ИПМ им.М.В.Келдыша РАН

Абрау-2016 • Труды конференции

А.Н. Андрианов, Т.П. Баранова,
А.Б. Бугеря, Е.Н. Гладкова,
К.Н. Ефимкин, П.И. Колударов

Методы трансляции непроцедурного языка НОРМА для графических процессоров

Рекомендуемая форма библиографической ссылки

Андрианов А.Н., Баранова Т.П., Бугеря А.Б., Гладкова Е.Н., Ефимкин К.Н., Колударов П.И. Методы трансляции непроцедурного языка НОРМА для графических процессоров // Научный сервис в сети Интернет: труды XVIII Всероссийской научной конференции (19-24 сентября 2016 г., г. Новороссийск). — М.: ИПМ им. М.В.Келдыша, 2016. — С. 23-35. — doi:10.20948/abrau-2016-35

Размещена также презентация к докладу

Методы трансляции непроцедурного языка НОРМА для графических процессоров

А.Н. Андрианов[1], Т.П. Баранова[1], А.Б. Бугеря[1], Е.Н. Гладкова[1], К.Н. Ефимкин[1], П.И. Колударов[2]

[1] Институт прикладной математики им. М.В. Келдыша РАН
[2] ООО «ЕМЕ»

Аннотация. Рассматриваются методы автоматического построения программ для графических процессоров по непроцедурным спецификациям на примере решения задач трансляции программ на языке НОРМА для графических процессоров с использованием технологии NVIDIA CUDA. Оценивается эффективность созданного на основе рассмотренных методов компилятора. Приводятся результаты применения компилятора для трёх различных задач.

Ключевые слова: параллельное программирование, графические процессоры, автоматизация программирования, непроцедурные спецификации, язык НОРМА

1. Введение

В современном научном сообществе задача разработки эффективных параллельных программ имеет важное стратегическое значение. Наверное, уже не осталось ни одной области науки или отрасли промышленности, так или иначе не связанной со сферой высокопроизводительных вычислений. Несмотря на то, что параллельное программирование появилось уже достаточно давно, успешно развивается и проводится масса исследований на эту тему, вопрос «как создать эффективную параллельную программу для решения такой-то задачи» до сих пор крайне актуален для программистов и математиков.

Достаточно быстрое развитие новых аппаратных возможностей для поддержки параллельных вычислений, наблюдаемое в последнее время, еще больше усложняет проблему. Например, появление массово доступных многоядерных процессоров поставило вопрос об эффективном программировании для них. Практически одновременно появились массово доступные графические ускорители (графические процессоры), а затем – ускорители Xeon Phi. И для каждого типа ускорителей опять возникает вопрос об эффективном программировании для них.

Конечно, нельзя не отметить, что в составе средств разработки программ для графических процессоров появляются различные новые библиотеки, такие как cuBLAS, cuSPARSE [1] и прочие, предоставляющие прикладному

программисту эффективные элементарные «кирпичики» для конструирования своей программы, но при этом всё равно оставляющие нерешёнными множество задач, такие как управление памятью, распределение вычислений между несколькими вычислителями и т.п. Надежды на автоматическое распараллеливание уже написанных последовательных программ на различные виды параллельных архитектур пока совершенно не оправдываются, несмотря на то, что фирмы-производители таких аппаратных решений давно активно поддерживают данное направление исследований. Из уже реализованных подходов можно отметить те, которые базируются на вполне разумном симбиозе распараллеливающего компилятора и подсказок со стороны программиста, выполненных в виде специальных программных директив, например OpenACC [2].

Работы по созданию и продвижению новых средств и языков программирования для графических процессоров, гибридных решений и различных нетрадиционных вычислительных архитектур (например, FPGA), ведутся весьма активно [3, 4], однако проблема простой разработки параллельных программ и утилизации новых возможностей вычислительной техники так и остается в настоящее время нерешенной.

2. Декларативный подход. Язык НОРМА

Одним из возможных подходов к решению задачи автоматизации параллельного программирования вычислительных задач, и, в частности, задачи автоматического построения эффективной программы для графических процессоров, является подход с использованием декларативных (непроцедурных) языков. При использовании этого подхода прикладной специалист программирует решение вычислительной задачи на непроцедурном языке (понятия, связанные с архитектурой параллельного компьютера, моделями параллелизма и прочие детали целевой системы при этом не используются), а затем компилятор автоматически строит параллельную программу (уже учитывая архитектуру целевого параллельного компьютера, модели параллелизма и прочие заданные параметры компиляции). С учетом отмеченных выше проблем привлекательность этого подхода в настоящее время только усиливается, и интерес к идеям непроцедурного декларативного программирования и реализации этих идей в языках программирования неуклонно растет.

Идеи декларативного программирования были сформулированы еще в прошлом веке, теоретические исследования этого подхода для класса вычислительных задач проведены в пионерских работах И.Б. Задыхайло еще в 1963 году [5]. Непроцедурный язык НОРМА и система программирования НОРМА [6, 7] разработаны в ИПМ им. М.В. Келдыша РАН также достаточно давно и предназначены для автоматизации решения вычислительных сеточных задач на параллельных компьютерах. Расчетные формулы записываются на языке НОРМА в математическом, привычном для прикладного специалиста

виде. Язык НОРМА позволяет описывать решение широкого класса задач математической физики. Программа на языке НОРМА имеет очень высокий уровень абстракции и отражает метод решения, а не его реализацию при конкретных условиях. Такое описание не ориентировано на конкретную архитектуру компьютера, поэтому оно предоставляет большие возможности для выявления естественного параллелизма и организации вычислений.

В систему программирования НОРМА [7] входит компилятор программ на языке НОРМА, позволяющий получать исполняемую программу на заданном процедурном языке программирования для определённой модели параллелизма. Далее будут рассмотрены методы и решения, реализованные в этом компиляторе при генерации исполняемой программы с использованием технологии NVIDIA CUDA для графических процессоров.

3. Методы построения CUDA программы по декларативным описаниям

При трансляции с языка НОРМА решается задача синтеза выходной параллельной программы. В результате анализа зависимостей по данным между операторами программы на языке НОРМА, в случае разрешимости этих зависимостей, представляется возможным построить так называемую «параллельную ярусную схему» выполнения программы. На каждом ярусе данной ярусной схемы располагаются операторы программы, которые не имеют зависимостей друг от друга и могут выполняться независимо и, соответственно, параллельно. В то же время каждый из этих операторов имеет зависимость от одного или более операторов, располагающихся на предыдущем ярусе ярусной схемы. Таким образом, группу операторов, располагающихся на одном уровне ярусной схемы, можно выполнять параллельно в результирующей программе, но только после того, как будут полностью выполнены все операторы предыдущего уровня ярусной схемы. Параллельная ярусная схема программы является, фактически, представлением идеального (естественного) параллелизма, определяемого соотношениями и зависимостями между расчетными переменными программы.

3.1. Компоновка вычислительных ядер

В результате ряда исследований по трансформации описанной параллельной ярусной схемы программы на языке НОРМА в программу с использованием технологии NVIDIA CUDA для графических процессоров предлагается использовать следующий подход для автоматического построения исполняемой программы для графических процессоров с использованием технологии NVIDIA CUDA.

Исполняемая программа стартует и завершается на центральном процессоре, на нём же выполняется ввод-вывод данных и итерационные циклы. Очевидно также, что вся логика по управлению вычислениями, выделением

памяти на графическом процессоре, организация обменов данными между памятью графического и центрального процессора, тоже создаётся в программном коде, выполняющемся на центральном процессоре. Вызовы других процедур и функций должны выполняться из программы на центральном процессоре, если это «обычные» пользовательские или библиотечные функции, имеющие реализации для центрального процессора, или же из вычислительного ядра, выполняющегося на графическом процессоре, если это пользовательские или библиотечные функции, имеющие реализации для графического процессора. Сами вычислительные операторы выполняются, по возможности, на графическом процессоре. Для этого они группируются в определённые наборы, каждый из которых может выполняться в пределах одного ядра программы с использованием технологии NVIDIA CUDA. Программа, выполняющаяся на центральном процессоре, осуществляет контроль над общим выполнением программы, запускает полученные ядра в определённом порядке, ведёт итерационные циклы и проверяет условие выхода из итерации. На рис.1 приведён пример общей схемы выполнения исполняемой программы.

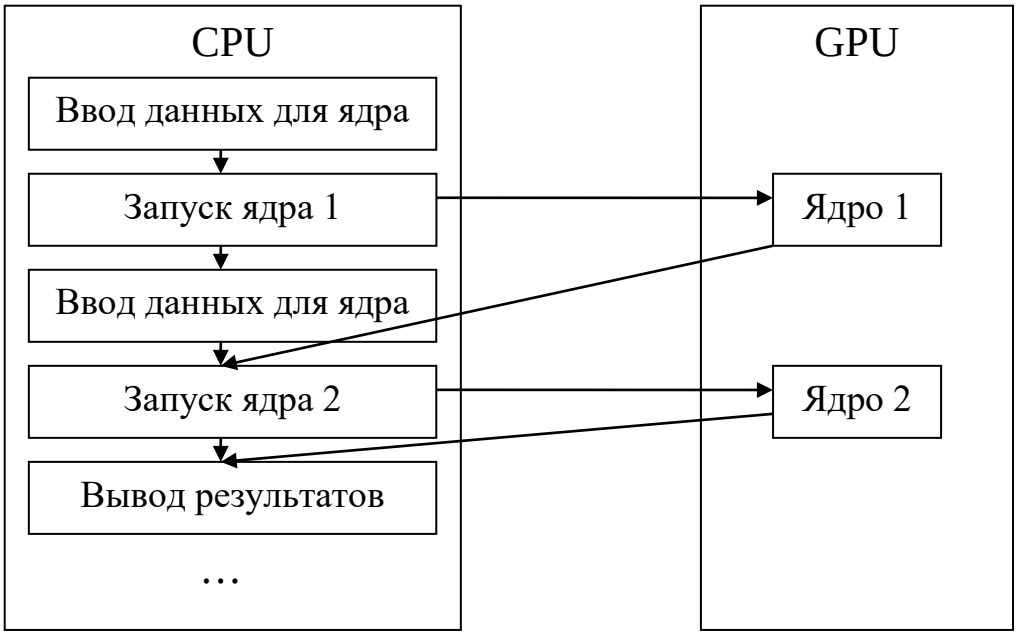

Рис.1. Пример общей схемы выполнения исполняемой программы.

Один из ключевых моментов в этой схеме организации исполняемой программы – это задача группировки вычислительных операторов в ядра, которые будут выполняться на графическом процессоре. С одной стороны, чем больше операторов будет содержать такое ядро, т.е. чем крупнее будут исполняемые ядра в программе, тем эффективнее она будет работать, т.к. будет меньше запусков ядер, меньше передачи параметров, в том числе и через память центрального процессора, и т.п. Но, с другой стороны, все операторы, сгруппированные в одно ядро, должны работать с одним и тем же

распределением рабочих областей на конфигурацию блоков и нитей, определяемую при запуске ядра. И может оказаться, что поиск такого распределения для большой группы операторов, объединённых в одно ядро, может дать худший результат, чем разбиение этой группы операторов на более мелкие отдельные ядра и поиск распределения для каждой группы в отдельности. Кроме того, при поиске таких группировок необходимо соблюдать ряд обязательных условий, накладываемых ярусной схемой и сущностью группируемых операторов. Например, мы можем менять порядок выполнения операторов, но только в пределах одного яруса. А операторы ввода-вывода всегда должны выполняться на центральном процессоре.

В результате обработки ярусной схемы программы на языке НОРМА получается линейный набор выполняемых элементов для центрального процессора и связанный с ним набор вычислительных ядер, включающих в себя один или более операторов исходной программы на языке НОРМА. Для каждого такого вычислительного ядра также определяются различные характеристики (распределение области на конфигурацию блоков и нитей, использованные переменные, вычисленные переменные и т.п.).

3.2. Спецификатор PLAIN

Вызовы других разделов и функций должны производиться кодом программы, выполняемым на центральном процессоре, если эти вызываемые разделы или функции не имеют реализаций для графического процессора. И, очевидно, что все стандартные и пользовательские функции, имеющие реализации для графического процессора, могут вызываться из программного кода, выполняемого на графическом процессоре. Компилятор программ на языке НОРМА имеет встроенный список стандартных функций, и при их вызове автоматически понимает, что они могут быть вызваны как на центральном процессоре, так и на графическом. Но как быть с пользовательскими функциями, имеющими реализацию на графическом процессоре? Для решения этой проблемы в язык НОРМА было введено новое понятие – спецификатор раздела или функции.

Спецификатор раздела или функции может быть указан как в реализации раздела или функции, так и в декларации EXTERNAL в вызывающих разделах и функциях. При этом раздел должен иметь один и тот же спецификатор во всех случаях. На данный момент поддерживаются 2 спецификатора: PLAIN и CUDA.

Спецификатор CUDA будет описан ниже. Спецификатор PLAIN предназначен для того, чтобы сообщить компилятору, что данная вызываемая функция или раздел является обычной «плоской» функцией или разделом: не содержит внутри себя какие-либо распараллеливающие конструкции и имеет реализацию для выбранного выходного языка – в данном случае реализацию для графического процессора с использованием технологии NVIDIA CUDA.

При вызове функций и разделов, объявленных внешними со спецификатором PLAIN, компилятор будет действовать так же, как и при вызове стандартных функций: генерировать код вызова там же, где происходит генерация всего оператора, содержащего этот вызов. Если же производится вызов функции или раздела, объявленного внешним без спецификатора PLAIN, то оператор, содержащий этот вызов, будет выполняться на центральном процессоре.

3.3. Эффективное управление памятью

При создании программ для графических процессоров, в силу того, что объём памяти графического процессора обычно гораздо меньше, чем объём памяти центрального процессора, всегда встаёт вопрос об эффективном использовании памяти графического процессора. Это становится особенно актуально при решении задачи автоматической генерации программ для графических процессоров, так как при автоматической генерации программ, как правило, возникают дополнительные вспомогательные переменные, необходимые для осуществления заданных операций, которые требуют выделения дополнительной памяти. Но несомненное достоинство автоматической генерации программ – это обладание полной информацией о том, где и как в программе используются переменные, а, следовательно, это возможность полностью контролировать процесс выделения и освобождения памяти и полностью исключить возможность «утечки» памяти в результате её не освобождения после использования.

Очевидно, что наиболее эффективным с точки зрения минимизации объёма требуемой памяти графического процессора был бы подход, при котором память запрашивалась бы непосредственно перед вызовом ядра, где она бы использовалась, и освобождалась бы сразу после такого использования. Но в целом ряде случаев буквальная реализация такого подхода приводит к катастрофическому провалу по производительности получающихся программ. Операции выделения, освобождения памяти и передача данных между памятью центрального и графического процессоров – очень дорогие по времени операции. Поэтому, например, если временная переменная требуется на каждом шаге итерации, то заводить её в памяти на каждом шаге, а потом сразу после использования уничтожать – это приведёт к тому, что основное время работы программы будет тратиться на операции с памятью, а не собственно на вычисления. Другой пример – если переменная требуется для вычислений в одном из ядер, а затем, ещё через несколько шагов – в другом ядре, и компилятор обладает информацией, что значение этой переменной между вызовами этих ядер изменено не было. Очевидно, что с точки зрения производительности гораздо эффективнее будет сохранить в памяти графического процессора значение такой переменной после вызова первого ядра, чем освободить занимаемую переменной память, а затем выделять её заново и заново загружать значения из памяти центрального процессора. Хотя с

точки зрения требуемого объёма памяти графического процессора это приводит к излишним накладным расходам (в памяти хранится переменная, не требующаяся для текущих расчётов).

При реализации механизма управления памятью в компиляторе программ на языке НОРМА для графических процессоров было решено использовать подходы, позволяющие достичь максимальной производительности получающихся программ, пусть даже за счёт повышенных требований к объёму памяти графического процессора.

3.4. «CUDA» соглашение о вызовах, спецификатор CUDA

Программа на языке Норма может состоять из нескольких разделов и функций. Разделы и функции вызывают друг друга, передают параметры. Если вызывается не «плоский» раздел (описанный со спецификатором PLAIN), то вызов осуществляется на центральном процессоре и параметры ему передаются как обычно – по значению или по указателю в памяти центрального процессора. Вызываемый раздел рассматривает свои формальные параметры как обычные переменные и дальше уже сам решает, нужно ли ему для вычислений выделять соответствующую область памяти в графическом процессоре и копировать туда значения своих формальных параметров, применяя механизм управления памятью, описанный выше, или нет. Такой подход прекрасно инкапсулирует происходящее в вызываемом разделе от вызывающего, но имеет существенный недостаток: если вызов происходит много раз в каком-то цикле, то при каждом вызове раздела для каждого передаваемого параметра будут производиться операции выделения/освобождения памяти графического процессора и копирования данных. Эта ситуацию аналогична описанной выше, когда начало и конец использования переменной находится внутри итерации. Но если в пределах трансляции одного раздела эту проблему можно решить, просто вынеся операции с памятью за пределы итерационного цикла, то в случае вызова раздела в цикле из другого раздела операции с памятью, которые должен производить вызываемый раздел, должны выноситься за пределы цикла в вызывающем разделе. Реализовать такие операции при условии раздельной трансляции разделов не представляется возможным.

Для решения этой проблемы была разработана и реализована в компиляторе следующая схема. Для каждого параметра, передаваемого через память центрального процессора, создаются дополнительно ещё два связанных с ним параметра: указатель на соответствующую память графического процессора и переменная-статус, определяющая, в какой памяти находится актуальное значение передаваемого параметра. Причём эти два параметра передаются по указателю, что даёт возможность вызываемому разделу их изменять, а вызывающему – учитывать сделанные изменения. Таким образом, если вызываемому разделу нужно произвести вычисления на графическом процессоре с использованием формального параметра, то при первом вызове он

выделит память на графическом процессоре и запишет её адрес в первом дополнительном параметре. А затем скопирует значение переменной в выделенную память и отразит это в статусе (втором дополнительном параметре). И потом, при всех последующих вызовах, при сохранении значений в дополнительных параметрах, вызываемый раздел будет понимать, что данные уже в памяти графического процессора и никаких лишних манипуляций производить не будет. А освобождать выделенную память будет уже вызывающий раздел, после цикла вызовов, проанализировав первый дополнительный параметр.

Такой алгоритм передачи параметров был назван соглашением о вызовах «CUDA». Для его успешного использования необходимо, чтобы он был применён как вызывающим, так и вызываемым разделом. Для этого в язык НОРМА был введён новый спецификатор раздела или функции CUDA. При его наличии и при генерации выходной программы для графических процессоров с использованием технологии NVIDIA CUDA компилятор будет генерировать актуальные и формальные параметры и их использование согласно данному соглашению.

Следует отметить, что использование соглашения о вызовах «CUDA» решает проблему эффективного использования памяти графического процессора для передаваемых параметров, но, к сожалению, аналогичную проблему для локальных и временных переменных вызываемого раздела решить не в состоянии. Для этого необходимо будет разработать и реализовать другие механизмы межпроцедурного взаимодействия.

4. Результаты применения компилятора

Приведённые выше методы и решения построения программы для графических процессоров фирмы NVIDIA с использованием технологии NVIDIA CUDA по декларативным описаниям были реализованы в компиляторе программ на языке НОРМА. Полученный компилятор был протестирован на реальной задаче из области газодинамики, на тесте CG из пакета NPB и на фрагменте задачи с большой вычислительной плотностью. Во всех трёх примерах применения компилятора удалось добиться получения полностью корректного результата и была проведена оценка эффективности получающихся исполняемых программ. Производительность программ для графических процессоров фирмы NVIDIA с использованием технологии NVIDIA CUDA сравнивалась с последовательной и OpenMP версиями той же самой программы. Для теста CG из пакета NPB также произведено сравнение с оригинальными версиями теста (последовательной и OpenMP), написанными вручную. Все запуски производились на вычислительном кластере К-100 [8]. Компиляция последовательных и OpenMP версий программ производилась компилятором Intel версии 15.0.0, компиляция CUDA программ – компилятором nvcc версии 6.5.

4.1. Прикладная задача из области газодинамики

В основном рабочем итерационном цикле данной программы производится расчёт различных величин для всех точек одномерной области. Для каждой точки вызывается «плоский» внешний раздел, который на основе предыдущих значений точки и её соседей высчитывает данные для новых значений. Все вычисления производятся с одинарной точностью. Времена выполнения версий программы приведены в таблице 1.

Последовательная программа, 1 ядро Intel Xeon X5670	OpenMP программа, 12 ядер Intel Xeon X5670	CUDA программа, 1 nVidia Fermi C2050
461 сек	45.6 сек	16.9 сек

Таблица 1. Время выполнения программы решения задачи из области газодинамики.

Полученный результат – ускорение в 27 раз по сравнению с последовательной программой – показывает, что для такого класса задач язык НОРМА подходит отлично. Построенная полностью автоматически по непроцедурным спецификациям на языке НОРМА программа для графических процессоров демонстрирует высокую скорость выполнения и эффективность применённых в компиляторе программ на языке НОРМА методов и решений.

4.2. Реализация теста CG из пакета NPB

Наиболее широко тесты для измерения производительности кластерных систем представлены в пакете NAS Parallel Benchmarks (NPB) [9], обладающим помимо объективных достоинств еще одним необъективным – авторитетностью программ и их авторов. Это послужило выбором данного пакета для исследования возможности реализации тестов на языке НОРМА и проверки эффективности кода, генерируемого компилятором с языка НОРМА.

После анализа исходных текстов программ тестового пакета NPB для эксперимента по портированию данного тестового пакета на язык НОРМА в первую очередь было выбрано ядро CG, как наиболее подходящее под концепцию языка НОРМА. Тестовое ядро CG (Conjugate Gradient) осуществляет приближение к наименьшему собственному значению большой разреженной симметричной положительно определенной матрицы с использованием метода обратной итерации вместе с методом сопряженных градиентов в качестве подпрограммы для решения СЛАУ. Тест применяется для оценки скорости передачи данных при отсутствии какой-либо регулярности. Вычисления производятся в определенных классах задач. Под классом понимается размерность основных массивов данных, используемых в тесте: «Sample code», «Class A» (маленькие), «Class B» (большие), «Class C»

(очень большие), «Class D» (огромные). Все вычисления производятся с двойной точностью.

После анализа исходных текстов ядра CG было принято решение переписать на языке НОРМА основную рабочую функцию теста, а также этапы инициализации теста и основной тестовый цикл. В то же время стало очевидно, что такие действия, как начальная инициализация разреженной матрицы, плохо подходит для реализации средствами языка НОРМА. Завершающий шаг – анализ правильности полученных результатов и вывод статистических данных о прохождении теста – также было решено оставить в оригинальном виде на Си для неоспоримости факта правильности анализа полученных результатов оригинальным кодом и сохранения формата вывода. Фактически, получается что тест начинает свою работу и заканчивает её кодом на Си, а та часть, в которой производятся все вычисления и при выполнении которой производится засечка времени и делается вывод о производительности системы, – реализуется на языке НОРМА. Поэтому в оставшемся коде на Си не нужно делать никаких предположений об архитектуре целевой системы и об используемых инструментах параллельного программирования. Этот код так и остаётся обычным последовательным кодом на Си и на быстродействие теста никак не влияет.

Среди конструкций, которые необходимо было портировать на язык НОРМА, оказались и такие, что не могут быть выражены средствами языка НОРМА. В первую очередь речь идёт о «сердце» теста – умножении разреженной матрицы на вектор. Но язык НОРМА поддерживает вызовы так называемых «внешних» функций, которые могут быть написаны на целевом языке программирования. Задача по реализации умножения разреженной матрицы на вектор (это делается с помощью косвенной адресации и цикла с переменными границами) была возложена на такую вспомогательную внешнюю функцию, а в программе на языке НОРМА в каждой точке области, соответствующей результирующему вектору, результат выполнения этой внешней функции в данной точке присваивается результирующей переменной.

Но подобная реализация умножения разреженной матрицы на вектор (фактически, оператор оставлен в том же виде, что был в оригинальном тесте), несмотря на очевидное достоинство – простота реализации (вся внешняя функция получилась из 8 строк) и хорошие показатели распараллеливания в версии OpenMP, плохо подошла для выполнения на графических процессорах. Поэтому были разработаны и реализованы ещё 2 варианта этой внешней функции, специально для версии для графических процессоров, которые учитывают особенности этой архитектуры. В первом из них каждая точка области рассчитывается одним блоком графического процессора. А цикл с переменными границами, в котором производится суммирование вычисленных значений, выполнен в виде редукции по нитям, где каждая нить предварительно высчитывает своё значение, соответствующее значению на определённом витке цикла. Такая реализация внешней функции получилась приблизительно в 120

строк кода. Второй вариант – использовать функцию из библиотеки cuSPARSE [1]. Такая реализация внешней функции получилась приблизительно в 60 строк кода. Результаты выполнения различных версий теста, как оригинальных, так и полученных в результате компиляции программ с использованием языка НОРМА, приведены в таблице 2. Измерения производились для классов теста A, B и C. Во всех случаях проверка результата прошла успешно и приведённые цифры – количество миллионов операций в секунду (Mop/s), основной показатель скорости выполнения теста.

	Class A		Class B		Class C	
	Си	Си + НОРМА	Си	Си + НОРМА	Си	Си + НОРМА
Последовательная программа, 1 ядро Intel Xeon X5670	1200	1175	718	716	513	615
OpenMP программа, 12 ядер Intel Xeon X5670	9737	8035	3595	3004	3240	2806
CUDA программа, простая реализация, 1 nVidia Fermi C2050	—	927	—	722	—	645
CUDA программа, реализация с редукцией, 1 nVidia Fermi C2050	—	2134	—	2178	—	1876
CUDA программа, использование cuSPARSE, 1 nVidia Fermi C2050	—	6521	—	4504	—	2704

Таблица 2. Результаты запуска теста CG (Mop/s).

Как можно заметить, результаты выполнения теста CG с использованием языка НОРМА практически идентичны (видимо, с точностью до погрешности измерений) соответствующим результатам оригинального теста CG, написанного вручную. Это даёт нам право утверждать, что тест CG прекрасно подошёл для записи его на языке НОРМА. И что задачи, имитирующиеся этим тестом, также должны хорошо подходить для программирования их (или по крайней мере их основных вычислительных ядер) на языке НОРМА. Ничуть не проиграв по скорости выполнения получившихся программ, компилятор программ с языка НОРМА полностью взял на себя задачу по автоматическому распараллеливанию выходной программы, и успешно её выполнил.

В то же время видно, что для эффективного выполнения на графических процессорах таких задач необходимо прибегать к более тонкому ручному программированию или, когда это возможно – использовать специальные библиотеки.

4.3. Фрагмент задачи с большой вычислительной плотностью

Данный фрагмент было решено привести здесь как пример вычислений, которые отлично подходят как для записи на языке НОРМА, так и для последующих вычислений на графических процессорах. Такие вычисления (с небольшим изменением) были выявлены в реальной прикладной задаче, когда один из авторов данной статьи занимался ручным распараллеливанием этой прикладной задачи. После выделения данного фрагмента было решено попробовать вынести его в отдельную процедуру и записать на языке НОРМА, чтобы посмотреть, каким будет результат автоматического распараллеливания его на различные архитектуры. Приведём этот фрагмент полностью:

```
DOMAIN PARAMETERS Nx=300,Ny=300.
ok1:(k1=1..Nx). ok2:(k2=1..Ny). ok:(ok1;ok2).
oi1:(i1=1..Nx). oi2:(i2=1..Ny). oi:(oi1;oi2).
VARIABLE X,Y DEFINED ON ok DOUBLE.
    FOR ok  ASSUME   X   = 1.0. // Initial data
CONSTANT d1 = 10.0D. CONSTANT d2 = 20.0D. CONSTANT pi2 = 3.14157D/2.0.
    FOR ok  ASSUME
        Y = SUM((oi)X[k1=i1,k2=i2]*sin(fmod(k1*i1*d1*k2*i2*d2,pi2))/i2).
```

В данном фрагменте в каждой точке двумерной области происходит суммирование по такой же двумерной области выражения, где синус вычисляется по данным, зависящим как от точки области результата, так и от точки области суммирования. Все вычисления производятся с двойной точностью. В итоге получаются массовые однородные независимые друг от друга вычисления. Времена выполнения этого фрагмента на различных архитектурах приведены в таблице 3.

Последовательная программа, 1 ядро Intel Xeon X5670	OpenMP программа, 12 ядер Intel Xeon X5670	CUDA программа, 1 nVidia Fermi C2050
894 сек	80.8 сек	15.3 сек

Таблица 3. Время выполнения фрагмента задачи с большой вычислительной плотностью.

Продемонстрированная высокая производительность вычислений на графическом процессоре показывает, что для такого класса задач применение языка НОРМА более чем оправдано. По достаточно короткой непроцедурной записи компилятор программ на языке НОРМА способен автоматически сгенерировать эффективное решение для параллельных архитектур.

5. Заключение

В настоящее время ведется разработка версии компилятора с языка НОРМА для высокопроизводительных вычислительных систем с гибридной

архитектурой. В генерируемой выходной программе будут одновременно использоваться технологии MPI, OpenMP и NVIDIA CUDA. Компилятор будет автоматически решать такие задачи, как распределение вычислительной нагрузки между графическими процессорами и центральным процессором, а также между узлами гибридной вычислительной системы, организация обмена данными между узлами и внутри одного узла и т.п.

Работа выполнена при финансовой поддержке гранта РФФИ № 15-01-03039-а.

Литература

1. CUDA Toolkit Documentation,
   URL: http://docs.nvidia.com/cuda
2. OpenACC,
   URL: http://openacc.org
3. В.А. Бахтин, И.Г. Бородич, Н.А. Катаев, М.С. Клинов, В.А. Крюков, Н.В. Поддерюгина, М.Н. Притула, Ю.Л. Сазанов. Распараллеливание с помощью DVM-системы некоторых приложений гидродинамики для кластеров с графическими процессорами. Научный сервис в сети Интернет: поиск новых решений: Труды Международной суперкомпьютерной конференции (17-22 сентября 2012 г., г. Новороссийск). — М.: Изд-во МГУ, 2012. с. 444 – 450.
4. Описание языка программирования COLAMO,
   URL: http://colamo.parallel.ru
5. И.Б. Задыхайло. Организация циклического процесса счета по параметрической записи специального вида. Журн. выч. мат. и мат. физ., т.3, N2, 1963, с.337-357.
6. Андрианов А.Н., Бугеря А.Б., Ефимкин К.Н., Задыхайло И.Б. НОРМА. Описание языка. Рабочий стандарт. — Препринты ИПМ им.М.В.Келдыша, 1995, № 120, 52 с.
7. Система НОРМА,
   URL: http://www.keldysh.ru/pages/norma
8. Гибридный вычислительный кластер К-100,
   URL: http://www.kiam.ru/MVS/resourses/k100.html
9. NAS Parallel Benchmarks,
   URL: http://www.nas.nasa.gov/publications/npb.html