Опыт разработки гибридных версий решателей разреженных СЛАУ

М.А. Кривов, С.А. Гризан

ООО «ТТГ Лабс»

В статье описан процесс портирования на графические ускорители двух популярных алгоритмов решения СЛАУ с разреженными матрицами, получаемыми при численном решении ряда уравнений аэродинамики на неструктурированных сетках. Рассмотрены подходы к оптимизации кода, заключающиеся в использовании различных вариантов распараллеливания и представления данных. Приведены результаты тестирования CUDA-версий решателей, а также дана оценка потери производительности при переходе от структурированных сеток к неструктурированным.

1. Введение

С появлением графических ускорителей, имеющих достаточно специфическую архитектуру и накладывающих ряд ограничений на реализуемый алгоритм, разработчики из многих предметных областей оказались перед непростым выбором — применить алгоритмически оптимальный метод и довольствоваться десятыми или сотыми процента от пиковой производительности, или же взять более простой и медленный алгоритм, который на графических ускорителях позволит достичь 10-30% пиковой производительности. Очевидно, что в первом случае вычисления можно ускорить, например, путем использования более высокого порядка аппроксимации, тогда как второй путь обеспечивает гарантированно высокое ускорение от эффективного переноса программы на графические ускорители.

Ещё одной дилеммой может оказаться выбор точности вычислений — даже на последних версиях ускорителей NVIDIA Tesla C2050 переход от одинарной к двойной точности способен заметно уменьшить итоговую производительность. В статье [1] было показано, что при перемножении матриц с помощью популярных GPU-версий BLAS переход от одинарной к двойной точности снизил эффективность использования вычислительных блоков ускорителя с 40% до 10%, что можно интерпретировать как замедление работы программы в восемь раз. Поэтому при использовании, например, абсолютно устойчивых итерационных методов, возможно, стоит ограничиться одинарной точностью вычислений, увеличив число требуемых для сходимости итераций, но при этом уменьшив суммарное время работы программы за счёт более эффективного использования вычислительных блоков GPU.

Озвученные проблемы особенно заметны при использовании сеточных разностных методов для решения ряда уравнений аэродинамики. Так, в зависимости от выбранной разностной схемы алгоритм решения итогового СЛАУ может оказаться не подходящим для использования на GPU, что не только приведет к дополнительным временным затратам на создание его эффективной реализации, но и может потребовать внесения ряда изменений в саму математическую модель.

В данной статье описываются ещё не завершённые работы по портированию существующих численных решателей для уравнения Лапласса, разработанных в рамках пакета SigmaFlow [2]. Целью работ стал поиск ответа на сформулированные выше вопросы, а также портирование существующих решателей на графические ускорители и их последующая оптимизация.

2. Переход к неструктурированным сеткам

При портировании упомянутых алгоритмов на графические ускорители ключевую роль играет выбор оптимальной модели представления данных. Использование

неструктурированных сеток позволяет существенно уменьшить количество узлов на целевой модели, тем самым сократив время вычислений. С другой стороны, при использовании структурированных сеток разработчик получает априорную информацию о расположении соседних узлов, что в дальнейшем позволяет выбрать подходящий формат массивов, а также препросчитать ряд констант.

Для выбора модели представления данных авторами была разработана небольшая тестовая программа, решающая трёхмерное уравнение Лапласа методом Якоби на структурированной и неструктурированной сетках. В обоих случаях использовалось равномерное разбиение параллелепипеда, однако в первом варианте данные представлялись как трёхмерный массив, а во втором – как некоторый граф, каждой вершине которого соответствует значение моделируемой характеристики в заданном узле, а ребру — связь между соседними узлами. Для неструктурированной сетки проводилось два типа тестов — оценка производительности при использовании «упорядоченного» графа, в котором все соседние вершины некоторого узла по возможности располагаются подряд в используемом для их хранения массиве, и «перемешанного» графа, где все вершины произвольным образом размешаны в соответствующем массиве. В соответствии с этим были разработаны версии данной программы для центрального и графического процессоров с использованием технологий OpenMP и NVIDIA CUDA.

Отдельно стоит остановиться на видах оптимизации, применявшихся при разработке GPU-версии программы. Так, при обсчёте с использованием структурированной сетки активно применялась разделяемая память. Легко заметить, что при использовании явной разностной схемы для вычисления значения в одной точке требуется семь обращений к глобальной памяти. Благодаря тому, что все вершины расположены в одном массиве, а для обращения к соседнему узлу требуется обратиться к элементу по некоторому заранее известному адресу, становится возможным загрузить в разделяемую память сразу некоторую подобласть, тем самым снизив количество обращений к глобальной памяти до одного, что значительно повышает скорость вычислений. Другим приёмом стало разбиение основного вычислительного ядра на два мини-ядра. Первое производит вычисления внутри каждой подобласти, помещаемой в разделяемую память, а второе — только на границе таких подобластей. Указанное разбиение позволило достичь выровненного доступа к памяти (в терминологии CUDA именуемого coalesced [3]), а также отказаться от трёх условных операторов, что, к примеру, на сетке размером 256x256x64 повысило скорость вычислений на 40-50%.

В случае неструктурированной сетки возможность использования разделяемой памяти фактически отсутствует, так как для получения индекса соседней вершины требуется прочитать значение из вспомогательного массива индексов, что делает выделение подобластей невозможным. Поэтому пришлось задействовать такие альтернативные механизмы, как текстурная память и L1-кэш глобальной памяти. В первом случае адресация массива со значениями происходила через специальные текстурные блоки графического ускорителя, кэширующие сразу некоторый участок памяти. Во втором предполагалось автоматическое использование кэша L1, который появился в архитектуре Fermi и который, по заявлению компании NVIDIA, позволяет достичь лучших результатов, чем при использовании текстурной памяти.

Результаты тестирования описанных выше программ на системе с ускорителем NVIDIA Tesla C2050 и процессором Intel Core 2 Quad приведены в Табл. 1 (все значения в гигафлопсах).

Таблица 1. Результаты тестирования на сетках различной структуры.

Сетка	Структурированная		Неструктурированная (упорядоченная)			Неструктурированная (перемешанная)		
	GPU	CPU	GPU (L1 кэш)	GPU (textures)	CPU	GPU (L1 кэш)	GPU (textures)	CPU
64x64x16	0,31	1,03	16,13	14,89	0,69	15,52	15,65	0,63
128x128x32	28,28	1,46	20,88	20,59	0,35	20,8	20,53	0,4
192x192x48	46,34	1,56	20,69	20,77	0,53	20,72	20,71	0,69
256x256x64	50,77	1,54	20,69	21,06	0,32	20,97	20,94	0,64
320x320x80	51,3	1,53	21	21,13	0,55	20,9	20,99	0,67

Подводя итоги, стоит отметить, что переход к неструктурированным сеткам вызвал потерю производительности всего в 2,5 раза, которая в дальнейшем может быть легко скомпенсирована уменьшением объёма вычислений посредством укрупнения ряда областей. Стоит отметить также, что графический ускоритель в данной задаче оказался в 20-30 раза быстрее четырёхядерного процессора, на котором были задействованы все ядра, причём независимо от используемой модели представления данных.

Ещё одним интересным результатом оказалось подтверждение гипотезы о том, что на архитектуре Fermi кэш L1 позволяет полностью заменить текстурную память. Оба названных механизма показали одинаковую производительность как на «упорядоченном», так и на «перемешенном» графах. Однако данные результаты верны лишь для ускорителя Tesla C2050: при попытке провести аналогичное сравнение на более старой карте Tesla C1060, в которой кэш L1 отсутствует, было зафиксировано 6-кратное падение производительности.

3. Вариационный решатель

Суть всех алгоритмов, рассматриваемых в данной статье, заключается в численном решении эллиптического уравнения в постановке задачи Дирихле в некоторой трёхмерной области, сетка на которой задана в виде графа, каждый из узлов которого имеет не более N соседей. Идея первого из оптимизируемых алгоритмов состоит в итерационном построении решения путём минимизации вектора сопряжённой невязки. С этой целью на каждой итерации строится и решается соответствующая сопряжённая система с использованием факторизации Якоби, а затем производится обновление приближенного значения искомой величины (в рассматриваемом случае – потенциала) [4].

В исходной реализации данного алгоритма используется весьма нестандартный формат представления сеток. Все вершины хранятся в памяти как один одномерный массив, содержащий значения искомой величины в соответствующих узлах, а для задания связей между вершинами служат два вспомогательных массива, длина которых равна числу рёбер в графе, и в которых содержатся индексы начальной и конечной вершин, соответственно. Таким образом, при проведении одной итерации алгоритма необходимо обойти все рёбра графа, внося изменения в соответствующие массивы вершин по заданным на дугах индексам. Очевидно, что данная реализация не может быть распараллелена, так как при независимой параллельной обработке нескольких рёбер будут возникать конфликты при адресации одних и тех же вершин, известные под названием data race.

При портировании данного алгоритма на графические ускорители были рассмотрено два варианта решения отмеченной проблемы. Первый заключается в использовании такой аппаратной возможности GPU, как атомарные операции. Данный механизм появился в архитектуре 1.1 и позволяет провести некую арифметическую операцию над произвольной переменной в глобальной памяти, гарантируя её атомарность и, как следствие, отсутствие влияния сторонних потоков на результат. К сожалению, данный механизм можно использовать не для всех типов переменных. Так, в архитектуре 2.0 (известной как Fermi) атомарные операции могут быть выполнены для чисел с одинарной и двойной точностью, в то время как в архитектуре 1.3 – только для чисел с одинарной точностью. Результатом применения атомарных операций стала разработка двух реализаций алгоритма, первая из которых оптимизирована под ускоритель NVIDIA Tesla C2050, имеющего архитектуру 2.0, а вторая — под более старую модель NVIDIA Tesla C1060 с архитектурой 1.3 и поэтому не поддерживающий атомарные операции с двойной точностью. Стоит отметить, что в последнем случае поддержка double-чисел всё-таки была добавлена путём замены недостающей команды циклом попыток записи результата до изменения исходной переменной другим потоком.

Альтернативным решением проблемы распараллеливания, очевидно, является выбор другого представления данных. Например, если к каждой вершине добавить в некотором виде список всех её соседей, то потребность в использовании массивов с рёбрами, а, следовательно, и атомарных операций отпадет. Однако здесь возникают два новых препятствия: 1) в исходной реализации используется довольно много массивов со вспомогательными коэффициентами, которые адаптированы под исходный формат данных и которые весьма трудно переделать под

новый, и (2) в общем случае у одной вершины может быть произвольное количество соседей, что заметно усложняет эффективное распараллеливание алгоритма под GPU из-за сильного ветвления. Для обхода указанных трудностей авторами была разработана специальная подсистема кэширования, которая для заданной сетки «на лету» создавала вспомогательные массивы с индексами, расположенные исключительно в памяти GPU и используемые в параллельной версии алгоритма. Поскольку данные преобразования выполнялись только на первой итерации алгоритма, а число итераций обычно оказывалось довольно большим, влияние динамического изменения формата данных на итоговую производительность не превысило одного процента.

На рис. 1 приведены результаты сравнительного тестирования двух описанных подходов на разных стеках при использовании чисел с одинарной точностью и GPU с архитектурой 2.0.

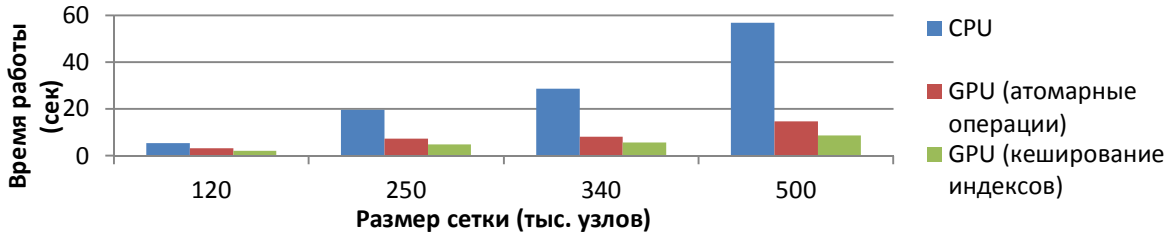

Рис. 1. Сравнение времени работы различных методов распараллеливания под GPU.

Легко заметить, что хотя при кэшировании индексов количество операций и выросло (из-за необходимости обхода дополнительных массивов), суммарная скорость работы увеличилась на 45-65% в зависимости от размера сеток. Поэтому при финальном тестировании ускоренной версии алгоритма, результаты которого приведены на рис. 2 и 3, использовался лишь этот вариант.

Оценка достигнутого ускорения проводилась на системе на базе процессора Intel Xeon E3 с пиковой производительностью 100 GFlops и ускорителя NVIDIA GeForce 580 GTX, обладающего пиковой производительностью 1600 GFlops.

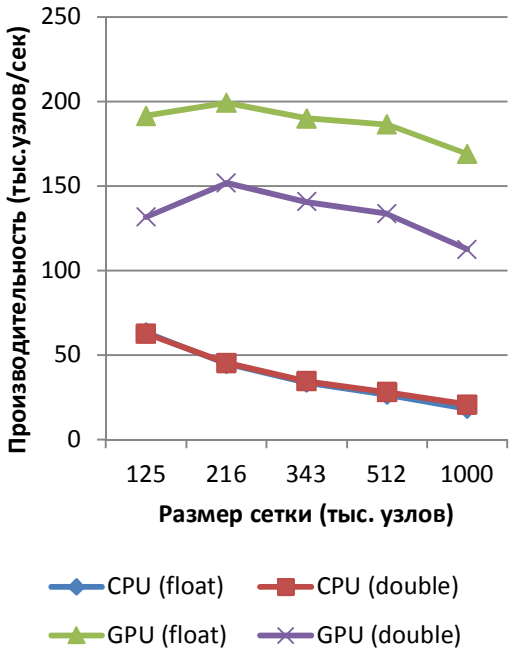

Рис. 2. Скорость вычислений для вариационного решателя.

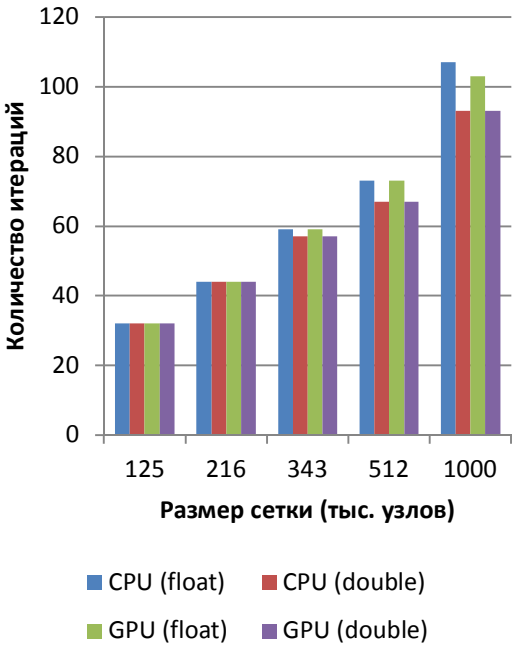

Рис. 3. Скорость сходимости для вариационного решателя.

На рис. 2 проиллюстрирована полезная производительность в пересчёте на количество узлов сетки, обработанных за секунду. На втором — число итераций, потребовавшихся для достижения заданной точности.

Поскольку узким местом данного алгоритма является пропускная способность памяти графического ускорителя, а не нехватка вычислительных ресурсов, то потеря в производительности при переходе от одинарной к двойной точности составила около 35%. В общем случае ускорение GPU-версии относительно CPU было 7-9-кратным, если учитывать пересылки данных, и 10-14-кратным при рассмотрении только времени непосредственных вычислений. Стоит отметить, что в настоящий момент ведутся работы по внесению изменений в архитектуру пакета, в результате чего от ряда пересылок данных удастся отказаться, тем самым приблизив итоговое ускорение к уровню 10-15 раз. Важно подчеркнуть, что все оценки проводились относительно последовательной версии программы, использующей только одно ядро процессора. Авторы считают такое сравнение вполне корректным, так как при разработке параллельной версии программы для CPU потребуется решать аналогичные проблемы синхронизации потоков, в результате чего эффект от многоядерности может оказаться относительно небольшим. Более того, в ряде работ (см., например, [5]) используется именно подобное сравнение, что позволяет сопоставить полученные результаты с уже известными попытками портирования подобных алгоритмов на GPU.

Последним моментом, на котором стоит акцентировать внимание, является более высокая точность float-вычислений на графическом ускорителе. На сетке из одного миллиона узлов GPU-версии потребовалось на четыре итерации меньше, чем исходной реализации для центрального процессора. Это легко объяснить изменением порядка суммирования, в результате чего при вычислении скалярного произведения массивов погрешность накапливается значительно медленнее. Подобное небольшое повышение точности наблюдается довольно часто и является своеобразным «бонусом» от портирования приложений на GPU.

4. Решатель D-ILU

В другом решателе реализован метод разложения матрицы системы на суперпозицию трёх вспомогательных матриц, что в дальнейшем упрощает процесс итерационного построения точного решения [6]. Данный метод активно применяется в популярном пакете OpenFOAM, в результате чего он и стал известен под сокращённым названием D-ILU.

В отличие от рассмотренного ранее вариационного решателя, в данной реализации изначально используется более удобный формат представления данных, в результате чего проблемы обеспечения атомарности операций отсутствовали. Зато трудности возникли при портировании на GPU алгоритма обращения одной из вспомогательных матриц. В изначальной реализации для этого служит паттерн вычислений, известный под названием scan. Он сводится к последовательному обходу массива, для обработки i-го элемента которого требуются результаты обработки всех предыдущих элементов с индексами i-1, i-2, ..., 1, 0. Очевидно, что данный алгоритм в принципе не подходит для переноса на графический ускоритель, поэтому пришлось его заменить итерационным аналогом, который строит приближённое значение искомой обратной матрицы. В результате получилась достаточно интересная реализация, в которой внешние итерации служат для построения приближённого решения искомой величины и, в свою очередь, содержат внутренние итерации для инвертирования вспомогательной матрицы.

Подобная реализация требует не только большего объема вычислений, но и нахождения того порога точности при построении вспомогательной матрицы, который обеспечит достаточную скорость сходимости внешних итераций и минимальное суммарное время работы всего решателя. Путем ряда тестов удалось определить, что 3-4 внутренние итерации гарантированно обеспечивают сходимость для всех тестовых данных, и при этом соответствующая реализация имеет максимальную производительность.

На рис. 4 и 5 приведены результаты тестирования ускоренной версии решателя D-ILU на той же вычислительной системе.

В отличие от вариационного решателя, скорость работы алгоритма D-ILU с увеличением размера сеток заметно понижается при проведении вычислений как на графическом ускорителе, так и на центральном процессоре. Итоговое ускорение от портирования оказалось 4-7-кратным, что объясняется значительно большим объёмом вычислений, обусловленным введением внутренних итераций. Здесь снова следует подчеркнуть, что после проведения ряда работ по

оптимизации пересылок данных между центральным и графическим ускорителем скорость работы GPU-версии удастся повысить более чем на 50%, сделав ускорение 6-12-кратным.

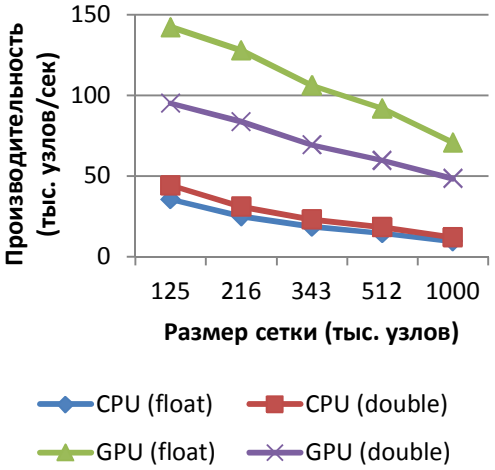

Рис. 4. Скорость вычислений для решателя D-ILU.

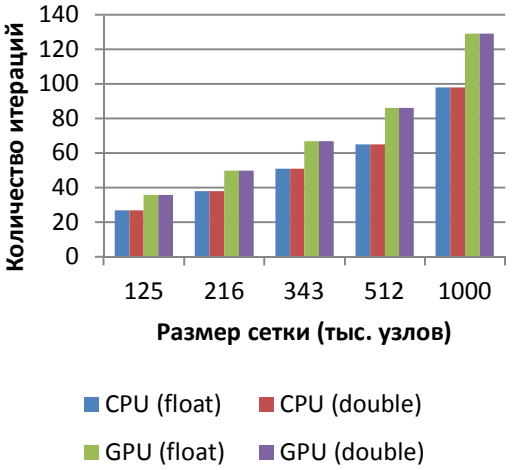

Рис. 5. Скорость сходимости для решателя D-ILU.

Возвращаясь к проблеме обращения вспомогательной матрицы, стоит отметить, что в предложенном варианте для достижения заданной точности потребовалось на 30% больше внешних итераций, что также негативно повлияло на итоговую производительность.

5. Заключение

В работе представлены промежуточные результаты портирования на графические ускорители двух решателей эллиптического уравнения на неструктурированных сетках. Рассмотрены проблемы, которые возникли при разработке параллельных версий соответствующих алгоритмов, а также проведено тестирование созданных реализаций на разных сетках и при вычислениях с двойной и одинарной точностью.

Следующим этапом данных работ является полное портирование на графические ускорители всего пакета SigmaFlow и проведение дальнейшей оптимизации отдельных решателей, а также добавление поддержки систем с несколькими GPU.

Литература

1. Кривов М.А., Казеннов А.М. Сравнение вычислительных возможностей графических ускорителей при решении различных классов задач // Труды Всероссийской научно-практической конференции "Применение гибридных высокопроизводительных вычислительных систем для решения научных и инженерных задач". Н.Н., 2011, с. 18-24.
2. Использование программы SigmaFlow для численного исследования технологических объектов / А.А. Дектерев, А.А. Гаврилов, Е.Б. Харламов, К.Ю. Литвинцев // Вычислительные технологии. 2003. Т. 8. Ч. 1. С. 250–255.
3. CUDA Programming guide: официальный сайт / NVIDIA Corporation - Santa Clara, 2009.
4. Жуков В.Т., Феодоритова О.Б., Янг Д.П. Итерационные алгоритмы для схем конечных элементов высокого порядка. / Математическое моделирование, т. 16, N 7, 2004.
5. M.J. Mawson. Designing numerical solvers for next generation high performance computing / University of Manchester, 2010.
6. T. Behrens. OpenFOAM's basic solvers for linear systems / Technical U. of Denmark, 2009.