Наука и Образование
МГТУ им. Н.Э. Баумана
Сетевое научное издание
ISSN 1994-0408

Наука и Образование. МГТУ им. Н.Э. Баумана. Электрон. журн. 2015. № 07. С. 129–145.

DOI: **10.7463/0715.0791286**

Представлена в редакцию: 04.06.2015
Исправлена: 21.06.2015

УДК 004.021

Применение технологии CUDA в задаче об определении матрицы жесткости

Берчун Ю. В.[1,*], **Киселёв И. А.**[1], **Хахалин А. С.**[1], **Сычёва Е. А.**[1], **Петрова Т. Д.**[1], **Яблоков В. Е.**[1]

[*]y_berchun@mail.ru

[1]МГТУ им. Н.Э. Баумана, Москва, Россия

В статье описывается ресурсоемкая часть по вычислению отображения матрицы жесткости в подпространстве собственных векторов в задаче по нахождению собственных частот колебаний методом итераций в подпространстве. Рассматривается технология программирования CUDA, как одна из технологий GPGPU. Перечисляются альтернативы данной технологии. Предлагается математическая интерпретация решаемой задачи с учетом специфики входных данных. Приводится описание программной реализаций данной операции с применением технологии CUDA. Содержится описание структур данных, используемых при реализации программы. Приводится описание и результаты тестирования.

Ключевые слова: графический процессор, GPU, GPGPU, CUDA, матрица жёсткости, матричные вычисления, собственные формы

Введение

До недавнего времени реализация алгоритмов общего назначения на графических устройствах требовала особых методов программирования. Помимо сложности в написании программ, ориентированных на параллельное исполнение, необходимо было использовать определенное *API* для графических устройств, которое не было предназначено для задач, несвязанных с визуализаций. Потребность в вычислениях общего назначения на ***GPU*** (*General-Purpose* computing on *GPU, GPGPU*) привела к появлению новых технологий. С их помощью стало возможно решение более широкого круга задач на графических картах. В конце 2006 года компанией *Nvidia* была представлена одна из таких технологий – ***CUDA*** (*Compute Unified Device Architecture*) [1].

Компания *Nvidia* не единственная, кто направил свои силы на *GPGPU*. Примерно в то же время *AMD* выпускает свой продукт ***AMD Close To Metal****(CTM)* – низкоуровневый программный интерфейс, нацеленный на вычисления общего назначения на картах компании. Данный продукт просуществовал недолго, и уже в декабре 2007 года был

объявлен выпуск ***Stream SDK***, дошедший до нашего времени как ***AMD APP SDK*** [2]. Решения, выпущенные компаниями *Nvidia* и *AMD*, предназначены только для графических устройств собственного производства. В 2008 году некоммерческим консорциумом *Khronos Group* [3] был представлен новый стандарт ***OpenCL*** [4] - фреймворк ориентированный на *GPGPU* и не зависимый от аппаратного обеспечения. Компания *Microsoft* так же приняла участие в данном направлении. Она разработала библиотеку ***C++ AMP*** [5], которая построена поверх *DirectX* 11 и предназначена для программирования гетерогенных (т.е. использующих одновременно вычислительные мощности, как центрального процессора, так и видеокарты) приложений. Одним из новых игроков в вычислениях общего назначения на графических устройствах является стандарт ***OpenACC*** [6]. Он описывает набор директив компилятора, предназначенных для простого и быстрого создания гетерогенных программ.

В настоящее время графические устройства применяются для высокопроизводительных параллельных вычислений в различных системах. Во многих суперкомпьютерах фирмы *Cray*, по состоянию на 2014 год [7], уже установлены, помимо чипов архитектуры x86-64, графические ускорители *Nvidia Tesla*. Применение *GPU* позволяет добиться многократного увеличения производительности в медицинских и в физических расчётах, в системах видеонаблюдения и во многих других направлениях [8]. В данной статье основное внимание уделено повышению производительности расчета за счет применения вычислительных возможностей графических процессоров при решении задачи об определении собственных частот и форм колебаний конечно-элементных моделей с большим числом степеней свободы.

Статья состоит из пяти частей. В первой приводится краткое описание технологии *CUDA*. Во второй части сформулирована задача и приведены некоторые замечания к выбору технологии. В третьей части рассмотрен алгоритм для решения поставленной задачи. В четвертой части описываются особенности программной реализации. В заключительной, пятой части рассмотрены результаты и представлен анализ проделанной работы.

Описание технологии CUDA

CUDA – это программно-аппаратная архитектура, разработанная компанией *Nvidia* для осуществления параллельных вычислений общего назначения на графическом процессоре. Данная платформа впервые появились на рынке с выходом чипа *G*80 и стала присутствовать во всех последующих сериях графических чипов компании. Программная модель архитектуры представляет собой набор библиотек и расширение языка, позволяющее удобно оперировать с возможностями графического устройства. И в отличие от шейдерных языков [9], используемых ранее для программирования на *GPU*, она несет в себе синтаксис, направленный на вычисления общего назначения.

Технология *CUDA* основывается на концепции, что графическое устройство (далее **устройство**, *device*) является массивно-параллельным сопроцессором к центральному

процессору (далее **хост**, *host*). Последовательная часть программы выполняется на хосте, а те места, которые подходят для распараллеливания, выполняются на устройстве. Функции, которые выполняются на устройстве, называются **ядрами** (*kernels*). В то же время для решения задач *CUDA* использует очень большое количество параллельно выполняемых **нитей** (*threads*), при этом обычно каждой нити соответствует один элемент вычисляемых данных. Все запущенные на выполнение нити организованы в иерархию (рис. 1).

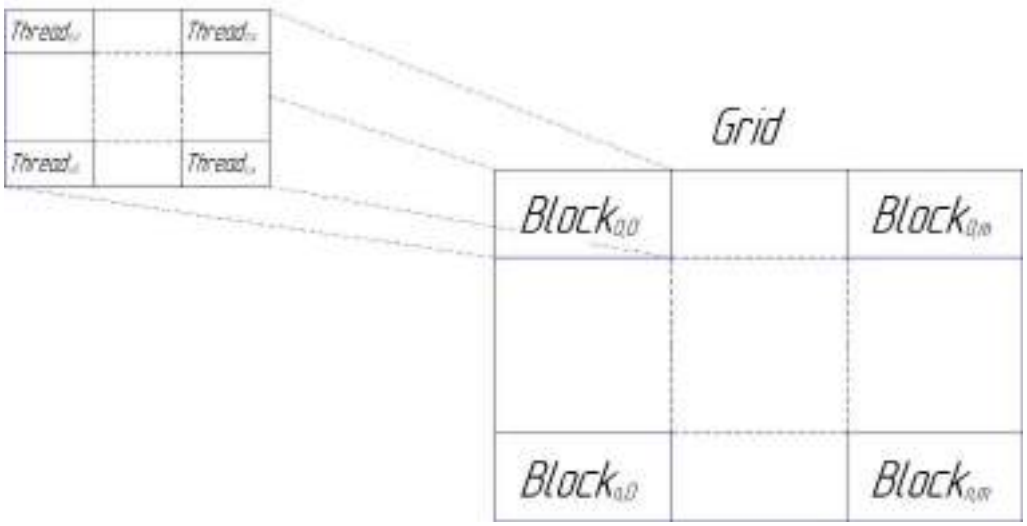

Рисунок 1. Иерархия нитей

На самом верхнем уровне находится **сетка** (*grid*), которая соответствует всем нитям, выполняющим данной ядро. Сетка представляет из себя одномерный или двухмерный массив **блоков** (*block*). Каждый блок, в свою очередь, представляет собой одномерный, двухмерный или трехмерный массив нитей (*threads*).

К основным программным особенностям технологии CUDA необходимо отнести расширения языка и набор поставляемых библиотек. Для управления местом выполнения кода программы были введены **спецификаторы функций**. Их необходимо указывать перед сигнатурой функции. Существует три спецификатора: *__global__*, *__device__* и *__host__*. Помимо места выполнения существует ограничения по месту вызова функций в зависимости от указанного спецификатора. Также к расширению языка относятся **спецификаторы переменных**, определяющие тип памяти, отводящейся под их хранение.

Помимо программных нововведений, компания *Nvidia* изменила и аппаратную часть своих графических устройств. Эффективная, понятная и простая аналогия между программными функциями и их аппаратной реализацией, позволяет практически на прямую оперировать с устройством. Так, например, у программистов имеется возможность управлять содержимым, так называемой, **разделяемой памяти** (*shared memory)* [10], которая физически является частью мультипроцессора. Скорость обращения к разделяемой памяти такая же, как и к кэшу первого уровня, причем физически кэш и

разделяемая память – одно и тоже, пропорция между этими двумя видами памяти может устанавливаться за счет настроек при инициализации видеокарты.

Более подробно с технологией *Nvidia CUDA* и её особенностями можно ознакомиться по официальной документации [11] и книгам [12, 13]. Или же по небольшой, но описывающей многие особенности, статье [14].

Постановка задачи

Как было упомянуто выше, основное внимание в статье уделено задаче механики деформированного твердого тела, а именно задаче по определению собственных частот колебаний конструкции на основе метода конечных элементов. Одним из методов решения данной задачи является **метод итераций в подпространстве**, состоящий из нескольких этапов, подробно описанных в [15, 16]. Одной из наиболее ресурсоемких стадий алгоритма является **определение матрицы жесткости в подпространстве собственных форм** (1), представляющая собой перемножение трех матриц, имеющих различную структуру и схему хранения. Повышение производительности расчета, за счет параллельной реализации данной операции с применением вычислительных возможностей графического процессора является целью настоящей статьи.

$$\boldsymbol{K}_{k+1} = \boldsymbol{X}_{k+1}^{T}\boldsymbol{K}\boldsymbol{X}_{k+1} \quad (1)$$

Где $\boldsymbol{K}$ – $(v \times v)$ матрица жесткости конечно-элементной модели конструкции, $\boldsymbol{X}$ – $(v \times n)$ матрица собственных форм колебаний, $\boldsymbol{K_{k+1}}$ – искомая $(n \times n)$матрица жесткости в подпространстве, v – количество степеней свободы, n – количество собственных форм, участвующих в решении.

В настоящей работе за основу взята реализация метод итераций в подпространстве для определения собственных частот и форм колебаний конечно-элементных моделей с большим числом степеней свободы, реализованный на *CPU* в программном комплексе *UZOR* 1.0, имеющего параллельную сетевую архитектуру клиент-сервер [17, 18].Этап вычисления проекции матрицы жесткости в рассматриваемой реализации выполняется в серверной части программы и занимает около 40-60% от всего времени выполнения 1 итерации алгоритма, в зависимости от характеристик конкретной конечно-элементной модели при осуществлении расчета на одном вычислительном узле. В случае использования сетевой параллельной версии программы с числом вычислительных узлов более двух, процент времени, затрачиваемый на операцию (1) в рамках 1 итерации алгоритма возрастает. При использовании 10 и более вычислительных узлов, доля операции (1) для большинства моделей превышает 95%, поскольку остальные вычислительно емкие этапы метода итераций в подпространстве выполняются параллельно на узлах локальной вычислительной сети. В связи с этим, повышение производительности данного этапа за счет применения графического процессора является актуальным, поскольку позволит существенно повысить общую производительность при решении задачи. В качестве метрики для оценки эффективности будем использовать время выполнения этапа (1).

Будем рассматривать операцию (1) с математической точки зрения, представляя её как перемножение матриц с определёнными особенностями. С учетом специфики матричного умножения, заметим, что она хорошо подходит для распараллеливания, поэтому является допустимой для применения ресурсов графического процессора. Эффективность переноса вычисления операций линейной алгебры на *GPU* уже было показано в статьях [19, 20].

Выбор *Nvidia CUDA*, в качестве технологии, для реализации рассматриваемого произведения, обусловлен аппаратными предпочтениями и эффективностью архитектуры [21].

При решении задачи необходимо учесть следующие особенности:

- реализуемая операция является промежуточной в работе алгоритма по отысканию собственных частот;
- размерности матриц велики (до 10^7);
- матрица жесткости $\boldsymbol{K}$ является симметричной и сильно разреженной, поэтому хранится в специализированных форматах (либо в виде глобальной матрицы жесткости, для которой хранятся лишь ненулевые элемент (например, ***CSR*** [22]), либо в виде матриц жесткости элементов. В работе рассмотрен второй вариант).

Описание алгоритма

Алгоритм основан на особенностях входных данных, которые поставляются из ранее упомянутой программы, выполняемой на *CPU*. В нашем случае они представлены двумя наборами:

- матрица собственных векторов – $\boldsymbol{X}$, которая представляет собой набор векторов и является плотно заполненной;
- матрицы жесткости элементов, представленные в виде набора симметричных матриц $\boldsymbol{L_i}$ и, соответствующих им, $(1 \times l)$ индекс-векторов $\boldsymbol{v}_i$, определяющих положение компонент данной матрицы в общей матрице жесткости конструкции при ансамблировании.

Тогда операцию (1) преобразуем в (2):

$$\boldsymbol{K}_{k+1} = \sum_{i=0}^{m} \boldsymbol{X}_{k+1,i}^{T} \boldsymbol{L}_i \boldsymbol{X}_{k+1,i} \qquad (2)$$

Где $\boldsymbol{L_i}$ - $(l \times l)$ матрица жесткости *i-го* элемента, $\boldsymbol{X}_{k+1,i}$ - $(l \times n)$ матрица значений, составленная из строк матрицы векторов собственных форм, индексы которых соответствуют значениям из вектора $\boldsymbol{v}_i$, m – количество элементов.

Составление $\boldsymbol{X}_{k+1,i}$ осуществляется путем выбора *j-ых* строк из матрицы $\boldsymbol{X}_{k+1}$, где j – значение из вектора индексов, соответствующего *i-ой* матрице жесткости элемента (3).

$$\forall\, i \in [0; m), \forall\, h \in [0; l) \mid (\boldsymbol{X}_{k+1,i})_{h,} = (\boldsymbol{X}_{k+1})_{(\boldsymbol{v_i})_h,} \qquad (3)$$

Схематическое представление операции (3) приведено на рисунке 2.

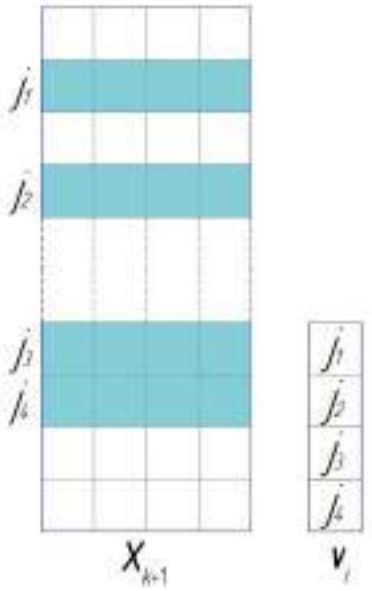

Рисунок 2. Выбор строк из матрицы собстенных форм по индекс-вектору матрицы жесткости элемента

Важной особенностью в поставленной задаче, является возможность её разделения на подзадачи для последующего параллельного вычисления. В рассматриваемом случае каждое слагаемое (4) в сумме (3) можно вычислять независимо друг от друга. Единственным критическим местом остается суммирование конечного результата.

Таким образом, алгоритм решения поставленной задачи сводится к трем частям: формирование матрицы $\boldsymbol{X}_{k+1,i}$ по известному $\boldsymbol{v}_i$, перемножение матриц (4) и суммирование промежуточных результатов (7).

$$\boldsymbol{K}_{k+1,i} = \boldsymbol{X}_{k+1,i}^T \boldsymbol{L}_i \boldsymbol{X}_{k+1,i}, \quad i = 0, \dots, m \tag{4}$$

Перемножение матриц разделим на два этапа (5), (6):

$$\boldsymbol{Y}_i = \boldsymbol{X}_{k+1,i}^T \boldsymbol{L}_i, \quad i = 0, \dots, m \tag{5}$$

$$\boldsymbol{K}_{k+1,i} = \boldsymbol{Y}_i \boldsymbol{X}_{k+1,i} \quad i = 0, \dots, m \tag{6}$$

Суммирование отображений локальных матриц жесткости:

$$\boldsymbol{K}_{k+1} = \sum_{i=0}^{m} \boldsymbol{K}_{k+1,i} \tag{7}$$

Программная реализация

Разработка программ под графический процессор для получения хороших результатов требует учета особенностей его работы. Большее количество ядер (по сравнению с *CPU*) в составе *GPU* позволяют создавать массивно-параллельные программы даже на домашнем компьютере. Но также это усложняет и сам процесс разработки. Необходимо преобразовывать алгоритм к данной архитектуре с учетом её особенностей, разрабатывать структуры и организовывать хранение данных таким образом, чтобы это являлось оптимальным при конкретных условиях.

В реализуемой программе элементы матриц хранятся в одномерном массиве по строкам. Такое хранение позволяет увеличить скорость доступа к произвольному элементу. Также разработана специальная структура для хранения матриц, предназначенных для устройства (рис. 3).

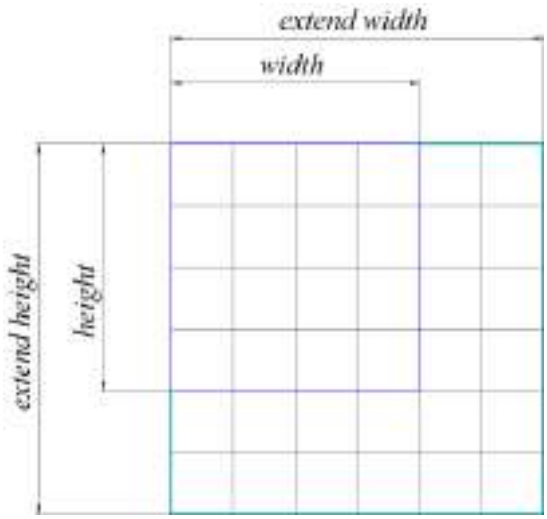

Рисунок 3. Схематическое изображение расширенной матрицы

Структура содержит указатель на первый элемент матрицы, основные размерности (высота и ширина), и дополнительные размерности (расширенная высота и расширенная ширина). Дополнительные размерности введены для оптимизации работы ядра. Их размер кратен размеру соответствующего измерения блока, что позволяет не учитывать выход потока за основные размеры при решении. В противовес данной методике можно поставить выполнение проверки выхода индекса нити за допустимые пределы (размерность матрицы) в ядре, но тогда в программе появится условные ветвления, что нежелательно для программ, выполняемых на *GPU*.

Основываясь на особенностях программной архитектуры *CUDA*, необходимо доработать алгоритм, описанный в предыдущей главе. С учетом того, что матрицы жесткости элементов являются малыми, но их количество велико, а запуск ядра является дорогостоящей операцией, было решено группировать $\boldsymbol{X}_{k+1,i}$ и $\boldsymbol{L}_i$ в матрицы (рис. 4) имеющие блочный характер, где каждый блок является матрицами $\boldsymbol{X}_{k+1,i}$ или $\boldsymbol{L}_i$. Таким образом сократим количество запусков и оптимизируем работу ядра. Так же с учетом того, что выполнение ядра является асинхронным, появляется возможность одновременно подготавливать следующий набор матриц. Объединять матрицы будем в группы по *num* блоков.

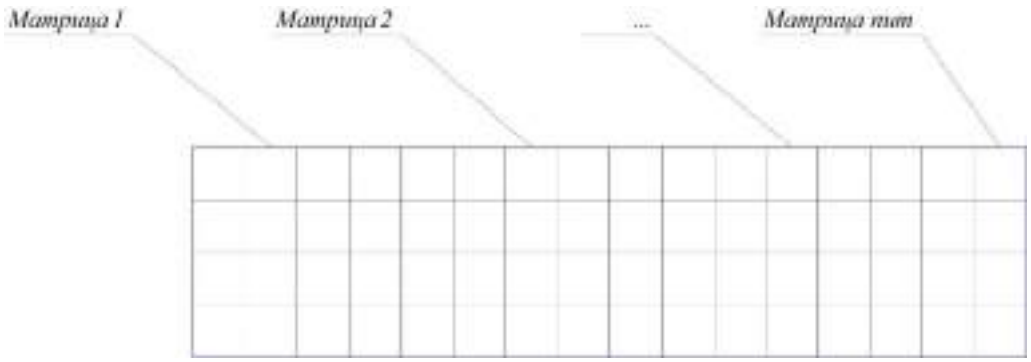

Рисунок 4. Горизонтальное представление блочной матрицы

С учетом такого представления, сначала будет производиться операция (5) для *num* блоков, затем выполняться операция (6) для этих же данных. Тогда операция (6) сводится к (8):

$$\sum_{i=a}^{a+num} \boldsymbol{K}_{k+1,i} = \sum_{i=a}^{a+num} \boldsymbol{Y}_i \boldsymbol{X}_{k+1,i}, \; a + num \le m, \; num > 0,$$

$$a = 0, 1num, 2num, \dots, \frac{m+num-2}{num-1} \qquad (8)$$

Где *num* – количество блоков в одной операции, *a* – индекс элемента, представленного первым блоком в блочной матрице (сдвиг).

Поблочное перемножение блочных матриц

Так как было решено оперировать блочными матрицами, то операция (5) требует изменений. Обозначим за $\boldsymbol{A}$ блочную матрицу наборов значений из матрицы собственных векторов (каждый блок – $\boldsymbol{X}_{k+1,i}^T$) , за $\boldsymbol{B}$ блочную матрицу из матриц жесткости элементов (каждый блок – $\boldsymbol{L}_i$), за $\boldsymbol{C}$ промежуточный результат (блочная матрица, где каждый блок – $\boldsymbol{Y}_i$). Необходимо перемножить каждый блок матрицы $\boldsymbol{A}$ на соответствующий блок матрицы $\boldsymbol{B}$, в результате получая один блок матрицы $\boldsymbol{C}$. Таким образом операция поблочного перемножения описывается формулой (9).

$$C_i = A_i \times B_i, \; i = 0, \dots, num - 1 \qquad (9)$$

Схематическое изображение данной операции представлено на рисунке 5.

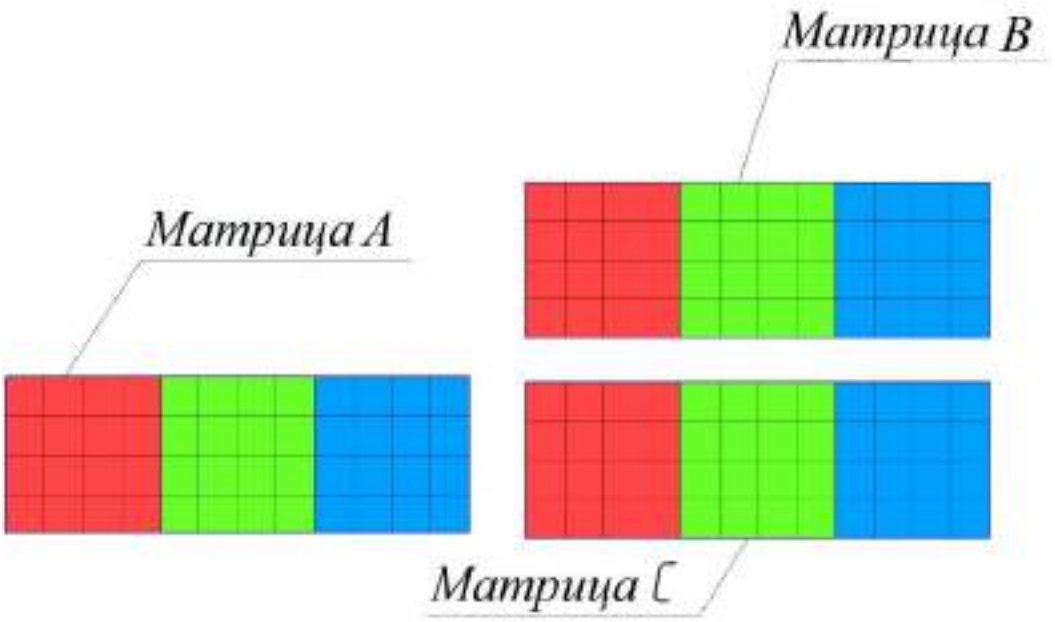

Рисунок 5. Схематическое изображение поблочного перемножения

Так как размер вычислительного блока *CUDA* может варьироваться, и в некоторых случаях не совпадать с размером блока матрицы, то нити, входящие в один блок *CUDA*, могут находится на стыке двух и более блоков матрицы. Поэтому необходимо

предусмотреть сложную индексацию, которая позволит выполнять перемножение без условий по определению блоков, к которым принадлежат элементы. Так же это определяет элементы, которые будут находится в разделяемой памяти.

Вторая операция

Второе ядро представляет собой реализацию операции (8), и выполняется над блочными матрицами (рис. 6). Каждая нить вычисляет 1 элемент в итоговой матрице, тем самым достигается параллельность. В отличие от предыдущего этапа, данная операция представляет простое перемножение двух матриц, тем самым исключается необходимость в сложной индексации в ядре. В результате получаем матрицу жесткости в подпространстве собственных форм (матрица ***R*** на рисунке 6).

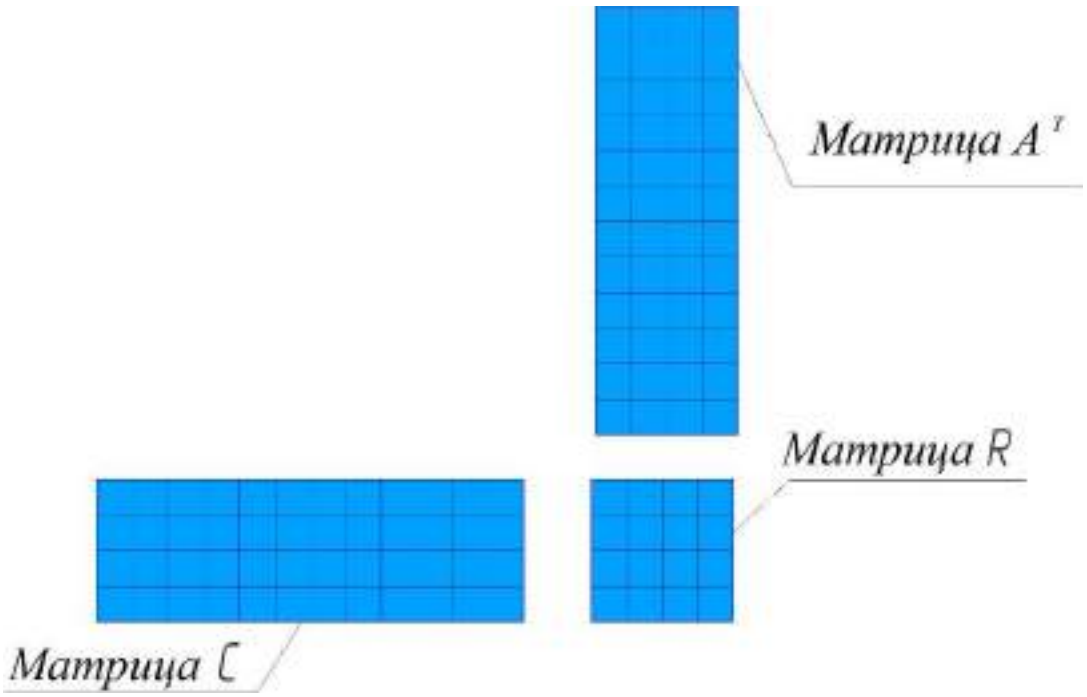

Рисунок 6. Схематическое изображение второй операции

Блок-схема алгоритма

Таким образом, программа на хосте считывает данные из файла, формируя блочные матрицы ***A*** и ***B***, количество блоков в которых записано в переменной *num.* Затем осуществляется асинхронный запуск двух ядер в одном потоке исполнения, что позволяет не беспокоиться о синхронизации этих операций. Если формирование следующего набора данных выполнилось до завершения вычислений на *GPU*, то синхронизируем устройство. Так как количество элементов может быть не кратно *num,* то по завершению основного цикла вызываются ядра для оставшегося количества данных. Блок-схема программной реализации поставленной задачи представлена на рисунке 7.

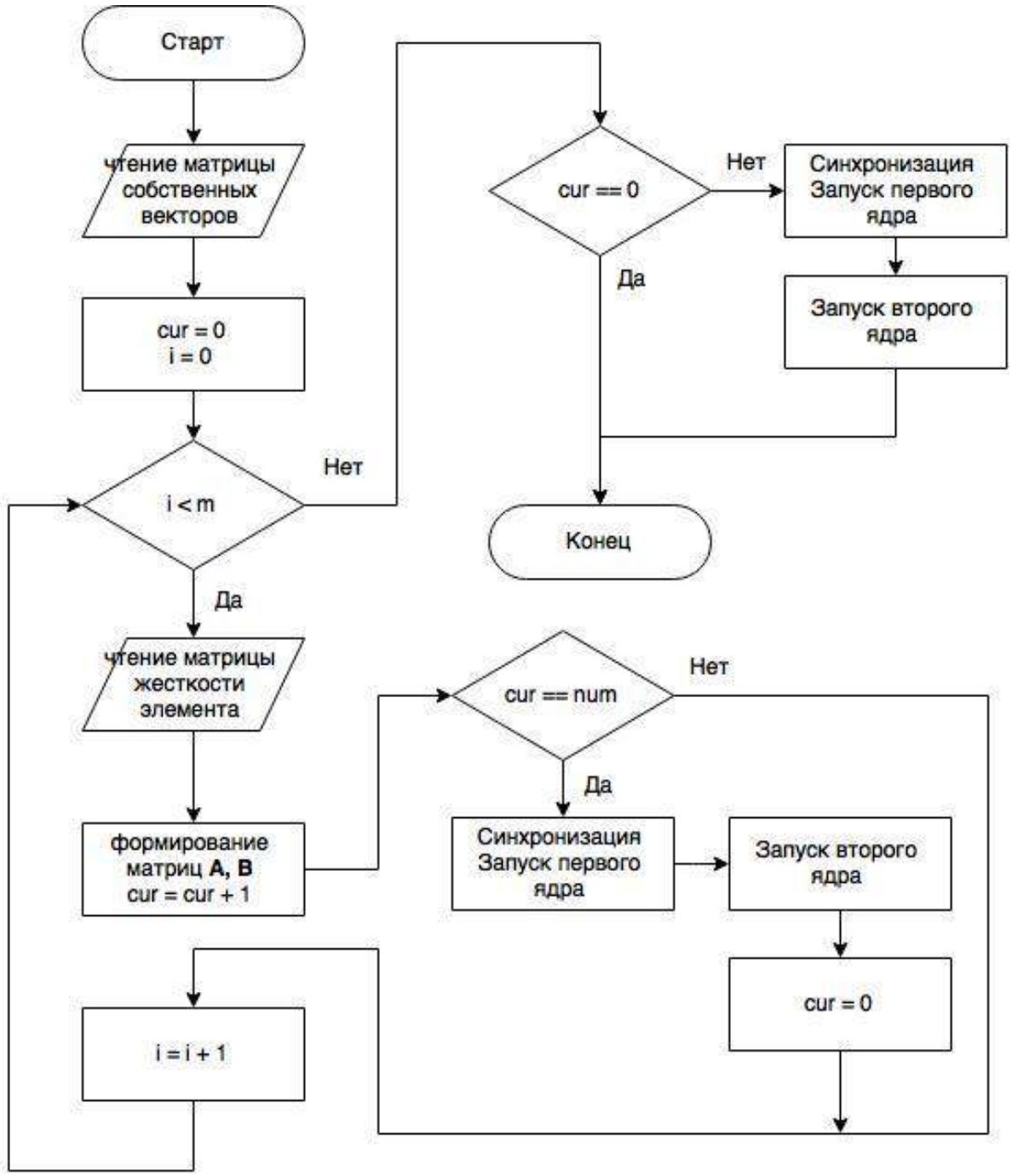

Рисунок 7. Блок-схема программной реализации

Тестирование

Для проведения тестов была выбрана упрощенная модель насосного агрегата, установленная на раме (рис. 8). Для нее требуется моделировать движение методом разложения по собственным формам [ссылка] в условиях нагрузки от землетрясения. Цилиндр сплошной, для него подобраны эквивалентные свойства, так чтобы он отражал массу, форму и размеры насоса. Такое приближение допустимо, поскольку в данном

расчете поведение внутренних частей насоса не вызывало опасений, и в целом он много жестче рамы.

Рисунок 7. Модель для тестирования

Приведенная модель имеет 1858824 степеней свободы, размер двоичного файла для хранения матрицы жесткости – 1324 Мб, а для матрицы собственных форм – 436 Мб. Задачей поиска являлось определение первых 27 форм колебаний. Тестирование проводилось на трех графических картах. Результаты (табл. 1) сравнивались с параллельной версией программы, выполняемой на центральном процессоре (Intel core i7 3.2ГГц, 6(12) ядер).

Таблица 1. Результаты

Наименование GPU	**Количество мультипроцессоров, шт.**	**Количество ядер, шт.**	**Время выполнения, с**	**Увеличение скорости, %**
GeForce GT 630M	2	96	31.86	254
GeForce GTX 470	14	448	6.72	1205
GeForce GTX 780	12	2304	4.36	1857

Результаты, приведенные в таблице 1 соответствуют, одной итерации при решении задачи по нахождению собственных частот и форм колебаний методом итераций в подпространстве. Для решения поставленной задачи (поиск собственных частот в диапазоне до 200 Гц, 27 шт.) потребовалось 18 итераций. Каждая итерация на центральном процессоре выполнялась порядка 160 секунд, из них на метод по определению матрицы жесткости в подпространстве собственных форм уходило 81 секунда. Таким образом на решение задачи по определению собственных частот и форм колебаний ушло 2880 секунд, из них 1458 секунд на рассматриваемый в статье этап. С применением графического процессора на данный этап уйдет 78 секунд (суммарное время на всех итерациях). Таким образом теоретическое общее время решения задачи с

применением графического процессора займет порядка 1500 секунд. Общий выигрыш за счет применения графического процессора в решении задачи по определению собственных частот и форм колебаний методом итераций в подпространстве составляет 192% (практически двукратное ускорение).

Заключение

Технология *Nvidia CUDA* является удобным и эффективным средством для вычислений общего назначения на графических устройствах. Скорость решения задачи по определению матрицы жесткости в подпространстве собственных форм с применением данной технологии увеличилась в несколько раз (в сравнении с *CPU* версией). Современные графические процессоры имеют большие возможности в решении массивно-параллельных задач. *CUDA* располагает простым механизмом выполнения параллельных программ на различных аппаратных платформах. Так один алгоритм, написанный с применением *CUDA*, может быть запущен на различных картах компании, и скорость выполнения будет зависеть только от аппаратных средств *GPU*. Таким образом данная технология позволяет системам быть легко масштабируемыми, а программное обеспечение переносимым и актуальным.

Список литературы

1. Параллельные вычисления CUDA // NVIDIA: сайт. Режим доступа: http://www.nvidia.ru/object/cuda-parallel-computing-ru.html (дата обращения 09.06.2015).
2. APP SDK – A Complete Development Platform // AMD: website. Available: http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk , accessed 15.06.2015.
3. Khronos Group, 25 Май 2014: промышленный консорциум // Википедия: сайт. Режим доступа: https://ru.wikipedia.org/wiki/Khronos_Group (дата обращения 11.06.2015).
4. OpenCL, 22 Июля 2013. Википедия: сайт. Режим доступа: https://ru.wikipedia.org/wiki/OpenCL (дата обращения 15.06.2015).
5. C++ AMP (C++ Accelerated Massive Parallelism) // Microsoft: сайт компании. Режим доступа: https://msdn.microsoft.com/ru-ru/library/hh265137.aspx (дата обращения 15.06.2015).
6. OpenACC: website. Available: http://www.openacc-standard.org/ , accessed 15.06.2015.
7. November 2014 // TOP500 list: website. Available at: http://www.top500.org/lists/2014/11 , accessed 21.05.2015.
8. Приложения для вычислений на GPU // NVIDIA: сайт. Режим доступа: http://www.nvidia.ru/object/gpu-computing-applications-ru.html (дата обращения 15.06.2015).
9. Шейдер (Shader). 18 Ноября 2005 // GameDev.ru — Разработка игр: сайт. Режим доступа: http://www.gamedev.ru/terms/Shader (дата обращения. 15.06.2015).

10. Using Shared Memory in CUDA C/C++, 28 January 2013 // NVIDIA CUDA Zone: website. Available at: http://devblogs.nvidia.com/parallelforall/using-shared-memory-cuda-cc/ , accessed 11.06.2015.
11. CUDA C Programming Guide, 5 March 2015 // NVIDIA Developer Zone: website. Available at: http://docs.nvidia.com/cuda/cuda-c-programming-guide , accessed 15.06.2015.
12. Боресков А.В., Харланов А.А. Основы работы с технологией CUDA. М.: ДМК Пресс, 2010. 232 с.
13. Сандерс Дж., Кендрот Э. Технология CUDA в примерах. Введение в программирование графических процессоров: пер. с англ. М.: ДМК Пресс, 2011. 232 с.
14. Казённов А.М. Основы технологии CUDA // Компьютерные исследования и моделирование. 2010. Т. 2, № 3. С. 295-308.
15. Bathe K.-J. The subspace iteration method – Revisited // Computers and Structures. 2013. Vol. 126, no. 15. P. 177-183. DOI: 10.1016/j.compstruc.2012.06.002
16. Вилсон Е., Батэ К. Численные методы анализа и метод конечных элементов: пер. с англ. М.: Стройиздат, 1982. 448 с.
17. Киселев А.С. Анализ эффективности методов расчета собственных частот трехмерных конструкций» // Труды 6-ой Российской НТК «Методы и программное обеспечение расчетов на прочность». 2010. С. 28-39.
18. Киселев А.С. Применение параллельных вычислений для повышения эффективности решения задач большой размерности в конечно-элементном программном комплексе UZOR 1.0 // Труды 8-ой Российской НТК «Методы и программное обеспечение расчетов на прочность». 2014. С. 117-131.
19. Карпенко А.П., Чернов С.К. Решение систем линейных алгебраических уравнений методом предобуславливания на графических процессорных устройствах // Наука и образование. МГТУ им. Н.Э. Баумана. Электрон. журн. 2013. № 1. С. 185-214. DOI: 10.7463/0113.0525190
20. Матвеева Н.О., Горбаченко В. И. Решение систем линейных алгебраических уравнений на графических процессорах с использованием технологии CUDA // Известия Пензенского государственного педагогического университета им. В.Г. Белинского. 2008. № 12. С. 115-120.
21. Ханкин К.М. Сравнение эффективности технологий OpenMP, NVIDIA CUDA И STARPU на примере задачи умножения матриц // Вестник Южно-Уральского государственного университета. Сер. Компьютерные технологии, управление, радиоэлектроника. 2013. Т. 13, № 1. С. 34-41.
22. cuSPARSE, 5 March 2015 // NVIDIA Developer: wwebsite. Available at: http://docs.nvidia.com/cuda/cusparse/#compressed-sparse-row-format-csr , accessed 28.06.2015.
23. Еременко С.Ю. Методы конечных элементов в механике деформируемых тел. Харьков: Изд-во "Основа" при Харьк. ун-те, 1991. 272 с.

24. Круглов В.Н., Папуловская Н.В., Чирышев А.В. Преимущества совместного использования CPU И CUDA-устройства // Фундаментальные исследования. 2014. № 8-2. С. 296-304.
25. Классификация Флинна // Лаборатория Параллельных информационных технологий НИВЦ МГУ: сайт. Режим доступа: http://parallel.ru/computers/taxonomy/flynn.html (дата обращения 15.06.2015).

Science & Education of the Bauman MSTU
Electronic journal
ISSN 1994-0408

Science and Education of the Bauman MSTU, 2015, no. 07, pp. 129–145.

DOI: **10.7463/0715.0791286**

Received: 04.06.2015
Revised: 21.06.2015

Using CUDA Technology for Defining the Stiffness Matrix in the Subspace of Eigenvectors

Yu.V. Berchun[1,*], I.A. Kiselev[1], A.S. Hahalin[1], E.A. Sycheva[1], T.D. Petrova[1], V.E. Yablokov[1]

[*] y_berchun@mail.ru

[1]Bauman Moscow State Technical University, Moscow, Russia

Keywords: графический процессор, GPU, GPGPU, CUDA, матрица жёсткости, матричные вычисления, собственные формы

The aim is to improve the performance of solving a problem of deformable solid mechanics through the use of GPGPU. The paper describes technologies for computing systems using both a central and a graphics processor and provides motivation for using CUDA technology as the efficient one.

The paper also analyses methods to solve the problem of defining natural frequencies and design waveforms, i.e. an iteration method in the subspace. The method includes several stages. The paper considers the most resource-hungry stage, which defines the stiffness matrix in the subspace of eigenforms and gives the mathematical interpretation of this stage.

The GPU choice as a computing device is justified. The paper presents an algorithm for calculating the stiffness matrix in the subspace of eigenforms taking into consideration the features of input data. The global stiffness matrix is very sparse, and its size can reach tens of millions. Therefore, it is represented as a set of the stiffness matrices of the single elements of a model. The paper analyses methods of data representation in the software and selects the best practices for GPU computing.

It describes the software implementation using CUDA technology to calculate the stiffness matrix in the subspace of eigenforms. Due to the input data nature, it is impossible to use the universal libraries of matrix computations (cuSPARSE and cuBLAS) for loading the GPU. For efficient use of GPU resources in the software implementation, the stiffness matrices of elements are built in the block matrices of a special form. The advantages of using shared memory in GPU calculations are described.

The transfer to the GPU computations allowed a twentyfold increase in performance (as compared to the multithreaded CPU-implementation) on the model of middle dimensions (degrees of freedom about 2 million). Such an acceleration of one stage speeds up defining the natural frequencies and waveforms by the iteration method in a subspace up to times.

References

1. CUDA Parallel Computing Platform. NVIDIA: website. Available at: http://www.nvidia.ru/object/cuda-parallel-computing-ru.html , accessed 09.06.2015.
2. APP SDK – A Complete Development Platform. AMD: website. Available at: http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk , accessed 15.06.2015.
3. Khronos Group, 25 May 2014: consortium. Wikipedia: website. Available at: https://en.wikipedia.org/wiki/Khronos_Group , accessed 11.06.2015.
4. OpenCL, 22 July 2013. Wikipedia: website. Available at: https://en.wikipedia.org/wiki/OpenCL , accessed 11.06.2015.
5. C++ AMP (C++ Accelerated Massive Parallelism). Microsoft: company website. Available at: https://msdn.microsoft.com/ru-ru/library/hh265137.aspx , accessed 15.06.2015.
6. OpenACC: website. Available at: http://www.openacc-standard.org/ , accessed 15.06.2015.
7. November 2014. TOP500 list: website. Available at: http://www.top500.org/lists/2014/11 , accessed 21.05.2015.
8. GPU computing applications. NVIDIA: website. Available at: http://www.nvidia.ru/object/gpu-computing-applications-ru.html , accessed 15.06.2015.
9. Shader. 18 November 2005. GameDev.ru - Game development: website. Available at: http://www.gamedev.ru/terms/Shader , accessed 15.06.2015. (in Russian).
10. Using Shared Memory in CUDA C/C++, 28 January 2013. NVIDIA CUDA Zone: website. Available at: http://devblogs.nvidia.com/parallelforall/using-shared-memory-cuda-cc/ , accessed 11.06.2015.
11. CUDA C Programming Guide, 5 March 2015. NVIDIA Developer Zone: website. Available at: http://docs.nvidia.com/cuda/cuda-c-programming-guide , accessed 15.06.2015.
12. Boreskov A.V., Kharlanov A.A. *Osnovy raboty s tekhnologiei CUDA* [Basics of working with CUDA technology]. Moscow, DMK Press, 2010. 232 p. (in Russian).
13. Sanders J., Kendrot E. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 2010. 312 p. (Russ. ed.: Sanders J., Kendrot E. *Tekhnologiya CUDA v primerakh. Vvedenie v programmirovanie graficheskikh protsessorov*. Moscow, DMK Press, 2011. 232 p.).
14. Kazennov A.M. Basic concepts of CUDA technology. *Komp'yuternye issledovaniya i modelirovanie = Computer Research and Modeling*, 2010, vol. 2, no. 3, pp. 295-308. (in Russian).
15. Bathe K.-J. The subspace iteration method – Revisited. *Computers and Structures*, 2013, vol. 126, no. 15, pp. 177-183. DOI: 10.1016/j.compstruc.2012.06.002
16. Bathe K.J., Wilson E.L. *Numerical methods in finite element analysis*. Prentice-Hall, Englewood Cliffs, NJ, 1976. (Russ. ed.: Bathe K.J., Wilson E.L. *Chislennye metody analiza i metod konechnykh elementov*. Moscow, Stroiizdat Publ., 1982. 448 p.).

17. Kiselev A.S. Analysis of effectiveness of methods for calculating natural frequencies of three-dimensional structures. *Trudy 6-oi Rossiiskoi NTK "Metody i programmnoe obespechenie raschetov na prochnost'"* [Proceedings of the 6th Russian NTC "Methods and software for strength calculations"]. 2010, pp. 28-39. (in Russian).
18. Kiselev A.S Application of parallel computing to improve the efficiency of solving large-scale problems in finite element software package UZOR 1.0. *Trudy 8-oi Rossiiskoi NTK "Metody i programmnoe obespechenie raschetov na prochnost'"* [Proceedings of the 8th Russian NTC "Methods and software for strength calculations"]. 2014, pp. 117-131. (in Russian).
19. Karpenko A.P., Chernov S.K. Solving systems of linear algebraic equations by preconditioning on graphics processing units. *Nauka i obrazovanie MGTU im. N.E. Baumana = Science and Education of the Bauman MSTU*, 2013, no. 1, pp. 185-214. DOI: 10.7463/0113.0525190 (in Russian).
20. Matveeva N.O., Gorbachenko V. I. Solution of systems of linear algebraic equations on GPUs using CUDA technology. *Izvestiya Penzenskogo gosudarstvennogo pedagogicheskogo universiteta im. V.G. Belinskogo = Proceedings of Belinskii PSPU*, 2008, no. 12, pp. 115-120. (in Russian).
21. Khankin K.M. Efficiency comparison of OpenMP, nVidia CUDA and StarPU technologies by the example of matrix multiplication. *Vestnik Yuzhno-Ural'skogo gosudarstvennogo universiteta. Ser. Komp'yuternye tekhnologii, upravlenie, radioelektronika = Bulletin of the Southern Ural State University. Ser. Computer Technologies, Automatic Control and Radioelectronics*, 2013, vol. 13, no. 1, pp. 34-41. (in Russian).
22. cuSPARSE, 5 March 2015. NVIDIA Developer: website. Available at: http://docs.nvidia.com/cuda/cusparse/#compressed-sparse-row-format-csr , accessed 28.06.2015.
23. Eremenko S.Yu. *Metody konechnykh elementov v mekhanike deformiruemykh tel* [Finite elements in deformable bodies mechanics]. Kharkiv, Osnova Publ., 1991. 272 p. (in Russian).
24. Kruglov V.N., Papulovskaya N.V., Chiryshev A.V. Advantages of combined CPU and CUDA devices use. *Fundamental'nye issledovaniya = Basic researches*, 2014, no. 8-2, pp. 296-304. (in Russian).
25. Klassifikatsiya Flinna [Flynn's Classification]. Laboratory of Parallel Information Technology RCC MSU: website. Available at: http://parallel.ru/computers/taxonomy/flynn.html , accessed 15.06.2015. (in Russian).