GPU версия CFD пакета SigmaFlow: портирование и оптимизация с использованием инструментария TTG Apptimizer

А.А. Гаврилов[1], М.А. Кривов[2], С.А. Гризан[2], А.А. Дектерёв[1]

Институт теплофизики СО РАН[1], ООО «ТТГ Лабс»[2]

В данной статье представлены результаты работ по переносу на GPU программы для пространственных тепло-гидродинамических расчетов SigmaFlow. Рассмотрены возникшие проблемы портирования и предложенные варианты их решения, в том числе описаны произведённые модификации форматов данных и самих алгоритмов, подходы к минимизации пересылок данных между вычислителями и узлами кластера. Отдельно приведены результаты оптимизации разработанных версий алгоритмов с целью повышения степени использования блоков GPU. Продемонстрировано итоговое ускорение от переноса вычислений на GPU порядка 7.4 - 11.3 раз относительно последовательной реализации и более 3 раз относительно параллельной версии, выполняющейся на четырёхядерном центральном процессоре.

1. Введение

В связи с массовым переходом к гетерогенным вычислительным системам, которые оснащены вспомогательными ускорителями, крайне актуальной становится задача адаптации существующих пакетов программ к новым подобным архитектурам. В настоящий момент данные гетерогенные системы в основном представлены кластерами, узлы которых содержат от двух до трёх графических ускорителей серии NVidia Tesla. Однако на смену (или же в дополнение) к ним приходят кластера, оснащённые сопроцессорами Intel Xeon Phi, а также начинают появляться прототипы суперкомпьютеров на базе гибридных процессоров, содержащих вычислительные ядра нескольких типов, каждый из которых лучше подходит для решения соответствующего класса задач.

Какой бы из трёх перечисленных выше типов гетерогенных вычислительных систем не стал доминирующим, проблема переноса большинства существующих программ на новые архитектуры по-прежнему остаётся актуальной. Во-первых, во всех концепциях гетерогенных вычислительных архитектур предполагается многократное повышение степени параллелизма. К примеру, если раньше количество независимых потоков вычислений, необходимых (но не достаточных!) для эффективной загрузки всех блоков центрального процессора равнялось количеству ядер, помноженному на количество выполняемых за такт одним ядром инструкций, и измерялось несколькими десятками, то для обеспечения аналогичной загрузки лишь одного графического ускорителя требуется не менее 512 потоков вычислений. Понятно, что далеко не все программные реализации и, тем более, алгоритмы обладают необходимым потенциальным параллелизмом, и поэтому при осуществлении их портирования на гетерогенные системы редко удаётся обойтись лишь «косметическими» изменениями.

Во-вторых, с появлением гетерогенных архитектур неизменно становится актуальной проблема разнородности памяти, причём сразу для всех уровней иерархии. Каждый ускоритель обладает своей оперативной памятью и своими кэшами, прямая адресация которых в общем случае невозможна ни для центрального процессора, ни других ускорителей. Поэтому достичь производительности, составляющей хотя бы десятки процентов от пиковой, удаётся лишь для тех задач, алгоритмы решения которых кроме необходимого уровня параллелизма также обладают свойством локальности данных. А точнее, позволяют «нарезать» исходные данные на необходимое количество независимых блоков, время обработки которых будет много больше времени их копирования. В противном же случае программисту часто приходится явно или неявно осуществлять барьерную синхронизацию, в результате чего вычислители большую часть времени простаивают, а достигнутая производительность начинает составлять лишь доли

процентов от пиковой.

Для рассматриваемой предметной области, а именно задач моделирования различных аэрогидродинамических процессов, обе вышеприведённые проблемы являются достаточно актуальными. Однако если необходимый потенциальный параллелизм обычно обеспечивается за счёт постоянного увеличения размера сеток, дискретизирующих расчётную область, то при попытке разбиения данных на необходимое количество блоков возникает ряд трудностей.

Практически все промышленные расчёты производятся на неструктурированных, а иногда даже и адаптивных сетках, поэтому возможности по их «разрезанию» непосредственно в процессе расчётов на необходимое количество блоков крайне ограничены. И если в случае обычных однородных кластеров подобное разбиение области на домены удаётся выполнить за счёт сторонних внешних программ типа MeTiS, то для осуществления аналогичных действий для гетерогенных систем данный подход оказывается малоприменимым — количество необходимых блоков увеличивается на порядок (в идеальном случае, требуется один блок на один мультипроцессор графического ускорителя, или же порядка 35-50 блоков в пересчёте на узел), а на их размеры накладывается ряд достаточно жёстких ограничений. И в результате процесс разбиение всей области на требуемые блоки с целью обеспечения эффективной загрузки всех вычислителей может оказаться более долгим, чем непосредственно сами расчёты. И это даже без учёта необходимости пересылок значений искомых величин на границах блоков, что является отдельной проблемой.

В частности, из-за озвученных проблем GPU версии большинства CFD-пакетов демонстрируют относительно небольшое ускорение, особенно по сравнению с пакетами из других предметных областей типа молекулярной динамики, где выигрыш от перехода на гетерогенные системы может доходить до нескольких сотен раз. Одним из ярких примеров может служить открытый программный комплекс OpenFOAM для проведения моделирования различных гидродинамических процессов, часть решателей которого в рамках проекта SpeedIT была адаптирована для работы на GPU. В соответствии с опубликованными результатами [1] тестирования двух предобуславливателей, полученное ускорение составило от 2.4 до 3.8 раз по сравнению с последовательной версией пакета. Если же учесть, что пиковые производительности ускорителя и одного ядра тестовой системы равнялись 450 и 12 гигафлопсам соответственно (т. е. соотносились как 37.5 к 1), то достигнутое ускорение демонстрирует явно неэффективное использование вычислительных блоков GPU. В частности, опять же из-за озвученных выше проблем.

Если же рассматривать закрытые платные CFD-пакеты, то ситуация оказывается примерно аналогичной. В случае пакета ANSYS конкретное ускорение от переноса вычислений на GPU сильно зависит как от решателя, так и от размера и формата данных. Если ориентироваться на опубликованные результаты от самой компании ANSYS [2], то ускорение составило примерно от 2 до 9 раз относительно последовательной версии при рассмотрении только решателей СЛАУ. В случае оценки ускорения для всего пакета оно становится более скромным — примерно от 1.3 до 4 раз относительно последовательной версии. В другой статье [3], выполненной уже независимыми авторами, проводилось тестирование GPU версии пакета ANSYS при решении собственной задачи, результаты которого оказались ещё менее радужными — ускорение от добавления GPU составило порядка 1.2 — 1.7 раз.

В качестве ещё одного примера можно привести опыт российской компании ООО «Тесис», разрабатывающей собственный CFD-пакет FlowVision. В статье от разработчиков данного решения [4], посвящённой реализации GPU-версии одного алгоритма для решения СЛАУ, описывается поход, позволивший получить ускорение для отдельных этапов более чем в 30 раз относительно одного ядра центрального процессора. К сожалению, в статье не приведены оценки для итогового ускорения, поэтому данные результаты сложно сравнивать с предыдущими.

В следующих разделах данной работы будут приведены результаты частичного портирования на GPU CFD пакета SigmaFlow, разрабатываемого институтом теплофизики СО РАН, которое было осуществлено совместно с компанией ООО «ТТГ Лабс». Предварительные результаты данных работ по портированию отдельных решателей СЛАУ были приведены в статье [5], в соответствии с которыми авторами было достигнуто 8-12 кратное ускорение относительного одного ядра центрального процессора.

2. Описание пакета SigmaFlow

Стоит сразу отметить, что в оптимизируемом пакете SigmaFlow реализованы решатели для различных видов уравнений и систем уравнений, однако GPU версия была разработана лишь для двух модулей, отвечающих за решение эллиптических уравнений и системы уравнений Навье-Стокса, описывающей движение ламинарных безвихревых ньютоновских жидкостей. Однако благодаря модульной структуре пакета, даже к текущей GPU версии могут быть подключены некоторые RANS-модели турбулентности — в определённые моменты (а точнее, после каждой итерации) все необходимые данные будут автоматически перемещены в память центрального процессора, обработаны, а после чего загружены обратно в память GPU. В случае же необходимости, добавление GPU поддержки для соответствующих моделей турбулентности может быть легко осуществлено в рамках дальнейших работ.

Все расчётные области задаются с помощью трёхмерных неструктурированных сеток, в случае GPU версии на которые накладывается дополнительное, но опциональное ограничение — каждый узел сетки имеет не более 6 соседей. Для хранения дискретных величин, определённых на вершинах и дугах построенного таким образом графа, независимо используются два внутренних формата данных. В обоих случаях все вершины и дуги перенумеровываются, после чего дискретизируемые на них величины сохраняется в массивы под соответствующими номерами. Далее, в соответствии с первым форматом данных, вводится вспомогательный массив для описания структуры графа, содержащий для каждой дуги индексы двух соединяемых ею вершин. Таким образом, для получения значения величины, заданной на дуге графа, достаточно найти её индекс в данном вспомогательном массиве, после чего с его помощью адресовать массив, содержащий данную величину. В соответствии со вторым форматом вводятся вспомогательные массивы индексов вершин-соседей, позволяющие для заданного узла сетки найти все другие узлы, с ним соединённые, а также индексы соединяющих дуг.

Подробное описание реализации решателя для уравнений Навье-Стокса, на примере которого далее будет описан процесс портирования, может быть найдено в статье [6], однако для краткого описания схемы решения следует отметить ряд моментов. Так, для аппроксимации конвективных членов уравнений переноса применяется противопоточная TVD схема второго порядка точности. Связь между полями скорости и давления, обеспечивающая выполнение уравнения неразрывности, реализуется при помощи SIMPLE-C процедуры на совмещенных сетках. Для устранения осцилляций поля давления используется подход Рхи-Чоу, заключающийся в специальной интерполяции вектора скорости на грани контрольных объемов. Полученные в результате дискретизации исходной системы дифференциальных уравнений разностные уравнения решаются итерационным способом с применением предобусловленного метода сопряженных невязок.

В исходной версии пакета была реализована возможность запуска расчётов на кластерных системах в соответствии с моделью «один MPI процесс на одно физическое ядро CPU», однако в рамках данных работ по разработке GPU версии осуществляется частичный переход к модели «один MPI процесс на один физический узел». При этом в качестве реализации MPI используется библиотека MPICH2, а для распараллеливания вычислений по узлам кластера производится разбиение всей расчётной области на домены с применением программы MeTiS.

3. Портирование пакета на GPU

3.1. Подготовительные работы

Так как оптимизируемый пакет содержит достаточно много модулей, суммарный размер которых составляет сотни тысяч строк кода, то перед началом непосредственного его переноса на GPU был выполнен ряд подготовительных работ. Во-первых, была встроена разработанная компанией ООО «ТТГ Лабс» небольшая вспомогательная утилита ttgUtils, опционально делающая замеры времени работы отдельных частей пакета и по завершению работы подготавливающая мини-отчёт, и также опционально сохраняющая на диск промежуточные

данные с целью дальнейшей проверки корректности вычислений между различными реализациями одного и того же алгоритма. Таким образом, в зависимости от опций компиляции, получается три сборки пакета — первая по завершению работы подготавливает мини-отчёт, сколько времени ушло на каждый этап вычислений, вторая проверяет, как сильно отличаются на каждой итерации промежуточные данные в CPU и GPU версиях (для чего используется L2 и C нормы), а третья предназначена для проведения расчётов в обычном режиме.

Использование данной утилиты, а не специализированных инструментов типа профайлера Intel Parallel Studio, было обусловлено необходимостью автоматизации тестирования создаваемой GPU версии пакета. К примеру, благодаря встроенной системе замеров времени и небольшого вспомогательного скрипта удалось полностью автоматизировать процесс тестирования сразу на наборе тестовых сеток разного размера, результаты которого группировались в отдельный файл с отчётом. Аналогично, с помощью другого скрипта была реализована функциональность сравнения результатов работы оптимизированной версии с эталонной, причём сравнивались именно промежуточные результаты (точнее, нормы массивов) на выбранных итерациях, а само тестирование, опять же, проводилось сразу на наборе сеток. Так как полный набор тестов занимал порядка нескольких часов, то данная схема в дальнейшем существенно упростила процесс портирования.

Во-вторых, опять же перед началом непосредственных работ по портированию, были изменены контейнеры, используемые для хранения массивов. Весь пакет SigmaFlow разработан с использованием языка C++ и принципов ООП, поэтому все контейнеры были реализованы с помощью шаблонных классов. Так как для работы GPU версии необходимо иметь возможность загружать нужные данные в память графического ускорителя, то было рассмотрено два сценария — модификация исходных контейнеров с целью авто-копирования хранящихся в них данных на GPU и «ручное» копирование массивов в память графического ускорителя непосредственно перед вызовом соответствующих CUDA-ядер. От второго варианта пришлось сразу же отказаться, так как в пакете не было единственного «узкого места», а сами вычисления были равномерно «размазаны» по многим модулям, между которыми вызывались легковесные CPU методы для корректировки данных. Как следствие, данный подход потребовал бы постоянного копирования массивов между CPU и GPU, что вместо ускорения привело бы лишь к замедлению вычислений.

Таким образом, было решено внести изменения в исходные контейнеры, позволившие не только адресовать хранящийся в них массив, но и получать ссылку на актуальную копию данных в памяти GPU. При этом в самом контейнере автоматически выделялся буфер в памяти GPU, запоминалось текущее местоположение данных и при соответствующем вызове незаметно осуществлялось копирование. К сожалению, данная схема, которая является вполне логичной и естественной при разработке программы «с нуля», оказалась крайне неудачной для внедрения в существующий пакет подобного размера, большую часть которого не планировалось модифицировать. Ниже перечислены основные трудности, с которыми пришлось столкнуться при встраивании данной схемы и дальнейшем портировании:

– Переход от контейнеров к указателям и обратно. В некоторых более старых частях пакета (например, отвечающих за подготовку данных и вызываемых лишь один раз при старте программы) работа с массивами осуществлялась через указатели, в результате чего контейнеры теряли информацию о местоположении актуальной версии данных. Как следствие, обновлённые массивы иногда затирались более старыми версиями, сохранёнными в памяти GPU.

– Работа с подмассивами. В одном из решателей обнаружилась не совсем стандартная схема представления данных, в соответствии с которой выделялся блок памяти размера 2*N, который далее «оборачивался» в два независимых контейнера размерностью N, что позволяло из первого массива адресовать элементы второго. Изначально такая возможность предусмотрена не была, в результате чего соответствующие им участки GPU памяти не были размещены последовательно.

– Создание контейнеров, ссылающихся на общий участок памяти. В отличие от предыдущего сценария, здесь используется общий блок памяти, который адресуется из-под разных контейнеров. Соответственно, в начальной версии для каждого из них создавался собственный GPU буфер, что в дальнейшем приводило к ошибкам.

В связи с большим размером оптимизируемого пакета подобные ошибки не удалось обнаружить на этапе проектирования, а что более страшно — они приводили не к аварийной остановке программы, а лишь к более высокой погрешности. В результате их поиск оказался крайне трудоёмким и был успешно осуществлён лишь благодаря подсистеме сравнения промежуточных результатов между CPU и GPU версиям.

3.2. Работы по портированию

В рамках работ по портированию пакета на GPU был осуществлён переход к ядро-ориентированной архитектуре. Соответственно, для каждой функции, в которой осуществляются непосредственные вычисления, было создано одно или несколько тривиальных вычислительных ядер и функция-обёртка, выполняющая базовые проверки, получение указателей на массивы и прочую подготовку, после чего вызывающая соответствующие вычислительные ядра. Причём переключение между реализациями ядер (в данном случае CUDA или исходный CPU код) было унифицировано и осуществлялось при компиляции с помощью макроса. Благодаря подобной структуре (вычислительный код полностью отделён от управляющего) удаётся минимизировать количество дублируемых участков пакета, а также появляется возможность в дальнейшем без дополнительных затрат добавить поддержку технологии OpenCL (и, как следствие, ускорителей от компании AMD), или же альтернативных версий ядер для CPU, написанных с использованием расширений SSE/AVX. Более того, в рамках текущей реализации для некоторых исходных функций были разработаны различные версии CUDA-ядер, отдельно оптимизированные под ускорители с разной архитектурой (в данном случае — CC 1.3 и CC 2.0). Стоит также отметить, что в результате данных модификаций было создано порядка 40 тривиальных CUDA-ядер.

В большинстве случаев разработка CUDA-аналогов для исходных CPU функций являлась сугубо технической работой, однако в некоторых из них при отображении на архитектуру графического ускорителя были обнаружены некоторые трудности. Так, в одной функции требовалось одновременно хранить множество значений вспомогательных переменных, в результате чего созданная компилятором NVCC GPU-подпрограмма завершалась с ошибкой из-за нехватки количества регистров. Наиболее простым вариантом решения данной проблемы является размещение части переменных в разделяемой памяти, однако в общем случае её также оказалось недостаточно. В результате пришлось сделать две версии данного ядра, одна из которых запускается с уменьшенным до 32 нитей размером блока (и, как следствие, с большим объёмом разделяемой памяти на одну нить), а вторая использует возможность архитектуры Fermi увеличить размер разделяемой памяти с 16 до 48 килобайт за счёт уменьшения размера кэша L1.

Другим проблемным моментом, потребовавшим внесения существенных изменений в ряд исходных CPU функций, оказалось активное использование операции +=, которая в случае CUDA-версии приводит к гонке данных, когда несколько разных нитей пытаются обновить один и тот же элемент массива, затирая изменения друг друга. Рекомендуемым решением данной проблемы является переход к аппаратно поддерживаемым атомарным операциям, однако в случае, если ускоритель имеет архитектуру CC 1.1, CC 1.2 или CC 1.3, они являются крайне неэффективными, в результате чего «ускоренная» версия пакета может оказаться намного медленнее исходной последовательной. В случае же более новых архитектур CC 2.0 и CC 2.1 атомарные операции хоть и являются более быстрыми, их использование всё равно на порядок замедляет соответствующее CUDA-ядро. Для решения данной проблемы потребовалось перейти к другому формату данных, благодаря которому удалось полностью избежать проблемной операции +=, и даже на архитектуре поколения Fermi получить 3-кратное ускорение для данного ядра.

Ещё одной проблемой, которая возникла при портировании двух ядер, используемых в предобуславливателе D-ILU, оказался специфический доступ к памяти, не позволяющий распараллелить исходную функцию. Фактически, для обработки i-того элемента массива требовалось обработать все элементы с индексами от 0 до i-1, что легко осуществимо в последовательной версии, но никак не реализуемо в рамках SIMT модели, используемой в программах для графических ускорителей. В результате работ был разработан аналог данного

предобуславливателя, который позволяет достичь схожих результатов, но не прямым, а итерационным путём, и для которого имеется возможность эффективного отображения на архитектуру графического ускорителя. В результате в GPU версии пакета SigmaFlow осуществляется несколько типов итераций — внешние для решения исходного СЛАУ и внутренние для применения предобуславливателя, выполняемые в рамках каждой внешней итерации. Соответственно, возникает вопрос — сколько требуется внутренних итераций, и как их количество влияет на скорость решения исходного СЛАУ? Частично ответ на данный вопрос был дан в рамках предыдущей статьи [5], в соответствии с которой достаточно трёх внутренних итераций. Однако в результате тестирования на дополнительных сетках выяснилось, что в ряде случаев этого недостаточно для обеспечения аналогичной скорости сходимости, поэтому в текущей версии их количество было повышено с 3 до 7-15 в зависимости от модификации исходного предобуславливателя. Как следствие, в разы возрос объём вычислений, однако при этом промежуточные данные на внешних итерациях в CPU и GPU версии пакета отличаются не более чем на 0.5% по норме C.

Резюмируя, стоит отдельно ещё раз отметить, что благодаря всем перечисленным модификациям удалось сохранить полную совместимость со всеми модулями пакета на уровне структур данных и сигнатур функций, при этом избежав необходимости копирования данных между CPU и GPU в основном цикле решения уравнения, время выполнения которого составляет более 99% от времени работы всей программы.

4. Оптимизация созданной GPU версии пакета

После завершения базового портирования пакета SigmaFlow на GPU итоговое ускорение на системе с центральным процессором Intel Xeon E3-1230 и ускорителем NVidia GeForce 580GTX при вычислениях на сетке из 512 тыс. узлов составило порядка 5 раз относительно исходной версии, выполняемой на одном ядре. В абсолютных значениях время работы соответствовало примерно 4200 и 840 секундам. Так как для решателей СЛАУ, являющихся в CPU версии наиболее вычислительноёмким этапом, было достигнуто 12-кратное ускорение, то было решено проводить дополнительную оптимизацию, основные направления которой рассмотрены ниже.

При более детальном анализе программы с помощью профайлера NVidia Nsight было установлено, что время работы CUDA-ядер составляет порядка 480 секунд, а в течение же остальных 360 секунд программа выполняет различные вспомогательные операции, никак не связанные с непосредственным вычислениями. Одним из таких мест оказалось выделение и освобождение памяти для временных GPU буферов, выполняемое на каждой итерации. Так как, фактически, осуществлялось 20 000 итераций, то суммарно на вызовы функции cudaMalloc() тратилось 160 секунд. В пересчёте же на одну итерацию затрачиваемое время получалось равным всего 8 микросекундам, в связи с чем при портировании данная операция казалась крайне незначительной. После введения механизма переиспользования всех выделяемых буферов от этих 160 секунд удалось полностью избавиться.

Другой достаточно «долгой» операцией было обнуление данных контейнеров, на которое тратилось порядка 80 секунд. Так как после внесённых изменений контейнеры стали содержать две копии данных (в памяти CPU и GPU), то их обнуление проводилось сразу во всех буферах. Как следствие, затраты на относительно меленный библиотечный вызов memset(), результаты которого фактически игнорировались, и составили те самые 80 секунд. Другой библиотечной операцией, внёсший свой существенный вклад в общее время работы программы, оказался текстовый вывод в консоль, осуществляемый с помощью оператора перенаправления потока языка C++. На весь вывод тратилось порядка 30 секунд, что составляло менее процента в исходной CPU версии пакета, но уже порядка 5% в оптимизированной. После замены вывода через потоки на вызов printf() и отказа от операции flush() после каждой записи затрачиваемое время удалось сократить в 3 раза (примерно до 10 секунд).

Наконец, последними функциями, непосредственно не связанными с вычислениями, но суммарное время выполнения которых было достаточно большим, оказались реализации скалярного произведения и редукции. В начальном варианте в них производилось несколько шагов редукции на графическом ускорители, после чего существенно уменьшенный массив копировался в память центрального процессора и окончательно редуцировался до числа.

Собственно, данный процесс копирования даже уменьшенного в несколько сотен раз массива суммарно занимал порядка 70-80 секунд. Поэтому потребовалось более точно определять необходимое число шагов редукции на GPU, в результате чего в обновлённой версии на центральный процессор копировался уже массив не более чем из 256 элементов, время на обработку которого стало действительно незначительным.

Следующим этапом была оптимизация непосредственно CUDA-ядер, которая свелась к трём действиям. Во-первых, в достаточно редко вызываемом ядре для вычисления градиента используемые атомарные операции были заменены на индексацию через вспомогательные массивы, что позволило сэкономить порядка 30 секунд. Далее, в ядре, отвечающем за проведение внутренних итераций при применении предобуславливателя, часть косвенно адресуемых коэффициентов была заранее сохранена во вспомогательный буфер, благодаря чему удалось отказаться от одного обращения с невыравненным паттерном доступа в память графического ускорителя и тем самым уменьшить время расчётов ещё на 20 секунд.

Таким образом, благодаря проведённым модификациям время работы GPU версии пакета на тестовой сетке сократилось с 840 до 440 секунд. Для проведения дальнейшей оптимизации были начаты работы по встраиванию разработанного компанией ООО «ТТГ Лабс» инструментария TTG Apptimizer, позволяющего динамически подстраивать параметры вычислительных ядер под связку «целевая система + обрабатываемые данные». В случае пакета SigmaFlow данными параметрами являлись размер блока нитей, которые будут выполняться на одном мультипроцессоре, а также, при возможности, количество узлов сетки, обрабатываемых одной нитью.

В связи с достаточно сложной архитектурой графических ускорителей задача обеспечения максимально эффективной загрузки все вычислительных блоков, в данном случае сводящаяся к подбору двух перечисленных параметров, является крайне нетривиальной и требующей учёта множество факторов, таких как размер входных данных, необходимые CUDA-ядру ресурсы, а также особенности модели ускорителя. С помощью же инструментария TTG Apptimizer данный процесс удалось полностью автоматизировать — на первых 100 итерациях данный инструмент для каждого оптимизируемого CUDA-ядра производит обучение, что выражается во временном понижении производительности, после чего подбирает параметры, оптимальные для данного конкретного случая, тем самым повышая итоговую скорость расчётов.

5. Оценка полученного ускорения

В качестве тестовой задачи, на которой производилась оценка достигнутого ускорения, была выбрана традиционная задача о стационарном ламинарном течении в пространственной каверне с подвижной верхней крышкой. Для расчетов использовалась неравномерная структурированная сетка, представляющая собой куб, количество узлов которого зависело от типа теста. При этом координатные линии были сгущены к поверхностях, а число Рейнольдса, посчитанное по скорости движения крышки и линейному размеру каверны, составляло 1000. Характерная картина течения в каверне приведена на рис.1.

Для оценки ускорения все тесты были разделены на две независимые группы. В первой производилось сравнение GPU версии пакета относительно MPI версии, запускаемой на одном ядре, на всех ядрах одного процессора и на кластере при выполнении расчётов на сетке из 4 миллионов узлов. Вторая группа содержит сравнение достигнутого ускорения относительно одного ядра центрального процессора, но на сетках различного размера.

Первая группа тестов выполнялась на обычном персональном компьютере, оснащённом графическим укорителем NVidia GeForce 580 GTX (пиковая производительность которого составляет порядка 1.5 терафлопс) и четырёхядерным центральным процессором Intel Core i7 2600k, частота которого была повышена до 4.4 гигагерц (в результате чего его пиковая производительность равнялась 140 гигафлопсам). В качестве кластера использовалась система из 12 узлов IBM Blade HS21, каждый узел которой включает в себя два 4-х ядерных процессора Intel Xeon, работающих на частоте 2.33 ГГц (пиковая производительность составляла 895 гигафлопс).

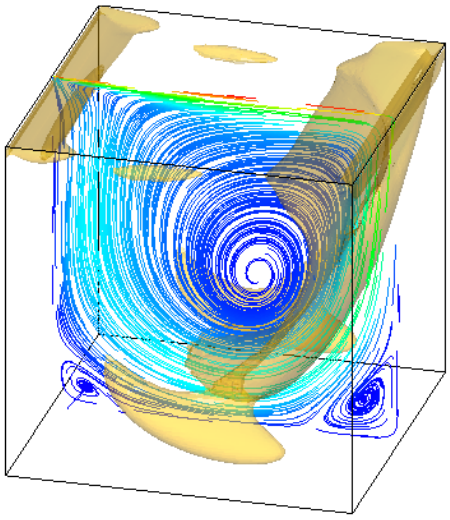

Рис.1. Траектории частиц жидкости и изоповерхность Q–критерия для течения в каверне.

Результаты тестирования описанной ранее задачи на всех целевых система приведены в следующей диаграмме, демонстрирующей ускорение относительно одного ядра процессора Intel Core i7.

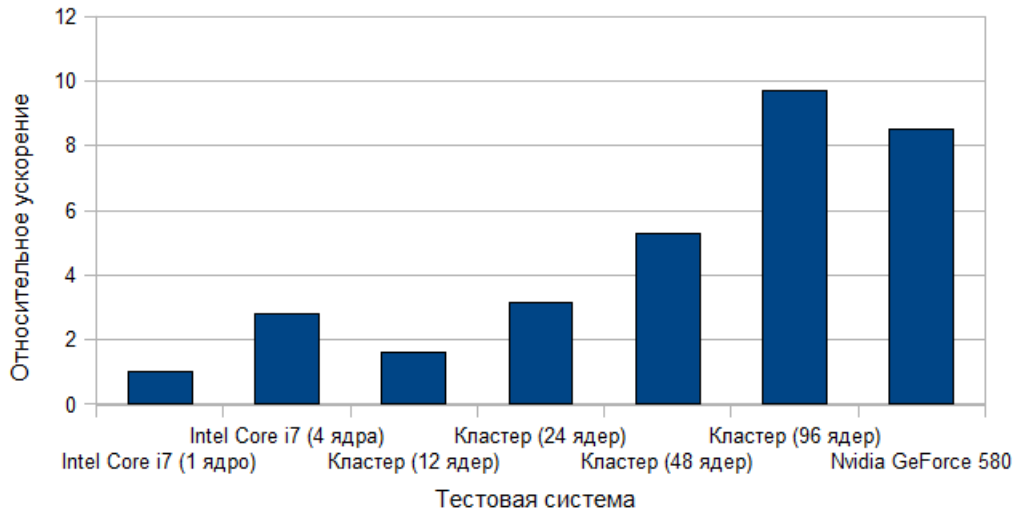

Рис.2. Ускорение относительно одного ядра процессора Intel Core i7 на сетке из 4 млн. узлов.

Полученные результаты оказались достаточно неоднозначными. С одной стороны, GPU версия пакета обеспечивает примерно такое же ускорение, как и кластер из 12 узлов (8.5 раз против 9.7). Однако если взять всего один современный центральный процессор с повышенной тактовой частотой и задействовать все его четыре ядра, то по сравнению с ним выигрыш от переноса вычислений на GPU становится более скромным и уже составляет ровно 3 раза. Таким образом, сравнение полученных ускорений с известными результатами портирования на графические ускорители CFD пакетов оказывается достаточно проблематичным, так как в зависимости от выбора системы заявляемое ускорение может изменяться в разы.

Для проведения тестов из второй группы использовался сервер с одним четырёхядерным центральным процессором Intel Xeon E3-1230 (порядка 100 пиковых гигафлопс) и, опять же, графическим ускорителем NVidia GeForce 580 GTX (обладающего, напомним, пиковой производительностью в 1.5 терафлопс). Приведённые ниже результаты первого теста из данной группы демонстрируют относительную производительность GPU версии пакета в сравнении с исходной версией, запущенной на одном ядре центрального процессора.

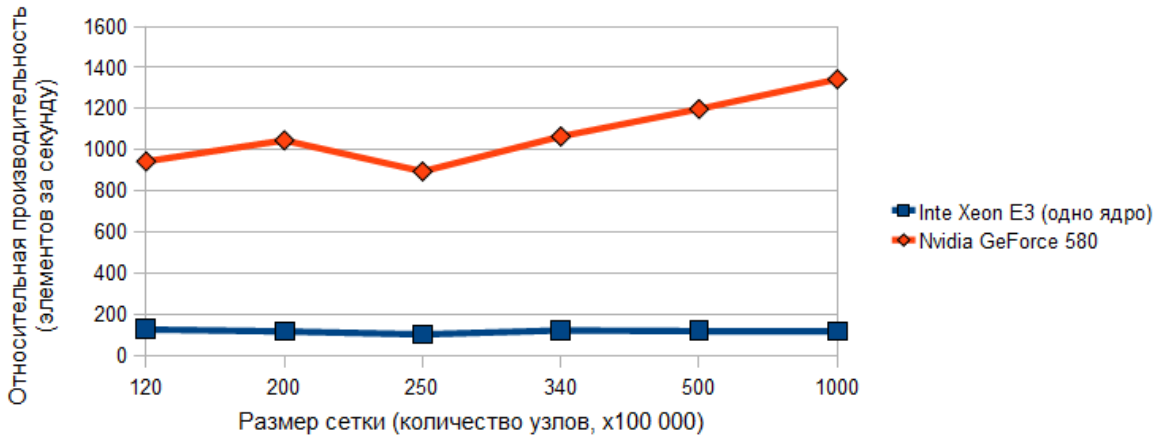

Рис.3. Производительность версий пакета в зависимости от размера сеток.

Согласно данным результатам, с ростом размера сетки производительность GPU также повышается, что можно объяснить большим количеством вычислений, благодаря которым уменьшается влияние достаточно медленных операций доступа к глобальной памяти. При этом ускорение относительно одного ядра центрального процессора в зависимости от размера сетки составило от 7.4 до 11.3 раз.

Другим достаточно интересным тестом было сравнение вклада в суммарное время вычислений различных этапов расчётов в зависимости от размера сетки. В данном случае для GPU версии пакета сравнивались суммарные времена трёх относительно независимых операций — (1) вычисления скалярного произведения, (2) вычисления градиента и (3) применения предобуславливателя. Результаты данного теста приведены на следующем графике.

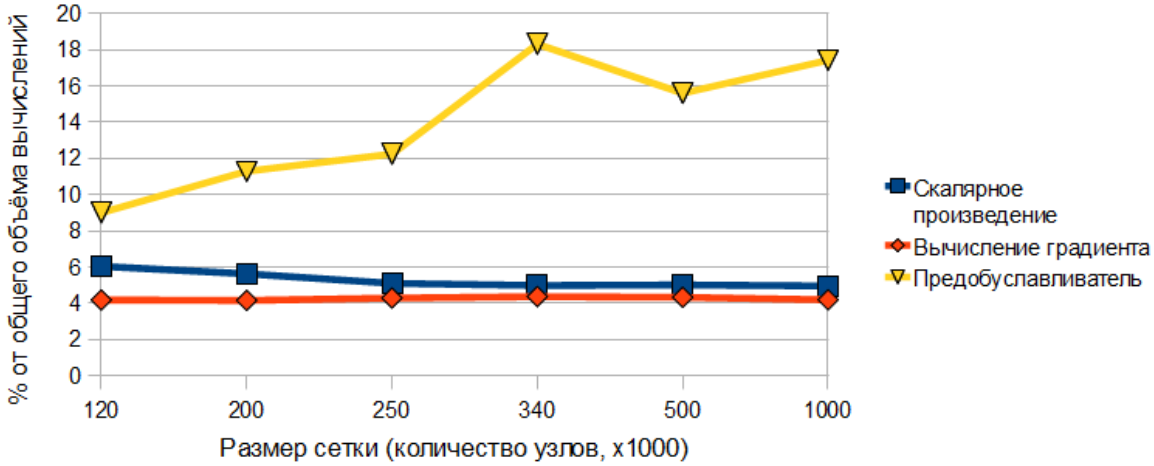

Рис.4. Вклад некоторых этапов в общее время вычислений в зависимости от размера сеток.

Как оказалось, доля времени выполнения некоторых операций типа вычисления градиента практически инвариантна и не зависит от размера сеток. С другой стороны, более сложные ядра типа предобуславливателя D-ILU относительно быстрее выполняются на небольших сетках.

Наконец, последним тестом была оценка разности промежуточных результатов из CPU и GPU версий пакетов в зависимости от номера итерации при решении СЛАУ. Так как из-за ряда особенностей, а именно таких как накопление погрешности при выполнении атомарных операций, изменение порядка суммирования при редукции массивов, различия в алгоритмах вычисления ряда функций тип синуса, результаты вычислений на CPU и GPU всегда немного отличаются. В данном же случае для предобуславливателя была разработана альтернативная реализация, что ещё больше усилило данные отличия. Результаты, представленные в следующем графике, были получены путём сравнения L2-норм для соответствующих массивов с промежуточными результатами.

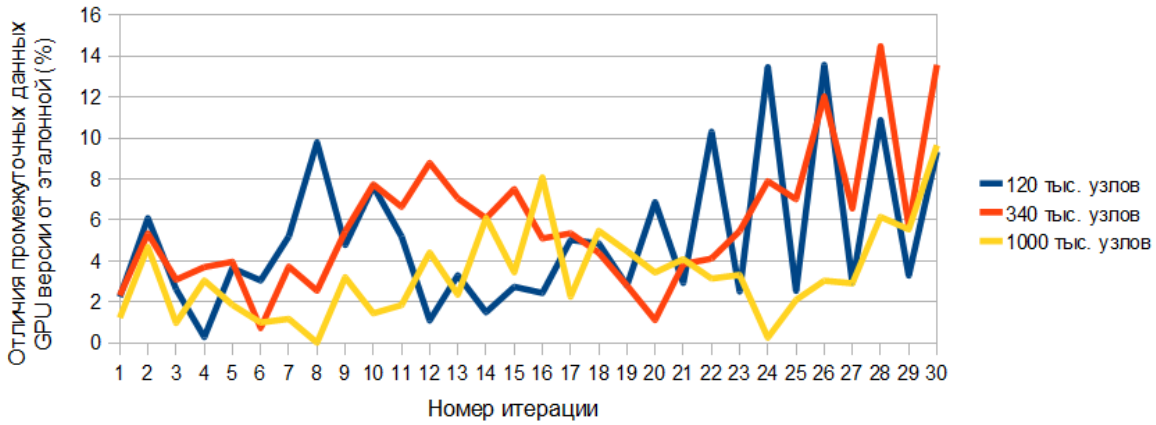

Рис.5. Отличия между промежуточными результатами в CPU и GPU версиях.

Стоит отметить, что в результате работы CPU и GPU версий пакета были получены практически идентичные решения, поэтому данный график больше иллюстрирует скорость сходимости, которая фактически оказывается также идентичной — на всех итерациях вплоть до конца вычислений данные отличия составляют не более 30%.

6. Заключение

В результате данных работ была создана GPU версия пакета SigmaFlow, предназначенная для моделирования ламинарных течений ньютоновских жидкостей. Были описаны проблемы, с которыми авторы столкнулись при осуществлении портирования, а также рассмотрены подходы для повышения производительности созданной GPU версии пакета. Согласно проведённому тестированию, было зафиксировано ускорение от 7.4 до 11.3 раз в зависимости от размера сетки относительно исходной версии, выполняемой одним ядром центрального процессора.

В рамках дальнейших работ планируется добавление в разработанную версию пакета поддержки систем с несколькими ускорителями, осуществление портирования ряда модулей, реализующих требуемые RANS-модели турбулентности, а также проведение дополнительных работ по оптимизации пакета с целью доведения итогового ускорения до 15 раз.

Литература

1. Acceleration of OpenFOAM with SpeedIT 2.1, Comparision of GAMG and DIC preconditioners URL: http://vratis.com/blog/?p=119 (дата обращения 01.12.2012)
2. Beisheim J., Speed Up Simulations with a GPU // Ansys Advantage, Vol. 4, Issue 2, 2010
3. Газизов Р.К., Касаткин А.А., Юлдашев А.В., Исследование ускорения решения термо-структурной задачи в пакете ANSYS средствами GPU // Труды конференции «Применение гибридных высокопроизводительных вычислительных систем для решения научных и инженерных задач», Нижний Новгород, 2011, с. 38 — 41.
4. Коньшин И.Н., Сушко Г.Б., Харченко С.А., Трёхуровневая MPI+TBB+CUDA параллельная реализация блочного итерационного алгоритма решения СЛАУ для мелкоблочных неструктурированных разреженных матриц // Научный сервис в сети Интернет: поиск новых решений: Труды Международной суперкомпьютерной конференции, Москва, Издательство МГУ, 2012, с. 522 — 528.
5. Кривов М.А., Гризан С.А., Опыт разработки гибридных версий версий решателей разреженных СЛАУ // Сборник трудов конференции «Параллельные вычислительные технологии 2012», Новосибирск, 2012.
6. Гаврилов А.А., Минаков А.В., Дектерев А.А., Рудяк В.Я. Численный алгоритм для моделирования ламинарных течений в кольцевом канале с эксцентриситетом // СибЖИМ. – 2010. – Т. 13, №4(44). – С. 46–61.