ОПТИМИЗАЦИЯ ПРЯМЫХ РЕШАТЕЛЕЙ СЛАУ С ПОМОЩЬЮ ГРАФИЧЕСКИХ УСКОРИТЕЛЕЙ

В.Л. Якушев, А.В. Филимонов, П.Ю. Солдатов, И.С. Никитин

Предложен способ адаптации прямого решателя СЛАУ для вычислительных систем, использующих графические ускорители (GPU). Описан опыт пошагового повышения быстродействия. Перечислены проблемы, возникшие при работе с графическими процессорами, рассмотрены варианты их решения. Исследовалось влияние различных факторов на эффективность решателя. Приведены результаты тестирования для разреженных матриц большой размерности.

Ключевые слова: графические процессоры (GPU), параллельные вычисления, разложение Холецкого.

Введение

Благодаря параллельным алгоритмам и развитию архитектур центральных процессоров можно существенно оптимизировать работу решателей. Тем не менее, повышать производительность вычислений на традиционных архитектурах становится все сложнее: рост тактовой частоты центральных процессоров (CPU) замедлился, требуются вложения в содержание кластеров, необходимо адаптировать коды под многопроцессорную архитектуру. Альтернативой дальнейшему наращиванию мощностей являются вычисления на графических процессорах (GPU). Благодаря большому количеству ядер и другим особенностям архитектуры использование GPU позволяет значительно увеличить производительность вычислений для некоторых задач [1]. В данный момент развитие технологий вычислений общего назначения на GPU идет быстрыми темпами, причем как на аппаратном, так и на программном уровне.

В данной работе рассмотрен прямой решатель СЛАУ, который реализует разложение Холецкого. Данный метод является прямым и эффективен при большом количестве правых частей. Метод применяется в конечно-элементных комплексах, например для расчета сооружений [2, 3, 4]. Разработанный решатель эффективно распараллелен для машин с общей памятью с помощью директив OpenMP.

Перспективы применения вычислений на GPU

Разработка оптимального для графических вычислительных устройств параллельного алгоритма для реализации разложения Холецкого с нуля или внедрением директив OpenACC потребовала бы много ресурсов [5]. Поэтому было решено сосредоточиться на перенаправлении наиболее ресурсоемких вычислений на GPU и обойтись минимальными изменениями кода. Решатель использует интерфейс BLAS (Basic Linear Algebra Subprograms), который является де-факто стандартом интерфейса программирования приложений для создания библиотек, выполняющих основные операции линейной алгебры, такие как умножение векторов и матриц. В то же время существует библиотека cuBLAS, которая реализует тот же интерфейс, адаптирована для GPU и входит в стандартный комплект разработки CUDA Toolkit, поставляемый компанией nVidia [6, 7]. Таким образом, можно динамически подключить две библиотеки, оптимизированные для разных архитектур. Для внедрения cuBLAS в работу решателя потребовалось разработать набор функций, которые устраняют отличия в обращении к BLAS и cuBLAS (например, в передаваемых типах данных) и корректно осуществляет передачу данных между вычислительными мощностями [8].

Профилирование алгоритма показало, что операция умножения матриц (GEMM) занимает 80–85% от времени факторизации. Используя GPU возможно серьезно уменьшить время выполнения умножений и, следовательно, серьезно уменьшить время работы решателя [9].

Однако, особенности работы GPU затрудняют применение данного подхода. Так, перенаправление абсолютно всех операций умножения матриц не ускорит, а замедлит работу решателя, поскольку будет затрачено слишком много времени на переписывание данных. Поэтому необходимо было определить критерии направления операции на GPU. Отдельный интерес представляет поиск максимально быстрого способа копирования данных на GPU и обратно. Подготовка данных для GPU происходит во многих параллельных потоках на CPU, и при их передаче образуется очередь, которая практически сводит на нет распараллеливание OpenMP. Оптимальный баланс, полученный на определенном примере и конфигурации оборудования, может быть нарушен при изменении конфигурации или задачи. Способы решения этих проблем будут описаны ниже.

Работа с cuBLAS

В ходе работы были разработаны тестовые программы сравнения скорости умножения матриц. По результатам их выполнения были определены размеры массивов, превышение которых могло сделать выгодным использование GPU. Очевидно, чем выше класс используемого GPU, тем ниже находится этот порог, тем быстрее выполняются сами вычисления, и уменьшается время работы решателя. Даже с учетом копирования данных на GPU и обратно для матриц, содержащих десятки тысяч элементов, умножение матриц на GPU может

выполняться на один-два порядка быстрее, чем при использовании многопоточных CPU, устанавливаемых в настольных ПК.

Любое обращение к GPU требует достаточно много времени, поэтому в первую очередь внимание было уделено работе с памятью. Было установлено, что выделять и освобождать участок памяти на GPU для каждого набора переменных менее эффективно, чем при инициализации устройства выделить под указатели память, размер которой сопоставим с доступной глобальной памятью на конкретном GPU. В этом случае копирование матриц производится каждый раз в один и тот же участок памяти без его освобождения. Освобождение памяти производится по завершении работы устройства. Также установлено, что при необходимости выполнения операций с подматрицами передаваемых матриц выгоднее переписать матрицу полностью, а не только нужную для вычислений часть, поскольку при этом не тратится лишнее время на вызов копирования.

В многопоточном режиме образование очередей заданий для GPU замедляет работу решателя, поскольку стандартными способами отправить новое задание на GPU можно только после завершения исполнения текущего. Быстрое умножение не перекрывает времени ожидания и копирования, и уменьшить время работы решателя не удается. В случае, если в алгоритме распределения прописать запрет на формирование заданий для GPU, когда GPU уже занят, то можно добиться незначительного ускорения. Эффект от работы GPU будет нивелирован относительно медленной работой ядер CPU с оставшимися в их распоряжении большими объемами информации, но общее время работы все же уменьшится. Было установлено, что применение такого алгоритма позволяет сократить время работы решателя вдвое в однопоточном режиме (без использования OpenMP) и на 20% в многопоточном режиме.

Запуск профилировщика GPU для тестового примера умножения матриц, аналог которого был встроен в решатель, показал, что загрузить видеокарту должным образом не получается, так как слишком мало ядер участвует в вычислениях, и соотношение времени вычислений к общему времени работы GPU незначительно (Рисунок 1). Несмотря на экономию времени за счет выделения и освобождения памяти, копирование выполнялось слишком долго. Поэтому был сделан вывод о неэффективности библиотеки, формирующей и передающей задания для GPU, и необходимости ее усовершенствования. Для получения приемлемого результата умножение матриц на GPU должно выполняться гораздо быстрее.

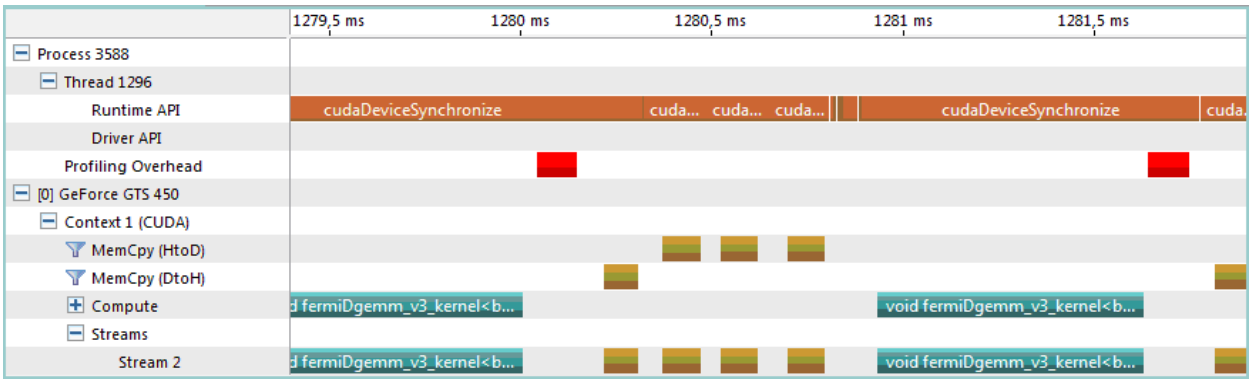

Рис. 1. Профилирование умножения матриц на GPU без оптимизации

Существуют методы распараллеливания операций на самой GPU [10]. Было использовано асинхронное копирование матриц. Благодаря cuda streams удается выполнять на видеокарте несколько операций умножения практически одновременно: каждый omp thread создает свой поток на GPU, и возникает параллельность выполнения некоторых команд. Так как программа после отправки задания на GPU продолжает выполнение команд на CPU, количество одновременно работающих потоков на GPU может превышать количество omp threads. Чтобы избежать переполнения памяти GPU, на некоторых видеокартах требуется ограничивать количество cuda streams, и направлять лишние задания на CPU.

С целью улучшения взаимодействия между CPU и GPU была применена pinned-память – прикрепленный буфер в оперативной памяти, который очень быстро может быть помещен в память GPU. Каждая матрица, которую нужно записать на GPU, сначала помещается в pinned-массив, а затем асинхронно копируется на GPU. Тестовый пример показал, что видеокарта стала простаивать гораздо реже (рис. 2), а скорость выполнения умножения, включая копирование матриц, увеличилась примерно в 2 раза по сравнению реализацией cublasDgemm без применения оптимизации (Рисунок 3).

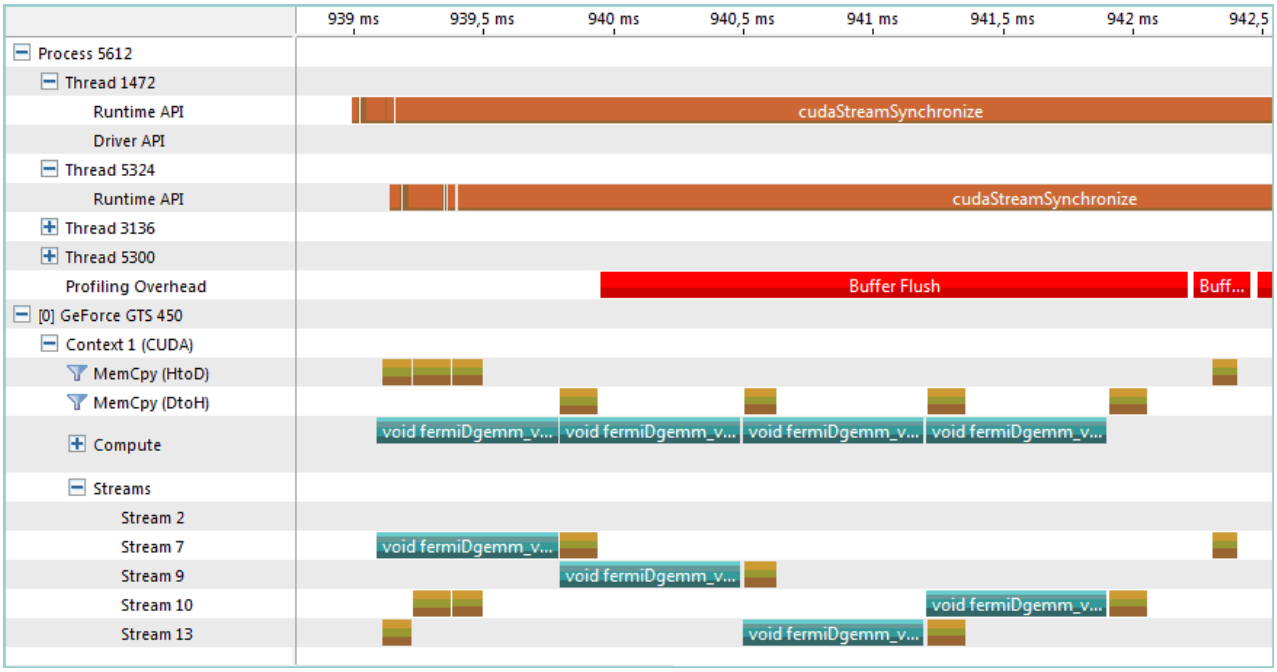

Рис. 2. Профилирование умножения матриц на GPU с использованием асинхронного копирования и pinned-памяти

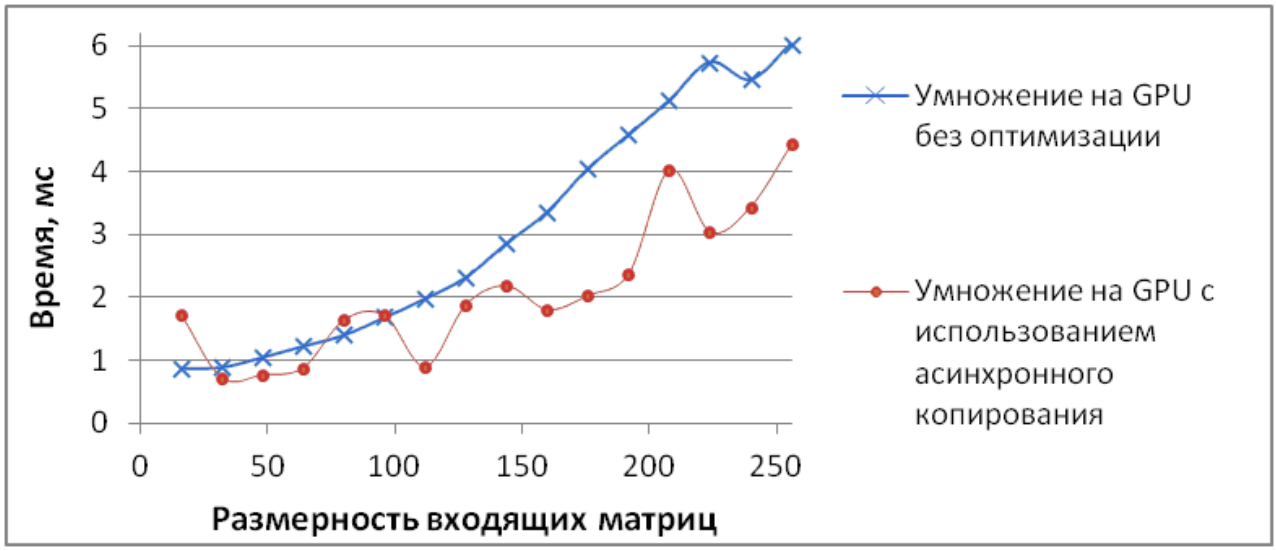

Рис. 3. Сравнение времени выполнения умножения матриц на GPU различными способами

Для настройки алгоритма, проверки правильности и оценки эффективности его работы был подобран набор разреженных матриц большой размерности, полученных с помощью метода конечных элементов. Численные эксперименты были проведены для различных конфигураций оборудования. Результаты тестирования решения – время расчета для каждой модели в различных режимах работы – приведены в Таблице 1. Все данные получены для двойной точности вычислений. Предполагается, что данный решатель будет использоваться в программных комплексах, при этом значительных затрат на новую технику со стороны компаний-пользователей не потребуется. Тестирование данного решателя производилось на оборудовании низкого и среднего ценового диапазона.

Таблица 1. Результаты тестирования (конфигурация 1: CPU Intel Core i7 3400 MHz (4 ядра, 8 потоков), GPU nVidia GeForce GTX 550Ti; конфигурация 2: 2x Intel Xeon X5680 3300 MHz (6 ядер, 12 потоков), GPU nVidia Tesla M2090)

№ п.п.	Размерность матрицы	Время факторизации, с	
		Конфигурация 1	Конфигурация 2

		CPU	CPU+GPU	Ускорение	CPU	CPU+GPU	Ускорение
1	921 600	64	36	1,78	59	25	2,36
2	2 263 338	127	76	1,67	115	44	2,61
3	2 428 323	108	64	1,69	99	37	2,68
4	2 545 314	267	128	2,09	141	53	2,66

Для некоторых задач добиться ускорения работы решателя не удается в силу особенностей решения СЛАУ, например, из-за низкого заполнения множителей разложения Холецкого. Рассматривается вариант написания собственного ядра для перемножения небольших матриц в первую очередь из-за непредсказуемого времени работы функций cuBLAS на небольших объемах данных (Рисунок 3). Возможно как варьировать способы копирования в зависимости от размеров входящих матриц, так и генерировать сами матрицы непосредственно на GPU. Другое направление для изучения – использование нескольких графических процессоров одновременно. Также необходимо стремиться к максимальному использованию пропускной способности канала передачи данных между вычислительными мощностями.

Выводы

При использовании графического ускорителя удалось уменьшить время работы решателя в 2,36-2,68 раза, при использовании nVidia Tesla M2090, и в 1,78-2,09, при использовании nVidia GeForce GTX 550Ti, при минимальном изменении исходного кода. Используемый метод оптимизации позволяет осуществить автоматическую балансировку для различных конфигураций CPU и GPU. Дальнейшее развитие представленного подхода видится в использовании нескольких графических ускорителей и применении более эффективных алгоритмов умножения матриц. После тестирования в различных условиях решатель может использоваться в расчетных программных комплексах.

ЛИТЕРАТУРА:

1. Cullinan C., Wyant C., Frattesi T. Computing Performance Benchmarks among CPU, GPU, and FPGA. Available at: http://www.wpi.edu/ accessed 03/26/2013.
2. Якушев В.Л., Жук Ю.Н., Симбиркин В.Н., Филимонов А.В. Реализация методов расчета для большеразмерных задач строительной механики в программном комплексе STARK ES. Вестник кибернетики 2011. № 10. С. 109–116.
3. Якушев В.Л., Симбиркин В.Н., Филимонов А.В. Решение большеразмерных задач строительной механики методом конечных элементов в программном комплексе STARK ES. Теория и практика расчета зданий, сооружений и элементов конструкций. Аналитические и численные методы: Сб. трудов международной научно-практической конференции. Москва, 2010. С. 516–526.
4. Yakushev V.L., Shuk U.N., Simbirkin V.N., Novikov P.A., Filimonov A.V. Solution of the generalized eigenvalue problem for 3D tensile structures in Stark ES. TensiNet symposium 2010 Tensile Architecture: Connecting Past and Future will be held 16-17-18 September 2010 at Sofia, University of Architecture, Civil Engineering and Geodesy. P. 103-104.
5. Hogg J.D., Reid J.K., Scott J.A. Design of a Multicore Sparse Cholesky Factorization Using DAGs: STFC Technical Report RAL-TR-2009-027. Science and Technology Facilities Council, 2009.
6. Сандерс Дж., Кэндрот Э. Технология CUDA в примерах: введение в программирование графических процессоров: Пер. с англ. Слинкина А.А., научный редактор Боресков А.В. М.: ДМК Пресс, 2011.
7. CUBLAS Library User Guide. // NVIDIA Corporation. Available at http://developer.nvidia.com/ accessed 03/26/2013.
8. Якушев В.Л., Филимонов А.В., Новиков П.А., Солдатов П.Ю. Создание эффективных решателей для GPU // Научн.-практ. конф. с междунар. участием с элементами научн. шк. для молодежи Высокопроизводительные вычисления на графических процессорах: Тез. докл., Пермь, 2012 г. С. 85–87.
9. Tan G., Li L., Triechle S., Phillips E., Bao Y., Sun N. Fast implementation of DGEMM on Fermi GPU // Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, ACM, New York, NY, USA , pp. 35:1–35:11.
10. CUDA C Programming Guide. Available at: http://docs.nvidia.com/ accessed 03/26/2013.