Интернет-журнал «Науковедение» ISSN 2223-5167 http://naukovedenie.ru/

Том 8, №4 (2016) http://naukovedenie.ru/index.php?p=vol8-4

URL статьи: http://naukovedenie.ru/PDF/54TVN416.pdf

Статья опубликована 02.08.2016.

Ссылка для цитирования этой статьи:
Цыганов Д.Л., Киселёв И.А., Заварзин Д.А. Повышение производительности численного моделирования колебаний упругих систем методом модальной суперпозиции за счет применения технологии Nvidia CUDA // Интернет-журнал «НАУКОВЕДЕНИЕ» Том 8, №4 (2016) http://naukovedenie.ru/PDF/54TVN416.pdf (доступ свободный). Загл. с экрана. Яз. рус., англ.

УДК

Цыганов Дмитрий Леонидович
ФГБОУ ВО «Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)», Россия, Москва[1]
Студент
E-mail: dmitrytsgn@gmail.com

Киселёв Игорь Алексеевич
ФГБОУ ВО «Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)», Россия, Москва
Кандидат технических наук, доцент
E-mail: i.a.kiselev@yandex.ru
РИНЦ: http://elibrary.ru/author_profile.asp?id=713365

Заварзин Денис Алексеевич
ФГБОУ ВО «Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)», Россия, Москва
Студент
E-mail: zavarzin.den@bk.ru

Повышение производительности численного моделирования колебаний упругих систем методом модальной суперпозиции за счет применения технологии Nvidia CUDA

Аннотация. Целью работы является повышение производительности интегрирования уравнений движения методом разложения по собственным формам для конечно-элементных моделей большой размерности с помощью параллельных вычислений на графическом процессоре. Приведена краткая историческая справка об использовании графических процессоров для общих вычислений и технологии NVIDIA CUDA. Описаны ключевые свойства архитектуры графических процессоров, произведённых NVIDIA, которые поддерживают CUDA. Указаны особенности организации потоков и памяти в NVIDIA CUDA. Выявлены ресурсоёмкие операции в рассматриваемом методе, которые могут быть преобразованы с использованием параллельных вычислений. Разобрано несколько возможных алгоритмов организации потоков на графическом процессоре, с указанием временной сложности. Произведён выбор алгоритма, обеспечивающего наибольшую

[1] 107207, г. Москва, ул. Байкальская, д. 38, корпус 2, кв. 106

производительность алгоритма. Указаны критерии выбора алгоритма. Аргументирован выбор алгоритма параллельного суммирования на основе дерева. Описаны дополнительные усовершенствования программной реализации выбранного алгоритма, включающие использование особенностей архитектуры графических процессоров от NVIDIA. Алгоритм реализован на языке программирования CUDA C. Выбрана и составлена конечно-элементная модель для тестирования программы. Произведено сравнительное тестирование реализованного алгоритма на графическом процессоре и последовательной реализации на центральном процессоре с использованием тестовой конечно-элементной модели. Измерено время работы алгоритма в обоих случаях. Сделаны выводы о преимуществах выбранного алгоритма вычислений с применением графического процессора на основе полученного многократного прироста производительности.

Ключевые слова: графический процессор; CUDA; параллельные вычисления; собственные формы; многопоточность; параллельное суммирование; организация потоков; оптимизация; конечные элементы

Введение

С каждым годом параллельные вычисления получают всё более широкое распространение. Развитие вычислительной техники за последние 20 лет показало, что этот способ компьютерных вычислений является перспективным. Основными преимуществами являются потенциальная экономия времени по сравнению с последовательной моделью вычислений, более низкая стоимость компонентов, возможность решать более сложные и объёмные задачи.

С 2002 года началось развитие GPGPU (General-Purpose Computation on Graphics Processing Units) – вычислений общего назначения на графических процессорах. Это использование графических процессоров для вычислений, которые обычно выполняются на центральном процессоре [1]. Началась разработка программного обеспечения, позволяющего использовать графический процессор для таких задач, как перемножение матриц [2]. Широкое распространение графических процессоров позволило им стать наиболее доступным средством для осуществления параллельных вычислений. Встала задача создания модели программирования, которая бы инкапсулировала специфическую работу с графическим процессором и предоставила разработчикам интерфейс, позволяющий использовать графический процессор аналогично центральному процессору.

Одной из первых попыток создать такую модель был разработанный в Стэнфордском университете язык Brook, который является расширением языка C [3]. В 2006 году компания Nvidia представила свою версию программной модели - технологию CUDA (Compute Unified Device Architecture) [5]. На сегодняшний день эта технология продолжает развиваться. Особенности 8-й версия CUDA анонсированы в апреле 2016-го года. Nvidia производит широкую линейку поддерживающих CUDA GPU, которая включает как дорогостоящие модели, предназначенные для профессиональной графики и моделирования сложных процессов, так и бюджетные модели, предназначенные для массового рынка. Это значит, что внедрение в программу параллельных вычислений с помощью комплекса CUDA может повысить производительность на большинстве современных компьютеров, в которых установлен графический процессор от Nvidia.

Технологии CUDA уже используются в таких областях, как механика деформированного твёрдого тела [15], биоинформатика, сейсморазведка, вычислительная гидродинамика [8], системы автоматизированного проектирования, компьютерное зрение, машинное обучение [9], науки о данных и др. В данной статье рассмотрено применение

параллельных вычислений для повышения производительности интегрирования уравнений движения методом разложения по собственным формам для конечно-элементных моделей большой размерности.

Статья разделена на пять частей. В первой части приведено краткое описание технологии CUDA. Во второй части приведена постановка задачи. В третьей части описаны различные варианты алгоритмов и аргументирован выбор реализуемого алгоритма. В четвёртой части описана программа. Пятая часть является заключением и содержит итоги выполненной работы.

1. Общая информация о CUDA

CUDA - архитектура параллельных вычислений, использующая графический процессор. NVIDIA CUDA включает в себя различные расширения существующих языков программирования и библиотеки. В этой статье рассматривается CUDA C - расширении для языка C/C++.

GPU хорошо подходят для параллельных вычислений из-за особенной архитектуры, которая отличается от архитектуры центральных процессоров. Архитектура GPU предполагает, что количество транзисторов, занятых в обработке данных, больше количества транзисторов, занятых управляющими функциями. Одним из основных элементов GPU являются потоковые мультипроцессоры. [5] Мультипроцессоры содержат потоковые процессоры, также называемые ядра CUDA (CUDA cores). [5] Все ядра внутри мультипроцессора единовременно выполняют одну и ту же инструкцию, но с разными данными. В GPU от NVIDIA использована разработанная в этой компании архитектура SIMT (Single-Instruction, Multiple-Thread). SIMT представляет собой модель параллельных вычислений, сочетающую в себе многопоточность и принцип SIMD. По классификации Флинна [4] SIMD предполагает одиночный поток команд и множественные потоки данных.

Код программ, написанных на CUDA C, делится на главный, управляющий код, который выполняется на процессоре (host code [5]), и код, который предназначен для выполнения на GPU (device code [5]). Функции, предназначенные для выполнения на GPU (также называемые kernel [8]), отмечаются спецификатором __global__. [5] Поток исполнения (thread) – основная единица обработки в CUDA. Потоки организованы в блоки (blocks). Блоки могут быть одномерные, двухмерные или трёхмерные. Количество потоков в блоке ограничено и различается в зависимости от конкретной модели GPU. Все потоки одного блока выполняются на одном потоковом мультипроцессоре [5]. Каждый поток имеет свой номер внутри блока. Этот номер можно получить с помощью встроенной переменной threadIdx. Так как блок может быть одномерным, двухмерным или трёхмерным, переменная threadIdx содержит три номера. Каждый номер соответствует одному из измерений. Размерность блока хранится в переменной blockDim. Эта переменная содержит 3 параметра, соответствующих размерностям по каждому из 3-х измерений. Она одинакова для всех блоков внутри одной решётки. Блоки организованы в решётку (grid). Решётка, как и блоки, может быть одномерной, двухмерной или трёхмерной.

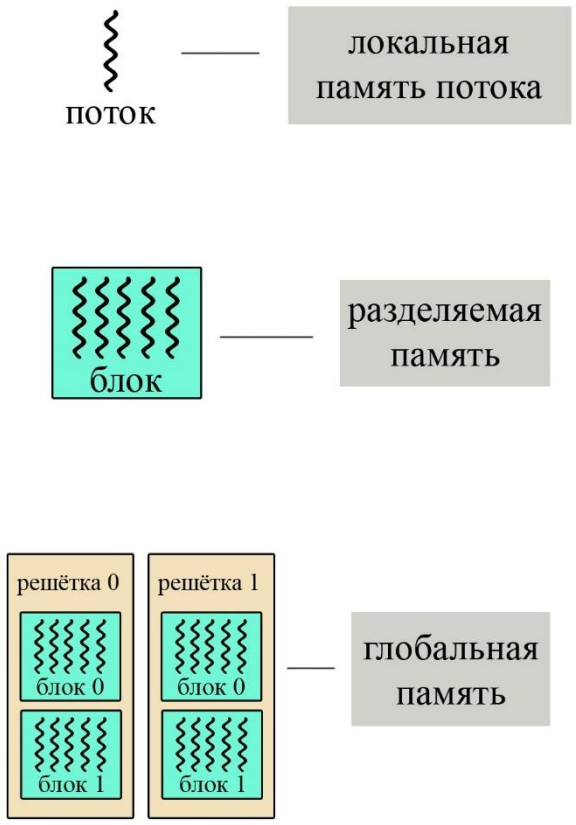

Рисунок 1. *Организация потоков и памяти в CUDA (разработано автором)*

GPU содержит модули динамической оперативной память. Память GPU делится на глобальную (global), разделяемую между потоками (shared) и локальную память потока. Разделяемая память представляет собой кэш память, которая находится в потоковом мультипроцессоре. Разделяемая память распределяется между блоками, выполняемыми на мультипроцессоре. Если к одним и тем же данным внутри блока необходимо обратиться несколько раз, загрузка этих данных в разделяемую память принесёт прирост производительности. [5]

Например, Nvidia GeForce GTX 780 Ti укомплектована 3072 мегабайтами глобальной памяти в соответствии со спецификацией. Этот графический процессор имеет вычислительные возможности (computing capability) уровня 3.5, а значит каждый мультипроцессор содержит по 64 килобайта кэш-памяти, которая распределена между разделяемой памятью и локальной памятью. [13] В зависимости от настроек, установленных пользователем, На разделяемую память может быть отведено 16 или 48 килобайт. При обработке данных, размер которых превышает объём разделяемой памяти, необходимо правильно выбрать, какие данные загрузить в неё для получения максимальной производительности.

2. Постановка задачи

Решение задачи о моделировании упругих колебаний деформируемого тела под действием внешних нагрузок, переменных во времени, может быть осуществлено при помощи эффективного и универсального численного подхода – метода конечных элементов [10]. В этом случае объект моделирования должен быть представлен дискретной конечно-элементной моделью, упругое движение которой полностью характеризуется перемещениями узлов сетки

по рассматриваемым степеням свободы. Движение дискретной модели для этого случая может быть представлено относительно вектора узловых перемещений в форме [10]:

$$K\{u\}+C\{\dot{u}\}+M\{\ddot{u}\}=\{Q(t)\} \quad (1)$$

где: K, C, M соответственно матрицы жёсткости, демпфирования и масс; {u(t)} – вектор-столбец узловых перемещений; {Q(t)} – вектор-столбец узловых сил.

При моделировании вибраций сложных объектов число степеней свободы модели может быть весьма велико (в частности может превышать 1 и 10-ки млн.). В таком случае прямое интегрирование по времени уравнения (1) приводит к значительным вычислительным затратам и целесообразно только при необходимости рассматривать быстропротекающие процессы (например, при ударном нагружении). В случае моделирования отклика на переменную нагрузку в низкочастотном диапазоне целесообразно применять метод модальной суперпозиции [6]. При использовании данного подхода на первом этапе требуется определить низшие собственные частоты и формы колебаний конечно-элементной модели в заданном диапазоне частот при помощи одного из известных методов [11] (обычно их необходимое количество лежит в диапазоне до нескольких десятков или сотен). Собственные формы колебаний представляют собой ортонормированный базис для рассматриваемой модели и обладают следующими свойствами:

$$u_i^T K u_j = 0, \quad u_i^T M u_j = 0 \text{ при } i \neq j \quad (2)$$

$$u_i^T M u_i = m_i, \quad u_i^T K u_i = k_i = \omega_i^2 m_i,$$

где: m_i – модальные массы; k_i – модальные жёсткости; ω_i – собственные частоты

На втором этапе движение системы может быть представлено в базисе выбранных собственных форм колебаний конечно-элементной модели в виде:

$$u(t) = a_1(t)u_1 + \ldots + a_n(t)u_n \quad (3)$$

После подстановки соотношения (3) в уравнение (1), умножении слева на транспонированный вектор собственной формы с номером i, а также с учетом соотношений (2), могут быть получены уравнения движения конечно-элементной модели в модальном пространстве:

$$m_i \ddot{a}_i(t) + [c_{i1}\dot{a}_1(t) + \ldots + c_{in}\dot{a}_n(t)] + k_i a_i(t) = q_i(t) \quad i = 1\ldots n \quad (4)$$

Уравнение движения по i собственной форме,

где: $c_{ij} = \{u_i\}^T C \{u_j\}$ - модальные коэффициенты демпфирования;

$q_i(t) = \{u_i\}^T \{Q(t)\}$ - модальные силы. (5)

Система дифференциальных уравнений движения в модальном пространстве:

$$[\mathfrak{M}]\{\ddot{a}(t)\} + [C]\{\dot{a}(t)\} + [\kappa]\{a(t)\} = \{q(t)\} \quad (6)$$

где: $\mathfrak{M}$ - матрица модальных масс; C - матрица модальных демпфирований; κ - матрица модальных жёсткостей.

В соотношении (4) индекс номера собственной формы i изменяется в пределах 1, ..., m, где **m** - количество учитываемых при расчете собственных форм колебаний. В связи с этим размерность системы уравнений (6) существенно меньше размерности системы дифференциальных уравнений (1). В то же время векторы $\{u_i\}$ и $\{Q(t)\}$ в соотношении для определения модальных сил (5) содержат по n элементов, где n – число степеней свободы рассматриваемой конечно-элементной модели. В случае, если расчет перемещений системы требуется только в отдельных узлах, а нагрузка прикладывается к большинству узлов конечно-элементной модели (например, при инерционной нагрузке), наибольшие вычислительные затраты сосредоточены в выражении (5) для вычисления модальных сил. Для получения $\{q(t)\}$ необходимо m раз скалярно перемножить эти векторы.

$$\{u_i\}^T \cdot \{Q(t)\} = \sum_{j=1}^{n} u_{ij} Q(t)_j = u_{i1} Q(t)_1 + \ldots + u_{in} Q(t)_n \quad (7)$$

В общем случае значения вектора $\{q(t)\}$ необходимо пересчитывать на каждом шаге интегрирования. При этом, для получения одного значения q_i необходимо произвести n операций умножения и n-1 операцию сложения. Каждая операция умножения не зависит от остальных. Так как необходимо произвести большое количество арифметических операций, которые могут быть выполнены в произвольном порядке, для нахождения значений вектора q_i рационально применить параллельные вычисления. Именно этот участок алгоритма в данной работе предлагается реализовать с использованием GPU.

3. Выбор алгоритма вычисления модальных сил

Для того, чтобы наиболее эффективно использовать вычислительные возможности GPU, необходимо правильно выбрать алгоритм вычисления модальных сил по выражению (5). При этом нужно учесть особенности решаемой задачи, исходных данных и архитектуры GPU.

Подбор алгоритма осуществляется на основе следующих требований:

- Временная сложность [12] алгоритма, выполняемого каждым потоком, должна быть минимальной.
- Использование последовательной организации кода должно быть сведено к минимуму.
- Алгоритм должен масштабироваться, т.е. зависеть от количества ядер и мультипроцессоров GPU.

Пример последовательного кода для вычисления $\{q(t)\}$ на языке C/C++:

```
for (int i=0; i<m; i++) {
        q[i] = 0.0;
        for (int j=0; j<n; j++) {
                q[i] += Q[j]*u[i][j];
        }
}
```

Элементы массивов $\{u_i\}$ и $\{Q(t)\}$ поочерёдно перемножаются и прибавляются к соответствующему элементу массива $\{q(t)\}$ (предполагается, что массив заранее инициализирован нулями). Временную сложность этого алгоритма можно представить как O(m) * O(n).

Стоит отметить, что этот код можно оптимизировать и без использования GPU. Так как современные процессоры имеют несколько ядер, можно создать несколько потоков, которые будут одновременно выполняться на процессоре. Однако, учитывая особенности задачи, архитектура GPU даст заведомо больший прирост производительности.

Одним из вариантов организации потоков, который можно применить для решения задачи, является вариант, в котором каждый элемент искомого массива $\{q(t)\}$ вычисляется параллельно отдельным потоком. Такая организация многопоточности используется, например, при перемножении матриц [5]. В нашей задаче понадобится составить m потоков, каждый из которых будет вычислять результат выражения (7) для i, определяемого на основе номера потока. Временная сложность алгоритма, выполняемого одним потоком, линейно зависит от длины обрабатываемых потоком векторов и равна O(n). Порядок m – $10^1 \ldots 10^2$, порядок n может достигать 10^7. Этот вариант организации потоков плохо подходит для данных, обрабатываемых в нашей задаче, так как количество потоков m зависит от количества учитываемых собственных форм, которое, как правило, существенно меньше числа потоковых процессоров видеокарты. Это означает, что вычислительные возможности GPU не могут быть полностью использованы. Кроме того, использование более мощных видеокарт не даст прироста производительности.

Так как для нахождения каждого элемента $\{q(t)\}$ требуется произвести большое количество операций, а элементов в векторе $\{q(t)\}$ сравнительно мало, имеет смысл подобрать алгоритм, позволяющий выполнить эту обработку с помощью параллельных вычислений, а затем последовательно применять его для каждого $\{u_i\}$ и $\{Q(t)\}$. Далее рассмотрены 2 варианта организации многопоточности, применяемых для нахождения одного элемента вектора $\{q(t)\}$.

Первый вариант предполагает разбиение векторов $\{u_i\}$ и $\{Q(t)\}$ на части с количеством элементов k. Каждый поток последовательно перемножает элементы u_{ij} и $Q(t)_j$ k раз, затем последовательно складывает полученные произведения (рисунок 2). Значение j вычисляется на основе номеров блока и потока. Затем все суммы, полученные каждым из потоков, последовательно суммируются для получения конечного результата. Количество потоков и, соответственно k, можно определять исходя из количества потоков, которое можно одновременно запустить на конкретном GPU.

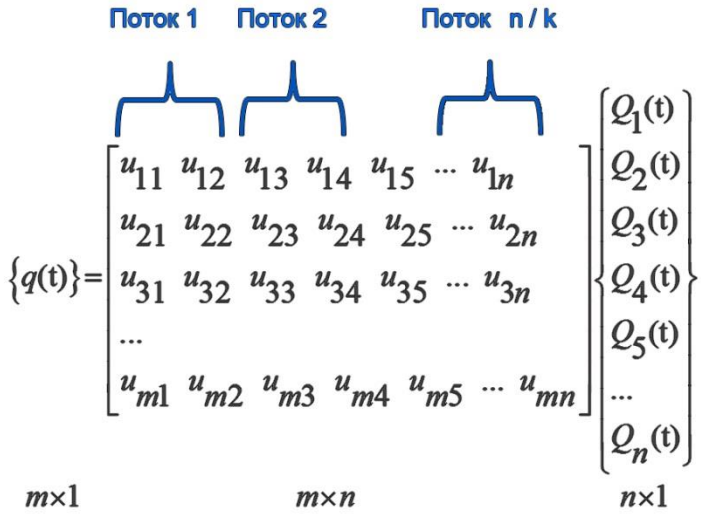

Рисунок 2. *Схема организации работы потоков, предполагающей разбиение векторов на части с количеством элементов k (разработано автором)*

Временная сложность алгоритма, выполняемого каждым потоком, как и в предыдущем случае линейно зависит от количества слагаемых $u_{ij}^{Q(t)_j}$, обрабатываемых потоком и равна O(k). Эта версия позволяет использовать вычислительные возможности конкретной GPU и способна работать быстрее на GPU с большим количеством мультипроцессоров. Однако в этом варианте реализации, как и в предыдущем, внутри каждого потока происходит производимая явно (в цикле) последовательная обработка элементов. Это говорит о том, что параллельные вычисления введены только частично.

Рассмотрим второй вариант - алгоритм параллельного суммирования, в основе которого лежит принцип дерева (или пирамиды) [7].

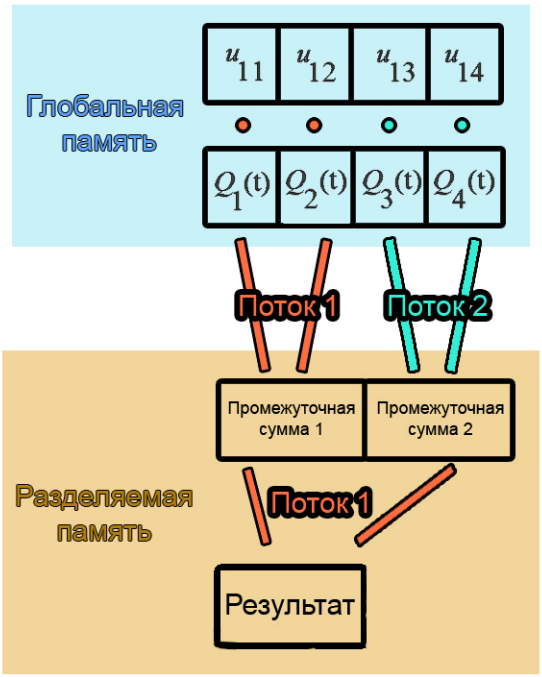

Рисунок 3. *Схема организации работы потоков, предполагающей суммирование по принципу дерева (разработано автором)*

Каждый поток за итерацию суммирует 2 элемента. В каждой следующей итерации участвует в 2 раза меньше потоков, чем в предыдущей. На первой итерации суммируются элементы исходного массива, на каждой следующей – промежуточные суммы, полученные в предыдущей итерации. Итерация, после которой останется только одна сумма, является последней. Эта сумма является конечным результатом. Временная сложность алгоритма O(log(n)), что меньше, чем в предыдущих случаях. Масштабируемость организуется с помощью блоков. Каждый блок обрабатывает k элементов, где k выбирается исходя из свойств конкретной GPU. Блоки распределяются между всеми мультипроцессорами исполняющей системой GPU [5], значит алгоритм масштабируется автоматически в зависимости от количества мультипроцессоров. Суммы, полученные всеми блоками, можно далее просуммировать по тому же алгоритму.

Алгоритм с разбиением векторов на части и алгоритм на основе дерева масштабируются лучше других рассмотренных алгоритмов. Алгоритм на основе дерева имеет временную сложность O(log(n)), то есть время выполнения алгоритма логарифмически зависит от длины массива входных данных. Остальные рассмотренные алгоритмы имеют сложность O(n), то есть зависимость времени выполнения от длины массива линейна. Следовательно, время выполнения древовидного алгоритма при росте длины массива входных данных растёт значительно медленнее, чем время выполнения остальных алгоритмов, что делает предпочтительным его применение для решения поставленной задачи.

4. Дополнительная оптимизация программной реализации алгоритма

Разберём выбранный алгоритм подробнее. После каждой итерации, кроме последней, образуются промежуточные суммы. Эти суммы необходимо хранить в памяти до следующей итерации, для которой они являются входными данными и снова попарно суммируются (рисунок 3). В этом случае можно выделить количество памяти, необходимое для первой итерации. На каждой следующей итерации требуется меньшее количество памяти, значит можно снова использовать выделенную для первой итерации память. Так как обращения к этой памяти происходят многократно, можно выделить её в разделяемой памяти, доступ к которой занимает меньше времени, чем доступ к глобальной памяти. Загрузку векторов в разделяемую память нужно провести не позже первой итерации.

Так как суммируются произведения элементов векторов $\{u_i\}$ и $\{Q(t)\}$, необходимо определить, на каком этапе вычислять эти произведения. Если делать это при загрузке элементов векторов в разделяемую память, это позволит избежать лишнего вызова kernel-функции и необходимости хранить в памяти массив произведений, который не нужен в дальнейших вычислениях.

Пример кода:

```
__shared__ T sdata[n/2];       // выделение разделяемой памяти
/*
Вычисление индекса i на основе номера потока и номера блока
*/
sdata[i] += ui[i] * Q[i] + ui[i+1] * Q[i+1];
```

5. Тестирование программы

Для тестирования программы была выбрана конечно элементная модель части 3-х координатного металлорежущего станка лёгкого класса, содержащая портал, двигатель, переходную пластину, рабочий инструмент и вал (рисунок 4). 3-х мерная модель содержит $6.5 * 10^5$ узлов или $1.95 * 10^6$ степеней свободы. Последний параметр показывает размерность векторов собственных форм.

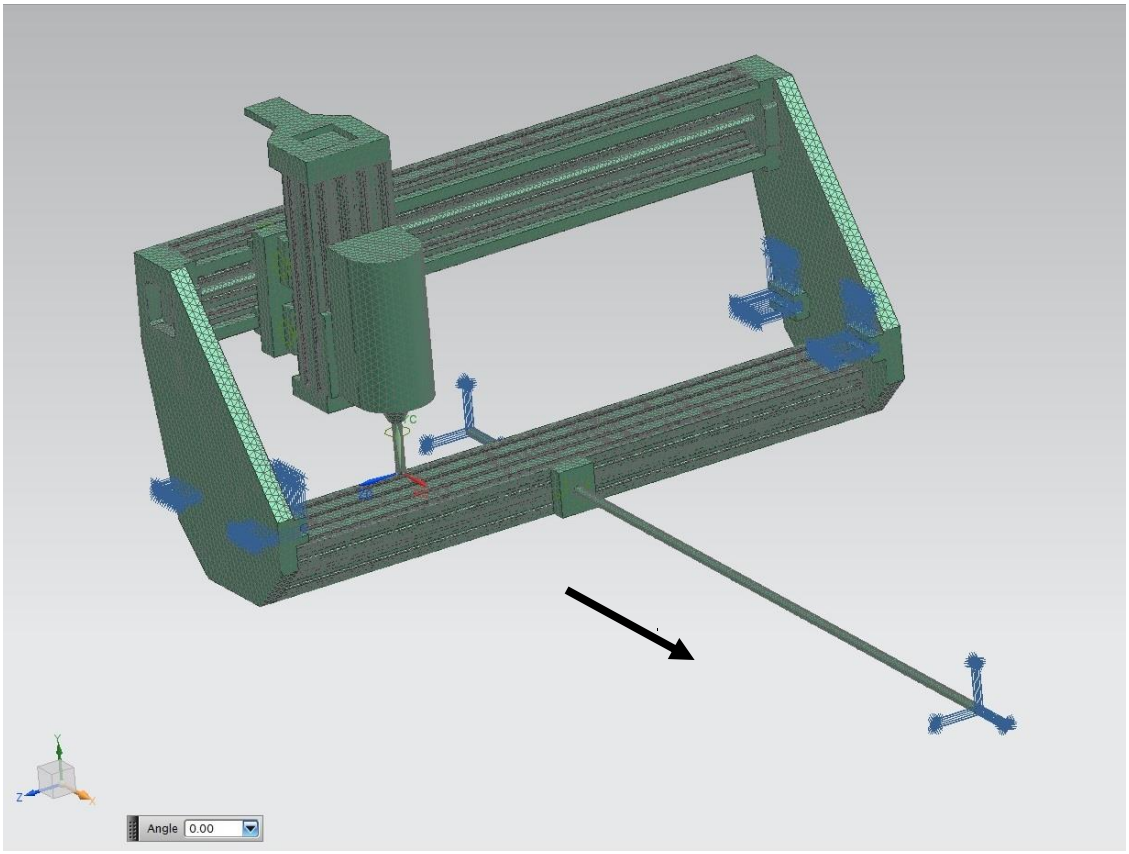

Рисунок 4. *Модель для тестирования (разработано автором)*

В тестовой задаче производится поиск перемещений конца инструмента при разгоне-торможении подвижной части станка. Подвижная часть закреплена по осям Y и Z и может перемещаться вдоль оси x. Ко всем частям приложено единичное ускорение в направлении оси X (рисунок 4). Система нагружена силами инерции. Моделирование движения системы производилось в течение 0.2 секунд. Численное интегрирование производилось с шагом по времени 0.0001 секунды. С помощью авторской программы PSE предварительно было найдено 15 собственных форм. Полученные перемещения модели в момент наибольшего отклонения представлены на рисунке 5. Перемещения конца инструмента показаны на рисунке 6.

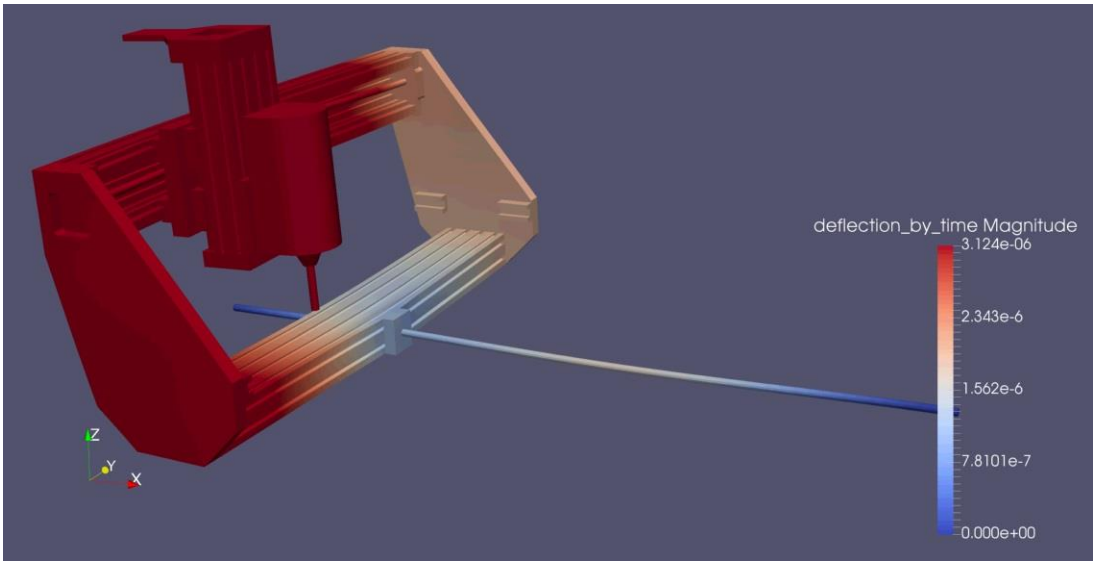

Рисунок 5. *Перемещения модели в программе ParaView [14] (разработано автором)*

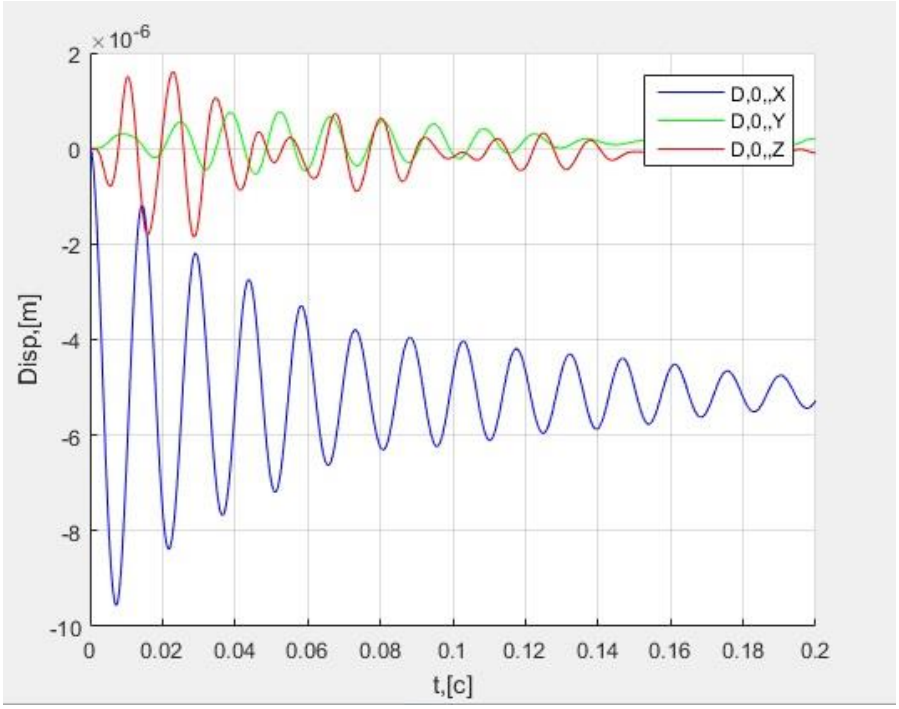

Рисунок 6. *Графики перемещения конца инструмента (разработано автором)*

Для анализа эффективности предпринятых оптимизаций задача была рассчитана с помощью последовательного алгоритма на процессоре (Intel Core i7-5930K 3.50 Ггц, 12 ядер). Было измерено время вычисления $\{q(t)\}$ - 104.032 мс на одном шаге по времени. Измерение времени производится с помощью встроенной в C++ функции clock(). Затем были произведены замеры времени вычисления $\{q(t)\}$ с помощью алгоритма на основе дерева на двух GPU с разным количеством мультипроцессоров и ядер. При этом время вычисления на процессоре взято за эталонное, относительно него вычисляется повышение производительности. Результаты приведены в таблице 1.

Таблица 1

Результаты тестов программы

GPU	Количество мультипроцессоров, шт.	Количество ядер, шт.	Время работы функции, мс	Прирост производительности, %
NVIDIA GeForce GTX 580	16	512	2.921	3462
NVIDIA GeForce GTX 780 Ti	15	2880	0.965	10680

Заключение

Внедрение технологии CUDA позволило повысить более чем в 100 раз производительность вычисления вектора модальных сил в рамках алгоритма интегрирования уравнений движения конечно-элементной модели методом разложения по собственным формам колебаний. Выбранный древовидный алгоритм вычисления модальных сил позволяет наиболее эффективно использовать возможности GPU при реализации общего случая интегрирования уравнений движения методом разложения по собственным формам. Полученное увеличение производительности расчета не может быть получено стандартными средствами организации многопоточных вычислений на CPU из-за небольшого количества их ядер по сравнению с GPU. Платформа параллельных вычислений CUDA является удобным средством для написания высокопроизводительных и масштабируемых программ, а также примером успешной реализации концепции GPGPU.

ЛИТЕРАТУРА

1. Fung, et al., Mediated Reality Using Computer Graphics Hardware for Computer Vision // Proceedings of the International Symposium on Wearable Computing 2002 (ISWC2002), 7–10 October 2002, P. 83–89.
2. Thompson C.J. Using modern graphics architectures for general-purpose computing: a framework and analysis // Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture. P. 306-317.
3. Buck I. et al. Brook for GPUs: stream computing on graphics hardware // Proceedings of ACM SIGGRAPH. 2004. P. 777-786.
4. Flynn M.J. Very high-speed computing systems // Proceedings of the IEEE. 2005. Vol. 54(12). P. 1901 – 1909. DOI:10.1109/PROC.1966.5273.
5. Sanders J., Kandrot E. CUDA by Example: An Introduction to General-Purpose GPU Programming // Addison-Wesley Professional. 2010.
6. Еременко С.Ю. Методы конечных элементов в механике деформируемых тел. // Харьков: Изд-во "Основа" при Харьк. ун-те, 1991.
7. Nguyen H. GPU Gems 3 // Addison-Wesley. 2008. ISBN: 9780321545428.

8. Суравикин А.Ю. Реализация метода SPH на CUDA для моделирования несжимаемых жидкостей // Интернет-журнал «НАУКА и ОБРАЗОВАНИЕ» # 07, июль 2012 http://technomag.neicon.ru/index.html (доступ свободный). Загл. с экрана. Яз. рус., англ. DOI.

9. Weninger F., Bergmann J., Schuller B. Introducing CURRENNT: The Munich Open-Source CUDA RecurREnt Neural Network Toolkit // Journal of Machine Learning Research 16 (2015) P. 547-551.

10. Зенкевич О. Метод конечных элементов в технике // М: МИР. 1975. 541 с.

11. Bathe K-J. Finite element procedures // New Jersey: Prentice Hall, 1996. 1037 p.

12. Sipser M. Introduction to the Theory of Computation // Course Technology Inc. 2005. ISBN:053494728X.

13. Harris M. Using Shared Memory in CUDA C/C++ // Nvidia: сайт. https://devblogs.nvidia.com/parallelforall/using-shared-memory-cuda-cc/ (доступ свободный). Загл. с экрана. Яз. англ.

14. Ayachit, Utkarsh, The ParaView Guide: A Parallel Visualization Application, Kitware, 2015, ISBN 978-1930934306.

15. Берчун Ю.В., Киселёв И.А., Хахалин А.С., Сычёва Е.А., Петрова Т.Д., Яблоков В.Е. Применение технологии CUDA в задаче об определении матрицы жесткости // Интернет-журнал «НАУКА и ОБРАЗОВАНИЕ» # 07, июль 2015 http://technomag.neicon.ru/index.html (доступ свободный). Загл. с экрана. Яз. рус., англ. DOI.

Tsyganov Dmitry Leonidovich
Bauman Moscow state technical university, Russia, Moscow
E-mail: dmitrytsgn@gmail.com

Kiselev Igor Alekseevich
Bauman Moscow state technical university, Russia, Moscow
E-mail: i.a.kiselev@yandex.ru

Zavarzin Denis Alekseevich
Bauman Moscow state technical university, Russia, Moscow
E-mail: zavarzin.den@bk.ru

Using Nvidia CUDA in numerical simulation of elastic bodies vibrations via modal superposition method

Abstract. The proposed research is devoted to the numerical simulation of the finite element model vibrations via modal superposition method by using graphical processing units. A brief historical background of GPGPU and NVIDIA CUDA was given. Key architecture properties of CUDA-Enabled NVIDIA GPUs were described. Main features of NVIDIA CUDA thread and memory hierarchies were mentioned. Resource-intensive procedures of reviewed method were identified. Possibility of parallel computing implementation in these procedures was examined. Several possible thread organization algorithms were described. The most optimal algorithm was selected. Criteria of algorithm selection was given. Selection of tree-based parallel reduction was explained. Additional program optimizations of the algorithm implementation were described, including taking advantage of CUDA GPU architecture features. Program implementation of the algorithm was written in CUDA C programming language. Finite element model intended for program testing was designed. Both parallel GPU algorithm and CPU concurrent algorithm implementations were tested. Run-time of both implementations was measured. Performance boost was calculated. Conclusions about selected algorithm and parallel calculation technology were made.

Keywords: graphical processor; CUDA; parallel computations; eigen vectors; multithreading; parallel reduction; thread hierarchy; optimizations; finite elements

REFERENCES

1. Fung, et al., Mediated Reality Using Computer Graphics Hardware for Computer Vision // Proceedings of the International Symposium on Wearable Computing 2002 (ISWC2002), 7–10 October 2002, P. 83–89.
2. Thompson C.J. Using modern graphics architectures for general-purpose computing: a framework and analysis // Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture. P. 306-317.
3. Buck I. et al. Brook for GPUs: stream computing on graphics hardware // Proceedings of ACM SIGGRAPH. 2004. P. 777-786.
4. Flynn M.J. Very high-speed computing systems // Proceedings of the IEEE. 2005. Vol. 54(12). P. 1901 – 1909. DOI:10.1109/PROC.1966.5273.
5. Sanders J., Kendrot E. CUDA by Example: An Introduction to General-Purpose GPU Programming. Addison-Wesley Professional, 2010. 312 p.

6. Eremenko S.Yu. Metody konechnykh elementov v mekhanike deformiruemykh tel [Finite elements in deformable bodies mechanics]. Kharkiv, Osnova Publ., 1991. 272 p. (in Russian).

7. Nguyen H. GPU Gems 3 // Addison-Wesley. 2008. ISBN: 9780321545428.

8. Suravikin A.Yu. Incompressible fluid simulation on CUDA using SPH method // Internet-zhurnal «NAUKA I OBRAZOVANIE» #7 july 2012 http://technomag.neicon.ru/index.html (dostup svobodnyy). Zagl. s ekrana. Yaz. rus., angl. DOI.

9. Weninger F., Bergmann J., Schuller B. Introducing CURRENNT: The Munich Open-Source CUDA RecurREnt Neural Network Toolkit // Journal of Machine Learning Research 16 (2015) P. 547-551.

10. Zienkiewicz O.C. Finite Element Method in Engineering Science // M. MIR. 1975. 541 p. (in Russian).

11. Bathe K-J. Finite element procedures // New Jersey: Prentice Hall, 1996. 1037 p.

12. Sipser M. Introduction to the Theory of Computation // Course Technology Inc. 2005. ISBN:053494728X.

13. Harris M. Using Shared Memory in CUDA C/C++ // Nvidia: site. https://devblogs.nvidia.com/parallelforall/using-shared-memory-cuda-cc/ (dostup svobodnyy). Zagl. s ekrana. Yaz. rus., angl.

14. Ayachit, Utkarsh, The ParaView Guide: A Parallel Visualization Application, Kitware, 2015, ISBN 978-1930934306.

15. Yu.V. Berchun1, I.A. Kiselev1, A.S. Hahalin1, E.A. Sycheva1, T.D. Petrova1, V.E. Yablokov. Using CUDA Technology for Defining the Stiffness Matrix in the Subspace of Eigenvectors // Internet-zhurnal «NAUKA I OBRAZOVANIE» #7 july 2015 http://technomag.neicon.ru/index.html (dostup svobodnyy). Zagl. s ekrana. Yaz. rus., angl. DOI.