ПРОГРАММНО–
ТЕХНИЧЕСКИЕ
КОМПЛЕКСЫ

Ф.И. АНДОН, А.Е. ДОРОШЕНКО, К.А. ЖЕРЕБ

УДК 681.3

ПРОГРАММИРОВАНИЕ ВЫСОКОПРОИЗВОДИТЕЛЬНЫХ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ: ФОРМАЛЬНЫЕ МОДЕЛИ И ГРАФИЧЕСКИЕ УСКОРИТЕЛИ

Ключевые слова: *высокопроизводительные параллельные вычисления, формальные методы программирования, алгебро-динамические модели программ, переписывающие правила, графические ускорители.*

ВВЕДЕНИЕ

Развитие архитектуры и методов программирования высокопроизводительных параллельных вычислительных систем в последнее десятилетие претерпело важные изменения. До средины 2000-х годов повышение производительности компьютеров могло осуществляться как за счет роста тактовой частоты процессоров, так и увеличения количества процессоров. Однако во второй половине прошедшего десятилетия в связи с отказом производителей от дальнейшего наращивания тактовой частоты из-за трудностей решения проблемы теплоотвода и появлением многоядерных архитектур [1] практически осталась только последняя из названных возможностей. Для программистов и исследователей такой поворот означал только одно — смену парадигмы и необходимость создания новых моделей программирования параллельных вычислений, соответствующих новым реалиям в развитии вычислительной техники. Первые попытки радикального решения проблемы на основе какой-либо отдельной технологии (такие как концепция транзакционной памяти, предложенная Intel [2], и координационная библиотека CCR, разработанная Microsoft [3]) успеха не имели. Это лишний раз показало, что разработка эффективных программ для нового класса многоядерных архитектур — масштабная научно-техническая проблема, успешное решение которой может быть обеспечено поэтапно, путем комплексного учета специфики предметных областей и глубокого охвата этапов жизненного цикла разрабатываемых программ с применением средств автоматизации проектирования и программирования — от написания первоначальных спецификаций до генерации выполняемого кода.

Основой для такой автоматизации является, прежде всего, высокоуровневая формализация конструирования многопоточных программ и автоматизация формальных трансформаций программ для оптимизации их производительности по заданным критериям (память, быстродействие, загрузка оборудования и др.). В данной работе предложены формальные алгебро-динамические модели параллельных вычислений, предназначенные для автоматизации программирования сравнительно нового класса мультипроцессорных систем — видеографических ускорителей общего назначения (GPGPU). Опыт применения таких моделей для автоматизации проектирования и программирования параллельных вычислений

берет начало от проекта макроконвейерного вычислительного комплекса [4], который разрабатывался в Институте кибернетики НАН Украины в 80–90-е годы. Несмотря на отличия этих проектов, их объединяют общие проблемы эффективного управления многоуровневой памятью многопроцессорной системы для получения высокой производительности вычислений.

В разд. 1 кратко описаны особенности архитектуры видеографических ускорителей с точки зрения программирования параллельных вычислений. Разд. 2 содержит описание алгебро-динамических моделей программ для обычного универсального процессора (CPU) и графического ускорителя (GPU). В разд. 3 описано применение переписывающих правил для преобразования программ, а также намечен переход к высокоуровневым моделям программ.

1. ГРАФИЧЕСКИЕ УСКОРИТЕЛИ КАК МУЛЬТИПРОЦЕССОРНЫЕ СИСТЕМЫ

Многопоточное программирование в настоящее время ассоциируется в основном с многоядерными процессорами общего назначения (multi-core CPUs [1]). Но существует еще одно активно развивающееся направление параллельного многопоточного программирования — программирование общецелевых задач для GPU [5]. Рыночные требования способствовали бурному развитию GPU, в результате их вычислительная мощность на данный момент значительно превышает возможности обычных процессоров. Поэтому возник интерес к использованию GPU для решения задач, не связанных напрямую с обработкой графики. Такой интерес поддерживается разработчиками графических ускорителей: в частности, компания NVidia представляет платформу CUDA [6] для вычислений на графическом ускорителе.

Схема устройства GPU показана на рис. 1 (рисунок взят из документации платформы CUDA [6]). Графическое устройство (device) содержит несколько мультипроцессоров, а также общую для них графическую память. Каждый мультипроцессор содержит несколько вычислительных ядер (скалярных процессоров) и один управляющий блок, поддерживающий многопоточное исполнение. В результате количество вычислительных ядер (а значит, и степень возможного параллелизма) существенно выше, чем у общецелевых многоядерных процессоров.

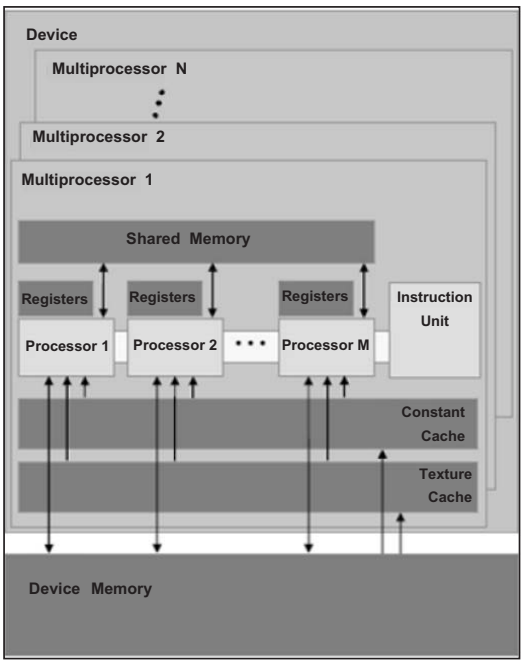

Рис. 1. Общая схема основных аппаратных компонентов GPU

GPU 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, но имеет ограниченный размер и требует явной синхронизации доступа. Имеется 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 ре-

жимы доступа, полезные для графических задач. Всем мультипроцессорам доступна также графическая память устройств, самая большая по объему и самая медленная для GPU.

Параллельная программа для GPU состоит из большого количества потоков. Особенности аппаратного обеспечения, а именно большое количество вычислительных ядер, позволяют использовать очень мелкозернистый параллелизм, вплоть до выделения отдельного потока для каждого элемента данных.

Для исполнения на GPU потоки объединяются в блоки. Каждый блок выполняется на одном мультипроцессоре. Различные блоки, соответствующие одной или нескольким программам, по возможности распределяются равномерно по доступным мультипроцессорам. Следует отметить, что наличие ветвлений в потоках одного блока понижает производительность, поскольку каждый из вариантов исполнения в таких случаях выполняется последовательно, а остальные потоки ожидают выполнения.

Несмотря на наличие специализированных средств CUDA, разработка программ для GPU остается трудоемкой, требующей от разработчика знания низкоуровневых деталей аппаратной и программной платформы. Поэтому актуальной является задача автоматизации процесса разработки. В данной статье предложено развитие формальных методов проектирования, основанное на концепциях алгебраического программирования [7] и алгебро-динамических моделей программ [8] с использованием техники переписывающих правил [9–11] для автоматизированной разработки эффективных программ для GPU. Разработаны высокоуровневые модели программ и модели исполнения программ для CPU и GPU. Описано использование переписывающих правил и высокоуровневых моделей для автоматизированного распараллеливания и оптимизации программ для GPU. Предложен метод автоматизированного перехода между высокоуровневой моделью программы и исходным кодом, основанный на использовании переписывающих правил специального вида. Разработанные формальные средства проиллюстрированы на конкретных задачах, которые показывают высокую эффективность преобразований.

Настоящая работа продолжает исследования, начатые в [12–14], по автоматизации процесса проектирования и разработки эффективных параллельных программ. Особенность данной публикации — использование новой аппаратной платформы для параллельных вычислений, GPU, что позволяет добиться значительного повышения производительности по сравнению с применением многоядерных процессоров общего назначения. Данная платформа исследовалась в [15], однако отличие настоящей работы состоит в использовании более формализованных методов и высокоуровневых представлений программ. Это позволяет упростить запись преобразований, а также применить одни высокоуровневые преобразования для программ на разных языках программирования.

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 GPU. Так, в работе [16] рассмотрен автоматический переход от многопоточной программы, реализованной с использованием технологии OpenMP [17], к реализации данной программы на платформе CUDA. В [18] описана платформа для оптимизации циклов в программах для GPU. Разработаны системы для автоматического распараллеливания и оптимизации программ из конкретной предметной области, например data mining [19] или обработка изображений [20]. Авторы работы [21] описывают библиотеку высокоуровневых структур данных для GPU. Также разрабатываются платформы их программирования, предоставляющие более высокоуровневые средства по сравнению с CUDA, такие как hiCUDA [22] и BSGP [23]. Отличие данной публикации заключается в использовании формальных моделей и методов, что позволяет совместить краткость описания и эффективность реализации преобразований.

2. АЛГЕБРО-ДИНАМИЧЕСКИЕ МОДЕЛИ ПРОГРАММ

2.1. Модель исходного кода программы. Алгебро-динамические модели программ состоят из двух частей: модель исходного кода программы и модель исполнения программы. В данном разделе рассмотрим следующую модель исходного кода. Программа состоит из множества процедур, соответствующих отдельным методам (функциям) языка программирования: $P = \{P_1, \dots, P_k\}$. Будем считать, что процедура описывается идентификатором (именем, уникальным в пределах программы), а также моделью кода: $P_i = (name_i, code_i)$. В дальнейшем для сокращения записи будем отождествлять обозначение для процедуры в целом P_i и ее имени $name_i$.

Для моделирования кода используем алгебру алгоритмов Глушкова. Процедурный код будем представлять в виде выражения этой алгебры. Алгебра Глушкова (АГ) является двухосновной алгеброй $A(Y, U)$, содержащей множества операторов Y и условий (предикатов) U. В алгебре определены стандартные операции: логические операции конъюнкции *AND*, дизъюнкции *OR* и отрицания *NOT*, последовательная композиция *THEN*, ветвление *IF*, итерация (цикл) *WHILE*.

На множествах операторов и условий вводятся базовые элементы, что дает возможность строить различные выражения алгебры, которые будут описывать сложные операторы и условия. Базовые операторы и условия обычно зависят от предметной области. Выделим один базовый оператор, общий для всех предметных областей: вызов процедуры $call(P_i)$.

Особенностью программ для GPU является их разделение на две части: код, который исполняется на CPU, и специализированный код для GPU. Эти части могут быть реализованы на разных языках. Например, в [15] рассматривались программы, в которых CPU-код реализован на C#, тогда как GPU-код — на C for CUDA, специальном расширении языка C.

В модели программы эту особенность будем учитывать следующим образом: считать, что каждая процедура относится либо к CPU-, либо к GPU-коду. При этом для CPU и GPU наборы базовых операторов могут различаться.

2.2. Модель исполнения последовательной программы. Для описания исполнения программ используем общие понятия теории дискретных динамических систем (transition systems) [8]. Дискретная динамическая система (ДДС) определяется как тройка (S_0, S, d), где S — множество, называемое пространством состояний; $S_0 \subseteq S$ — множество начальных состояний; $d \subset S \times S$ — бинарное отношение переходов в пространстве состояний. Система может перейти из состояния s_i в состояние s_j, если $(s_i, s_j) \in d$.

Для построения модели исполнения необходимо определить интерпретацию выражений АГ, из которых состоит программа. Пусть V — множество переменных программы. Для простоты будем считать, что переменные не имеют типа и принимают значения в некотором универсальном множестве D. Частичные отображения $b\colon V \to D$ переменных на их значения будем называть состояниями памяти. Информационной средой является множество состояний памяти: $B = \{b\colon V \to D\}$. Тогда операторы алгебры алгоритмов интерпретируются как функции на множестве B, а условия — как предикаты на том же множестве. Интерпретация базовых операторов и условий определяется в рамках соответствующей предметной области.

Теперь определим ДДС, которая описывает исполнение последовательной программы. Состояния ДДС имеют вид $s = (b, R, F)$, где $b \in B$ — текущее состояние памяти, $R \in Y$ — текущее состояние управления, представленное в виде остаточной программы, $F \in (P \to Y)^*$ — стек вызовов процедур, т.е. последовательность идентификаторов процедур и операторов АГ, описывающих состояние управления данной процедуры. Начальное состояние для заданной программы имеет вид $s_0 = (b_0, Y(P_m), (P_m \to \varnothing))$, где P_m — основная процедура (точка входа в программу). Отношение перехода задается следующими правилами перехода:

1) $(b, yR, F) \to (y(b), R, F)$, где $y \in Y$ — базовый оператор;

2) $(b, if\,(u, P, Q)R, F) \to \begin{cases} (b, PR, F), u(b) = 1, \\ (b, QR, F), u(b) = 0; \end{cases}$

3) $(b, while(u, P)R, F) \to \begin{cases} (b, Pwhile(u, P)R, F), u(b) = 1, \\ (b, R, F), u(b) = 0; \end{cases}$

4) $(b, call(P_j)R, (\ldots, P_i \to \varnothing)) \to (b, Y(P_j), (\ldots, P_i \to R, P_j \to \varnothing))$;

5) $(b, \varepsilon, (\ldots, P_i \to R, P_j \to \varnothing)) \to (b, R, (\ldots, P_i \to \varnothing))$.

Правило 1 определяет выполнение базовых операторов. Правила 2 и 3 описывают ветвление и циклическую конструкцию. Работа с процедурами описана правилами 4 (вызов процедуры) и 5 (возврат из процедуры). Финальные состояния (из которых невозможен переход ни в какое другое состояние) имеют вид $s_f = (b, \varepsilon, \varnothing)$.

Определенную таким образом ДДС обозначим S^{ser}. Она моделирует исполнение заданной последовательной программы. Отметим, что система S^{ser} детерминированная, поскольку для каждого состояния переход определен однозначно.

2.3. CPU-программа. Опираясь на построенную модель для последовательных программ, опишем аналогичную модель для параллельных программ, исполняемых на графических GPU. При построении модели исполнения программ для данных ускорителей будем отдельно рассматривать исполнение на CPU и на GPU. Для каждого режима построим соответствующую модель в виде ДДС. Модель исполнения программы в целом, S^{gc}, строится как объединение этих двух ДДС.

Код, относящийся к CPU, исполняется так же, как и любая обычная программа, поэтому для моделирования этой части можно использовать модель S^{ser}, в которую нужно добавить средства взаимодействия с GPU. Рассмотрим следующие новые операторы: $init\_gpu$ — инициализация GPU, $copy\_gpu(m_G, m_C)$ — копирование данных в видеопамять, $copy\_back(m_C, m_G)$ — копирование результатов из видеопамяти и $call\_gpu(P_i, block, grid)$ — запуск кода (ядра CUDA) на GPU. Соответственно потребуется добавить правила переходов, которые будут менять состояние ДДС S^{gc} в целом, а не только один из ее компонентов. Формальное описание этих правил приведено в п. 2.6.

2.4. GPU-программа: уровень блоков. Модель S^{gpu} исполнения кода на GPU строится по многоуровневому принципу [8]: состояния более высоких уровней ДДС являются комбинацией состояний более низких уровней. На низшем уровне моделируется исполнение отдельных потоков; для этого используется модифицированная ДДС S^{ser}. Переходы ДДС отдельных потоков объединяются в переход ДДС более высокого уровня.

Модель S^{gpu} содержит три уровня вложенности состояний: потоков, блоков и программы в целом. Отдельные потоки (первый уровень) объединяются в блоки (второй уровень), которые, в свою очередь, формируют GPU-программу в целом (третий уровень).

Важной особенностью модели S^{gpu} по сравнению с последовательной моделью S^{ser} является наличие иерархии памяти. В CUDA используется пять видов памяти: регистры, разделяемая (shared) память, кеш констант, кеш текстур и глобальная память. В рамках модели ограничимся рассмотрением двух наиболее часто используемых видов памяти, а именно shared-память и глобальная память.

Различные виды памяти в модели проявляются в виде дополнительных компонентов состояния. Для низшего уровня (потоков) состояние имеет вид $s = (b_g, b_s, R, F)$, где $b_g \in B_g$ — состояние глобальной памяти, $b_s \in B_s$ — состояние shared-памяти. Однако для упрощения модели будем объединять состояния различных видов памяти в общее состояние $b = b_g \cup b_s$, $b \in B = B_g \times B_s$, считая, что все операторы и предикаты АГ также действуют на объединенном множестве B.

На уровне потоков действуют правила перехода 1–5 (правило 4) имеет дополнительное ограничение — вызываемая функция должна работать на GPU, что описывается модификатором `__device__` в C for CUDA.

На уровне блоков состоянием является множество потоков, каждый со своим состоянием $s^2 = \{T_i \to s_i^1\}$. Переходы на уровне блоков объединяются из переходов на уровне потоков. Эта процедура проводится следующим образом: выбирается некоторое подмножество потоков; для каждого из них осуществляется переход в соответствии с отношением d^{gpu1}, и новое состояние программы получается объединением состояний отдельных потоков. Новые состояния отдельных потоков используют состояние управления, определенное отношением переходов для S^{ser}. Состояние общей памяти определяется функцией $merge: B \times B^* \to B$. Эта функция меняет состояние каждой переменной, которая была изменена хотя бы в одном из потоков. Формально

$$merge(b_0, b_1, \ldots, b_k) = \left\{ v_i \to \left\{ \begin{array}{l} !\{b_j(v_i) \mid b_j(v_i) \neq b_0(v_i),\ \ j = \overline{1,k}\} \\ b_0(v_i) \end{array} \right. \right\}$$

(здесь использовано обозначение $!A$ для произвольного элемента множества A). Функция *merge* применяется к обоим видам памяти: b_g, b_s (или к объединенной памяти b).

Следует отметить важное ограничение на допустимые переходы: все одновременно срабатывающие потоки должны исполнять одинаковые операторы. Это объясняется тем, что в архитектуре GPU на каждый блок выделяется только одно управляющее устройство [6]. Поэтому потоки могут работать одновременно, только если они исполняют одну инструкцию.

Для синхронизации потоков добавим оператор *_ Barrier*. Он используется для синхронизации состояния потоков в пределах блока: каждый поток, достигший данного оператора, ожидает, пока все остальные потоки также достигнут его. Действие оператора *_ Barrier* описывается следующими правилами перехода:

6) $(b, \_\ Barrier; R, F) \to (inc(b), waiting\_\ Barrier; R, F)$;

7) $(b, waiting\_\ Barrier; R, F) \to (zero(b), R, F)$ при условии $bc = threads$.

Используется дополнительная переменная *bc*, описывающая количество потоков, ожидающих исполнения. Операторы $inc(b)$: $bc \leftarrow bc + 1$ и $zero(b)$: $bc \leftarrow 0$ соответственно добавляют новый поток к количеству ожидающих и очищают список ожидания. В C for CUDA оператор *_ Barrier* реализован примитивом `__syncthreads()`.

2.5. GPU-программа: уровень программы. Уровень программы в целом строится из уровня блоков таким же образом, как уровень блоков строился из уровня потоков. Состояниями программы являются отображения из множества блоков во множество состояний уровня блоков: $s^3 = \{B_j \to s_j^2\}$. Как и на втором уровне, количество блоков постоянно и определяется параметрами запуска ядра.

В отличие от уровня блоков, shared-память уникальная для каждого блока, а не общая, как глобальная память. Поэтому объединение результатов функцией *merge* на данном уровне используется только для глобальной памяти b_g.

Еще одной особенностью объединенных переходов на уровне всей программы является следующее ограничение: не может начаться исполнение нового блока (который находится в начальном состоянии уровня блоков), если существует хотя бы один блок, который начал исполняться, но не задействован в данном переходе. Иными словами, не происходит переключение между разными одновременно исполняемыми блоками; каждый блок, начавший исполняться, исполняется до конца.

На уровне программы существуют свои средства синхронизации (например, атомарные операции [6]). Однако они доступны не во всех устройствах, сильно замедляют выполнение программы, к тому же противоречат идеологии CUDA,

согласно которой блоки должны исполняться независимо и произвольно, поэтому эти средства в модель не включаются.

2.6. Взаимодействие CPU и GPU. Совместная работа двух частей программы координируется из CPU-кода с использованием операторов *copy_gpu*, *copy_back* и *call_gpu* (оператор *init_gpu* используется однократно в начале программы и не влияет на дальнейшее исполнение). Первые два оператора копируют данные между CPU и GPU. Формально это сводится к тому, что некоторое множество переменных (обычно массив) в состоянии одного устройства принимают те же значения, что и соответствующие переменные в состоянии другого.

Оператор *call_gpu* собственно запускает код для исполнения на графическом ускорителе. Его работа описывается двумя правилами:

8) $(s^c, (b_g, \varnothing)) \to (s^c, s_0^g(P_i, block, grid))$;

9) $(s^c, s_f^g) \to ((b, R, F), (b'_g, \varnothing))$.

Здесь использовано обозначение $s^c = (b, call\_gpu(P_i, block, grid)R, F)$. Правило 8 описывает создание начального состояния ДДС S^{gpu} при исполнении оператора *call_gpu*. Параметры этого состояния — количество блоков, потоков, исполняемый модуль, которые извлекаются из параметров оператора. Отметим, что оператор не удаляется из состояния управления: с точки зрения CPU, этот оператор выполняется все время, пока идут вычисления на GPU. Правило 9 выполняется, когда вычисления на GPU окончены. Оно очищает состояние управления GPU и завершает выполнение оператора *call_gpu*. Состояние памяти CPU в процессе вычислений на GPU не меняется: требуется явное копирование результатов вычислений оператором *copy_back*.

2.7. Применение алгебро-динамических моделей. Построенные алгебро-динамические модели исполнения программ для GPU позволяют проводить формальный анализ работы таких программ, поэтому такие модели полезны при разработке программ. В частности, в работе [14] авторы предложили использовать алгебро-динамические модели для доказательства корректности преобразований многопоточных программ. При этом формулируются некоторые свойства программ (например, отсутствие тупиков или конфликтных ситуаций) и доказывается корректность определенного класса преобразований при наличии определенных свойств программ. Такой же подход применим и к программам для GPU.

Еще одним направлением использования алгебро-динамических моделей исполнения программ может быть оценка времени выполнения программы. В работе [13] описан общий подход к оценке сложности алгоритма, представленного в виде выражения АГ; однако там использована абстрактная модель вычислителя, которая не позволяет учесть аппаратные особенности GPU. Предложенные в данной работе алгебро-динамические модели достаточно подробно описывают процесс исполнения программы на GPU, поэтому появляется возможность более точной оценки времени исполнения. В частности, можно описать влияние на производительность программы способов доступа к памяти и ветвлений в близких потоках.

3. ПЕРЕПИСЫВАЮЩИЕ ПРАВИЛА И ПРЕОБРАЗОВАНИЯ ПРОГРАММ

3.1. Система Termware. Для автоматизации преобразований программ используем систему переписывающих правил Termware [9, 10]. Она предназначена для описания преобразования над термами, т.е. выражениями вида $f(t_1, \ldots, t_n)$. Для задания преобразований используются правила Termware, т.е. конструкции вида

```
source [condition] -> destination [action].
```

Здесь **`source`** — исходный терм (образец для поиска), **`condition`** — условие применения правила, **`destination`** — преобразованный терм, **`action`** — дополнительное действие при срабатывании правила. Каждый из четырех компонентов

правила может содержать переменные (которые записываются в виде **$var**), что обеспечивает общность правил. Компоненты **condition** и **action** необязательны. Они могут исполнять произвольный процедурный код, в частности использовать дополнительные данные о программе.

3.2. Переход от последовательной программы к программе для GPU. Переписывающие правила позволяют автоматизировать преобразования программ, в частности переход от последовательной программы для CPU к параллельной программе, исполняющейся на GPU. Для этого программа представляется в виде высокоуровневой модели, описанной в п. 2.1. Код каждой процедуры $code_i$ моделируется в виде выражения АГ, 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:

$$Ser1 = for(i, 0, m, body(i)) . \quad (1)$$

Здесь использован оператор цикла со счетчиком $for(var, min, max, body)$, который выражается общим оператором цикла *while.* Оператор $body(i)$ описывает тело цикла; в общем случае это достаточно сложный оператор, в частности, он может содержать вложенные циклические конструкции. Рассмотрим следующее преобразование: данный участок программы переходит в параллельный эквивалент:

$$Gpu1 = init\_gpu; copy\_gpu; call\_gpu(gbody1, block1d, grid1d); copy\_back. \quad (2)$$

Здесь использованы операторы взаимодействия с GPU, описанные в п. 2.3. Собственно тело цикло перемещается в новую процедуру $gbody1$, исполняющуюся на GPU. Эта процедура имеет следующий вид:

$$gbody1 = assign(i, \_GetCoor(x)); \_CpuToGpu(body(i)). \quad (3)$$

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). Затем 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. $for(\$iter,0,\$itlm,\$body) \rightarrow$
   $[init\_gpu; copy\_gpu; call\_gpu(gbody1, block1d, grid1d(\$itlm)); copy\_back]$
   $[\_AddMethod(gbody1, \_CreateKernel1d(\$iter, \$body))]$.
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 новую процедуру. Правило 2 генерирует тело новой процедуры $gbody1$. Правило 3 задает размеры вычислительной решетки для запуска ядра на основании количества итераций исходного цикла $\$itlm$.

Заметим, что переписывающие правила, задающие переход, имеют достаточно простой вид и благодаря использованию высокоуровневых моделей программ непосредственно следуют из алгебраических равенств (1)–(3). Для сравнения приведем аналогичные преобразования, описанные в [15], которые использовали низкоуровневую модель программы (дерево синтаксического разбора) и поэтому были более громоздкими и содержали большое количество технических деталей.

3.3. Оптимизация программ для GPU. Переписывающие правила могут использоваться также для выполнения оптимизирующих преобразований. В этом случае переписывающие правила применяются таким же образом, как и при

распараллеливании, описанном в предыдущем пункте. Единственное отличие заключается в том, что исходной программой является параллельная программа для GPU, которая может быть создана вручную или получена путем преобразований.

В качестве примера оптимизирующего преобразования рассмотрим переход от применения глобальной памяти GPU 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 *gbody*1 из предыдущего пункта. Преобразованное ядро будет иметь вид

$$gbody1.1 = assign(i, \_GetCoor(x)); copy\_shared(i); \_Barrier;$$

$$\_GlobalToShared(gbody(i)); \_Barrier; copy\_global(i).$$

Здесь использовано обозначение $gbody(i) = \_CpuToGpu(body(i))$, подчеркивающее, что тело исходного ядра не обязательно должно 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 необходима синхронизацияи потоков с использованием оператора $\_Barrier$. Преобразование также использует оператор $\_GlobalToShared$ для перехода от операторов, действующих над глобальной памятью B_g, к операторам над shared-памятью B_s.

Правила, реализующие данное преобразование, аналогичны рассмотренным в п. 3.2 и не приводятся из-за ограниченного объема статьи.

3.4. Переход между высокоуровневыми и низкоуровневыми моделями программы. Как уже упоминалось, преимущество использования высокоуровневых моделей программ заключается в возможности более краткой и выразительной записи преобразований программы. Однако при этом возникает необходимость перехода от модели программы к исходному коду.

Для более низкоуровневых моделей, фактически соответствующих дереву синтаксического разбора, такое преобразование осуществляется с помощью синтаксического анализатора и генератора для данного языка программирования. Этот подход использовался, например, в [15]. Однако для построения более высокоуровневых моделей необходимы дополнительные знания о предметной области, которые можно выразить в виде наборов базовых операторов и предикатов АГ.

В данной работе для перехода от исходного кода к высокоуровневой модели программы и в обратном направлении также используются переписывающие правила. Переход проводится в два этапа: между исходным кодом и низкоуровневой моделью (деревом синтаксического разбора), а затем между низкоуровневой и высокоуровневой моделью (операторами АГ). На первом этапе используются синтаксический анализатор и генератор данного языка. Второй этап осуществляется с помощью 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 [14]). 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 перейти от низкоуровневой к высокоуровневой модели. Аналогично правила R_g осуществляют обратный переход. В качестве примера паттернов рассмотрим функцию *_GetCoor*, которая используется для вычисления номера исходной итерации цикла по параметрам потока и блока. В этом случае $t_p = \_GetCoor(\$c)$, $t_g = Dot(blockIdx,\$c) * Dot(blockDim,\$c) + Dot(threadIdx,\$c)$. Таким образом, возможно преобразование элемента высокоуровневой модели $\_GetCoor(x)$ 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 применяется правило R_p паттерна для выделения элемента высокоуровневой модели.

Еще одна важная особенность высокоуровневых моделей — 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.

Рассмотрим, как один оператор высокоуровневой модели по-разному выглядит в различных языках (реализация оператора *init_gpu* для C и C#). В первом случае этот оператор сводится к вызову одной функции: $t_g^c = FunctionCall(InitCUDA, NIL)$. В результате генерируется фрагмент кода **`InitCUDA()`**. Для C# тот же оператор приводит к созданию объекта специального типа: $t_g^{cs} = DeclarationAssignment(cuda, CUDA, New(CUDA, [0, true]))$. Соответствующий код имеет вид CUDA **`cuda = new CUDA(0, true)`**.

Таким образом, использование высокоуровневых моделей позволяет описывать преобразования программ независимо от конкретного языка реализации.

3.5. Пример использования преобразований. Рассмотрим использование описанных преобразований на примере задачи умножения матриц. Исходная программа реализована на языке C#. По этой программе построена сначала низкоуровневая модель (с использованием синтаксического анализатора языка C#), а затем и высокоуровневая модель (с использованием паттернов для операторов арифметических действий и работы с двумерными массивами). К высокоуровневой модели сначала применялись распараллеливающее, а затем оптимизирующее преобразования, описанные в пп. 3.2 и 3.3. Далее к полученной высокоуровневой модели распараллеленной и оптимизированной программы применялся обратный процесс: сначала использовались паттерны для перехода к низкоуровневой модели, которая затем переводилась в исходный код с применением генератора кода. Этот процесс проводился для двух языков: C# и C. (Различия между полученными программами заключалось только в CPU-части программы: GPU-программа в обоих случаях была одинаковой и реализованной на языке C for CUDA.)

Результаты измерения производительности различных вариантов программ приведены в табл. 1. Как видно из таблицы, для данной задачи даже неоптимизированный вариант дает четырехкратное ускорение, а оптимизированный позволяет уменьшить время работы более чем в 20 раз. Заметим,

Таблица 1

Версия программы	Время исполнения, с	Коэффициент ускорения
Последовательная, C#	4,02	1
Параллельная, C#	0,97	4,1
Оптимизированная, C#	0,19	21,4
Последовательная, C	3,58	1
Параллельная, C	0,96	3,7
Оптимизированная, C	0,15	23,2

что основное время исполнения GPU-программы занимают не сами вычисления, а копирование данных и результатов. Если данные уже имеются в видеопамяти и результат необходимо поместить там же, оптимизированная версия даст ускорение в 124 раза, что близко к количеству вычислительных ядер GPU (128 ядер).

Задача умножения матриц хорошо подходит для параллельной реализации (хотя, как видно из табл. 1, для достижения производительности необходимо применить оптимизирующие преобразования). Для сравнения рассмотрим задачу сложения элементов числового массива, подробно описанную в [15]. Эта задача использует похожие операторы арифметических действий и работы с массивами, однако операторы в различных потоках исполняются в различном порядке и доступ к памяти осуществляется нерегулярно. В результате применение распараллеливающего преобразования даже понижает производительность программы, тогда как использование оптимизирующего преобразования из п. 3.3 повышает производительность всего в два раза по сравнению с последовательной программой. Это подтверждает тот факт, что разработка программ для GPU — сложная задача, требующая учета специфики задачи. Использование переписывающих правил и высокоуровневых моделей в данном случае позволяет применить стандартные преобразования, прежде чем концентрироваться на более тонких деталях оптимизации, специфических для конкретной задачи. В работе [15] описано, как применение переписывающих правил для реализации специфических преобразований позволяет получить ускорение в 20 раз по сравнению с последовательной программой; при этом специфические преобразования могут применяться только после использования универсальных преобразований для распараллеливания и оптимизации.

ЗАКЛЮЧЕНИЕ

В настоящей работе рассмотрены проблемы программирования высокопроизводительных вычислений и их преломление, в частности, для архитектуры видеографических ускорителей. Предложены формальные методы для разработки эффективных параллельных программ для GPU. Алгебро-динамические модели исполнения программ позволяют доказывать корректность преобразований и оценивать время исполнения программ. Переписывающие правила способствуют автоматизации распараллеливания и оптимизации программ. Использование высокоуровневых моделей программ позволяет сократить размер правил для преобразования, а также применить одни и те же правила для описания преобразований на разных языках. Экспериментальные данные подтвердили высокую эффективность преобразований: достигнуто ускорение более чем в 20 раз по сравнению с последовательной программой.

Дальнейшие исследования в данном направлении предполагают разработку дополнительных преобразований для распараллеливания и оптимизации кода для GPU и оценку их эффективности на различных примерах. Кроме того, возможно уточнение построенных моделей для учета дополнительных аппаратных и программных возможностей платформы CUDA; предполагается также поддержка других технологий программирования для графических ускорителей, таких как OpenCL.

СПИСОК ЛИТЕРАТУРЫ

1. Akhter S., Roberts J. Multi-core programming. Increasing performance through software multi-threading. — Hillsboro: Intel Press, 2006. — 336 p.
2. Adl-Tabatabai A.-R., Kozyrakis C., Saha B. Unlocking concurrency // Comput. Architect. — Dec. 2006 / Jan. 2007. — **4**, N 10. — P. 24–33.
3. Chrysanthakopoulos G., Singh S. An asynchronous messaging library for C#. — http://research.microsoft.com/~tharris/scool/papers/sing.pdf.
4. Дорошенко А.Е. Математические модели и методы организации высокопроизводительных параллельных вычислений. Алгебро-динамический подход. — Киев: Наук. думка, 2000. — 177 с.
5. General-purpose computation using graphics hardware. — http://www.gpgpu.org.

6. NVidia CUDA technology. — http://www.nvidia.com/cuda.
7. Letichevsky A.A., Kapitonova J.V., Konozenko S.V. Computations in APS // Theoret. Comput. Sci. — 1993. — **119**. — P. 145–171.
8. Андон Ф.И., Дорошенко А.Е., Цейтлин Г.Е., Яценко Е.А. Алгеброалгоритмические модели и методы параллельного программирования. — Киев: Академпериодика, 2007. — 631 с.
9. Doroshenko A., Shevchenko R. A rewriting framework for rule-based programming dynamic applications // Fund. Inform. — 2006. — **72**, N 1–3. — P. 95–108.
10. TermWare. — http://www.gradsoft.com.ua/products/termware_rus.html.
11. Летичевский А.А., Хоменко В.В. Переписывающая машина и оптимизация стратегий переписывания термов // Кибернетика и системный анализ. — 2002. — № 5. — С. 3–17.
12. Дорошенко А.Е., Жереб К.А., Яценко Е.А. Формализованное проектирование эффективных многопоточных программ // Пробл. программирования. — 2007. — № 1. — С. 17–30.
13. Дорошенко А.Е., Жереб К.А., Яценко Е.А. Об оценке сложности и координации вычислений в многопоточных программах // Там же. — 2007. — № 2. — С. 41–55.
14. Дорошенко А.Е., Жереб К.А. Алгебро-динамические модели для распараллеливания программ // Там же. — 2010. — № 1. — С. 39–55.
15. Дорошенко А.Е., Жереб К.А. Разработка высокопараллельных приложений для графических ускорителей с использованием переписывающих правил // Там же. — 2009. — № 3. — С. 3–18.
16. Lee S., Min S., and Eigenmann R. OpenMP to GPGPU: a compiler framework for automatic translation and optimization // Proc. of the 14th ACM SIGPLAN Symp. on Principles and Practice of Parallel Program. (PPoPP'9), Raleigh, NC (USA) Febr. 14–18, 2009. — P. 101–110.
17. OpenMP specification. — http://openmp.org/wp/.
18. A compiler framework for optimization of affine loop nests for gpgpus / M. Baskaran, U. Bondhugula, S. Krishnamoorthy, et al. // Proc. of the 22nd Ann. Intern. Conf. on Supercom. (ICS'08), Island of Kos (Greece), June 07–12, 2008. — New York: ACM, 2008. — P. 225–234.
19. Ma W. and Agrawal G. A compiler and runtime system for enabling data mining applications on gpus // Proc. of the 14th ACM SIGPLAN Symp. on Principles and Practice of Parallel Program (PPoPP'09), Raleigh, NC (USA), Febr. 14–18, 2009. — New York: ACM, 2009. — P. 287–288.
20. Allusse Y., Horain P., Agarwal A., and Saipriyadarshan C. GpuCV: an open source GPU-accelerated framework for image processing and computer vision // Proc. of the 16th ACM Intern.Conf. on Multimedia (MM'08), Vancouver, British Columbia (Canada), Oct. 26–31, 2008. — New York: ACM, 2008. — P. 1089–1092.
21. Glift: Generic, efficient, random-access GPU data structures / A.E. Lefohn, S. Sengupta, J. Kniss, et al. // ACM Trans. Graph. — 2006. — **25**, N 1. — P. 60–99.
22. Han T.D. and Abdelrahman T.S. hiCUDA: a high-level directive-based language for GPU programming // Proc. of 2nd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-2), Washington, D.C., March 08, 2009. — New York: ACM, 2009. — **383**. — P. 52–61.
23. Hou Q., Zhou K., Guo B. BSGP: bulk-synchronous GPU programming // ACM SIGGRAPH 2008 Papers, Los Angeles, Aug. 11–15, 2008. — New York: ACM, 2008. — P. 1–12.

Поступила 24.02.2011