СИСТЕМЫ АВТОМАТИЗИРОВАННОГО ПРОЕКТИРОВАНИЯ

УДК 681.31

А.Д. Иванников

МЕТОДЫ ДЕКОМПОЗИЦИИ ЗАДАЧИ ОТЛАДКИ ПРОЕКТА ЦИФРОВЫХ СИСТЕМ С ПОМОЩЬЮ МОДЕЛИРОВАНИЯ

На основе использования в качестве модели проекта цифровой системы семейства стационарных динамических систем формулируется задача отладки проекта методом моделирования. В связи со сложностью и большой размерностью задачи отладки цифровых систем в целом рассматриваются и обобщаются методы декомпозиции этой задачи, а именно: вертикальная и горизонтальная структурная декомпозиция, функциональная декомпозиция, декомпозиция по типам ошибок.

Ключевые слова: *модель цифровых систем, отладка методом моделирования, проектирование систем на кристалле, логическое моделирование, логико-временной анализ.*

Введение. В настоящее время проектирование сложных цифровых систем на кристалле возможно только с использованием средств автоматизации проектирования. Причем обычно сжатые сроки проектирования делают необходимым широкое использование уже спроектированных ранее блоков.

Спроектированная цифровая система задается в виде принципиальной электрической схемы *Сх* технических средств и текста программного или микропрограммного обеспечения *П*, т. е. в виде пары (*Сх*, *П*).

Основой отладки цифровых систем является отладочный эксперимент, в котором на входы объекта отладки (компьютерной модели цифровой системы) подается отладочный тест, определяется выходное воздействие и возможно какие-то внутренние переменные в зависимости от времени. Если выходное воздействие не соответствует требуемому, осуществляется локализация ошибки и ее исправление.

Сравнительно малое количество выводов и высокая сложность технических средств и программно-микропрограммного обеспечения реальных цифровых систем не дают возможность локализовать ошибку, исходя лишь из наблюдения выходных воздействий цифровой системы. С целью получения более полной информации о работе отлаживаемой системы разработчик анализирует изменения содержимого ячеек ЗУ и регистров блоков, а также логических сигналов на выходах блоков, из которых состоит цифровая система, если модель цифровой системы, используемая в качестве объекта отладки, это позволяет. Однако и при этом условии высокая сложность современных цифровых схем и микросистем, большое количество возможных источников ошибок затрудняют локализацию последних в связи с высокой размерностью задачи. Эффективным средством в этом случае является декомпозиция задачи отладки на ряд задач отладки меньшей размерности. *Целью данной работы* является формулировка принципов декомпозиции задачи отладки современных цифровых систем методом моделирования.

Семейство стационарных динамических систем как модель проекта цифровой системы. Исходными данными для проектирования цифровой системы является техническое задание, в котором определяются входы и выходы системы, возможные значения входных и выходных переменных, функции, выполняемые системой, зависимости выходных сигналов от входных, задаются временные ограничения на функционирование системы, протоколы обмена информацией с внешней средой. Обычно техническое задание содержит указанную информацию в виде описания, то есть в неформализованном виде. Вместе с тем техни-

ческое задание должно полностью определять цифровую систему с точки зрения ее внешнего поведения.

При проектировании техническое задание определяет прежде всего внешнее поведение цифровой системы, то есть временные диаграммы входных и выходных сигналов. Назовем эти временные диаграммы допустимыми взаимодействиями. Будем рассматривать взаимодействия цифровой системы с внешней средой (временные диаграммы входных и выходных сигналов системы) от начального момента времени (например, включения питания) до какого-то текущего момента времени. Множество взаимодействий цифровой системы с внешней средой назовем множеством допустимых взаимодействий, если каждое взаимодействие (каждая временная диаграмма) множества удовлетворяет требованиям технического задания.

Отличие допустимых взаимодействий друг от друга может быть обусловлено:

- различной длительностью взаимодействий;
- различной последовательностью функций, выполняемых цифровой системой в зависимости от внешних сигналов;
- различной последовательностью появления ряда входных или выходных сигналов, если эта последовательность безразлична, при выполнении одной и той же последовательности функций; наличием безразличных переключений входных сигналов, не оказывающих влияния на функционирование цифровой системы;
- разбросом временных параметров блоков цифровой системы, что обусловливает разброс моментов времени появления выходных сигналов, и разбросом временных параметров внешней среды, что обусловливает различное время реакции внешней среды на выходные сигналы цифровой системы.

Не рассматривая подробную математическую теорию, отметим, что множество допустимых взаимодействий характеризует с точки зрения внешнего поведения множество стационарных динамических систем [1].

Задача отладки проекта цифровой системы. Итак, техническое задание на проектирование цифровой системы неформальным образом задает семейство **D** стационарных динамических систем с конечными множествами входных, выходных и внутренних состояний и непрерывным временем.

Проект цифровой системы задается в виде (*Сх*, *П*), т.е. в виде принципиальной схемы соединения блоков системы и текста программного и/или микропрограммного обеспечения. Цифровая система, заданная таким образом, при правильном проектировании определяет непустое множество стационарных динамических систем **M**.

Задача отладки проекта цифровой системы заключается в следующем. Проверить

$$M \neq \varnothing, M \in B. \qquad (1)$$

Если соотношение (1) не выполняется, то внести в пару (*Сх*, *П*) такие исправления, чтобы приведенные соотношения выполнялись.

В связи с отмеченной выше сложностью и большой размерностью задачи целесообразно использовать декомпозицию задачи отладки, то есть обнаружение и локализацию ошибок проектирования каждого типа по отдельности.

Структурная вертикальная декомпозиция. Цифровая система может быть представлена как вертикальная иерархическая композиция из нескольких компонентов различных уровней. Таких уровней, как минимум, два: технические средства, определяющие архитектуру или микроархитектуру цифровой системы и являющиеся интерпретатором последовательности команд или микрокоманд; программное или микропрограммное обеспечение, которое в комплексе с техническими средствами определяет переработку (является интерпретатором) последовательностей входных сигналов в выходные.

Цифровая система в ряде случаев содержит большее количество уровней. Достаточно часто цифровая система строится из отлаженных ранее блоков, которые в свою очередь представляют из себя схемы соединения блоков более низкого уровня. Также в ряде цифровых систем можно выделить уровень технических средств, определяющих микроархитектуру системы (систему микрокоманд), уровень микропрограмм, определяющих архитектуру системы (систему команд, методы адресации, дисциплину обслуживания прерываний) и уровень программного обеспечения. Программное обеспечение может быть в ряде случаев представлено как имеющее два и более уровней.

Цифровая система в процессе работы выполняет некоторую последовательность функций из конечного алфавита **K,** свойственного конкретной системе. Каждую такую последовательность $f \in \mathbf{F}$ будем считать программной, выполняемой (интерпретируемой) цифровой системой. «Командами» таких программ являются отрезки входных взаимодействий, вызывающих выполнение той или иной операции из алфавита **K.**

Цифровую систему в целом необходимо рассматривать совместно с множеством **F** выполняемых последовательностей функций.

Цифровая система может быть представлена как

$$(\mathbf{L}_1, \mathbf{L}_2, \dots, \mathbf{L}_n, \mathbf{F}_n),$$

где L_1 – схема технических средств;
$\mathbf{L}_2, \dots, \mathbf{L}_n$ – программы или микропрограммы i-го уровня, представляющие собой определенные последовательности команд этого уровня;
n – количество уровней, имеющихся в системе;
$\mathbf{F}_n$ – множество программ, выполняемых цифровой системой в целом.

Кортеж $(\mathbf{L}_1, \dots, \mathbf{L}_i)$ задает правила интерпретации команд i-го уровня, то есть виртуальную машину этого уровня с системой команд $\mathbf{K}_i$.

Отладка цифровой системы в целом представляет собой выбор из множества $\boldsymbol{F}_n$ конечного множества конечных по времени отладочных тестов $TST_n \subset \mathbf{F}_n$, выполнения TST_n с помощью интерпретатора $(\mathbf{L}_1, \mathbf{L}_2, \dots, \mathbf{L}_n)$, определение правильности или ошибочности интерпретации, в случае наличия ошибки – ее идентификацию и исправление.

В соответствии с принципом структурной вертикальной декомпозиции задачу отладки цифровой системы $(\mathbf{L}_1, \mathbf{L}_2, \dots, \mathbf{L}_n, \mathbf{F}_n)$ можно заменить отладкой виртуальной машины первого уровня, т. е. отладкой интерпретатора $(\mathbf{L}_1)$ на наборе отладочных тестов TST_1; отладкой виртуальной машины второго уровня, т. е. отладкой интерпретатора $(\mathbf{L}_1,\mathbf{L}_2)$ на наборе отладочных тестов TST_2 и так далее, заканчивая отладкой машины n – го уровня (цифровой системы в целом), т. е. отладкой интерпретатора $(\mathbf{L}_1, \mathbf{L}_2, \dots, \mathbf{L}_n)$ на наборе отладочных тестов $TST_n \subset \mathbf{F}_n$. Такая декомпозиция существенно снижает размерность задачи отладки в связи с меньшим количеством возможных ошибок и их типов в каждом уровне $\mathbf{L}_i$ по сравнению с цифровой системой в целом.

При этом для каждого уровня необходимо:

а) иметь возможность выделить множество допустимых программ $\mathbf{F}_i$ для виртуальной машины i–го уровня;

б) построить набор отладочных тестов $TST_i \subset \mathbf{F}_i$;

в) знать множество допустимых реакций для всех возможных программ i – го уровня или, по крайней мере, для всех отладочных тестов TST_i;

г) иметь машинную модель интерпретатора i – го уровня (модель технических средств, модели цифровой системы на уровнях микроархитектуры, архитектуры, базовых макрокоманд, из которых формируется общий алгоритм работы системы) с возможностью слежения за внутренними переменными модели этого уровня.

В качестве такой модели всегда может быть использован интерпретатор первого уровня $(\mathbf{L}_1)$, т.е. модель технических средств в сочетании с программами $(\mathbf{L}_2, \dots, \mathbf{L}_i)$. Однако такие модели требуют больших затрат машинного времени в связи с моделированием внутренних переменных всех уровней. В этих случаях более экономично в смысле затрат времени инструментальной ЭВМ использовать специально созданные и отлаженные эмуляторы i–го уровня, моделирующие внутренние переменные только этого уровня. Например, для отладки программного обеспечения могут быть использованы функционально-логические модели системы на уровне соединения типовых блоков. Но во многих случаях для отладки программ информация о логических сигналах на выводах блоков не является необходимой. Тогда целесообразно использование эмуляторов цифровых систем на уровне архитектуры (системы команд), позволяющих следить только за изменениями содержимого программно-доступных регистров блоков и ячеек ЗУ [2].

Структурная горизонтальная декомпозиция. Во многих случаях определенный уровень $\mathbf{L}_i$ цифровой системы можно представить как некоторое множество связанных между собой блоков. Возможны случаи, когда такие блоки функционируют одновременно (процессорные блоки в мультипроцессорной системе; блоки каналов ввода-вывода и другие специализированные периферийные блоки, функционирующие параллельно с выполнением основной программы) или в определенной очередности (блоки ЗУ, ввода-вывода, подключенные к общей шине; процессорный блок и блок контроллера прямого доступа к памяти).

На микропрограммном уровне могут быть выделены микропрограммы, реализующие отдельные команды, микропрограмма обслуживания пульта, микропрограмма, реализующая дисциплину обслуживания прерываний. В программном обеспечении цифровых систем реального времени могут быть выделены фрагменты, запускаемые при поступлении сигнала прерывания с заданным приоритетом.

Разбиение уровня $\mathbf{L}_i$ на блоки обычно осуществляется таким образом, чтобы их интерфейс был наиболее простым и легко описываемым.

Итак, каждый иерархический уровень $\mathbf{L}_i$ цифровой системы может быть представлен как

$$\mathbf{L}_i = \begin{pmatrix} u_i^1 \\ u_i^2 \\ u_i^j \end{pmatrix},$$

где u_i^j – блок, выделенный в уровне $\mathbf{L}_i$; j – количество блоков, на которые разбит уровень $\mathbf{L}_i$.

В соответствии с принципом структурной горизонтальной декомпозиции отладка $\mathbf{L}_i$ может быть заменена автономной отладкой блоков $u_i^1;u_i^2;...;u_i^j$ и отладкой их взаимодействия. Так, в случае уровня технических средств возможна отдельная отладка блоков ОЗУ, ПЗУ, блока микропрограммного управления и отладка обмена информацией между блоками по общей шине. В случае микропрограммного обеспечения – отдельная отладка микропрограмм, реализующих каждую команду.

Функциональная декомпозиция. Цифровая система обычно реализует конечный набор функций, например, контроль температуры объекта, индикацию состояния объекта на табло, управление шаговыми двигателями и т.д. Виртуальные машины более низких уровней ($\mathbf{L}_1$, $\mathbf{L}_2$,…) или их блоки также реализуют конечный набор функций или команд. Так, блок микропрограммного управления может реализовать переход к микропрограмме с адресом на единицу большем текущего адреса, условный или безусловный переход к указанному адресу микрокоманды, переход к микроподпрограмме или возврат из нее. Какой-либо контроллер внешнего устройства, подключенный к общей шине системы, может осуществлять только три операции (цикла): чтение, запись, чтение-пауза-запись, что определяется стандартом на шину.

В соответствии с принципом функциональной декомпозиции при отладке цифровой системы или ее составляющих необходимо проводить последовательную отладку функций, выполняемых системой или ее частью, а также проверку правильности выполнения последовательности функций.

На принципе функциональной декомпозиции, в частности, основаны многие современные методы выбора отладочных тестов для цифровых систем [3]. Следует отметить, что результат применения функциональной декомпозиции может совпадать с результатом структурной горизонтальной декомпозиции, например, в случае отладки микропрограмм, реалиизующих отдельные команды процессорного блока.

Декомпозиция по типам ошибок. При отладке цифровых систем методом моделирования могут быть выделены группы отладочных экспериментов для выявления ошибок определенных типов. Так при проведении отладки могут использоваться машинные модели на уровне архитектуры системы и на уровне сети составляющих ее блоков, что в свою очередь подразделяется на модели для логической отладки и модели для верификации временных диаграмм. Более того, верификация временных диаграмм может проводиться с различной степенью подробности [4]. Модели каждой степени детализации ориентированы на выявление определенных типов ошибок. Разделение задачи проверки технических средств цифровой системы на верификацию временных диаграмм и логическую отладку является результатом применения принципа декомпозиции задачи отладки по типам ошибок.

Принцип декомпозиции по типам ошибок может быть использован совместно с другими принципами декомпозиции. Примером такого подхода может служить моделирование на определенном уровне для выявления и устранения ошибок проектирования какого-либо типа, но проводимого не для всего проекта сразу, а для его отдельных структурных частей.

Заключение. Рассмотренные методы декомпозиции отладки проектов цифровых систем на этапе проектирования могут модифицироваться и, как указывалось выше, использоваться в смешанном варианте. Примером этого является смешанное моделирование проектов цифровых систем, когда одни части проекта моделируются на более высоком уровне, скажем, на уровне просто логических сигналов, а другие, наиболее критические на более детальном уровне, например, с подробным анализом фронтов сигналов.

Библиографический список

1. *Иванников А.Д.* Моделирование микропроцессорных БИС на основе теории динамических систем // Сб. трудов «Автоматизация проектирования радиоэлектронной аппаратуры и средств вычислительной техники». Свердловск: Изд-во УПИ им. С.М.Кирова. – 1986. Вып. 6. – С. 39–41.

2. URL: http:// msyst.ru/index.php/activities/86-forth-cat/116-quark-rossijskaya-sapr-sistemnogo-urovnya (дата обращения: 03.03.2014).

3. *Иванников В.П., Камкин А.С., Косачев А.С., Кулямин В.В., Петренко А.К.* Использование контрактных спецификаций для представления требований и функционального тестирования моделей аппаратуры // Программирование. – 2007. – Т. 33. № 5. – С. 47-62.

4. *Стемпковский А.Л., Гаврилов С.В., Глебов А.Л.* Методы логического и логико-временного анализа цифровых КМОП СБИС. – М.: Наука, 2007. – 220 с.

УДК 004.272

Д.В. Лунин, С.В. Скворцов

ОРГАНИЗАЦИЯ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ НА ПЛАТФОРМЕ CUDA

Рассмотрены возможности организации параллельных вычислений на графических ускорителях. Предложен подход к повышению производительности программных приложений общего назначения за счет их распараллеливания на платформе CUDA.

Ключевые слова: *параллельные вычисления, графические ускорители, CUDA, потоки.*

Введение. Технологические ограничения не позволяют увеличивать быстродействие микропроцессоров бесконечно, а области их применения расширяются постоянно. Чтобы удовлетворить возрастающие запросы пользователей, разработчики вынуждены применять новые и модифицировать существующие архитектурные решения. Классическим подходом к увеличению производительности является использование различных параллельных архитектур [1-3].

Понятие параллельной архитектуры является достаточно широким, поскольку отражает и способ обработки данных, используемый в системе, и организацию памяти, и топологию связей между процессорными модулями, и способ исполнения системой арифметических операций и другие аспекты.

Основная сложность при эксплуатации систем параллельной обработки данных заключается в разработке прикладных программ, учитывающих как параллелизм реализуемых алгоритмов, так и потенциальные возможности системы.

Цель работы – исследовать возможности применения технологии CUDA для повышения производительности общедоступных компьютерных систем за счет решения задач параллельной обработки данных в многоядерной среде графических процессоров фирмы nVidia.

Постановка задачи. Параллельное программмирование становится все более актуальным при разработке программ для стандартных, широкодоступных компьютеров, а не только для специализированных высокопроизводительных систем. Этому способствует бурное развитие многоядерных процессоров, которые сейчас устанавливаются в большинстве настольных и мобильных компьютеров.

Создание многоядерных процессоров обусловлено двумя обстоятельствами, направленными на повышение их быстродействия. Во-первых, рост тактовой частоты процессора ограничен технологией изготовления. Во-вторых, с ростом тактовой частоты сильно увеличивается тепловыделение процессора. Выходом из такой ситуации стало создание многоядерных процессоров. Многоядерность – это размещение на одном кристалле нескольких ядер, т.е. как бы два, четыре и более процессоров в одном. Но на практике *n*-ядерные процессоры не производят вычисления в *n* раз быстрее одноядерных: хотя прирост быстродействия и оказывается значительным, но при этом он во многом зависит от типа приложения [4].

В настоящее время есть еще одно направление параллельного программирования, которое интенсивно развивается. Речь идет о программировании общецелевых задач для графических ускорителей. Рыночные требования привели к бурному развитию графических ускорителей, в результате чего их вычислительная мощность на данный момент значительно превышает возможности обычных процессоров. Поэтому возник интерес к использованию графических ускорителей для решения задач, не связанных напрямую с обработкой графики.

Интерес к использованию графических ускорителей в качестве средств высокопроизводительных вычислений поддерживается усилиями ведущих разработчиков аппаратуры. Так, компания nVidia предоставляет платформу CUDA (Compute Unified Device Architecture) для вычислений на графическом ускорителе [5]. Аналогично компания AMD выступила с инициативой Stream [6].

Такие платформы облегчают реализацию различных задач на графических ускорителях, поскольку поддерживают модель программи-

рования, которая более приспособлена к разработке произвольных вычислений, по сравнению со средствами программирования графики. Тем не менее, разработка приложений для графических ускорителей остается достаточно сложной задачей. Разработчик должен быть знаком с устройством графического ускорителя, он должен понимать особенности работы его компонент и принципы организации вычислений, включая взаимодействие с центральным процесссором.

Таким образом, задача организации вычислений средствами графических ускорителей является актуальной и новой по сравнению как с последовательным программированием, так и с многопоточным параллельным программированием [7-9]. Многие аспекты этой задачи требуют достаточно низкоуровневой реализации и оптимизации кода, подобно методам и алгоритмам, изложенным в работах [10-12].

Архитектура графических процессоров nVidia. 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.

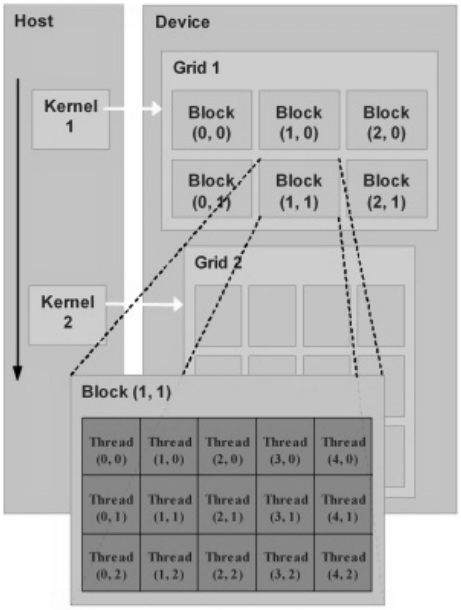
Графический ускоритель (device – по терминологии CUDA) содержит несколько мультипроцессоров (grid), а также общую для них разделяемую графическую память (см. рисунок). Каждый мультипроцессор содержит несколько вычислительных ядер (скалярных процессоров), а также один управляющий блок (block), поддерживающий многопоточное исполнение. Например, средняя по характеристикам видеоплата nVidia Quadro FX1800 включает 8 мультипроцессоров, каждый из которых содержит 512 блоков. Для более мощных графических ускорителей количество мультипроцессоров достигает 128.

Таким образом, количество вычислительных ядер (а значит, и степень возможного параллелизма) оказывается существенно выше, чем у общецелевых многоядерных процессоров.

Параллельные вычисления средствами графического ускорителя выполняются под управлением *центрального процессора* (ЦП, host). Для обращения к ускорителю в программу для ЦП включаются специальные *подпрограммы* (kernel), каждая из которых может состоять из большого количества *потоков* (thread). Особенности аппаратного обеспечения, а именно большое количество вычислительных ядер графического ускорителя, позволяют использовать очень мелкозернистый параллелизм, вплоть до выделения отдельного потока для каждого элемента данных. Все инструкции программы выполняются по принципу SIMD, когда одна инструкция применяется ко всем потокам в блоке.

Программирование в CUDA предполагает группирование потоков. Они объединяются в *блоки потоков* (thread block) - одномерные или двумерные сетки потоков, взаимодействующих между собой при помощи разделяемой памяти и точек синхронизации. Программа (kernel) исполняется над *сеткой блоков потоков* (grid), как это показано на рисунке.

Важно отметить, что одновременно исполняется только одна сетка потоков, т.е. если несколько подпрограмм (kernel) претендуют на ресурсы ускорителя, то они выполняются последовательно и без прерываний. Каждый блок может быть одно-, двух- или трехмерным по форме и может состоять из 512 потоков (для существующих аппаратных средств ускорителей).



Обобщенная архитектура графического процессора nVidia

Разделяемая потоками графическая память имеет объем 16 килобайт на каждый мультипроцессор. Она позволяет обмениваться информацией между потоками одного блока, которые всегда выполняются одним и тем же мультипроцессором. При этом важно отметить, что потоки из разных блоков обмениваться данными не могут.

Кроме того, мультипроцессоры могут обращаться и к системной видеопамяти, но с большими задержками и худшей пропускной способ-

ностью. Для ускорения доступа и снижения частоты обращения к видеопамяти каждый мультипроцессор имеет 8 килобайт кэша на константы и текстурные данные. Системная видеопамять и кэш каждого мультипроцессора на рисунке не показаны.

Методика применения CUDA. Возможными вариантами применения технологии CUDA являются как создание новых параллельных приложений для решения прикладных задач, так и модификация существующего последовательного программного обеспечения.

Первым шагом при переносе существующего приложения на CUDA является определение участков кода, снижающих скорость работы приложения в целом. Если среди таких участков есть подходящие для быстрого параллельного исполнения, то выполнение этих функций переносится на графический ускоритель с помощью расширений CUDA для языка Си. Заметим, что для решения такой задачи могут использоваться методы и алгоритмы анализа зависимостей по данным и управлению, позволяющие выявить скрытый параллелизм в программах.

При использовании технологии CUDA прикладная программа формируется с помощью поставляемого nVidia компилятора, который генерирует код и для ЦП, и для графического ускорителя. При работе программы центральный процессор выполняет свои процедуры, а графический ускоритель выполняет CUDA код с параллельными вычислениями. Эта часть программы, предназначенная для графического ускорителя, по терминологии CUDA также называется ядром (kernel). В ядре определяются операции, которые будут исполнены над данными. Необходимо различать программное ядро – подпрограмму (kernel) и вычислительное ядро – часть мультипроцессора.

Таким образом, программа для графического ускорителя, написанная на платформе CUDA, разделяется на две части: код для исполнения на графическом устройстве и код для исполнения на центральном процессоре. Код для исполнения на графическом устройстве (kernel) может быть написан как на специальном низкоуровневом языке (PTX), так и на расширении языка Си (Си для CUDA). В последнем случае ядро имеет вид функции языка Си, описывающей поведение одного потока.

При создании приложении программист имеет возможность задавать количество блоков и потоков. Оптимальным является использование от 64 до 512 потоков в блоке. Группировка блоков в сетки позволяет применить ядро к большему числу потоков за один вызов. Это помогает и при масштабировании. Если у графического процессора недостаточно ресурсов, он будет выполнять блоки последовательно. В обратном случае блоки могут выполняться параллельно, что важно для оптимального распределения работы на видеоплатах разного уровня.

Платформа CUDA предлагает два разных способа вызова ядер. Более простой способ Си для CUDA является расширением языка Си. Вызов ядра при этом похож на вызов функции Си, но с передачей дополнительных параметров. Эти параметры определяют число потоков в блоке (размер блока) и количество одновременно исполняемых блоков в сетке (grid). Каждый поток работает со своими данными, которые определяются номером потока в блоке и номером блока в сетке.

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. Применение driver API может привести к появлению ошибок, так как требует ручного выполнения многих действий, которые автоматизированы в Си для CUDA. Пример таких действий – передача параметров ядра. Однако driver API дает разработчику больший контроль над выполнением программы, но требует более высокой квалификации программиста.

Разработка программ на платформе CUDA. В качестве практического примера использования описанного подхода рассмотрим построение программы умножения двух матриц на графическом ускорителе.

Пусть исходные матрицы A и B являются квадратными размером $N \times N$. Результирующая матрица C будет иметь такой же размер, а ее элементы c_{ij} вычисляются по формуле:

$$c_{ij} = \sum_{k=1}^{N} a_{ik} b_{kj} \quad (i = 1,2,...,n; j = 1,2,...,n),$$

где a_{ij} и b_{ij} – элементы матриц A и B соответственно.

Последовательная реализация умножения матриц на ЦП в соответствии с приведенной формулой имеет следующий вид:

```
for (int i = 0; i < n; i++)
 {
   for (int j = 0; j < n; j++)
    {
      c[i][j] = 0;
      for (int k = 0; k < n; k++)
        c[i][j] += a[i][k] * b[k][j];
    }
 }
```

Для организации параллельного умножения матриц A и B представим матрицу C в виде совокупности квадратных подматриц размером BLOCK_SIZE×BLOCK_SIZE, чтобы вычисление всех элементов каждой подматрицы производилось бы последовательно в одном потоке (thread). Все такие потоки, сформированные в среде CUDA, выполняются параллельно аппаратными средствами графического ускорителя (device).

Реализация вычислений в одном потоке представлена в виде функции matrMult.

```
__global__ void matrMult(float*a, float*b,
                         int n, float*c)
{
  int bx = blockIdx.x; // индексация блоков
  int by = blockIdx.y;
  int tx = threadIdx.x;// индексация потоков
  int ty = threadIdx.y;
  float sum = 0.0f;
  int ia = n*BLOCK_SIZE*by + n*ty;
  int ib = BLOCK_SIZE*bx + tx;
// вычисление элементов одного блока матрицы
  for (int k = 0; k < n; k++)
    sum += a[ia + k] * b[ib + k*n];
// запись блока матрицы в оперативную память
  int ic = n*BLOCK_SIZE*by + BLOCK_SIZE*bx;
  c[ic + n*ty + tx] = sum;
}
```

В CUDA существует несколько особых переменных, которые существуют внутри каждого вычислительного ядра. Эти переменные позволяют отличать один поток от другого:

uint3 blockIdx - координаты текущего блока внутри сетки (grid);

dim3 blockDim - размерность блока графического ускорителя (block) при запуске ядра;

uint3 threadIdx - координаты текущего потока внутри блока.

Тип dim3 - это специальный тип, основанный на стандартном типе uint3, имеющем нормальный конструктор. Этот конструктор позволяет задавать не все компоненты вектора. Недостающие компоненты инициализируются единицами. Тип dim3 используется для задания параметров запуска ядра в основном методе main.

Чтобы функция matrMult выполнялась на графическом ускорителе, используется квалификатор __global__ , который и говорит о том, что эта функция должна исполняться на графическом ускорителе, а не на ЦП.

Обращение к функции matrMult в основном методе main несколько отличается от стандартного и имеет следующий вид:

```
matrMult <<<blocks,threads>>> (av,bv,N,cv) ,
```

где параметры в угловых скобках указывают среде CUDA способ запуска ядра.

Первый параметр dim3 blocks задает размер сетки (grid) для запуска потоков, т.е. количество параллельных блоков, в которых ускоритель должен исполнять ядро. Сетка может быть одно- и двухмерной, причем максимальное значение по каждому измерению составляет (65536, 65536, 1), а максимальное число блоков в сетке не может превышать 4294967296.

Второй параметр dim3 threads определяет размер блока при запуске, т.е. количество потоков (threads) в каждом блоке. Блок может одно-, двух- или трехмерным, причем максимальное значение по каждому измерению равно (512, 512, 64), а максимальное число потоков в блоке не может превышать 512. Следует заметить, что значения этих параметров влияют на скорость выполнения программы и должны выбираться с учетом особенностей решаемой задачи и модели графического ускорителя.

Для выполнения функции matrMult графическим ускорителем предварительно требуется передать необходимые данные в системную видеопамять, которая имеет линейную адресацию. Поэтому все матрицы A, B и C представляются построчно в виде линейных последовательностей элементов – векторов av, bv, cv размером

```
numBytes = N*N.
```

Самым простым способом выделения и освобождения системной видеопамяти является использование функций cudaMalloc и cudaFree.

Первая из этих функций выделяет память на графическом процессоре, например:

```
cudaMalloc ((void**)&av, numBytes) ,
```

а вторая – освобождает ее:

```
cudaFree(av).
```

Для обмена данными между графическим ускорителем и ЦП используется функция cudaMemCpy. Обращение к этой функции для передачи исходных данных на ускоритель будет иметь вид:

```
cudaMemcpy(av,a,N*N*sizeof(int),
           cudaMemcpyHostToDevice).
```

Возврат полученных результатов на ЦП осуществляется следующим образом:

```
cudaMemcpy(cv,c,N*N*sizeof(int),
           cudaMemcpyDeviceToHost).
```

Далее приведен полный текст метода main.

```
int main(int argc, char * argv[])
{
   int N;
   setlocale(LC_ALL,"russian");
   cout <<"Введите размер матриц"; cin >> N;
   int numBytes = N*N*sizeof(float);
   float * a = new float [N*N];
   float * b = new float [N*N];
   float * c = new float [N*N];
// заполнение матриц исходными данными
```

```
    for (int i = 0; i < N; i++)
      for (int j = 0; j < N; j++)
        {
          int k = N*i + j;
          a[k] = i; b[k] = i*3;
        }
// инициализация векторов с данными
    float * av = NULL;
    float * bv = NULL;
    float * cv = NULL;
// выделение системной видеопамяти
    cudaMalloc ((void**)&av, numBytes);
    cudaMalloc ((void**)&bv, numBytes);
    cudaMalloc ((void**)&cv, numBytes);
// определение параметров запуска ядра
    dim3 threads (BLOCK_SIZE, BLOCK_SIZE);
    dim3 blocks (N/threads.x, N/threads.y);
// передача данных на ускоритель
    cudaMemcpy (av, a, numBytes,
                cudaMemcpyHostToDevice);
    cudaMemcpy (bv, b, numBytes,
                cudaMemcpyHostToDevice);
// параллельное вычисление подматриц
    matrMult<<<blocks,threads>>>(av,bv,N,cv);
// возврат результатов на ЦП
    cudaMemcpy (c, cv, numBytes,
                cudaMemcpyDeviceToHost);
// освобождение видеопамяти
    cudaFree(av); cudaFree(bv); cudaFree(cv);
// освобождение памяти ЦП и завершение работы
    delete a; delete b; delete c;
    return 0;
}
```

Сравнение производительности. Для оценки эффекта от применения распараллеливающих преобразований на платформе CUDA были проведены измерения скорости работы программ умножения матриц, представленных в двух версиях: последовательная реализация на ЦП; многопоточная реализация на базе графического ускорителя. При этом вторая версия программы запускалась на выполнение с несколькими вариантами параметров ядра для значений `BLOCK_SIZE`, равных 16, 32, 64 и 128.

При проведении экспериментов использовались следующие аппаратные средства: ЦП Intel Core-i7; графический ускоритель nVidia Quadro FX1800 с видеопамятью 1 ГБ. Для каждого размера матрицы выполнено по 10 опытов с разными исходными данными. Средние значения времени работы программ приведены в таблице.

Полученные результаты показывают, что использование графического ускорителя позволяет значительно увеличить скорость работы прикладных программ. В зависимости от размера умножаемых матриц время работы многопоточного приложения на графическом ускорителе сокращается от 40 до 300 раз по сравнению с последовательной реализацией на ЦП, причем с увеличением величины N выигрыш возрастает. Из таблицы также видно, что скорость работы многопоточной программы зависит от параметра `BLOCK_SIZE`, значение которого выбирается экспериментально.

Среднее время умножения квадратных матриц (мс)

Программа умножения матриц	Параметр `BLOCK_SIZE`	Размер матриц N		
		256	512	1024
Последовательная для ЦП	не применяется	66	588	6621
Многопоточная для графического ускорителя	16	21	167	1318
	32	2.35	6.32	10.73
	64	0.53	1.33	4.26
	128	0.86	1.52	4.69

Заключение. В статье исследованы возможности организации параллельных вычислений общего назначения на графическом ускорителе. Определены основные особенности параллельных многопоточных приложений для графических ускорителей, отличающие их от последовательных однопоточных программ, выполняемых на ЦП. Предложена методика разработки параллельных программ на платформе CUDA, обеспечивающих существенное повышение производительности прикладных приложений посредством организации их многопоточного исполнения на графическом ускорителе.

Дальнейшие исследования в данном направлении предполагают разработку дополнительных алгоритмов распараллеливания и оптимизации кода для графических ускорителей, а также оценку их эффективности при различных режимах запуска ядра. В качестве прикладных задач, допускающих параллельные вычисления на графическом ускорителе, могут быть выбраны задачи организации самодиагностики и тестирования многопроцессорных систем [13-17].

Библиографический список

1. *Корячко В.П., Скворцов С.В., Телков И.А.* Модель планирования параллельных процессов в суперскалярных процессорах // Информационные технологии. 1997. № 1. С. 8-12.

2. *Михеева Л.Б., Скворцов С.В.* Синтез параллельного кода для RISC-процессоров с оптимизацией загрузки регистровой памяти // Информационные технологии. 2002. № 7. С. 2-9.

3. *Гершанов В.И., Скворцов С.В., Телков И.А.* Методы повышения отказоустойчивости вычислительных систем, основанных на принципе ассоциативной селекции потоков данных // Вопросы радиоэлектроники. Серия Электронная вычислительная техника. 1992. № 7. С. 50-58.

4. *Козлов М.А., Скворцов С.В.* Алгоритмы параллельной сортировки данных и их реализация на языке Clojure // Вестник Рязанского государственного радиотехнического университета. 2013. № 4-1 (46). С. 92-96.

5. CUDA Parallel Computing Platform. [Электронный ресурс]. URL: http://www.nvidia.com/cuda (дата обращения 28.03.14).

6. AMD Quick Stream Technology. [Электронный ресурс]. URL:http://amd-quick-stream.software.informer.com (дата обращения 15.04.14).

7. *Эндрюс Г.* Основы многопоточного, параллельного и распределенного программирования. М.: ИД "Вильямс", 2003. 512 с.

8. *Камерон Х., Трейси Х.* Параллельное и распределенное программирование с использованием C++. М.: Вильямс, 2004. 672 с.

9. *Богачев К.Ю.* Основы параллельного программирования. М.: БИНОМ. Лаборатория знаний, 2010. 342 с.

10. *Скворцов С.В.* Оптимизация кода для суперскалярных процессоров с использованием дизъюнктивных графов // Программирование. 1996. № 2. С. 41-52.

11. *Скворцов С.В.* Целочисленные модели оптимизации кода по критерию времени // Информационные технологии. 1997. № 10. С. 2-7.

12. *Першин А.С., Скворцов С.В.* Распределение регистровой памяти в системах параллельной обработки данных // Системы управления и информационные технологии. 2007. № 1 (27). С. 65-70.

13. *Скворцов С.В.* Применение симметричной диагностической модели при организации активной отказоустойчивости многопроцессорных систем // Вестник Рязанского государственного радиотехнического университета. 1998. № 4. С. 57–64.

14. *Скворцов Н.В., Скворцов С.В., Хрюкин В.И.* Дешифрация диагностического синдрома многопроцессорной системы в реальном времени // Системы управления и информационные технологии. 2010. № 1 (39). С. 49–53.

15. *Скворцов Н.В., Скворцов С.В., Хрюкин В.И.* Синтез диагностических графов для многопроцессорных систем с активной отказоустойчивостью // Вестник Рязанского государственного радиотехнического университета. 2012. № 39–2. С. 83-89.

16. *Скворцов Н.В., Скворцов С.В.* Автоматизация проектирования процессов самодиагностики для многопроцессорных систем с активной отказоустойчивостью // Вестник Рязанского государственного радиотехнического университета. 2013. № 4–2 (46). С. 71–77.

17. *Рудаков В.Е., Скворцов С.В.* Построение базового множества независимых путей потокового графа для тестирования программных модулей // Системы управления и информационные технологии. 2012. Т. 50. № 4. С. 67-70.