Применение GPU в рамках гибридного двухуровневого распараллеливания MPI+OpenMP на гетерогенных вычислительных системах*

А.В. Горобец[1], С.А. Суков[1], А.О. Железняков[2], П.Б. Богданов[2], Б.Н. Четверушкин[1]

ИПМ РАН им М.В. Келдыша[1], НИИСИ РАН[2]

В работе рассматривается применение расширенного распараллеливания для расчетов задач газовой динамики и аэроакустики на гетерогенных кластерах с узлами, сочетающими вычислительные элементы принципиально разной архитектуры, CPU и GPGPU. Двухуровневая модель распараллеливания MPI+OpenMP дополняется применением OpenCL для загрузки GPGPU, таким образом, реализуется третий уровень параллелизма. Представлена параллельная модель алгоритма для неструктурированных сеток.

1. Введение

В настоящее время наблюдается определенная тенденция в развитии вычислительных систем. С одной стороны, как и прежде, продолжается рост производительности за счет увеличения числа процессорных ядер. С другой стороны, становятся всё более популярными гетерогенные системы, высокая производительность которых обусловлена применением вычислительных элементов принципиально иной архитектуры. Первое направление развития требует от алгоритма все большей степени параллелизма и мотивирует переход от MPI модели на двухуровневую параллельную модель MPI+OpenMP, которая лучше соответствует современной архитектуре суперкомпьютеров с многоядерными узлами. Второе направление требует адаптации алгоритмов к гетерогенной архитектуре и еще более сложной параллельной модели, сочетающей принципиально различные типы параллелизма.

В отличие от "обычного" кластера, на котором вычисления выполняются на однотипных процессорных ядрах, на гетерогенном кластере вычисления также выполняются на GPU, или более точно GPGPU (General-purpose graphics processing units — графические процессоры общего назначения), имеющих существенно отличающуюся архитектуру. GPU устройства имеют свою собственную оперативную память, доступ к которой осуществляется посредством шины PCI-Express. Программа для такой архитектуры состоит из кода для CPU, другими словами host-кода, который написан на обычном языке программирования C, C++, и кода для GPU – kernel кода, написанного на специальном языке, как правило, производном от C. Для выполнения вычислений на GPU необходимо передать входные данные в память GPU, выполнить набор kernel-подпрограмм (называемых ниже вычислительными ядрами) и получить результаты обратно в память CPU.

Для расчетов на гетерогенных вычислительных системах предлагается использовать многоуровневую параллельную модель. На первом уровне, как и на “обычном” кластере, используется MPI для объединения узлов в рамках модели с распределенной памятью на основе традиционного геометрического параллелизма. На втором уровне применяется OpenMP в рамках того же MIMD (multiple instruction multiple data - множественные потоки команд и данных) параллелизма, но в рамках модели с общей памятью для распараллеливания по CPU ядрам внутри узла. По сравнению с MPI, использование OpenMP имеет некоторые особенности, связанные с одновременным доступом к общей памяти, специфическими узкими местами многоядерных компьютеров и так далее. Более подробно типичные проблемы , возникающие в случае применения OpenMP, рассматриваются, например, в [1]. Гибридная двухуровневая модель распараллеливания MPI+OpenMP является достаточно хорошо известным подходом в вычислительной

* Работа выполнена при поддержке совета по грантам президента Российской Федерации, грант МК-7559.2010.1. Для расчетов использовался суперкомпьютер Ломоносов НИВЦ МГУ, суперкомпьютер К100 ИПМ РАН и тестовый стенд GPGPU НИИСИ РАН.

газовой динамике. Алгоритмы на основе MPI+OpenMP и сравнение с “плоским” MPI подходом приводятся, например, в [2–4]. Сравнительный анализ показывает, что гибридный подход повышает производительность вычислений, но, как правило, не намного. В [5], анализ производительности для MPI и MPI+OpenMP подходов к распараллеливанию приводит к вполне закономерному выводу, что второе имеет преимущество над первым только при расчетах на большом числе процессоров. Однако, основной мотивацией использования более сложной параллельной модели MPI+OpenMP является не столько выигрыш в производительности, сколько возможность задействовать существенно большее число процессоров.

Далее параллельная модель должна быть дополнена другим типом параллелизма для того, чтобы задействовать GPU, в которых, помимо MIMD параллелизма, также используется SIMD (single instruction multiple data – один поток команд на множество данных) параллелизм на уровне потоковых процессоров. Таким образом, появляется третий уровень, на котором из одной или нескольких OpenMP нитей CPU кода вызываются вычислительные ядра, выполняющиеся на одном или нескольких графических процессорах. Перенос вычислений на архитектуру GPU сам по себе представляет достаточно сложную задачу, требующую учета множества факторов. Примеры адаптации алгоритмов под графические процессоры представлены, например, в работах [6, 7].

В данной работе на упрощенном примере рассматривается проблема переноса газодинамического алгоритма для неструктурированных сеток на GPU. Далее в разделе 2 приводится краткое описание математической модели и двухуровневого распараллеливания MPI+OpenMP, в разделе 3 рассматривается адаптация упрощенного алгоритма к GPU архитектуре и в разделе 4 приводятся результаты измерения производительности вычислений на графических процессорах. Раздел 5 посвящен общему подходу к гетерогенным вычислениям.

2. Математические модели и параллельная реализация алгоритмов на CPU

Рассматривается моделирование сжимаемых течений с использованием неструктурированных сеток. Класс моделей аэроакустики [8], построенных на основе уравнений Эйлера, включает в себя три основные модели: линейная (линеаризованные уравнения Эйлера), нелинейная для пульсационных компонент течения, описываемая различными формами NLDE (NonLinear Disturbance Equations) уравнений, нелинейная для всего течения, описываемая полными уравнениями Эйлера. Действие молекулярной вязкости и теплопроводности реализовано в рамках моделей на основе уравнений Навье–Стокса и их линейных аналогов.

Для пространственной дискретизации в рамках конечно-объемного подхода используются неструктурированные тетраэдральные и гибридные сетки (которые также могут содержать шестигранники, четырехугольные пирамиды и треугольные призмы).

Конвективный поток через грани контрольных объемов вычисляется с использованием схемы Роу. Аппроксимация потоков имеет повышенный порядок точности (до шестого включительно) и реализуется на расширенном шаблоне, включающем два противопотоковых тетраэдра и все соседние узлы их вершин [9]. Для интегрирования по времени используются явные схемы Рунге-Кутты до 4-го порядка и неявные схемы до 2-го порядка на основе линеаризации по Ньютону.

Для параллельной реализации численных алгоритмов используется гибридный двухуровневый подход с MPI на первом уровне и OpenMP на втором. Применение OpenMP позволяет повысить параллельную эффективность на вычислительных системах с многоядерными узлами при использовании большого числа процессоров, поскольку в T раз сокращается число MPI процессов, где T – число OpenMP нитей. Это дает следующие преимущества:

1) MPI-процессу доступно примерно в T раз больше памяти.
2) Сокращается количество обменов данными, поскольку в T раз уменьшается количество участвующих в обмене данными MPI процессов
3) Примерно во столько же раз уменьшается общий объем пересылки, поскольку уменьшается протяженность границ между подобластями.

4) Не происходит простаивание процессов в ожидании своей очереди на доступ к разделяемым коммуникационным ресурсам узла.

Однако OpenMP также имеет некоторые особенности. Одна из основных проблем, это пересечение по данным между параллельными нитями. Присутствие критических секций и атомарных операций, которые ограничивают одновременный доступ нитей к данным, существенно снижает производительность. Избежать пересечения можно простым способом, реплицируя данные для каждой нити, однако это ведет к росту затрат оперативной памяти. Более универсальный и эффективный способ, который был реализован, это двухуровневое разбиение расчетной области, когда подобласть MPI-процесса разбивается далее на подобласти OpenMP нитей. Элементы сетки переупорядочиваются таким образом, чтобы внутренние элементы подобластей были сгруппированы в памяти и отделены от интерфейсных элементов (т. е. элементов сетки, принадлежащих более чем одной OpenMP подобласти). Это позволяет локализовать пересечение по данным. Нити OpenMP параллельно обрабатывают данные, соответствующие внутренним элементам, а затем последовательно (или параллельно, но с наложением вычислений) обрабатываются интерфейсные элементы. При этом, поскольку количество интерфейсных элементов, как правило, намного меньше, чем внутренних, достигается высокая параллельная эффективность.

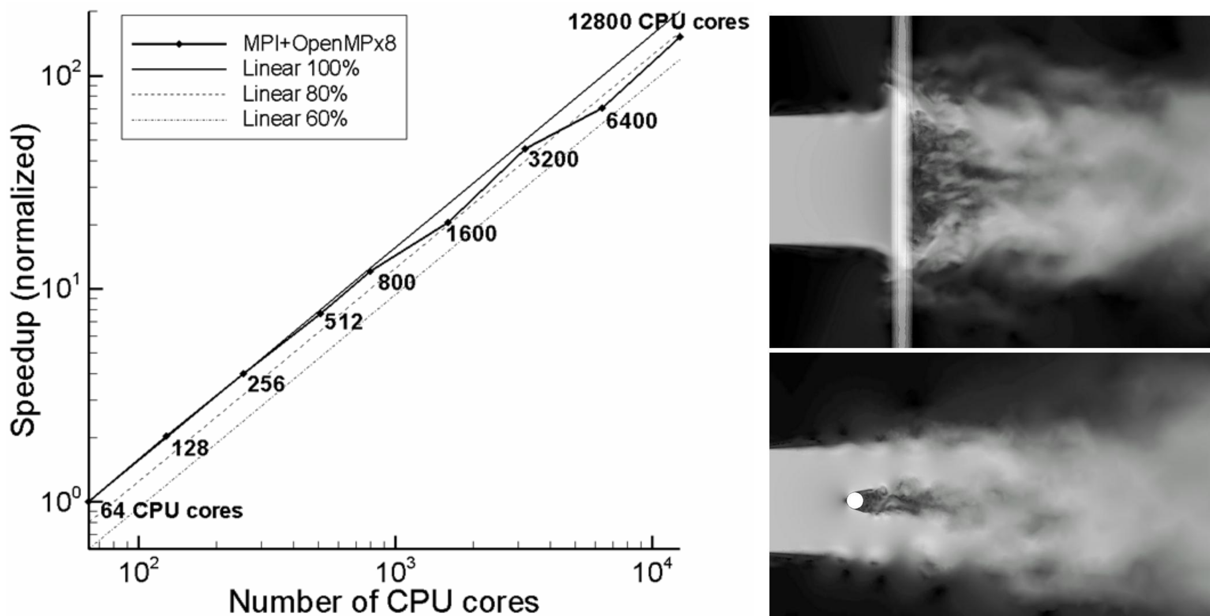


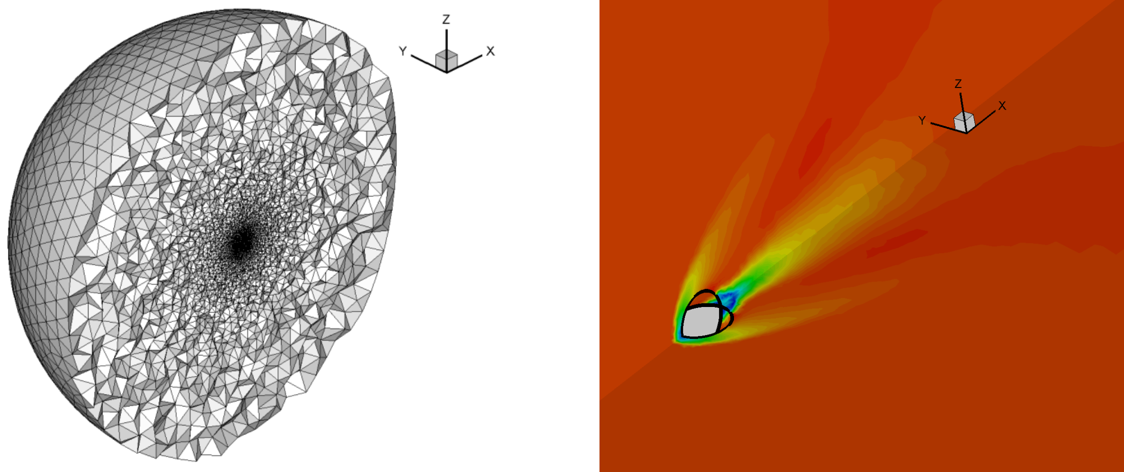
Рис. 1 Ускорение на суперкомпьютере Ломоносов, нормированное по времени на 64 ядрах (слева), полученное на расчете обтекания цилиндра дозвуковой струей (справа).

Тест на ускорение с двухуровневым распараллеливанием MPI+OpenMP для схемы повышенного порядка с центрами в узлах был выполнен на неструктурированной тетраэдральной сетке, содержащей 16 миллионов узлов и 100 миллионов тетраэдров, для задачи об обтекании цилиндра дозвуковой струей. Моментальная картина течения показана для наглядности на рис. 1 справа. Ускорение, нормированное по времени на 64 ядрах, составило на 12800 ядрах 152. Результаты показаны на рис. 1 слева. Один шаг по времени 4-шаговой схемы Рунге-Кутты занимал 26.8 и 0.18 секунды на 64 и 12800 ядрах соответственно.

Детальное таймирование по различным составляющим алгоритма показывает, что имеется достаточный запас параллелизма, и для расчетов на более подробных сетках могут быть эффективно задействованы несколько десятков тысяч процессорных ядер. Теперь основная проблема, которую требуется решить, это переход к гетерогенной архитектуре с графическими процессорами.

Применение GPU исследуется на упрощенном, но репрезентативном примере, воспроизводящем характерную вычислительную нагрузку. В качестве математической модели взята система безразмерных уравнений Эйлера, в качестве модельной тестовой задачи - сверхзвуковое обтекание сферы (число Маха равно 2.75). Расчетная область представляет собой сферу

(D = 50), внутри которой расположено обтекаемой тело - сфера единичного диаметра. Центр обтекаемой сферы совпадает с началом координат. Для дискретизации по пространству применяется конечно-объемный метод с определением значений сеточных функций в центрах элементов сетки, то есть контрольные объемы (расчетные ячейки) совпадают с элементами сетки. Используется неструктурированная тетраэдральная сетка, равномерно сгущающаяся к поверхности обтекаемого тела (рисунок 2 слева). Конвективный поток через грани контрольных объемов вычисляется с использованием схемы Роу. Интегрирование по времени – схема Рунге-Кутты первого порядка точности.



а) поверхностная сетка б) срединное сечение Y = 0

Рис. 2. Тетраэдральная сетка: 79 608 узлов, 472 114 элементов, 942 088 внутренних граней расчетных ячеек (слева) и картина течения (справа).

На рисунке 2 справа показана построенная по результатам проведенных вычислительных экспериментов картина распределения модуля скорости в плоскостях Y = 0 и Z = 0 вблизи поверхности сферы. Далее для данного примера подробно рассматривается реализация упрощенного алгоритма на GPU.

3. Использование графических процессоров

3.1 Выбор средства разработки

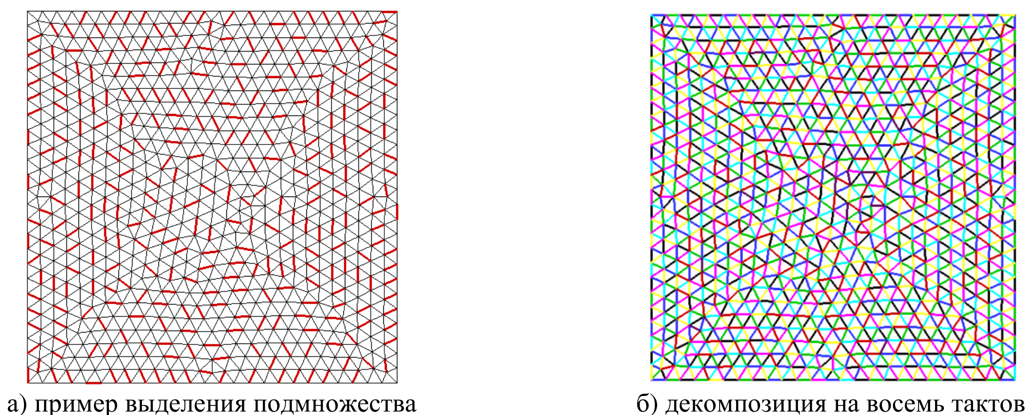
Существуют несколько методик и средств разработки для написания вычислительных ядер и компиляции программ для выполнения на GPU. Это в частности CUDA (Compute Unified Device Architecture) для графических процессоров NVidia, поддерживающих технологию GPGPU (произвольных вычислений на видеокартах) и OpenCL (Open Computing Language - открытый язык вычислений) для GPU различных производителей [10–12]. CUDA является наиболее популярным средством в настоящее время. Однако комплексы программ, использующие CUDA, не переносимы и могут использоваться только на оборудовании NVidia. Это может являться существенным недостатком, поскольку часто необходимо, чтобы расчеты можно было выполнять на любых доступных вычислительных системах. Поэтому выбор был сделан в пользу OpenCL, который обеспечивает переносимость программы и позволяет использовать графические процессоры как производства NVidia, так и ATI (AMD). Для сравнения на начальном этапе были выполнены реализации алгоритма с использованием обоих средств, CUDA и OpenCL, для того чтобы убедиться, что реализация на OpenCL не уступает в производительности реализации на CUDA. Измерение производительности было выполнено на GPU NVidia C1060 и показало отсутствие заметных различий в скорости работы программы.

3.2 Адаптация программных алгоритмов к GPU

С точки зрения программной реализации, численный алгоритм, по сути, представляет собой множество требующих обработки однотипных заданий по вычислению потоков через грани расчетных ячеек. Для вычисления потока через внутреннюю грань необходимо знать значения газодинамических переменных в ячейках, которым эта грань принадлежит, а так же ее геометрические параметры (площадь, вектор нормали и т.д.). Для вычисления потока через граничную грань помимо ее геометрических параметров необходимо знать значения газодинамических переменных только в одной ячейке. Поскольку значения газодинамических переменных считаются постоянными для одного шага интегрирования по времени, то задания по вычислению потоков независимы по входным данным. То есть они могут обрабатываться параллельно, что идеально подходит для перехода на GPU. Но, как и в случае распараллеливания таких алгоритмов в рамках модели общей памяти, остается проблема суммирования потоков. Определенный с использованием схемы Роу поток через общую грань двух расчетных ячеек i и j суммируется в соответствующие массивы потоков по ячейкам, которые далее используются для нахождения значений газодинамических переменных на новом слое по времени. Тогда, если, например, параллельно вычислять потоки через грани между ячейками i → j и i → k, то существует большая вероятность возникновения ошибки при одновременном суммировании потоков для ячейки i двумя разными нитями. В данном случае при разработке алгоритма для GPU эта проблема была решена двумя способами: вычислением потоков с репликацией данных и потактовым вычислением потоков.

В первом случае, с целью исключения конфликтов по доступу к памяти на этапе вычисления потоков создается дополнительный массив для записи потоков по граням ячеек, что несколько увеличивает потребление оперативной памяти: для каждой грани записывается два набора из 5 значений (плотность, 3 компоненты скорости и давление). Для использованной тетраэдральной сетки, имеющей 942 088 внутренних граней, это около 75 МБ для чисел двойной точности. Кроме того, после вычисления потоков через грани, необходимо просуммировать потоки, чтобы получить конечные значения в ячейках. Для этого необходимо хранить в памяти так называемый дуальный граф связей контрольных объемов, который содержит $(2 \cdot N_S + N_E + 1)$ целочисленных значений. Здесь N_E – число ячеек в сетке, а N_S – число внутренних граней контрольных объемов. Для рассматриваемого случая объем памяти для хранения дуального графа сетки составляет порядка 9 МБ, а общее увеличение затрат оперативной памяти равно 84 МБ, что в общем случае не является критичным. Плюс этого подхода заключается в том, что все задания по обработке внутренних граней контрольных объемов могут быть запущены одновременно.

Во втором случае к дуальному графу сетки применяется алгоритм раскраски ребер. Идею данного алгоритма покажем на примере дуального графа сетки, изображенного на рисунке 3. Узлы графа – ячейки сетки, ребра – связи ячеек через общие грани контрольных объемов.



а) пример выделения подмножества   б) декомпозиция на восемь тактов

Рис.3. Декомпозиция дуального графа для реализации потактового алгоритма.

Попытаемся выделить из множества ребер графа такое подмножество ребер, чтобы каждый из узлов встречался не более одного раза (рисунок 3 а). Обработка запущенными на GPU нитями такого подмножество ребер может происходить параллельно и по определению исключает возможность конфликтов нитей по модификации одних и тех же ячеек памяти. Далее разобьем дуальный граф на такие подмножества (рисунок 3 б), а саму процедуру вычисления потоков на такты. На каждом из таких тактов обрабатывается одно из выделенных подмножеств ребер сетки. Число тактов по вычислению потоков не может быть меньше максимального числа граней для контрольных объемов, то есть, например, для гексаэдральной сетки не будет меньше 6. При этом число одновременно обрабатываемых на каждом из тактов граней не может быть больше, чем $N_E/2$. Необходимость многократного запуска процедуры вычисления потоков для каждого из тактов и снижение числа обрабатываемых одновременно граней – отрицательные стороны данного метода. Но по сравнению с репликацией данных в потактовом алгоритме нет необходимости в выделении дополнительных объемов оперативной памяти и лишнем суммировании вычисленных по граням потоков в ячейки.

Производительность обоих способов была сравнена на GPU NVidia C1060, первый вариант оказался быстрее на 17% и был принят в качестве основного. Было также оценено влияние на производительность способа размещения блочных векторов в памяти. Например, каждому контрольному объему соответствует 5 значений физических переменных. Таким образом, данные могут быть развернуты в линейный массив путем записи N_E блоков по 5 значений. В этом случае адресация j-й переменной для i-го контрольного объема будет иметь вид $V[i*5+j]$. Такой вариант больше подходит для архитектуры CPU, но для GPU архитектуры намного эффективнее разворачивать данные в 5 блоков по N_E значений, что соответствует адресации $V[i+j*N_E]$. Такое размещение соответствует слиянию доступа к памяти (coalescing), что существенно увеличивает производительность (в данном случае более чем в полтора раза).

4. Производительность вычислений на GPU

Для тестовой реализации алгоритма было выполнено измерение производительности на GPU и CPU с помощью таймирования с высоким разрешением временных затрат на выполнение как всего шага по времени, так и отдельных операций. В частности, замерялся расчет потоков через грани контрольных объемов, суммирование потоков с граней в ячейки, граничные условия (Г. У.), переход на новый временной слой по схеме Рунге-Кутты (Р. - К.). Измерения выполнялись для последовательной host-программы, работающей на одном CPU ядре, и программы, использующей одно GPU устройство. Измерения усреднялись на интервале порядка 100 шагов по времени, чтобы повысить точность таймирования и избежать случайных отклонений в результатах измерений. Все вычисления выполняются для чисел с плавающей точкой **двойной точности** (double, 64 бита). Результаты приведены в таблице 1 для нескольких моделей GPU и CPU.

Таблица 1. Время вычислений (сек.) на GPU и CPU.

Операция	**Intel Xeon E5504 2.0GHz**	**Intel Xeon X5670 2.93GHz**	**NVidia C1060**	**NVidia GTX470**	**NVidia C2050**	**AMD Radeon 5870**	**AMD Radeon 6970**
Расчет потоков	0.255	0.172	0.0095	0.0053	0. 0056	0.0041	0.0031
Суммирование	--- [1]	---	0.0043	0.0033	0.0026	0.0030	0.0022
Г. У.	0.0011	0.0008	0.00045	0.00021	0.0003	0.0004	0.00028
Р. - К .	0.029	0.022	0.0015	0.00103	0.0014	0.0016	0.0019
Всего	0.285	0.195	0.0157	0.00984	0.0099	0.0091	0.0075

В данном случае алгоритм полностью выполнялся на GPU, при интегрировании по времени не требовалось передачи данных между CPU и GPU. Если бы на шаге по времени присутст-

[1] На CPU суммирование не выполняется, потоки сразу рассчитываются в узлах.

вовали этапы вычислений, выполняющиеся только на CPU, что возможно будет иметь место в перспективе, то обмен данными составил бы две операции (запись и чтение) по $5 \cdot N_E$ значений, которые бы заняли примерно 0.01 секунды, что могло бы увеличить время вычислений на GPU.

Для проверки корректности вычислений на GPU, как корректности реализации алгоритма, так и корректности работы устройства, вычисления дублировались на CPU, и через заданное число шагов выполнялось сравнение значений величин во всех контрольных объемах. На одной из тестируемых аппаратных конфигураций, а именно на системе, имеющей 4 GPU NVidia Tesla C1060, была отмечена некорректная работа 3-х устройств из 4-х: возникали расхождения с результатами CPU в произвольные моменты, меняющиеся от запуска к запуску. При интенсивном доступе к памяти GPU в процессе вычислений некорректно срабатывала выборка данных из памяти, что было проверенно на отдельном тесте с копированием массивов. При уменьшении числа нитей проблема пропадала, но при этом производительность снижалась в полтора раза. На других аналогичных устройствах такой проблемы не возникло, однако обнаруженная ненадежность GPU в случае выполнения интенсивных вычислений приводит к выводу о том, что необходима разработка дополнительного механизма проверки результатов вычислений на графических процессорах. К примеру, в данном случае, при отсутствии сравнения с CPU кодом, проблему невозможно было бы обнаружить, так как ошибки выборки из памяти не приводили к возникновению некорректных значений (NaN, inf, отрицательная плотность или давление, и т.д.). Однако, в реальных расчетах сравнение вычислений GPU и CPU на каждом шаге по времени естественно не имеет смысла и нужно искать другие подходы.

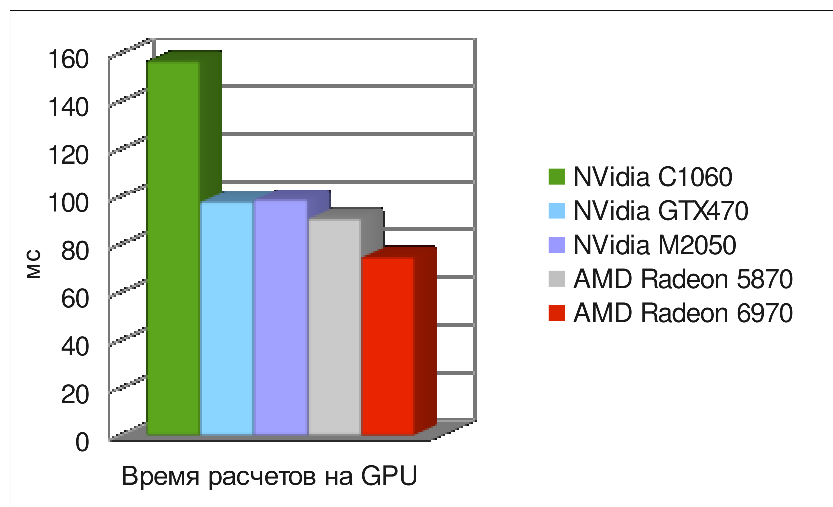


Рис.4. Сравнительная диаграмма времен исполнения алгоритма для разных моделей GPU

Работа тестовой реализации алгоритма была проверена на графических процессорах ATI (AMD) и NVidia. Здесь следует отметить, что на ATI 5870 и ATI 6970 была получена более высокая производительность, чем на NVidia C2050, при том, что последнее имеет примерно в 10 раз большую стоимость (на московском рынке декабря 2010 года). В то же время, действия по оптимизации при написании кода вычислительных ядер имели на ATI намного больший эффект, чем на NVidia. Из этого можно сделать вывод, что ATI имеет более слабый компилятор, в котором не реализованы основные оптимизационные алгоритмы, тогда как на NVidia компилятор сам выполняет оптимизацию кода достаточно хорошо. На ATI доработка кода повысила производительность примерно в два раза, в то время как на NVidia только на 10%. Также на AMD Radeon 6970 подпрограммы Г. У. и Р. - К. работали медленнее, чем на более старой модели, что тоже говорит о недостаточной эффективности компилятора AMD (ATI) для 6970. К особенностям модели 6970 надо также отнести энергосберегающий режим, в котором понижаются частоты процессора и памяти, поэтому необходимо «прогревать» карту перед началом расчетов для достижения максимальной производительности. Действия по оптимизации кодов вычислительных ядер, которые были выполнены, включают в себя:

1. Замена всех переменных, известных до выполнения ядра, на константы. Поскольку компиляция ядра выполняется в процессе работы CPU программы, на момент компиляции, например, уже известны размерности массивов, число контрольных объемов и граней, различные параметры (число Маха, параметры численной схемы и т.д.). Поэтому эти значения дописываются в виде #define в текст исходного кода ядра. Это действие на NVidia дало выигрыш 5%.
2. Минимизация числа операций деления. На центральном процессоре разница в скорости между выполнением операций умножения и деления сказывается намного меньше, в то время как на GPGPU эта разница может достигать двух порядков
3. Как уже было сказано, существенное, в полтора раза, увеличение скорости произошло при переходе к слиянию доступа к памяти (coalescing), за счет переупорядочивания массивов.
4. Существенным для ATI стало добавление директивы #pragma unroll <N> перед циклами внутри ядер.
5. Ядро Г. У. для ATI удалось ускорить в 50 раз после анализа результатов профайлера и сведения к нулю количества фальшивых регистров (scratch register).

5. Общий подход к гетерогенным вычислениям

Как уже было сказано, во многих реальных задачах потребуется эффективное использование ресурсов одновременно и CPU и GPU. Поэтому, необходима проработка общей методологии программирования гетерогенных вычислительных систем, то есть систем, в состав которых входят вычислители принципиально отличных архитектур. Современный вычислительный узел может одновременно нести на борту несколько универсальных многоядерных процессоров (SMP CPU), потоковые ускорители на базе графических процессоров (GPGPU), модули программируемой логики (FPGA), гибридные процессоры-ускорители (CELL) и т.д. Каждое устройство имеет свою модель программирования. В этой связи, эффективное программирование неоднородного вычислительного узла представляет собой нетривиальную задачу, сопряженную с глубоким знанием архитектуры каждого вычислителя и сбалансированным распределением аппаратных ресурсов под конкретную задачу. Проблему зоопарка программных моделей призван решить открытый стандарт языка для вычислений OpenCL, разработанный консорциумом Khronos[10]. Стандарт OpenCL оперирует с обобщенной моделью вычислительного устройства, под которую подходит большинство известных на текущий момент архитектур.

Вопрос планирования задач, пересылки данных, балансировки нагрузки между вычислителями, синхронизации этапов вычислений и т.п. решает модель StarPU - пакет, разработанный в национальном исследовательском институте информатики и автоматизации Франции[13]. На базе StarPU реализовано новое поколение эффективных библиотек численного анализа для гетерогенных систем [14].

Новые стандарты и подходы к программированию гетерогенных систем должны лечь в основу будущих кодов и пакетов прикладных программ. Отдельной задачей стоит перенос кодов, написанных в старой парадигме программирования многоядерных систем.

6. Заключение

Реализованное двухуровневое распараллеливание MPI+OpenMP позволяет выполнять расчеты на сетках с числом узлов до 200 миллионов и числом тетраэдров до 1.2 миллиарда с использованием более 10 тысяч процессорных ядер. На примере упрощенной математической модели был выполнен перенос вычислений на графические процессоры. Численный алгоритм для моделирования внешнего обтекания невязким сжимаемым газом на основе явной схемы первого порядка с центрами в контрольных объемах для неструктурированных сеток был адаптирован к вычислениям на GPU и реализован в виде вычислительных ядер на OpenCL. Сравне-

ние производительности вычислений с двойной точностью на одном ядре CPU и на одном GPU устройстве показало 20 ~ 38-кратное ускорение всего алгоритма.

Литература

1. Aubry R., Houzeaux G., Vazquez M., Cela J. M., Some useful strategies for unstructured edge-based solvers on shared memory machines, International Journal for Numerical Methods in Engineering (2010) n/a. doi:10.1002/nme.2973.

2. Itakura K., Uno A., Yokokawa M., Ishihara T., KanedaY., Scalability of hybrid programming for a CFD code on the Earth Simulator, Parallel Computing 30 (12) (2004) 1329–1343

3. Nakajima K., Three-level hybrid vs. flat MPI on the Earth Simulator: Parallel iterative solvers for finite-element method, Applied Numerical Mathematics 54 (2) (2005) 237–255.

4. Heuveline V., Krause M.J., Latt J., Towards a hybrid parallelization of lattice Boltzmann methods, Computers and Mathematics with Applications 58 (5) (2009) 1071–1080.

5. Chorley M.J., Walker D.W., Performance analysis of a hybrid MPI/OpenMP application on multi-core clusters, Journal of Computational Science 1 (3) (2010) 168–174.

6. Monakov A., Lokhmotov A., Avetisyan A., Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures, High Performance Embedded Architectures and Compilers, Lecture Notes in Computer Science, 2010, Volume 5952/2010, 111-125, DOI: 10.1007/978-3-642-11515-8_10

7. Buatois, Luc and Caumon, Guillaume and Levy, Bruno, Concurrent number cruncher: a GPU implementation of a general sparse linear solver, Int. J. Parallel Emerg. Distrib. Syst., 24 (3) (2009) 205—223, DOI: 10.1080/17445760802337010

8. Abalakin I., Dervieux A., Kozubskaya T., Computational Study of Mathematical Models for Noise DNS. – AIAA-2002-2585 paper.

9. Abalakin I., Dervieux A., Kozubskaya T., Ouvrard H., Accuracy Improvement for Finite-Volume Vertex-Centered Schemes Solving Aeroacoustics Problems on Unstructured Meshes, – AIAA paper 2010-3933 (2010)

10. Khronos OpenCL Working Group, The OpenCL Specification, Version: 1.1, http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf (2010)

11. Advanced Micro Devices, Inc, AMD Accelerated Parallel Processing OpenCL Programming Guide, http://developer.amd.com/gpu/AMDAPPSDK/assets/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf (2011).

12. NVIDIA, OpenCL Programming Guide for the CUDA Architecture Version 2.3 , http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_OpenCL_ProgrammingGuide.pdf, (2011)

13. INRIA RUNTIME team, A Unified Runtime System for Heterogeneous Multicore Architectureshttp://runtime.bordeaux.inria.fr/StarPU/ (2010)

14. Dongarra J., Tomov S., Agullo I., Ltaief H., Augonnet C., Namyst R., Thibault S., Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs, (2010)