшения характеристик повторного использования и сопровождаемости. Она отличается от большинства других систем этого класса как назначением и семантикой используемых средств, так и технологией их реализации. Язык Termware не является универсальным языком программирования в том смысле, что он не предназначен для написания полнофункциональных программных систем. Это координационный язык-оболочка, предназначенный для написания предметно-ориентированных частей приложений, встраиваемых в прикладную систему для реализации функций взаимодействия этой системы с программным окружением.

Для системы Termware характерны следующие особенности:

- использование декларативного описания;
- реализация только требуемой части приложения, а не всего приложения;
- ориентация на тесное взаимодействие с другими частями приложения.
- Переписывающие правила описываются на языке Termware и хранятся в отдельных файлах. Основой языка являются термы, т.е. выражения вида $f(x_1,...,x_n)$. В качестве атомарных термов используются переменные (которые записываются в виде *$var*), а также константы определенных типов данных (числовых, логических, строковых и атомарного – неизменяемые строки). Для упрощения записи и восприятия используются сокращения для многих термов, например *x+y* – для *plus(x,y);* [*x:y*] – для *cons(x,y)* (список с первым элементом *x* и остатком списка *y*)*; x ? y: z* – для *ifelse(x,y,z)* и другие. Весь набор правил является термом, записанным с использованием этих сокращений.
- Правила Termware имеют следующий общий вид:
- *source [condition]* $\rightarrow$ *destination [action]*
- Здесь используются четыре терма:
- *source* – входной образец;
- *destination* – выходной образец;
- *condition* – условие, определяющее применимость правила;
- *action* –действие, выполняемое при срабатывании правила.

Выполняемые действия и проверяемые условия являются необязательными компонентами правила, которые могут вызывать императивный код. Для этого разработчик должен реализовать «базу фактов» – класс, реализующий интерфейс *IFacts* и предоставляющий методы, которые могут вызываться из переписывающих правил. Таким образом, осуществляется связь между декларативными правилами на языке Termware и императивным кодом на традиционных объектно-ориентированных языках (таких как Java или C#). Кроме того, возможно написание собственной стратегии (в виде класса, реализующего интерфейс *ITermRewritingStrategy*), определяющей порядок применения правил.

- Система Termware была изначально реализована в виде библиотеки языка Java, предназначенной для встраивания в прикладные программы [8]. Также был разработан интерфейс командной строки для интерактивного исполнения простых переписывающих правил. Авторы данной работы перенесли систему Termware на платформу Microsoft .NET, что позволяет использовать средства переписывающих правил из программ на языке C#, а также на других языках, поддерживаемых платформой .NET. Кроме того, был реализован графический интерфейс пользователя для более удобного создания и применения переписывающих правил [19]. Также были разработаны синтаксический анализатор (парсер) и генератор кода для языка C#, что позволяет системе Termware работать с программами, представленными в виде исходного кода на этом языке. Разработанные компоненты составляют инструментарий для преобразования программ на платформе Microsoft .NET, описанный в следующем пункте.

2.3. Структура инструментария. Разработанный программный инструментарий предназначен для автоматизированного преобразования программ с использованием техники переписывающих правил. Инструментарий состоит из следующих основных компонентов:

• Анализатор для перевода программного кода с языков высокого уровня (таких как C#, Java, C++) в модель программы в виде термов;

• Система применения переписывающих правил для преобразования программ;

• Генератор для обратного перевода из модели программы в виде термов в язык программирования;

• Графический интерфейс пользователя для просмотра и редактирования термов и переписывающих правил, а также управления другими компонентами.

Взаимодействие между этими компонентами схематически представлено на рис. 2.

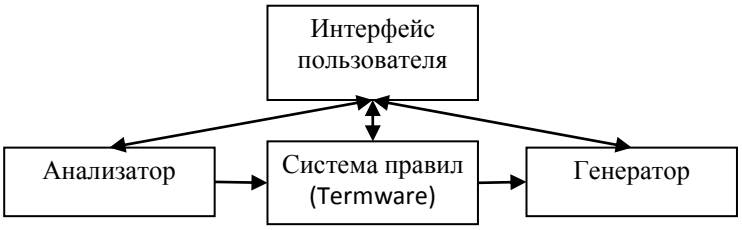

Рис. 2. Основные компоненты системы

Синтаксический анализатор осуществляет переход от исходного текста программы на языке C# к его модели в виде термов. Аналогично, генератор кода осуществляет переход от модели программы к исходному коду на C#. Заметим, что данная модель является низкоуровневой и явно описывает все синтаксические конструкции языка. Для удобства пользователя инструментарий поддерживает переход к высокоуровневой алгебраической модели; для этого используются переписывающие правила специального вида, описанные в п. 3.1.

На данный момент синтаксический анализатор и генератор кода поддерживают язык C# версии 2.0. Инструментарий предусматривает возможность добавление новых анализаторов и генераторов для поддержки других языков программирования. Для этого необходимо предоставить классы, реализующие интерфейсы *IParser* и *ICodeGenerator*, которые осуществляют преобразование между текстовым представлением исходного кода на заданном языке и синтаксической моделью (деревом синтаксического разбора), представленной в виде термов.

Для автоматического применения переписывающих правил к заданным термам (в частности, модели программы) инструментарий использует библиотеку Termware для платформы Microsoft .NET. Инструментарий поддерживает загрузку и сохранение систем правил на языке Termware. Кроме того, есть возможность реализации баз фактов (содержащих методы, доступные для вызова из переписывающих правил) или дополнительных стратегий переписывания на языке C# или на других языках платформы Microsoft .NET.

Графический интерфейс разработанного инструментария представлен на рис. 3. В левой части расположено древовидное представление текущего терма (модели программы). Правая часть интерфейса поддерживает редактирование правил, в текстовом или графическом (древовидном) представлении.

Графический интерфейс позволяет пользователю просматривать и редактировать термы в визуальном древовидном представлении, а не в текстовом виде. Поддерживаются стандартные функции для работы с элементом управления дерево, такие как сворачивание отдельных ветвей. Редактирование терма осуществляется с помощью контекстного меню, поддерживающего добавление и удаление узлов, изменение содержимого узлов, работу с буфером обмена. Те же возможности доступны и при редактировании правил в древовидном представлении. Таким образом, графический интерфейс упрощает понимание сложных моделей программ и позволяет работать с ними без изучения специфического синтаксиса системы переписывающих правил. При этом опытные пользователи могут работать с термами также и в текстовом представлении. В частности, при редактировании дерева каждый листовой узел может содержать не только атомарный символ, но и подтерм произвольной сложности, представленный в текстовом виде. Это позволяет быстро добавлять сложные элементы модели, такие как арифметические выражения.

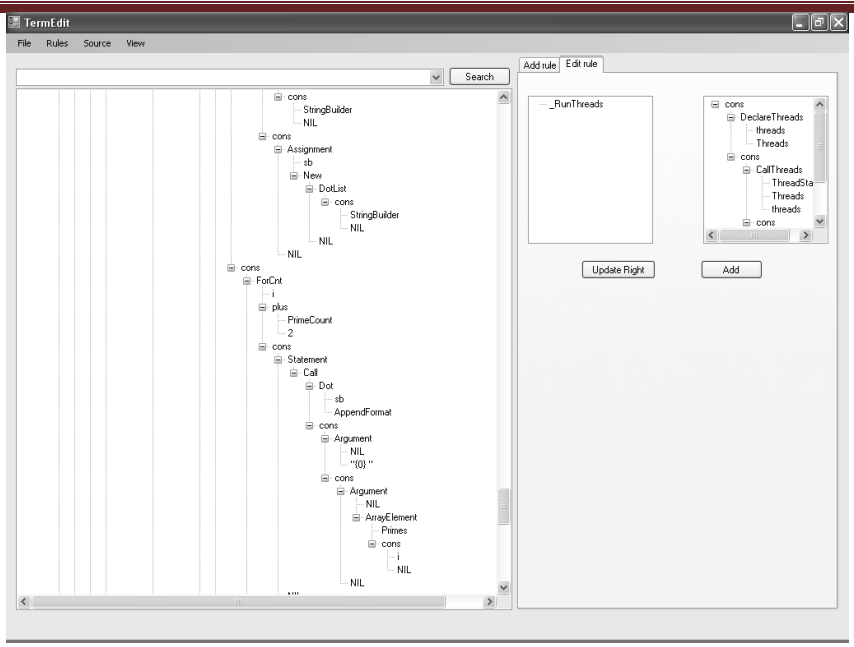

Рис. 3. Графический интерфейс пользователя

2.4. Дополнительные возможности инструментария. Кроме базовых возможностей системы Termware [7], разработанный инструментарий поддерживает дополнительные возможности, направленные на упрощение работы пользователя с системой. К таким возможностям относятся

- метки;
- TODO-термы;
- мультисистемы;
- заготовки правил;
- модификаторы;
- паттерны.

TODO-термы и метки – термы специального вида, которые могут использоваться в переписывающих правилах вместе с другими термами, но также могут дополнительно обрабатываться инструментарием. *Метки* имеют вид *_MARK_...*, т.е. имя терма начинается с символов *_MARK_*, после чего идут произвольные другие символы и/или подтермы. Метки используются для выделения отдельных элементов (подтермов) модели и не имеют самостоятельного значения в модели. Выделение с помощью меток используется для указания участка кода (модели), к которому применяются преобразования (см. п. 3.2). Кроме того, метки могут использоваться в термах, которые являются результатами преобразования, чтобы предотвратить дальнейшее применение правил к этим термам. Инструментарий обрабатывает метки особым образом: доступна команда для удаления всех меток из модели; кроме того, генератор игнорирует метки при создании кода по модели. Таким образом, метки не влияют на генерируемый код, но позволяют правилам работать корректно.

В качестве примера использования меток рассмотрим правила для добавления элемента *Field(b,int)* (целочисленное поле класса с именем *b*) непосредственно после элемента *Field(a,int)*. Простейшее правило, реализующее данное преобразование, имеет вид

[Field(a,int):$next]->[Field(a,int):[Field(b,int):$next]] (1)

Здесь использована переменная *$next* для обозначения хвоста списка. Однако правило в таком виде приведет к зацикливанию: после его первого срабатывания, в модели остается элемент *Field(a,int)*, к которому применимо то же правило. Для того чтобы избежать зацикливания, можно пометить элемент *Field(a,int)* в правой части правила меткой _MARK_Processed:

[Field(a,int):$next]->[Field(a,int, **_MARK_Processed**):[Field(b,int):$next]] (2)

В этом случае после однократного применения правила, результирующий терм отличается от исходного, и правило не может быть применено вторично. Еще одним вариант реализации преобразования состоит в том, чтобы пометить исходный элемент *Field(a,int)* меткой _MARK_Target, и удалить эту метку в процессе работы правила:

*[Field(a,int, **_MARK_Target**):$next]->[Field(a,int):[Field(b,int):$next]]* (3)

На практике могут применяться оба описанных подхода, в зависимости от особенностей задачи. Если необходимо преобразовать только один элемент модели (например, поле с

именем *a* только в одном классе), используется метка на исходном терме. Если же преобразование должно работать со всеми элементами заданного вида (добавить новое поле после всех полей с именем *a*), имеет смысл использовать метки на результирующем терме.

TODO-термы используются для задания правил общего вида, в которых часть результирующих термов заранее неизвестна. Аналогично меткам, TODO-термы имеют вид *_TODO_...*, где за символами *_TODO_* следуют произвольные символы и подтермы. Инструментарий поддерживает список всех термов такого вида, и предлагает пользователю заменить их на конкретные элементы модели. Кроме того, поддерживается автоматическая замена TODO-терма на терм, полученный из него удалением символов *_TODO_*.

В качестве примера использования TODO-термов рассмотрим немного измененное преобразование (1). Предположим, что необходимо заменить элемент *Field(a,int)* на элемент вида *Field(b,int)*, но с именем, которое должен задать пользователь. Для этого используется правило

Field(a,int)->Field(_TODO_b, int) (4)

В этом случае пользователь может выбрать из списка TODO-термов элемент *_TODO_b* и заменить его на нужное имя. Если в модели присутствует несколько термов вида *_TODO_b*, все они будут заменены. Если же пользователь не меняет данный терм, он автоматически заменяется на терм *b* при генерации кода. Возможно также правило вида

Field(a,int)-> _TODO_Field(b, int) (5)

В этом случае весь терм в правой части правила может быть изменен, хотя значение по умолчанию остается таким же.

Более сложный пример использования *TODO*-термов для задания оптимизирующих преобразований многопоточных программ приведен в [20].

Мультисистемы позволяют объединять применять несколько систем правил в заданной последовательности. При этом каждая из систем срабатывает только один раз. Например, правило (2) можно объединить с правилом

Field(a,int, _MARK_Processed)->Field(a,int) (6)

Правила (2) и (6) нельзя объединить в одной системе, поскольку они приведут к зацикливанию (после применения правила (2), а затем правила (6), снова применимо правило (2)). Однако их можно объединить в рамках одной мультисистемы из двух систем. В этом случае, сначала применяется правило (2), затем правило (6), но после его применения правило (2) уже неприменимо. Таким образом, можно реализовывать преобразования, которые содержат одинаковые элементы в исходном и результирующем термах, при этом избегая зацикливания правил.

Заготовки правил позволяют упростить создание сложных наборов правил, которые часто используются для решения определенных задач. Это облегчает работу с системой начинающих пользователей и повышает эффективность работы более опытных пользователей.

В качестве примера рассмотрим добавление элемента в список после данного. Для создания такого правила пользователь выбирает из контекстного меню для терма, соответствующего элементу списка, команду “Create Special Rule – List Insert After”. Эта команда, примененная к терму *T(x1,…,xn)*, создает следующее правило:

[T(x1,…,xn):$next]->
[T(x1,…,xn,_MARK_Processed):[_TODO_Add: $next]] (7)

Такое правило вставляет в список, содержащий терм *T(x1,…,xn),* сразу за ним, терм *_TODO_Add*. Кроме того, терм *T* помечается меткой *_MARK_Processed* . Правило (2) было создано с помощью заготовки (7). Заготовки могут содержать не только отдельные правила, но и системы и даже мультисистемы правил.

Список заготовок может расширяться пользователем. Для этого необходимо указать имя правила и создать правило-заготовку, содержащее подтерм *_CURRENT*. Этот подтерм при вызове команды Create Special Rule будет заменен на текущий подтерм (т.е. тот подтерм, для которого было вызвано контекстное меню). При создании правила-заготовки можно использовать специальные функции *_AddMark* для добавления метки и *_ClearMark* для удаления меток. Например, описанное выше правило (7) задается заготовкой

[_CURRENT: $next]->[_AddMark (_CURRENT, Processed):[_TODO_Add: $next]] (8)

Модификаторы используются для сокращения размера древовидного представления терма. Используются модификаторы двух видов: модификаторы-термы и модификаторы-списки. *Модификаторы-термы* имеют вид $m_t = m_t(f) =< f, string_f >$, где $f \in \Sigma_t$ – некоторый терминальный символ,

$string_f : T_f \to STRING$ – функция, преобразующая в строковое представление термы из множества $T_f = \{ f(t_1,...,t_n) \mid t_i \in T_v \}$ (термы с именем f). Функция $string_f$ может возвращать просто строковое представление терма (в виде $f(t_1,...,t_n)$). Кроме того, она может использовать сокращения для арифметических и других операций, аналогичные описанным в п. 2.2. Например, для терма $t = plus(a,b)$ строковое представление имеет вид $string_{plus}(t) = a + b$.

Модификаторы-списки имеют вид пары $m_l = m_l(f) =< f, R_f >$, где символ $f \in \Sigma_t$ используется для обозначения списка, а правила R_f переводят список в стандартный вид $cons(t_1, cons(t_2,..., cons(t_n, NIL)...))$. В результате применения модификатора-списка стандартный список приобретает вид одноуровневого дерева с вершиной f и узлами $t_1, t_2,..., t_n$. При этом удаляются узлы с символами *cons* и NIL, которые носят технический характер и препятствуют восприятию модели.

Паттерны (patterns) также способствуют сокращению размеров дерева, представляющего модель, однако в отличие от модификаторов паттерны вносят изменение в саму модель, а не только в ее графическое представление. Паттерны описывают соответствие между часто встречающимися комбинациями элементов модели и их сокращенным обозначением. Набор паттернов позволяет осуществлять переход между низкоуровневой (более подробной) моделью программы и ее высокоуровневым вариантом. Подробное описание паттернов приведено в п. 3.1.

3. Инструментарий на основе переписывающих правил

В данном разделе предложены три основных способа применения техники переписывающих правил для инженерии программного кода приложений для графических ускорителей:

1. Автоматизированный переход между низкоуровневыми и высокоуровневыми моделями программ;

2. Автоматизация преобразований программ с целью перехода от последовательных к параллельным программам (распараллеливание), а также повышения производительности параллельных программ для графических ускорителей по времени исполнения (оптимизация);

3. Генерация вспомогательных программ, таких как автоматизированные тесты, на основе кода основной программы.

Использование единого подхода (техники переписывающих правил) и единого инструментария для реализации различных автоматизированных преобразований программного кода упрощает работу разработчика, способствует более быстрому изучению техники и инструментальных средств переписывающих правил. Кроме того, возможно повторное использование правил, разработанных для решения одной задачи, в рамках других задач. Например, разработанные паттерны для перехода между низкоуровневыми и высокоуровневыми моделями могут использоваться также для реализации распараллеливающих и оптимизирующих преобразований.

3.1. Переход между высокоуровневыми и низкоуровневыми моделями программы.

Как описано в п. 2.3, разработанный инструментарий содержит синтаксический анализатор и генератор кода языка C#, позволяющие строить на основе исходного кода модель кода в виде термов, а также переходить от модели к исходному коду. Однако такая модель является низкоуровневой, поскольку в ней явно описаны все синтаксические конструкции языка. Например, рассмотрим достаточно простую и распространенную конструкцию языка C# – цикл типа for со счетчиком. Этому циклу соответствует фрагмент кода

```
for(int $var=$start; $var <$end; $var++) {$body}
```
(9)

Здесь использованы переменные *$var, $start, $end, $body* для указания участков кода, которые меняются в зависимости от программы. Фрагменту кода (9) соответствует следующая низкоуровневая (синтаксическая) модель:

```
For(cons(Declaration(NIL,DotList(cons(DotList(cons(int,NIL)),NIL)),cons(Assignment($var,
$start),NIL)),NIL),less($var,$end),cons(PostIncrement($var),NIL),$body)
```
(10)

Этот терм является достаточно громоздким и неудобным для понимания и изменения. Поэтому вместо него используется более высокоуровневое (алгебраическое) представление

```
ForCnt($var,$start,$end,$body)
```
(11)

Высокоуровневая модель представлена в виде термов, как и низкоуровневая модель про-

граммы. Однако в отличие от последней, высокоуровневая модель описывает не синтаксические конструкции языка программирования, а операторы алгебры алгоритмов Глушкова [18].

Преимущество использования высокоуровневых моделей программ заключается в возможности более краткой и выразительной записи преобразований программы. Однако при этом возникает необходимость перехода между моделью программы и исходным кодом. Для низкоуровневых (синтаксических) моделей такое преобразование осуществляется с использованием синтаксического анализатора и генератора для данного языка программирования. Однако для построения более высокоуровневых моделей необходимы дополнительные знания о предметной области, которые могут быть выражены в виде наборов базовых операторов и предикатов алгебры Глушкова.

В данной работе для автоматизированного перехода от исходного кода к высокоуровневой модели программы и в обратном направлении используется техника переписывающих правил. Переход производится в два этапа: между исходным кодом и низкоуровневой моделью (деревом синтаксического разбора), а затем между низкоуровневой моделью и высокоуровневой моделью (операторами алгебры Глушкова). На первом этапе используются синтаксический анализатор и генератор данного языка. Второй этап осуществляется с использованием переписывающих правил: поскольку оба вида моделей представимы в виде термов, преобразования между ними записываются в виде правил. При этом правила представлены в виде паттернов Termware (более подробно описанных в [21]). В общем случае, паттерн определяется двумя системами правил: R_p – система правил для выделения паттерна из произвольного терма, R_g – система правил для расшифровки паттерна. В более частном случае паттерн задается парой термов t_p – обозначение паттерна (элемент модели высокого уровня) и t_g – образец, задающий паттерн (элемент модели низкого уровня). В этом случае $R_p = \{t_g \to t_p\}$ и $R_g = \{t_p \to t_g\}$.

Такого рода паттерны задаются для каждого высокоуровневого оператора. Последовательное применение правил R_p для всех операторов позволяет осуществить переход от низкоуровневой к высокоуровневой модели. Аналогично, правила R_g осуществляют обратный переход. Например, для случая цикла for со счетчиком, можно использовать в качестве терма t_p выражение (11), а в качестве терма t_g – выражение (10). Полученный таким образом паттерн можно использовать для автоматического перехода от синтаксического к алгебраическому представлению цикла.

В качестве более специфического примера использования паттернов рассмотрим функцию $\_GetCoor$, которая используется для вычисления номера исходной итерации цикла по параметрам потока и блока в платформе CUDA [4]. В этом случае $t_p = \_GetCoor(\$c)$,

$$t_g = Dot(blockIdx, \$c) \ * \ Dot(blockDim, \$c) \ + \ Dot(threadIdx, \$c).$$

Таким образом, элемент высокоуровневой модели $\_GetCoor(x)$ может быть преобразован в элемент низкоуровневой модели

$$Dot(blockIdx, x) \ * \ Dot(blockDim, x) \ + \ Dot(threadIdx, x),$$

который затем преобразуется в фрагмент исходного кода *blockIdx.x* * *blockDim.x* + *threadIdx.x*. Возможно преобразование и в обратном направлении, когда фрагмент исходного кода переходит в элемент низкоуровневой модели с использованием синтаксического анализатора, а затем используется правило R_p паттерна для выделения элемента высокоуровневой модели.

Еще одной важной особенностью высокоуровневых моделей является независимость от языка реализации. Одна высокоуровневая модель программы может соответствовать низкоуровневым программам на различных языках (или с использованием различных платформ). Для поддержки разработки программ на различных языках необходима поддержка низкоуровневой модели (т.е. наличие анализатора и генератора) для каждого языка, а также набор паттернов, поддерживающих данный язык.

3.2. Переход от последовательной программы к программе для графических ускорителей. Наиболее важным применением техники переписывающих правил является автоматизация преобразований программ, в частности, переход от последовательной программы для

CPU к параллельной программе, исполняющейся на GPU. Для этого программа представляется в виде высокоуровневой модели. При этом все методы представлены в виде выражения алгебры Глушкова, которое естественным образом представляется в виде терма. К этим термам применяются переписывающие правила Termware, которые переводят исходную программу в преобразованную версию.

Рассмотрим распараллеливающие преобразования для определенных циклических конструкций. Пусть фрагмент исходной программы имеет следующий вид:

$$Ser1 = ForCnt(i, 0, m, body(i)) \quad (12)$$

Здесь использован оператор цикла со счетчиком *ForCnt*, описанный в предыдущем пункте. Оператор $body(i)$ описывает тело цикла; в общем случае это достаточно сложный оператор, в частности, он может содержать циклические конструкции. Рассмотрим следующее преобразование: участок программы (12) переходит в параллельный эквивалент:

$$Gpu1 = init\_gpu; copy\_gpu; call\_gpu(gbody1, block1d, grid1d); copy\_back \quad (13)$$

Здесь использованы операторы взаимодействия с GPU: *init_gpu* – инициализация платформы CUDA, *copy_gpu* и *copy_back* – копирование данных между памятью CPU и GPU, *call_gpu* – вызов ядра CUDA. Эти операторы также реализованы в виде паттернов: их реализация для языка C# и платформы CUDA .NET описана в [22].

Тело исходного цикла перемещается в новую функцию (ядро CUDA) $gbody1$, исполняющуюся на GPU. Это ядро имеет следующий вид:

$$gbody1 = assign(i, \_GetCoor(x)); \_CpuToGpu(body(i)) \quad (14)$$

Сначала вычисляется номер исходной итерации *i*, для этого используются параметры текущего потока (функция $\_GetCoor(x)$, которая вычисляет номер итерации по положению текущего потока в блоке и текущего блока в решетке). После этого исполняется тело цикла для этого значения. При этом используется функция $\_CpuToGpu$ для преобразования между операторами CPU- и GPU-программ.

Преобразование последовательной программы $Ser1$ в параллельную версию для графических ускорителей $Gpu1$ описывается следующими переписывающими правилами:

1. $ForCnt(\$iter, 0, \$itlm, \$body, \_MARK\_Parallel) \rightarrow$
$[init\_gpu; copy\_gpu; call\_gpu(gbody1, block1d, grid1d(\$itlm)); copy\_back]$
$[\_AddMethod(gbody1, \_CreateKernel1d(\$iter, \$body))]$
2. $\_CreateKernel1d(\$iter, \$body) \rightarrow assign(\$iter, \_GetCoor(x)); \_CpuToGpu(\$body)$
3. $grid1d(\$itlm) \rightarrow (\$itlm + block1d - 1) / block1d$

Правило 1 описывает переход фрагмента программы от $Ser1$ к $Gpu1$. При этом правило содержит действие $\_AddMethod$, которое создает новый метод. Правило 2 генерирует тело нового компонента $gbody1$. Правило 3 задает размеры вычислительной решетки для запуска ядра на основании количества итераций исходного цикла $\$itlm$. Заметим, что в правиле 1 цикл *ForCnt* помечен меткой *_MARK_Parallel*, которая указывает, какие участки кода должны быть распараллелены.

Заметим, что переписывающие правила, задающие переход, имеют достаточно простой вид и непосредственно следуют из алгебраических равенств (12–14). Это становится возможным благодаря использованию высокоуровневых моделей программ. Для сравнения можно привести аналогичные преобразования, описанные в работе [22], которые использовали низкоуровневую модель программы (дерево синтаксического разбора) и поэтому были более громоздкими и содержали большое количество технических деталей.

3.3. Оптимизация программ для графических ускорителей. Переписывающие правила могут использоваться также для выполнения

оптимизирующих преобразований. В этом случае переписывающие правила применяются таким же образом, как и в случае распараллеливания, описанного в предыдущем пункте. Единственным отличием является тот факт, что в качестве исходной программы выступает параллельная программа для GPU, которая может быть создана вручную или получена путем применения преобразований.

В качестве примера оптимизирующего преобразования рассмотрим переход от использования глобальной памяти графического ускорителя к использованию shared-памяти. Такое преобразование позволяет существенно повысить быстродействие программы, поскольку задержки при доступе к shared-памяти намного меньше, чем в случае глобальной памяти.

Преобразование такого типа не затрагивает CPU-часть программы и меняет только соответствующее GPU-ядро. В качестве исходной программы рассмотрим ядро $gbody1$ из предыдущего пункта. Преобразованное ядро будет иметь вид

$$gbody1.1 = assign(i, \_GetCoor(x)); copy\_shared(i); \_Barrier;$$
$$\_GlobalToShared(gbody(i)); \_Barrier; copy\_global(i) \quad (15)$$

Здесь использовано обозначение $gbody(i) = \_CpuToGpu(body(i))$, подчеркивающее тот факт, что тело исходного ядра не обязательно должно было быть получено преобразованием последовательной программы. Новое ядро использует два оператора $copy\_shared(i)$ и $copy\_global(i)$ для копирования данных из глобальной в shared-память и в обратном направлении. Эти операторы аналогичны операторам $copy\_gpu$ и $copy\_back$ для копирования данных между памятью CPU и GPU. Основное отличие заключается в том, что операторы $copy\_gpu$ и $copy\_back$ копируют сразу все данные, тогда как при использовании $copy\_shared(i)$ и $copy\_global(i)$ каждый поток копирует свою часть данных. Поэтому появляется необходимость синхронизации потоков с использованием оператора $\_Barrier$. Преобразование также использует функцию $\_GlobalToShared$ для перехода от операторов, действующих над глобальной памятью, к операторам над shared-памятью. Данное преобразование реализуется в виде переписывающих правил аналогично п. 3.2.

3.4. Автоматизированные тесты. Еще одним направлением использования переписывающих правил является генерация вспомогательных программ, таких как автоматизированные тесты. Такие тесты могут применяться для проверки корректности преобразования последовательной программы в параллельную для графических ускорителей. Большое количество тестов могут быть сгенерированы автоматически, благодаря применению техники переписывающих правил.

В качестве примера рассмотрим генерацию тестов для проверки независимости итераций цикла в программе $Ser1$ (более подробно этот вопрос рассмотрен в [21]). Независимость итераций необходима для распараллеливания: если итерации являются зависимыми, они должны исполняться последовательно в том же порядке, что и в исходной программе.

Для проверки этого условия можно использовать результаты исполнения программы. При этом генерируются специальные тестовые программы, которые запускают необходимый код, получают результаты его исполнения и сравнивают эти результаты с ожидаемыми. Ожидаемые результаты можно получить путем исполнения исходной программы (которая предполагается правильной).

В простейшем случае в качестве тестов можно использовать преобразованную программу для графических ускорителей (полностью или частично – только те методы, которые были изменены). При этом к программе необходимо добавить проверку результатов, что реализуется простыми правилами Termware.

Преимуществом такого подхода является то, что проверяется исполнение именно той программы, которая является результатом преобразования. Недостаток может заключаться в том, что некоторые ошибки в параллельных программах могут проявляться только при специальных условиях. Поэтому вполне возможно, что при исполнении тестов результаты совпадут с ожидаемыми, тогда как при реальном использовании программы будут возникать ошибки. Кроме того, тестовая программа получается достаточно сложной, поскольку она использует графический ускоритель и требует большое количество технического кода для связи между CPU и GPU.

В силу описанных недостатков, более предпочтительным является генерация специальным образом модифицированной последовательной тестовой программы. Например, для проверки независимости итераций цикла можно сгенерировать последовательную программу, содержащую тот же цикл, но с обращенным порядком итераций. Для этого используется такое правило:

ForCnt(*$var,$start,$end,$body,_MARK_Transform*) ->
ForStep(*$var,$end, $start,-1,$body,_MARK_Transformed*) (16)

Если в исходной программе итерации были независимыми, то такая модифицированная программа должна давать тот же результат. В противном случае результат может как отличаться, так и совпадать (если при конкретном способе перестановки операторов случайно получается такой же результат). Можно использовать и более сложные перестановки, например, развернуть цикл на 2 итерации и перемешать операторы из этих итераций.

Особенность тестов такого вида заключается в том, что если тест не проходит, это свидетельствует о невыполнении заданного условия. Если же тест проходит, это не может гарантировать выполнение условия. Поэтому необходимо использовать несколько различных тестов. Преимущество Termware заключается в том, что переписывающие правила позволяют автоматизировать создание большого количества тестов для каждой конкретной программы.

3.5. Пример: умножение матриц. В качестве примера использования предложенного подхода рассмотрим создание программы для графических ускорителей, реализующей умножение матриц. В качестве входных данных рассматривалась последовательная программа на языке C#, реализующая простейший алгоритм умножения матриц. К этой программе были применены распараллеливающие и оптимизирующие преобразования, в результате чего была получена программа для графических ускорителей, реализующая тот же алгоритм.

Исходный код последовательной программы был преобразован в низкоуровневую синтаксическую модель с использованием синтаксического анализатора языка C#. Далее к этой модели были применены паттерны для перехода к высокоуровневой алгебраической модели. По этой модели были сгенерированы тесты для проверки независимости итераций цикла. Исполнение этих тестов подтвердило независимость итераций, т.е. возможность распараллеливания программы.

Далее к высокоуровневой модели последовательной программы были применены преобразования, описанные в п. 3.2. Был использован двумерный вариант преобразований, при котором два вложенных цикла переходят в ядро CUDA, исполняемое на двумерной решетке. В полученной параллельной программе каждый элемент результирующей матрицы вычисляется на отдельном потоке.

К полученной параллельной программе для графических ускорителей были применены оптимизирующие преобразования, описанные в п. 3.3. Эти преобразования были направлены на переход к использованию быстрой shared-памяти. Участки памяти (фрагменты матриц) копировались в shared-память, вычисления производились в этой памяти, а окончательный результат копировался обратно в глобальную память.

Все варианты программы – последовательная SEQ, неоптимизированная для графических ускорителей GPU1 и оптимизированная для графических ускорителей GPU2 – исполнялись для квадратных матриц размера 512x512 и 1024x1024. Результаты измерений производительности приведены в таблице 1.

Таблица 1. Сравнение производительности

Программа	512x512		1024x1024		Увеличение времени при удвоении
	Время исполнения, с	Ускорение	Время исполнения, с	Ускорение	
SEQ	1,25	1	63,67	1	50,8
GPU1	0,35	3,5	2,89	22,0	8,2
GPU2	0,03	41,0	0,21	300,3	7,0

Как видно из таблицы, распараллеливание позволило достичь значительного повышения производительности. Для матриц размера 512x512, неоптимизированная программа для графических ускорителей продемонстрировала ускорение в 3,5 раза, тогда как оптимизация

позволила достичь ускорения в 41 раз. Это демонстрирует важность применения оптимизирующих преобразований, в частности таких, которые используют детали реализации параллельной платформы (в данном случае иерархию памяти). Для матриц размера 1024x1024 было достигнуто еще большее ускорение: в 22 раза для неоптимизированной программы и в 300 раз после оптимизации. Заметим, что ускорение для оптимизированной программы в данном случае оказалось больше, чем количество вычислительных ядер в GPU (в данном случае 128). Это объясняется особенностями использования памяти в CPU- и GPU-программах. Для небольших матриц CPU-программа может использовать кеш, что существенно повышает производительность. Однако для больших матриц все данные не помещаются в кеш, поэтому производительность программы резко падает. Для GPU-программ нет такого ограничения: shared-память (аналог кеша) используется одинаково, независимо от размера матрицы. Поэтому для больших матриц программа для графических ускорителей работает существенно более эффективно, что подтверждается полученным ускорением. Это наблюдение подтверждается последним столбцом табл.1, где приведено увеличение времени работы при удвоении размера матрицы. Поскольку алгоритм умножения матриц имеет сложность $O(n^3)$, при удвоении размера время работы должно увеличиваться в 8 раз. Это выполнено для программ GPU1 и GPU2 (для последней получено увеличение в 7,2 раза, поскольку большее количество вычислений маскирует потери на копирование данных между CPU- и GPU-памятью). Однако для последовательной программы время исполнения увеличилось в 50 раз, что подтверждает тот факт, что CPU-программа перешла к менее эффективной работе с памятью.

Выводы

В работе описан подход к проектированию и разработке эффективных параллельных программ для графических ускорителей, основанный на технике переписывающих правил. Использование автоматизированных преобразований программ в виде переписывающих правил упрощает переход на новую параллельную платформу, позволяет добиться высокой производительности программ, способствует повторному использованию программных артефактов. Разработанный инструментарий содержит средства для применения переписывающих правил к инженерии программного кода: синтаксический анализатор и генератор кода языка C#, систему переписывающих правил, графический интерфейс для визуального представления и редактирования правил и моделей программ. Кроме того, инструментарий поддерживает ряд дополнительных возможностей, направленных на упрощение изучения техники переписывающих правил и повышение производительности разработчика. Описано использование инструментария для различных задач в области инженерии программного кода: работа с высокоуровневыми моделями программ, автоматизация распараллеливающих и оптимизирующих преобразований, создание тестов. Применение описанного подхода и инструментария к вычислительной задаче продемонстрировало высокую производительность полученных параллельных программ для графических ускорителей.

Дальнейшие исследования в данном направлении предполагают реализацию дополнительных преобразований в виде переписывающих правил и создание библиотеки готовых правил и заготовок для повторного использования. Также планируется апробация разработанных средств на более сложных проектах. Еще одним направлением исследований является поддержка других высокоуровневых языков (таких как Java) и платформ программирования графических ускорителей (таких как OpenCL).

Литература

1. *Ryoo S., Rodrigues C.I., Baghsorkhi S.S., Stone S.S., Kirk D.B., and Hwu W.W.* Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08, Salt Lake City, UT, USA, February 20–23, 2008). – P. 73–82.

2. *General-Purpose* Computation Using Graphics Hardware. http://www.gpgpu.org .

3. *Fatahalian K., Sugerman J., and Hanrahan P.* Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In ACM SIGGRAPH/EUROGRAPHICS Conf. on Graphics Hardware, 2004. – P. 133–137.

4. *NVidia* CUDA technology. http://www.nvidia.com/cuda .

5. *AMD* (ATI) Stream technology. http://www.amd.com/stream .

6. *N. Anderson, J. Mache, W. Watson.* Learning CUDA: lab exercises and experiences. In Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion (SPLASH '10). – 2010. – P. 183– 188.

7. *Doroshenko A., Shevchenko R.* A Rewriting Framework for Rule-Based Programming Dynamic Applications, Fundamenta Informaticae. – 2006, Vol. 72, N 1–3. –P. 95–108.

8. *TermWare*. – http://www.gradsoft.com.ua/products/termware_rus.html .

9. *CUDA* .NET http://www.gass-ltd.co.il/en/products/cuda.net/ .

10. *Lee S., Min S., and Eigenmann R.* OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '09). Raleigh, NC, USA, February 14–18, 2009. – P. 101–110.

11. *OpenMP* specification. http://openmp.org/ wp/

12. *Baskaran M., Bondhugula U., Krishnamoorthy S., Ramanujam J., Rountev A., and Sadayappan P.* A compiler framework for optimization of affine loop nests for gpgpus. In Proceedings of the 22nd Annual international Conf. on Supercomuting (ICS '08). Island of Kos, Greece, June 07–12, 2008). – P. 225–234.

13. *Ma W. and Agrawal G.* A compiler and runtime system for enabling data mining applications on gpus. In Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '09). Raleigh, NC, USA, February 14 – 18, 2009. – P. 287–288.

14. *Allusse Y., Horain P., Agarwal A., and Saipriyadarshan C.* GpuCV: an open source GPU-accelerated framework for image processing and computer vision. In Proceeding of the 16th ACM International Conf. on Multimedia (MM '08).Vancouver, British Columbia, Canada, October 26–31, 2008. – P. 1089–1092.

15. *Lefohn A. E., Sengupta S., Kniss J., Strzodka R., and Owens J. D.* Glift: Generic, efficient, random-access GPU data structures. ACM Trans. Graph. 25, 1 Jan. 2006. – P. 60–99.

16. *Han T. D. and Abdelrahman T. S.* hiCUDA: a high-level directive-based language for GPU programming. In Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-2). Washington, D.C., March 08 – 08, 2009)., Vol. 383. – P. 52–61.

17. *Hou, Q., Zhou, K., and Guo, B.* BSGP: bulk-synchronous GPU programming. In ACM SIGGRAPH 2008 Papers. Los Angeles, California, August 11 – 15, 2008. – P. 1–12.

18. *Андон Ф.И., Дорошенко А.Е., Цейтлин Г.Е., Яценко Е.А.* Алгеброалгоритмические модели и методы параллельного программирования. – К.: Академпериодика, 2007. – 631 с.

19. *Жереб К.А.* Программный инструментарий, основанный на правилах, для автоматизации разработки приложений на платформе Microsoft .NET// Управляющие системы и машины. – 2009. – № 4. – С. 51–59.

20. *Дорошенко А.Е., Жереб К.А., Яценко Е.А.* Об оценке сложности и координации вычислений в многопоточных программах // Проблемы программирования. – 2007. – № 2. – С. 41–55.

21. *Дорошенко А.Е., Жереб К.А.* Алгебродинамические модели для распараллеливания программ // Проблемы программирования. – 2010. – № 1. – С. 39–55.

22. *Дорошенко А.Е., Жереб К.А.* Разработка высокопараллельных приложений для графических ускорителей с использованием переписывающих правил // Проблемы программирования. – 2009. – № 3. – С. 3–18.

Сведения об авторах

Дорошенко Анатолий Ефимович доктор физико-математических наук, профессор, заведующий отделом теории компьютерных вычислений Института программных систем НАН Украины.
Научные интересы – формальные методы разработки программного обеспечения, методы программирования, параллельные вычисления и интеллектуализация компьютерных приложений
03680, Киев-187, Проспект Академика Глушкова, 40.
E-mail: doroshenkoanatoliy2@gmail.com

Жереб Константин Анатольевич, младший научный сотрудник отдела теории компьютерных вычислений Института программных систем НАН Украины.
Научные интересы – автоматизация разработки программного обеспечения, параллельные вычисления, техника переписывающих правил и процессов разработки программного обеспечения.
03680, Киев-187, Проспект Академика Глушкова, 40.
E-mail: zhereb@gmail.com

Статья поступила в редакцію 16.10.2010 г.

