**КОМПЬЮТЕРНЫЕ ИССЛЕДОВАНИЯ
И МОДЕЛИРОВАНИЕ 2014 Т. 6 № 1 С. 79–97**

Ки&М

ЧИСЛЕННЫЕ МЕТОДЫ И ОСНОВЫ ИХ РЕАЛИЗАЦИИ

УДК: 519.6

Параллельная реализация конечно-элементных алгоритмов на графических ускорителях в программном комплексе FEStudio

С. П. Копысов[a], И. М. Кузьмин[b], Н. С. Недожогин[c], А. К. Новиков[d], В. Н. Рычков[e], Ю. А. Сагдеева[f], Л. Е. Тонков[g]

Институт механики УрО РАН,
Россия, 426067, г. Ижевск, ул. Т. Барамзиной, 34

E-mail: [a] s.kopysov@gmail.com, [b] imkuzmin@gmail.com, [c] Nedozhogin@inbox.ru, [d] sc_work@mail.ru, [e] bob.r@mail.ru, [f] sagdeeva@yandex.ru, [g] tnk@udman.ru

Получено 26 ноября 2013 г.

Рассматриваются новые подходы и алгоритмы распараллеливания вычислений метода конечных элементов, реализованные в программном комплексе FEStudio. Представлена программная модель комплекса, позволяющая расширять возможности распараллеливания на различных уровнях вычислений. Разработаны параллельные алгоритмы численного интегрирования динамических задач и локальных матриц жесткости, формирования и решения систем уравнений с использованием модели параллелизма данных CUDA.

Ключевые слова: метод конечных элементов, параллельные алгоритмы, гибридные вычислительные системы, объектно ориентированное программирование

Parallel implementation of a finite-element algorithms on a graphics accelerator in the software package FEStudio

S. P. Kopysov, I. M. Kuzmin, N. S. Nedozhogin, A. K. Novikov, V. N. Rychkov, Y. A. Sagdeeva, L. E. Tonkov

Institute of Mechanics UB RAS, 34 T. Baramzinoy st., Izhevsk, 426067, Russia

Abstract. — In this paper, we present new parallel algorithms for finite element analysis implemented in the FEStudio software framework. We describe the programming model of finite element method, which supports parallelism on different stages of numerical simulations. Using this model, we develop parallel algorithms of numerical integration for dynamic problems and local stiffness matrices. For constructing and solving the systems of equations, we use the CUDA programming platform.

Keywords: finite element method, parallel computing, hybrid HPC platforms, object-oriented programming

Citation: *Computer Research and Modeling*, 2014, vol. 6, no. 1, pp. 79–97 (Russian).

Работа выполнена при финансовой поддержке РФФИ (грант 14-01-00055-а, 13-01-00101-а, 14-01-31066-мол_а) и программы Президиума РАН № 18 при поддержке УрО РАН (проект 12-П-1-1005).

Введение

Применение сложных конечно-элементных моделей на практике сдерживается значительной трудоемкостью их программной параллельной реализации. Для того чтобы ускорить и усовершенствовать процесс создания расчетных программ, необходимо разработать пакет прикладных программ для конечно-элементного анализа — программную среду, в основе которой лежит модель конечных элементов и которая позволяет разработчику сосредоточить внимание непосредственно на создании своей модели, используя для решения указанных общих задач уже готовые процедуры. При создании параллельных программ, основанных на модели конечных элементов, можно применить технологию объектно ориентированного моделирования, в результате чего получаются объектно ориентированная модель конечных элементов, соответствующая математической модели, и ее программная реализация в виде объектно ориентированного кода [Рычков и др., 2002].

Одним из основных подходов к распараллеливанию решения многомерных дифференциальных уравнений по праву считаются методы декомпозиции области [Копысов и др., 2003а]. Большинство программных реализаций методов декомпозиции области построены на основе того или иного метода аппроксимации дифференциальной задачи, чаще всего на основе метода конечных элементов (МКЭ). В данной работе предлагается фундаментальная система объектов, реализованная в программном комплексе FEStudio (`https://www.udman.ru/projects/FEStudio`), общая для всех методов декомпозиции области, далее она расширяется путем добавления новых объектов, реализующих специфические, в том числе и параллельные, алгоритмы.

Параллельные вычисления на графических процессорах (GPU) позволяют уменьшить время решения в десятки раз, но требуют разработки новых алгоритмов и программного обеспечения. В последнее время основное внимание уделяется алгоритмам решения на GPU систем линейных алгебраических уравнений. Вместе с тем в конечно-элементном анализе при решении двумерных и особенно трехмерных задач актуально распараллеливание процесса формирования сеточных систем уравнений, который связан с численным интегрированием по объему и поверхности конечных элементов (локальные матрицы жесткости и масс, векторы распределенной нагрузки). В методе конечных элементов генерируется множество плотных матриц жесткости элемента. Матрицы жесткости формируются и хранятся различными способами: поэлементно, пореберно и в глобальной матрице жесткости без нулевых элементов. Введение той или иной схемы вычисления матрично-векторного и скалярного произведения в итерационных методах решения систем линейных уравнений позволяет выбрать уровень распараллеливания при одном и том же разделении области на разных гибридных архитектурах, содержащих центральные процессоры и графические ускорители.

Работа построена следующим образом: в первом параграфе приводится трехуровневая объектно ориентированная модель метода конечных элементов, включающая классы расчетных моделей, численные классы и классы решения, и рассматривается модель метода декомпозиции области, основанная на конечно-элементной аппроксимации. Второй параграф посвящен развитию модели, включающей параллельные алгоритмы и классы решения на гибридных вычислительных системах. В третьем параграфе рассматриваются численные классы и алгоритмы численного интегрирования, формирования и решения систем линейных алгебраических уравнений с использованием графических ускорителей. Приводятся оценки параллельного ускорения и эффективности использования параллельных алгоритмов. Полученные результаты и сделанные выводы обобщаются в заключении.

Метод декомпозиции области, основанный на конечно-элементной аппроксимации

Программная реализация FEStudio представлена трехуровневой объектно ориентированной моделью метода конечных элементов, включающей классы расчетных данных, численные классы и классы решения [Рычков и др., 2002].

Функциональная модель МКЭ в FEStudio (рис. 1) описывает: построение модели решения `AnalysisModel` по конечно-элементной сетке, определяемой классом `FEMDomain`; введение граничных условий посредством класса `ConstraintHandler`; упорядочивание степеней свободы с помощью класса `DOFNumberer`; формирование классом `Integrator` и решение классом `Solver` системы уравнений `SOE`.

Классы расчетной модели описывают дискретное представление задачи. Класс `FEMDomain` содержит методы для создания и редактирования расчетной области, представленной классом, который является контейнером для геометрии с расчетными данными и конечно-элементной сетки, записанной с помощью узлов (`Node`), рёбер (`Edge`), многоугольников (`Polygon`), элементов (`Element`) различных видов, граничных условий. Вопросы распараллеливания алгоритмов данной расчетной модели в пакете FEStudio рассмотрены в работах авторов [Kopyssov, Novikov, 2001; Копысов, Новиков, 2002; Копысов и др., 2008; Копысов и др., 2007].

Классы решения используются для анализа КЭ расчетной модели и решения задачи. Класс решения `Analysis` — это сборка основных шагов решения, которые выполняются под управлением класса алгоритма решения `SolutionAlgorithm`. На некоторых из них остановимся далее.

Численные классы представляют численные методы, применяемые в МКЭ. Класс системы линейных алгебраических уравнений `LinearSOE` (рис. 1) содержит матрицу, правую часть и решение; включает следующие основные методы: добавление блока в систему и введение в систему граничных условий. Потомки `LinearSOE` соответствуют системам с разными видами матриц (ленточными, профильными, CSR и т. д.). Класс метода решения системы линейных алгебраических уравнений `LinearSolver` отвечает за решение системы. Потомки этого класса позволяют решать системы уравнений различными методами (предобусловленным методом сопряженных градиентов, LU-разложение и др.). Примеры классов данной модели будут более подробно рассмотрены далее.

Определим основные шаги, общие для всех методов декомпозиции шагов области, основанных на конечно-элементной аппроксимации [Копысов и др., 2003а]:

Шаг 1. Построение **и разделение** конечно-элементной сетки с заданными физическими параметрами и граничными условиями.

Шаг 2. Установка соответствия между узловыми степенями свободы и номерами уравнений **в локальных и глобальной** системах уравнений.

Шаг 3. Формирование **локальных (и глобальной)** систем уравнений с учетом вкладов от элементов и узлов в соответствии со схемами сборки.

Шаг 4. Введение граничных условий.

Шаг 5. Решение системы уравнений **с независимым выполнением матрично-векторных операций на подобластях**.

Шаг 6. Обновление узловых степеней свободы в соответствии с полученным решением.

Шаг 7. Определение расчетных параметров в элементах.

Основные шаги метода декомпозиции области незначительно отличаются от характерных операций метода конечных элементов, поэтому целесообразно представить объектно ориентированную модель метода декомпозиции как расширение модели конечных элементов.

При переходе от объектно ориентированной модели метода конечных элементов к модели метода декомпозиции области использовались все механизмы объектно ориентированного программирования, при этом первичным требованием было повторное использование кода (наследование) с его минимальной модификацией (полиморфизм), после чего определялись взаимосвязи новых классов между собой, а также с уже существующими.

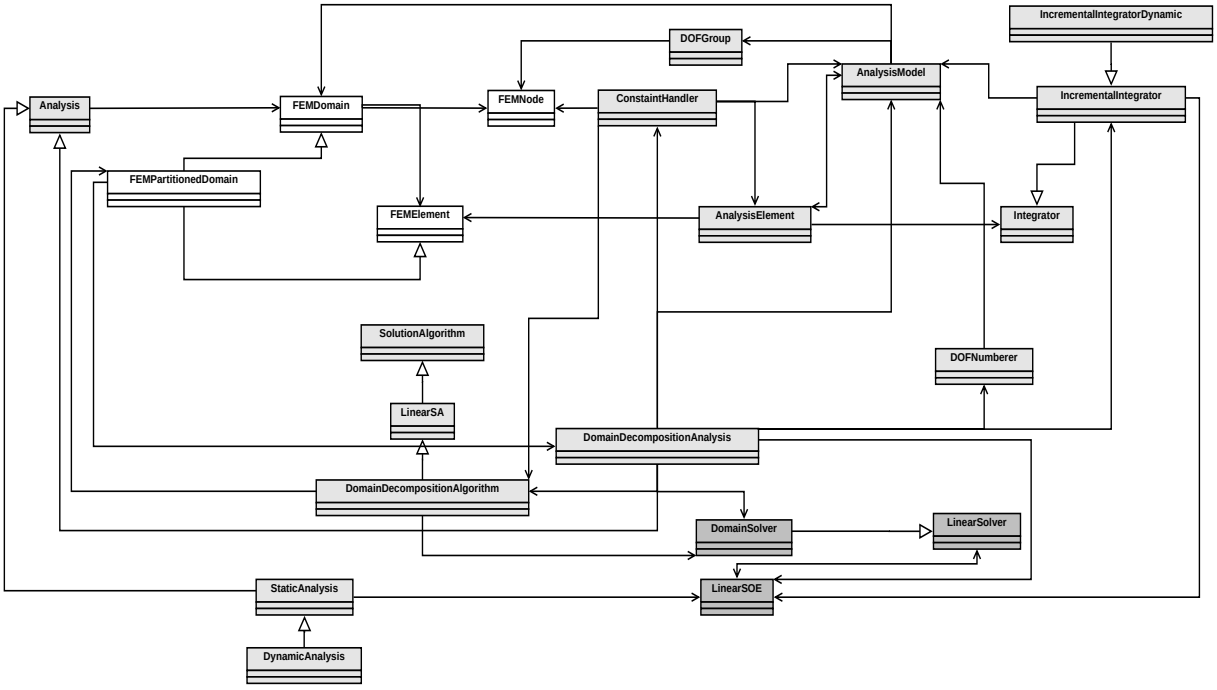

Рис. 1. Объектная модель метода конечных элементов (□ — классы расчетных данных, ■ — численные классы, ■ — классы решения)

Реализованная в FEStudio модель метода декомпозиции области является расширением всех подсистем модели метода конечных элементов для области и подобластей [Копысов и др., 2003b]. В расчетную модель области введен класс разделителя области `DomainPartitioner`, который отвечает за формирование графа, соответствующего конечно-элементной сетке, за его разделение и распределение объектов сетки между подобластями. От метода формирования графа (граф узлов или граф элементов) зависит, какой алгоритм лучше применить для его разделения и по какому принципу нужно перераспределить объекты между подобластями (все эти детали определены в потомках этих классов). Расчетная модель расширена классом подобласти `FEMPartitionedDomain`. В численную модель добавлен класс решателя в подобласти `DomainSolver`, который отвечает за выполнение численных операций в подобласти для того или иного метода декомпозиции. Их наборы конкретизируются в потомках этого класса; например, для метода подструктур в классе `GenSubstructureSolver` определены методы формирования блока для дополнения Шура [Копысов и др., 2012]. Из соображений эффективности для некоторых потомков решателей на подобласти введены классы систем линейных алгебраических уравнений специального вида. Класс системы уравнений для метода подструктур `SubstructureSOE`, например, содержит систему в разобранном виде, по блокам, соответствующим внутренним и граничным степеням свободы. В модель решения

подобласти включены классы метода декомпозиции `DomainDecompositionAnalysis` и алгоритма `DomainDecompositionAlgorithm`. Метод класса решения ассоциирован с подобластью и является контейнером всех объектов решения. Алгоритм управляет объектами, собранными в методе, для обновления состояния подобласти.

Построение моделей метода декомпозиции области

Среди основных шагов метода декомпозиции области определяющими являются ***шаги 1 и 5***, поэтому рассмотрим, как они формализованы в объектно ориентированной модели FEStudio.

Разделение области. Для разделения области введены три класса: разделитель области, разделитель графа и балансировщик нагрузки. Первый класс отвечает за формирование графа по данным конечно-элементной сетки области и перераспределение сеточных объектов между подобластями в соответствии с разделенным графом; второй — обеспечивает подключение различных алгоритмов разделения графов, третий — предусмотрен для статической и динамической балансировки вычислительной нагрузки путем обработки весов графа подобластей.

Сетку удобно представить в виде графа связности элементов и применить какой-либо алгоритм разделения графов. Объект класса разделителя области `DomainPartitioner` строит граф элементов и делит его, используя тот или иной алгоритм разделения графов. Для разделения графа использовалось несколько алгоритмов [Копысов, Новиков, 2002]. Класс `FEMPartitionedDomain` расширяет интерфейс `FEMDomain` для выделения внутренних и граничных узлов подобласти. Потомки классов `FEMPartitionedDomain` предназначены для разделения как на непересекающиеся, так и на пересекающиеся подобласти. В аддитивных методах Шварца, например, удобно работать с объектами пересечений синхронно, в таком случае объекты пересечений создаются в одном экземпляре, а в мультипликативных — асинхронно, в подобластях `FEMPartitionedDomain` создаются свои копии объектов пересечений, которые рассчитываются независимо и периодически синхронизируются.

Подобласть как область в расчетной модели и элемент в модели решения

В объектно ориентированной модели метода декомпозиции [Копысов и др., 2003а] подобласть рассматривается одновременно как область с определенной геометрией, материальными характеристиками и нагрузками и как элемент — суперэлемент — со степенями свободы, в том числе соответствующими граничным условиям. Суперэлемент отвечает за статическую конденсацию, проводимую над расчетными данными подструктуры, и за выполнение ряда матрично-векторных операций в подобластях.

Представление подобласти как области и элемента приводит к тому, что при построении объектно ориентированной модели метода декомпозиции целесообразно дважды использовать модель метода конечных элементов: на уровне подобласти и на уровне разделенной области.

Таким образом, построена модель метода декомпозиции области, основанного на конечно-элементной аппроксимации, которая обеспечивает подключение различных алгоритмов разделения области, основанных на графовом представлении конечно-элементной сетки. Представление подобласти классом расчетной модели и модели решения позволило повторно использовать структуру классов метода конечных элементов в области и в подобластях. Модель метода декомпозиции позволяет быстро реализовать сложные алгоритмы, например методы декомпозиции с адаптивным уточнением [Kopyssov, Novikov, 2001]. Данная модель не является окончательной — предполагается ее постепенное уточнение, а также расширение иерархии классов.

Для вычислений на гибридных архитектурах (CPU + GPU) с переносом части вычислений на графические процессоры в программном комплексе FEStudio используется технология

параллельного программирования CUDA. С применением данной технологии разработаны библиотеки матричных вычислений (CUSMA, CUda Sparse Matrix Algebra) и численного интегрирования конечно-элементных матриц (CUNInt, CUda Numerical Integration). Методы соответствующих классов FEStudio (решения систем уравнений, формирования матриц) вызывают функции из данных библиотек как внешние (`extern`) функции. При использовании графических ускорителей не всегда удобны структуры данных объектно ориентированной модели FEStudio, которые применяются для расчета на центральном процессоре. В этом случае перед использованием библиотек, реализованных на CUDA, данные преобразуются в структуры, адаптированные для вычислений на GPU. Некоторые параллельные алгоритмы, реализованные в рамках данной модели, представлены в следующих разделах.

Параллельные классы решения

Одним из примеров использования сложных конечно-элементных моделей, требующих распараллеливания вычислений, является моделирование процессов динамического деформирования.

При решении задач динамики особенно актуальным представляется поиск оптимальных схем интегрирования уравнений динамического деформирования тел и конструкций в условиях геометрически и физически нелинейного поведения. Здесь необходимо отметить, что традиционно такие постановки используют неявные схемы интегрирования типа Ньюмарка и модифицированную отсчетную конфигурацию, что приводит к решению нелинейных систем большого порядка на каждом временном шаге решения задачи, и требуют перерасчета пространственных производных в каждый момент времени.

Решение динамических задач теории упругости

Рассмотрим динамическую задачу теории упругости для тела, занимающего область $\Omega \in \mathbb{R}^d$ с границей $\Gamma = \Gamma_1 \bigcup \Gamma_2$ и находящегося под действием объемных сил $f : \Omega \times \mathbb{R} \to \mathbb{R}^d$, зависящих от времени t. В отсчетной лагранжевой формулировке уравнения движения в общем случае принимают вид

$$\rho_0 \frac{\partial^2 u}{\partial t^2} = \nabla_0 \cdot (F \cdot S) + \rho_0 f, \qquad u(t_0) = u^0, \qquad \dot{u}(t_0) = \dot{u}^0, \qquad n \cdot (F \cdot S) = p, \tag{1}$$

где u — вектор перемещений; ρ_0, ∇_0 — плотность и оператор дивергенции в отсчетной конфигурации; f, p — векторы массовых и поверхностных сил; F — градиент деформации; $S = F^{-1} \cdot \sigma \cdot F^{-T} detF$ — симметричный тензор напряжений Пиола–Кирхгоффа; здесь σ — тензор напряжений Коши. Для простоты запишем, что определяющие соотношения имеет вид $S = \lambda(trE)I + 2\mu E$, где $E = 1/2(\nabla u + (\nabla u)^T + \nabla u \cdot (\nabla u)^T)$ — тензор деформаций Грина-Лагранжа, а λ, μ — упругие постоянные.

Аппроксимируя уравнения движения (1) методом конечных элементов, получим систему уравнений

$$M\ddot{u}(t) + D\dot{u} + Ku(t) = R(t), \tag{2}$$

где M, D, K — глобальные матрицы масс, демпфирования, жесткости; R — вектор нагрузок.

Для решения задач нелинейной динамики с учетом физической и геометрической нелинейности в пакете используются отсчетная лагранжева формулировка уравнений движения и явная центрально-разностная схема для аппроксимации во времени с диагонализацией матриц масс и демпфирования:

$$^{t+\Delta t}u_i = a_0(^tR_i - {}^tG_i) + {}^tu_i a_1 + {}^{t-\Delta t}u_i a_2,$$

здесь $a_0 = 1/(D_{ii}/(2 \triangle t) + M_{ii}/(\triangle t)^2)$; $a_1 = 2a_0M_{ii}/(\triangle t)^2$; $a_2 = a_0D_{ii}/(2 \triangle t) - a_1/2$; ${}^tR_i$ — вектор объемных и поверхностных сил; ${}^tG = \int_{{}_0V} {}^t_0B^T_{L_0} {}^t\hat{S}\, d^0V$ — вектор внутренних сил, в котором используются вектор ${}^t_0\hat{S} = ({}^t_0S_{11}, {}^t_0S_{22}, {}^t_0S_{33}, {}^t_0S_{12}, {}^t_0S_{23}, {}^t_0S_{33})^T$ и матрица градиентов на текущем шаге вида ${}^t_0B_{L_0} = {}_0B_{L_0} {}^t_0F^T$. Значительным преимуществом оказывается то, что все производные по пространственным координатам вычисляются к исходной конфигурации и, следовательно, могут быть предварительно получены и рассматриваемый алгоритм будет менее затратным по числу операций. Явная центрально-разностная схема, кроме того, позволяет достаточно быстро изменять и использовать различные определяющие соотношения для гиперупругих материалов.

Учет нелинейностей в этом случае производится без итераций, необходимых на каждом шаге по времени, других алгоритмов. Кроме того, исключается операция сборки глобальной матрицы жесткости, а вклады в вектор узловых сил могут быть вычислены на уровне отдельных конечных элементов. Вместо решения системы уравнений выполняются матричные операции, которые эффективно распараллеливаются и тем самым позволяют существенно сократить вычислительные затраты на каждом шаге по времени.

Алгоритм 1 Вариант параллельного алгоритма с явной схемой и операциями на уровне конечных элементов

1: Предварительно вычисляются: ${}_oB^{e^T}_{L_o}$ — матрица градиентов; матрицы масс M^e; коэффициенты a_0,a_1,a_2. {Параллельно в цикле по элементам с помощью OpenMP в несколько потоков.}
2: **while** $t + \Delta t < t_*$ **do**
3: **while** $e < m$ **do**
4: Вычисление градиента деформации ${}^tF^e$.
5: Вычисление вектора внешних сил ${}^tR^e$.
6: Переопределение текущей матрицы градиента ${}^t_0B^e_{L_0} = {}_0B^e_{L_0} {}^t_0F^{e^T}$.
7: Вычисление напряжений ${}^t\hat{S}^e$.
8: Вычисление вклада внутренних напряжений ${}^tG^e = \int_{{}_0V^e} {}^t_0B^{e^T}_{L_0} {}^t\hat{S}^e d^0V^e$.
9: $e \leftarrow e + 1$
10: **end while**
11: Сборка элементных векторов ${}^tG = \sum_{e=1}^m C_e^T {}^tG^e$, ${}^tR = \sum_{e=1}^m C_e^T {}^tR^e$.
12: Вычисление вектора перемещений ${}^{t+\Delta t}u_i = a_0({}^tR_i - {}^tG_i) + {}^tu_ia_1 + {}^{t-\Delta t}u_ia_2$.
13: **end while**

Параллельный **алгоритм** 1 решения задачи нелинейного динамического деформирования позволяет выполнять многие операции независимо, на уровне отдельных конечных элементов до основного этапа решения задачи, и не требует сборки глобальных матриц и обращения матрицы жесткости, что обеспечивает сокращение числа операций на 30–40 %, уменьшение затрат памяти, независимость вычислений на уровне отдельных конечных элементов и высокую параллельную эффективность для реализации на GPU-архитектуре.

Для решения динамических задач в линейной постановке также используется распараллеленный метод Ньюмарка [Кузьмин и др., 2012].

Наиболее трудоемкая операция алгоритма Ньюмарка — матрично-векторные произведения, которые применяются как в основном алгоритме, так и на каждом шаге метода сопряженных градиентов для решения системы (3).

Рассмотрим основные возможности распараллеливания алгоритма на n_p потоков с учетом совместного использования многоядерных CPU и графического ускорителя GPU.

1. Формирование глобальных матриц жесткости и масс. Распараллеливание цикла по конечным элементам осуществляется с помощью OpenMP:

$$K = \sum_{e=1}^{m} C_e^T K^e C_e, \qquad M = \sum_{e=1}^{m} C_e^T M^e C_e.$$

2. Матрично-векторное произведение при вычислении эффективной нагрузки выполняется на каждом временном шаге на уровне отдельных конечных элементов в цикле по элементам ($a_0 = 1/(\alpha(\Delta t)^2)$, $a_2 = 1/(\alpha\Delta t)$, $a_3 = 1/(2\alpha) - 1$, $\alpha \geqslant 0.25(0.5 + \delta)^2$, $\delta \geqslant 0.5$:

$$^{t+\Delta t}\hat{R} = \sum_{e=1}^{m} C_e^T \Big({}^{t+\Delta t}R^e + M^e(a_0\, {}^t u + a_2\, {}^t\dot{u} + a_3\, {}^t\ddot{u}) \Big) C_e.$$

3. Решение системы уравнений

$$\hat{K}\, {}^{t+\Delta t}u = {}^{t+\Delta t}\hat{R} \tag{3}$$

осуществляется методом сопряженных градиентов с разными вариантами реализаций матрично-векторных и скалярных произведений на OpenMP и CUDA на:

(a) GPU с помощью CUDA;

(b) CPU, с использованием OpenMP для вычисления матрично-векторного произведения $\hat{K}p$, в два потока.

Таким образом, параллельный алгоритм схемы Ньюмарка имеет вид:

Алгоритм 2 Вариант параллельного алгоритма Ньюмарка

1: Формируются матрицы жесткости и масс с использованием OpenMP:

$$K = \sum_{e=1}^{m} C_e^T K^e C_e, \qquad M = \sum_{e=1}^{m} C_e^T M^e C_e.$$

2: Выбираются временной шаг Δt, параметры α и δ, вычисляются постоянные a_0–a_7 в один поток на CPU.

3: Формируется эффективная матрица жесткости $\hat{K}$: $\hat{K} = K + a_0 M$ на CPU.

4: **while** $t + \Delta t < t_*$ **do**

5: Вычисляется эффективная нагрузка с использованием OpenMP:

$$^{t+\Delta t}\hat{R} = \sum_{e=1}^{m} C_e^T \Big({}^{t+\Delta t}R^e + M^e(a_0\, {}^t u + a_2\, {}^t\dot{u} + a_3\, {}^t\ddot{u}) \Big) C_e.$$

6: Решается система (на GPU или в два потока с помощью OpenMP для матрично-векторного произведения МСГ $\hat{K}p_i$ и $\hat{K}^T p_i$):

$$\hat{K}\, {}^{t+\Delta t}u = {}^{t+\Delta t}\hat{R}.$$

7: Вычисляются ускорения и скорости для момента времени $t + \Delta t$ последовательно на CPU.

8: **end while**

Сравнение параллельного ускорения некоторых шагов **алгоритма 2**, вычисляемого как $S = t_1/t_{n_p}$, где t_1 — время выполнения одним потоком CPU, а t_{n_p} на n_p потоках CPU или GPU, представлено в таблице 1. Матрицы $\hat{K}, M$ — хранились с учетом симметрии (верхний треугольник, включая диагональ) в построчно сжатом формате CSR.

Таблица 1. Время выполнения и ускорения **алгоритма 2** для сетки из 67244 тетреэдров, 19890 узлов с системой из уравнений $N = 59670$, с матрицией коэффициентов, содержащей ненулевых элементов $Nnz = 1118694$

Варианты	Формирование K, M t, с	Вычисление $\hat{R}_{t+\Delta t}$ t, с	Решение $\hat{K}u_{t+\Delta t} = \hat{R}_{t+\Delta t}$ t, с	Суммарное ускорение, S
	CPU		OpenMP	
CPU+OpenMP	7.75	4.13	2.23	2.1
	OpenMP		OpenMP	
OpenMP+OpenMP	1.46	2.11	2.22	3.5
	OpenMP		CUDA	
OpenMP+CUDA	1.45	2.11	0.11	10.05

Для решения динамической задачи в рамках объектно ориентированной модели вводятся два класса: `DynamicAnalysis` и `IncrementalIntegratorDynamic`. Первый класс является потомком `Analysis` и отвечает за алгоритм решения задачи. В нем, согласно **алгоритму 2**, реализован метод `Analyze`. Второй класс — потомок `IncrementalIntegrator`, в котором переопределены методы формирования матриц жесткости и векторы правых частей. Кроме того, в класс конечных элементов `FEMElement`, который является базовым для всех конечных элементов, вводится дополнительный метод, отвечающий за формирование вектора правых частей `DynamicRightHandVector`. Реализация этого метода зависит от типа конечного элемента и выполнена в его потомках.

Численные классы

Вычисление локальных матриц жесткости четырехугольных и шестигранных конечных элементов, а также элементов высоких порядков связано с численным интегрированием. Одним из подходов к распараллеливанию численного интегрирования является применение массового параллелизма графических ускорителей. В рамках FEStudio разрабатывается библиотека CUNint, в которой численное интегрирование заданного множества локальных матриц жесткости распараллеливается при помощи технологии CUDA. Интерфейс к данной библиотеке обеспечивает метод `FormTangent` класса `GPUIncrementalIntegrator`, в котором формируется структура данных для передачи на GPU, содержащая координаты узлов, характеристики материала, степени аппроксимирующих полиномов, а также вызывается host-функция `GPUMSInt`, обеспечивающая численное интегрирование на графическом ускорителе. Далее в этом же методе `GPUIncrementalIntegrator::FormTangent` в цикле по конечным элементам проинтегрированные локальные жесткости помещаются во вспомогательный объект класса `numeric::AlgebraUSymMatrix`, в котором отвечает за механизм внесения граничных условий Дирихле (при помощи `AnalysisElement::HandleMatrixConstraints`), после чего методом `LinearSOE::AddToA` локальная матрица жесткости добавляется в матрицу системы уравнений. Соответствие локальных и глобальных степеней свободы определяет метод `get_EquationNumbers` из класса `AnalysisElement`. В случае решения системы уравнений на GPU механизм внесения граничных условий Дирихле и формирования глобальной матрицы жесткости переносится на графический ускоритель.

Для решения СЛАУ на графических ускорителях была разработана и интегрирована в FEStudio библиотека CUSMA. Доступ к функциям библиотеки осуществляется через метод

`Solve` класса `GPUCGSolver` — для решения предобусловленным методом сопряженных градиентов системы с глобальной матрицей жесткости или `GPUElementCGSolver` — для решения в системы в поэлеметной форме без сборки матрицы.

Интегрирование и поэлементные схемы

Необходимость численного интегрирования в методе конечных элементов появляется при вычислении в (2) локальных матриц K^e — жесткости и M^e — масс, поверхностных интегралов.

В общем случае алгоритм формирования локальной матрицы K^e жесткости включает следующие шаги:

- построение функций формы элемента ψ_e;
- переход от глобальных системы координат к локальной, вычисление матрицы Якоби J, обратного преобразования J^{-1} и якобиана $detJ$;
- вычисление частных производных от функций формы элемента для формирования матрицы градиентов, связывающей деформации и перемещения на элементе B^e;
- численное интегрирование по площади (или объему в трехмерном случае) для окончательного вычисления матрицы жесткости.

Численное интегрирование выполняется по естественным координатам $-1 \leqslant r, s, t \leqslant 1$, через которые выражается элемент объема $\det J(r,s,t)drdsdt$, здесь $\det J(r,s,t)$ — определитель матрицы Якоби, связывающей локальные (r,s,t) и глобальные координаты (x,y,z). Локальная матрица жесткости обычно вычисляется как

$$K^e_{\alpha\beta} \approx \sum_{i=1}^{n_i}\sum_{j=1}^{n_j}\sum_{k=1}^{n_k} B^{e\,T}_{\alpha}(r_i,s_j,t_k)D^e B^e_{\beta}(r_i,s_j,t_k)w_i w_j w_k \det J(r,s,t), \tag{4}$$

где $B^e(r,s,t) = B(x,y,z)J^{-1}(r,s,t)\Phi^e(r,s,t)$ — матрица производных функций формы, $B(x,y,z)$ — матрица производных, вид которой зависит от пространственной размерности и дифференциального оператора задачи; $\Phi^e(r,s,t)$ — матрица функций формы конечного элемента e, D^e— матрица характеристик материала (среды); $B^e(r_i,s_j,t_k)$ — матрица производных функций формы в точке интегрирования (r_i,s_j,t_k). Полагая B и D^e равными единичной матрице, получаем выражение для матрицы масс M^e; n_i, n_j, n_k — число точек интегрирования, w_i, w_j, w_k — весовые коэффициенты квадратур Гаусса. Число точек интегрирования при использовании квадратур Гаусса–Лежандра определим как $n_G \geqslant \underbrace{[(3p-2)/2]\times\ldots\times[(3p-2)/2]}_{d}$, где p — максимальная степень полинома в конечном элементе.

Выделим следующие уровни распараллеливания при вычислении локальных матриц жесткости на графическом ускорителе (см. **алгоритм** 3):

I — вычисление матрицы K^e в отдельных параллельных процессах, далее в нитях CUDA;

II — вычисление матриц $K^e(r_i,s_j,t_k)$ отдельными нитями CUDA.

Следует отметить, что матрица K^e в (4) представима в блочном виде, когда блоки матрицы соответствуют определенным узлам сетки. В таком случае каждый блок вычисляется отдельной параллельной нитью (см. **алгоритм** 3, циклы по переменным α и β). Кроме того, возможно параллельное выполнение умножения матриц $B^{e\,T}_{\alpha}D^eB^e_{\beta}$. Результатом работы **алгоритма** 3 является множество локальных матриц жесткости $\{K^e\}$, $e = 1,2,\ldots,m$, или его подмножество (в случае нескольких GPU).

Алгоритм 3 Уровни распараллеливания алгоритма численного интегрирования матриц K^e

```
В CUDA-нити th ∈ [1, m] выполнить: {I уровень}
for i = 1 to n_i do
  for j = 1 to n_j do
    for k = 1 to n_k do
      if levels = 2 then
        В CUDA-нити th ∈ [1, m · n_G] выполнить: {II уровень}
      end if
      Вычисление частных производных ψ_e по r, s, t.
      Вычисление матрицы Якоби J(r_i, s_j, t_k, x_i, y_j, z_k), detJ и J^{-1}.
      Вычисление матрицы производных B^e(x_i, y_j, z_k).
      for α = 1 to N_e do
        for β = α to N_e do
          K^e_{αβ}(x_i, y_j, z_k) = B^{e T}_α(x_i, y_j, z_k) D^e B^e_β(x_i, y_j, z_k) w_i w_j w_k det J.
          K^e_{αβ} ← K^e_{αβ} + K^e_{αβ}(x_i, y_j, z_k).
        end for
      end for
    end for
  end for
end for
```

Параллельный алгоритм, соответствующий уровню распараллеливания I, состоит в одновременном выполнении m копий последовательного алгоритма интегрирования, где m — число конечных элементов. Потенциально этот вариант в m раз быстрее последовательного алгоритма. При реализации, в связи с аппаратными ограничениями на число одновременно выполняемых нитей (размер `warp`-а), использованием арифметики с двойной точностью и необходимостью передачи данных из оперативной памяти в память графического ускорителя достигается ускорение $t_{CPU}/t_{GPU} \ll m$.

Теоретически на уровне II возможно $m \times n_G$ вычислений матриц $K^e(x_i, y_j, z_k)$. Практически хранение такого числа матриц не только затратно по памяти, но и малоэффективно по скорости доступа к данным, которые будут размещены в глобальной памяти ускорителя. Также теоретически этот вариант может выполняться в n_G раз быстрее, чем первый алгоритм, но необходимо обеспечить суммирование n_G матриц (появляются точки синхронизации), и хранение в n_G раз больше промежуточных данных: матрица Якоби и обратная к ней, матрица $B^e(x_i, y_j, z_k)$, что становится критичным при высоких порядках аппроксимации.

Программно уровни распараллеливания **алгоритма** 3 реализованы в виде отдельных `kernel`-функций CUDA, что позволяет управлять размещением данных в памяти ускорителя и группировать нити, излишне не усложняя логику программы.

В случае распараллеливания уровня I необходимо выполнить m CUDA-нитей, каждая из которых вычисляет одну матрицу жесткости и не имеет общих данных с другими нитями. Для этого каждой нити передаются координаты узлов конечного элемента, характеристики материала или среды, порядок аппроксимирующего полинома, число точек интегрирования. Множество локальных матриц жесткости хранится в глобальной памяти графического ускорителя.

Уровень распараллеливания II рассматривается в рамках статической модели с фиксированным числом нитей. В этом случае создаются $m \cdot n_G$ нитей, при этом необходимо обеспечить суммирование матриц K^e, получаемых нитями в разных точках интегрирования. Как показали проведенные исследования, вычисления с использованием атомарной операции суммирования оказались медленнее, чем реализация уровня I, обладающая меньшей степенью параллелизма. Поэтому при реализации уровня II используется разделяемая память ускорителя, а вычисления организуются следующим образом: блоком нитей CUDA параллельно вычисляются элемен-

ты матрицы во всех точках интегрирования одного конечного элемента, каждая нить выполняет вычисления в соответствующей точке интегрирования; для суммирования $K_{\alpha\beta}$ элементы $K^e_{\alpha\beta}(x_i, y_j, z_k)$ помещаются в разделяемую память ускорителя; создается `__shared__` массив (в смысле CUDA) размерности $\min\{2^k | 2^k \geqslant n_G\}$, $k \in \mathbb{N}$, в который все нити блока записывают соответствующие им значения $K^e_{\alpha\beta}(x_i, y_j, z_k)$, вычисленные в точках интегрирования (x_i, y_j, z_k). При суммирования элементов данного массива применяется алгоритм сдваивания, а результат операции передается в глобальную память ускорителя, где хранится K^e. Корректность результатов суммирования обеспечивает выбор числа нитей в блоке NB_{th}, которое полагается равным общему числу точек интегрирования для конечного элемента n_G.

В варианте алгоритма IIa блоком нитей обрабатываются сразу несколько локальных матриц жесткости, внутри блока выделяются группы нитей, которые выполняют вариант распараллеливания уровня II применительно к отдельному конечному элементу. В этом случае также создается `__shared__` массив, но при суммировании над его частями одновременно выполняется несколько копий алгоритма сдваивания. Число конечных элементов, обрабатываемых блоком нитей, полагается равным NB_{th}/n_G для конечных элементов первого порядка.

Предложенные варианты распараллеливания применялись при решении динамической задачи теории упругости в плоской и трехмерной постановке. Время выполнения численного интегрирования и ускорение, при использовании графических процессоров, полученные на одном временном шаге задачи, представлены в таблице 2. Вычислительные эксперименты проведены на одном из вычислительных узлов кластера X4 Института механики УрО РАН (два четырехъядерных центральных процессора Intel Xeon E5-2609, 2.4GHz, 32 ГБ оперативной памяти и две видеокарты GeForce GTX 680 на основе графического процессора GK104, созданного на ядре Kepler. Каждая видеокарта обладает вычислительными возможностями Compute capability 3.0, содержит 1536 ядер CUDA и 4 ГБ графической памяти.

Таблица 2. Время выполнения (с) и ускорение

m	t_{CPU}	$\frac{t_{CPU}}{t_{GPU}}$		
$p = 1$	–O3	I	II	IIa
Сетка 2D, $n_G = 4$				
10 000	0.01	1.48	1.10	2.09
100 000	0.10	2.25	1.35	3.54
1 000 000	0.99	2.35	1.49	3.83
10 000 000	10.03	2.44	1.51	4.28
Сетка 3D, $n_G = 8$				
10 000	0.14	0.74	0.67	1.01
100 000	1.35	1.64	1.38	3.14
500 000	6.77	1.83	1.50	3.85

В рассматриваемых случаях под ускорением понимается отношение времени вычисления m матриц жесткости на центральном процессоре t_{CPU} к времени вычисления на графическом ускорителе t_{GPU}. Приведенное в таблице 2 ускорение получено относительно варианта вычислений на центральном процессоре с оптимизацией исполняемого кода `-O3`, которая, как показали эксперименты, на данном CPU до восьми раз ускоряет неоптимизированный код и в два раза — по сравнению с оптимизацией `-O2`. Сетка 2D состоит из четырехугольных конечных элементов первого порядка, сетка 3D — из шестигранных; соответствующие локальные матрицы жесткости имеют размеры 8×8 и 24×24. Вычисления выполнялись с двойной точностью. Отметим, что время вычислений на GPU включало также копирование данных в память ускорителя и возвращение результата в оперативную память.

Как видно из таблицы 2, максимальное ускорения для каждого значения m получено при интегрировании блоком нитей (32 нити) нескольких матриц жесткости (восемь в двумерном случае и четыре — в трехмерном). С увеличением m увеличивается и ускорение, что говорит о неполной загруженности ядер графического процессора. Более полная загруженность GPU при распараллеливании на уровне IIa позволила получить ускорение до двух раз по сравнению с уровнем I, когда каждая нить интегрировала матрицу K^e отдельного конечного элемента e.

Вычисленные матрицы жесткости K^e хранятся построчно в симметричном формате (верхняя треугольная часть, включая диагональ матрицы) на ускорителе при использовании поэлементных схем, в другом случае передаются в оперативную память для решения системы с собранной глобальной матрицей жесткости.

В пакете FEStudio применяются следующие варианты распределения вычислений между графическим ускорителем и центральным процессором.

1. Вычисление только локальных матриц жесткости на GPU с передачей получаемых матриц в оперативную память для сборки глобальной матрицы жесткости на CPU. В этом случае вычисления ограничиваются только нахождением множества матриц K^e, $e = 1, 2, \ldots, m$.

2. Вычисление матриц K^e, задание граничных условий Дирихле на GPU и сборка системы уравнений на графическом ускорителе. В проинтегрированных матрицах K^e элементы строк и столбцов, соответствующие степеням свободы с заданными граничными условиями первого рода полагаются равными нулю, кроме диагональных элементов матрицы. Для этого необходимо передать на GPU: номера этих степеней в конечных элементах, значения граничных условий. Кроме того, на GPU формируется вектор поправок к правой части, получаемый при исключении заданных степеней свободы из других уравнений. В этом случае передача матриц исключается, а центральный процессор выполняет функцию управляющего процессора, обеспечивает ввод/вывод и постпроцессную обработку результатов.

3. Интеграция вычисления локальных матриц жесткости и задания граничных условий Дирихле в вычисление матрично-векторного произведения для методов подпространств Крылова, выполняемых на GPU. В этом случае вычисляется матрица K^e, в ней учитываются граничные условия Дирихле (см. предыдущий вариант), после чего вычисляется произведение $K^e p^e$, здесь p^e — соответствующая часть вектора направления p, например, в методе сопряженных градиентов.

Элементные схемы решения СЛАУ на графическом ускорителе. Вычислительные схемы решения конечно-элементных СЛАУ, использующие способ формирования матрицы коэффициентов сложением матриц отдельных конечных элементов, принято называть поэлементными. Такие схемы позволяют не только обходиться без хранения матрицы коэффициентов системы, но и более эффективно использовать кэш-память за счет локализации данных [Копысов, Новиков, 2010].

Основным отличием поэлементного алгоритма является способ вычисления матрично-векторного произведения, при котором используются два варианта организации вычислений: с извлечением данных в вектор меньшего размера и разнесением данных в вектор большего размера.

Схема с извлечением. Необходимые компоненты вектора $u \in R^N$ извлекаются в вектор $u_e \in R_e^N$, $N_e \ll N$, который умножается на локальную (эффективную) матрицу жесткости K^e:

$$\underbrace{\begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_N \end{bmatrix}}_{u} \xrightarrow{C_e} \underbrace{\begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{N_e} \end{bmatrix}}_{\bar{u}_e} \times \underbrace{\begin{bmatrix} k^e_{11} & k^e_{12} & \ldots & k^e_{1N_e} \\ k^e_{11} & k^e_{22} & \ldots & k^e_{2N_e} \\ \vdots & \vdots & \ddots & \vdots \\ k^e_{N_e1} & k^e_{N_e2} & \ldots & k^e_{N_eN_e} \end{bmatrix}}_{K^e} = \underbrace{\begin{bmatrix} \tilde{v}_1 \\ \tilde{v}_2 \\ \vdots \\ \tilde{v}_{N_e} \end{bmatrix}}_{\tilde{v}_e} \xrightarrow{\sum_{e=1}^{m} C_e^T \tilde{v}_e} \underbrace{\begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_N \end{bmatrix}}_{v}.$$

Схема с извлечением реализованна в FEStudio в классе `ElementCGSolver` (метод `Solve`). Извлечение компонент вектора осуществляется с помощью косвенной индексации, без копирования компонент p в вектор меньшей размерности. Доступ к необходимым компонентам вектора p происходит через элементный массив номеров уравнений `numeq`, получаемый методом `ElementCGSOE::GetLocalNumEquations`.

При вычислении вектора v_e может использоваться кэш-память при вычислениях на CPU или разделяемая память в случае GPU. В случае нескольких GPU, все необходимые компоненты векторов u и v должны быть доступны соответствующей нити CUDA.

Схема с разнесением. Для вычислений на графических ускорителях в интегрированной библиотеке CUSMA реализована схема с разнесением. В этом случае элементные матрицы (локальные матрицы жесткости) записываются в виде блоков формата BCSR [Копысов и др., 2012]. Такое преобразование формата хранения матриц осуществляется в методе `GPUElementCGSolver` пакета FEStudio, после чего для решения системы уравнений вызывается внешняя `host`-функция `GPUElemCGSolve` из библиотеки CUSMA.

На каждой итерации метода сопряженных градиентов операция матрично-векторного произведения (функции `MatrVectMul`) заменяется на вызовы следующих функций: `Scatter_p`, отвечающая за разнесение вектора; `ElemMatrVectMul`, выполняющая произведение разнесенного вектора на элементную матрицу; `MatrVectMul` — произведение вектора результата на матрицу связности.

В схеме с разнесением большая часть вычислений осуществляется над векторами размерности $\tilde{N} = m \times N_e$, которые являются объединением элементных векторов $u_e \in R^{N_e}$ и $v_e \in R^{N_e}$. Операция $C_{\tilde{N}\times N} v$ — *разнесение* (расширение в пространстве индексов) вектора $v \in R^N$, а $C^T_{\tilde{N}\times N} \tilde{v}$ — суммирование компонент вектора $\tilde{v}$:

$$\underbrace{\begin{bmatrix} \overline{u}^{(1)} \\ \overline{u}^{(2)} \\ \vdots \\ \overline{u}^{(m)} \end{bmatrix}}_{\overline{u}} \underbrace{\times}_{GPU} \underbrace{\begin{bmatrix} \tilde{K}_{11} & 0 & \dots & 0 \\ 0 & \tilde{K}_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \tilde{K}_{mm} \end{bmatrix}}_{\tilde{K}} = \underbrace{\begin{bmatrix} \tilde{v}^{(1)} \\ \tilde{v}^{(2)} \\ \vdots \\ \tilde{v}^{(m)} \end{bmatrix}}_{\tilde{v}} \underbrace{\xrightarrow{C^T_{N\times\tilde{N}}}}_{GPU} \underbrace{\begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_N \end{bmatrix}}_{v} \underbrace{\xrightarrow{C_{\tilde{N}\times N}}}_{GPU} \underbrace{\begin{bmatrix} \overline{v}^{(1)} \\ \overline{v}^{(2)} \\ \vdots \\ \overline{v}^{(m)} \end{bmatrix}}_{\overline{v}}.$$

В данной схеме конфликты при параллельном доступе переносятся на работу с вектором результата v.

Реализация проекционных методов на подпространствах Крылова, например алгоритма сопряженных градиентов, без непосредственной сборки глобальной матрицы системы линейных алгебраических уравнений и соотношений (3) позволяет эффективно реализовать метод решения на параллельной вычислительной системе как с распределенной, так и с общей памятью (см. **алгоритм 2**). Главные преимущества поэлементной схемы заключаются в следующем: использование любого типа элементов, простота представления и отображения на много- и мультиядерные системы. Элементные матрицы могут быть вычислены заново без изменения всей матрицы. Основной недостаток: требуется использование эффективных поэлементных предобуславливателей.

Согласно принятому разделению, подобласть, находящаяся на процессоре, содержит блоки матриц и векторов для внутренних и граничных узлов. Пусть подобласть $\Omega^{(i)}$ связана с конечно-элементной подсеткой $\mathcal{T}^{(i)}$, причем элементы внутри подобласти пронумерованы как $e = 1, m_I^{(i)}$, а элементы, лежащие на границах подобластей, имеют номера $e = m_I^{(i)} + 1, m_B^{(i)}$. Сгруппировав таким образом неизвестные в подобласти, можно выполнять операции матрично-векторного и скалярного произведения в двух процессах для внутренних и граничных узлов. Тогда, совместив

глобальные коммуникации, связанные с вычислением, с теми же операциями для внутренних узлов, не требующих обменов, можно записать

$$Kp = \sum_{e=1}^{m^i} C_e^T K^e C_e p = \sum_{e=1}^{m} C_e^T K^e C_e p + \sum_{e=m_I^i+1}^{m_B^i} C_e^T K^e C_e p.$$

Совмещение вычислений и обменов в совокупности с уменьшением числа обменов позволило уменьшить время выполнения **алгоритма 2** на 15–20 %.

Приведенный алгоритм сравнивался со схемами, использующими другие типы представления элементных векторов и матриц, и показал более высокие характеристики ускорения и эффективности. Кроме того, адаптированный вариант поэлементной декомпозиции был использован и для разрывного метода Галеркина [Копысов и др., 2011].

Таблица 3. Время решения одного шага по времени Δt, двумерный случай

Тип геометрии элемента	Число элементов/ узлов	N/Nnz	Решение системы, с CPU/GPU	
			CSR	EBE
Треугольник	200000/101101	202202/1782214	180/1.1	160/6.1
	394800/397902	397902/3513958	508/2.6	460/15.5
	1195600/600495	1200990/10628410	2758/11.6	2423/60.7
Четырехугольник	100000/101101	202202/2206990	134/1.2	106/3.2
	197400/397902	397902/4352690	346/3.1	286/9.1
	597800/600495	1200990 /13168770	1876 / 13.8	1675/39.1

Для оценки ускорения параллельных алгоритмов на разных типах элементов проведены вычислительные эксперименты по решению ряда тестовых задач. Исследования проводились на восьмиядерном (два четырехъядерных процессора Intel Xeon E5420, 8 ГБ оперативной памяти) вычислительном узле кластера X4 Института механики УрО РАН, который оснащен двумя видеокартами GeForce GTX 680 (2 ГБ оперативной памяти на каждой карте). Результаты исследований представлены в таблицах 3, 4.

Результаты исследований показали, что в двумерном случае на четырехугольных элементах (сетки с числом ячеек $\lg m \geqslant 5$) время выполнения элементной схемы при решении системы (3) на CPU примерно на 20 % меньше, чем при использовании собранных матриц. Однако при решении задачи в трехмерной постановке применение глобальных матриц жесткости в случае шестигранных конечных элементов уменьшает время выполнения более чем в два раза, в случае тетраэдальных элементов — в 3.5 раза (в обоих случаях применялись сетки с числом ячеек $\lg m \geqslant 5$). При вычислениях на CPU в двумерном случае предпочтительнее оказалась элементная схема, а в трехмерном — схема с собранными матрицами.

При решении схемой с собранными матрицами на графическом ускорителе двумерной задачи на четырехугольных элементах достигнуто ускорение в 106–127 раз, для трехмерной на шестигранных конечных элементах 38–56 раз и в 76–83 раза — на тетраэдрах, что подтверждает высокую параллельную эффективность предложенных алгоритмов. Вместе с тем при выборе алгоритма необходимо учитывать особенности гибридной архитектуры вычислительной системы, структуру и заполненность матриц, что определяет дальнейшие направления исследований элементных схем, обеспечивающих большую локализацию данных.

Таблица 4. Время решения одного шага по времени Δt, трехмерный случай

Тип конечного элемента	Число элементов/ узлов	N/Nnz	Время решения системы, с CPU/GPU	
			CSR	EBE
Тетраэдр	119360/23493	70479/1545935	19/0.3	72/3.4
	221184/41377	124131/2867779	40/0.5	209/3.9
	647731/119007	357021/8269721	126/1.3	471/12.1
Шестигранник	21970/25676	77028/2915736	10/0.3	21/0.6
	49130/55404	166212/6453944	30/0.6	62/1.3
	106480/116909	350727/13881759	83/1.4	175/1.4

Решение систем линейных уравнений и предобуславливание

В большинстве работ, посвященных реализации итерационных методов на GPU, рассматривается предобуславливатель Якоби или его блочный аналог. Оптимальным выбором для вычислений на графических процессорах представляются предобуславливатели, в которых считается, что известна аппроксимация обратной матрицы системы $\bar{M} \approx K^{-1}$. Тогда дополнительные операции, связанные с предобуславливанием, сводятся к вычислению произведения $z_{k+1} = \bar{M} r_{k+1}$.

Для вычисления на GPU решение системы методом сопряженных градиентов распараллелено с помощью технологии CUDA. Все вспомогательные массивы, в частности r, p, q, z, а также матрица системы, предобуславливатель, вектор правых частей и вектор решения, хранятся в памяти графического ускорителя. После завершения работы метода сопряженных градиентов массив u, который хранит приближение вектора решения, копируется в оперативную память.

Для реализации операций суммы, скалярного произведения, копирования векторов и умножения вектора на скаляр использовались функции библиотеки CUBLAS. При выполнении матрично-векторного произведения вектор хранится в текстурной памяти GPU, которая кэшируется, что дает более быстрый доступ и уменьшает временые затраты.

Для решения систем в FEStudio реализован блочный вариант, использующий несколько GPU. При разделении на блоки матрица K системы уравнений представлялась графом, имеющим число вершин, равное размерности K, где k_{ij} — элемент матрицы, расположенный в i-той строке и j-ом столбце. Каждой вершине графа матрицы K на основе вычисленного разделения многоуровневым алгоритмом ставится в соответствие номер графического ускорителя, исходя из которого вершины графа делятся на внутренние и граничные (связанные хотя бы с одной вершиной, имеющей другой номер ускорителя).

$$K = \begin{pmatrix} K_1^{[i_1,i_1]} & K_1^{[i_1,b_1]} & \cdots & 0 & 0 & \cdots & 0 & 0 \\ K_1^{[b_1,i_1]} & K_1^{[b_1,b_1]} & \cdots & 0 & K_1^{[b_1,b_l]} & \cdots & 0 & K_1^{[b_1,b_{n_p}]} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & K_l^{[i_l,i_l]} & K_l^{[i_l,b_l]} & \cdots & 0 & 0 \\ 0 & K_l^{[b_l,b_1]} & \cdots & K_l^{[b_l,i_l]} & K_l^{[b_l,b_l]} & \cdots & 0 & K_l^{[b_l,b_{n_p}]} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & 0 & \cdots & K_{n_p}^{[i_{n_p},i_{n_p}]} & K_{n_p}^{[i_{n_p},b_{n_p}]} \\ 0 & K_{n_p}^{[b_{n_p},b_1]} & \cdots & 0 & K_{n_p}^{[b_{n_p},b_l]} & \cdots & K_{n_p}^{[b_{n_p},i_{n_p}]} & K_{n_p}^{[b_{n_p},b_{n_p}]} \end{pmatrix}.$$

При полученном разделении в каждом блоке формируется несколько матриц K_l, где l — номер блока. В каждом блоке выделяется несколько типов матриц: $K_l^{[i_l,i_l]}$ — матрица, элементы

которой связывают внутренние вершины; $K_l^{[i_l,b_l]}$, $K_l^{[b_l,i_l]}$ — матрицы, связывающие внутренние вершины с граничными; $K_l^{[b_l,b_m]}$ — матрица, связывающая граничные вершины l-го блока с граничными вершинами m-го блока. Здесь $l, m \in [1, n_p]$, где n_p — число блоков. При матрично-векторном произведении $q = Kp$ на каждом GPU вычисляются два вектора: $q_b^l = K_l^{[b_l,i_l]} p_l + \sum_{m=1}^{m\leqslant N} K_l^{[b_l,b_m]} p_b^m$, $q_l = K_l^{[i_l,i_l]} p_l + K_l^{[i_l,b_l]} p_b^l$, где l — номер GPU. Это позволяет снизить затраты связанные с обменом между блоками на каждой итерации метода сопряженных градиентов, так как для выполнения последующих операций требуется обмен векторами q_b^l, которые при минимизации границ имеют размер существенно меньший, чем размер вектора q.

Алгоритм метода сопряженных градиентов для нескольких GPU выглядит следующим образом:

Алгоритм 4 Метод сопряженных градиентов с предобуславливателем

1: $K, \overline{\mathcal{M}} \in \mathbb{R}^{N\times N}$ {$\overline{\mathcal{M}}$ формируется на GPU, матрицы хранятся в CSR формате}
2: $u, r, p, q, z \in \mathbb{R}^N$ {Векторы хранятся в памяти GPU, копии на CPU нет}
3: $r_0 = f$ {cublasDcopy}
4: $u_0 = 0$; $z_0 \leftarrow \overline{\mathcal{M}} r_0$ {на GPU}
5: $p_0 = z_0$ {cublasDcopy}
6: $\rho_0 = (r_0, z_0)$ {здесь и далее $(\cdot,\cdot) = \sum_{j=0}^{n_p} ((\cdot_j^i, \cdot_j^i) + (\cdot_j^b, \cdot_j^b))$}
7: **while** $\|r_i\|_2/\|b\|_2 > \varepsilon$ **do**
8: $(p_{id}^b)_j \to p^k$ {p^k — локальные переменые на каждом GPU, $k \in [1, n_p]$}
9: $(q_{id}^i)_j = K_{id}^{[i_{id},i_{id}]}(p_{id}^i)_j + K_{id}^{[i_{id},b_{id}]}(p_{id}^b)_j$ {на GPU}
10: $(q_{id}^b)_j = K_{id}^{[b_{id},i_{id}]}(p_{id}^i)_j + K_{id}^{[b_{id},b_{id}]}(p_{id}^b)_j$ {на GPU}
11: $(q_{id}^b)_j += \sum_{k=0,k\neq id}^{n_p} K_{id}^{[b_{id},b_k]} p^k$ {на CPU}
12: $\alpha_j = (r_j, z_j)/(q_j, p_j)$ {cublasDdot}
13: $u_{j+1} = u_j + \alpha_j p_j$; $r_{j+1} \leftarrow r_j - \alpha_j q_j$ {cublasDasxpy}
14: $z_{j+1} = \overline{\mathcal{M}} r_{j+1}$ {на GPU}
15: $\rho_{j+1} = (r_{j+1}, z_{j+1})$ {cublasDdot}
16: $\beta_{j+1} = \rho_{j+1}/\rho_j$
17: $p_{j+1} = z_{j+1} + \beta_{j+1} p_j$ {последовательно cublasDscal и cublasDasxpy}
18: **end while**

В таблице 5 представлены результаты работы **алгоритма 4** на вычислительных узлах кластера «Уран» ИММ УрО РАН (центральный процессор CPU — Intel Xeon E5675; GPU — восемь NVIDIA Tesla M2090) для матриц из коллекции `http://www.cise.ufl.edu/research/sparse/matrices/` (N/Nnz — размерность системы и число ненулевых элементов). Решение СЛАУ для этих матриц методом сопряженных градиентов требует меньше временных затрат в алгоритме, использующем GPU, при этом максимальное ускорение относительно центрального процессора CPU достигало двадцати одного для сильно разряженных матриц при использовании четырех ускорителей, а для более заполненных матриц ускорение вычислений равнялось девяти и в последнем случае получено при использовании только двух GPU. Использование нескольких GPU для решения системы сокращает затраты по памяти и времени.

Таким образом, построена программная модель метода декомпозиции области, основанного на конечно-элементной аппроксимации, которая обеспечивает расширение и параллельную реализацию различных шагов алгоритма на гибридных вычислительных системах. Представление подобласти классом расчетной модели и модели решения позволило дважды использовать

Таблица 5. Сравнение времени решения систем на CPU и GPU, с

Матрица (N/Nnz)	CPU	$1 \times GPU$	$2 \times GPU$	$4 \times GPU$	$6 \times GPU$	$8 \times GPU$
Flan_1565						
1564794 / 117406044	1675.5	144.6	101.4	77.9	83.5	95.5
Audikw 1						
943695 / 77651847	846.5	85.3	66.4	57.1	65.7	78.9
Emilia 923						
923136 / 41005206	706.5	94.7	78.6	74.1	86.3	107.7
Inline 1						
503712 / 36816342	1537	166.3	163.9	169.0	220.3	279.1

структуру классов метода конечных элементов в области и в подобластях. Созданная модель метода декомпозиции позволяет быстро реализовать сложные алгоритмы, например методы декомпозиции с адаптивным уточнением [Kopyssov, Novikov, 2001; Копысов, Новиков, 2002]. Данная программная модель не является окончательной — предполагается ее постепенное уточнение, а также расширение иерархии классов. Часть параллельных алгоритмов, реализованных в рамках модели, была представлена выше. Кроме того, модель включает многомасштабный анализ, основанный на асимптотическом осреднении и вейвлет-преобразовании (см. [Копысов, Сагдеева, 2005]), решении связанных задач [Копысов и др., 2013; Копысов и др., 2013] и др.

Список литературы

Копысов С. П., Краснопёров И. В., Рычков В. Н. Объектно ориентированный метод декомпозиции области // Вычислительные методы и программирование. — 2003a. — Т. 4, № 1. — С. 176–193.

Копысов С. П., Краснопёров И. В., Рычков В. Н. Реализация объектно ориентированной модели метода декомпозиции области на основе параллельных распределенных компонентов CORBA // Вычислительные методы и программирование. — 2003b. — Т. 4, № 1. — С. 194–206.

Копысов С. П., Кузьмин И. М., Недожогин Н. С., Новиков А.К. Параллельные алгоритмы формирования и решения системы дополнения Шура на графических ускорителях // Ученые записки Казанского университета. Серия «Физико-математические науки». — 2012. — Т. 154, № 3. — С. 202–215.

Копысов С. П., Кузьмин И. М., Тонков Л. Е. Алгоритмическое и программное обеспечение решения задач взаимодействия конструкции с жидкостью/газом на гибридных вычислительных системах // Компьютерные исследования и моделирование. — 2013. — Т. 5, № 2. — С. 153–164.

Копысов С. П., Новиков А. К. Параллельные алгоритмы адаптивного перестроения и разделения неструктурированных сеток // Матем. моделирование. — 2002. — Т. 14, № 9. — С. 91–96.

Копысов С. П., Новиков А. К. Метод декомпозиции для параллельного адаптивного конечно-элементного метода // Вестник Удмуртского университета. Математика. Механика. Компьютерные науки. — 2010. — № 3. — С. 141–154.

Копысов С. П., Новиков А. К., Пономарёв А. Б. О параллельном построении неструктурированных сеток // Сб. трудов Межд. конф. «Параллельные вычислительные технологии ПаВТ’2007» / ЮУрГУ. — Т. 2. — г. Челябинск: Изд. ЮУрГУ, 2007. — 29 янв.–2 фев. — С. 65–76.

Копысов С. П., Новиков А. К., Сагдеева Ю. А. Решение систем уравнений метода Галёркина с разрывными базисными функциями на графическом ускорителе // Вестник Удмуртского университета. Математика. Механика. Компьютерные науки. — 2011. — № 4. — С. 121–131.

Копысов С. П., Пономарёв А. Б., Рычков В. Н. Параллельная распределенная модель расчетной сетки // Вестник Удмуртского университета. Математика. Механика. Компьютерные науки. — 2008. — № 2. — С. 194–196.

Копысов С. П., Сагдеева Ю. А. Вычислительные особенности и реализация вейвлет-осреднения в задачах многомасштабного анализа // Вычислительные методы и программирование. — 2005. — Т. 6. — С. 1–8.

Копысов С. П., Тонков Л. Е., Чернова А. А. Двухстороннее связывание при моделировании взаимодействия сверхзвукового потока и деформируемой пластины. Сравнение численных схем и результатов эксперимента // Вычислительная механика сплошных сред. — 2013. — Т. 6, № 1. — С. 78–85.

Кузьмин И. М., Недожогин Н. С., Новиков А. К., Сагдеева Ю. А. Конечно-элементное решение динамических задач деформирования на GPU // IX Всероссийская конференция «Сеточные методы для краевых задач и приложения». — Казань: Изд. «Отечество», 2012. — С. 250–253.

Рычков В. Н., Краснопёров И. В., Копысов С. П. Объектно ориентированная параллельная распределенная система для конечно-элементного анализа // Матем. моделирование. — 2002. — Т. 14. — С. 81–86.

Kopyssov S., Novikov A. Parallel Adaptive Mesh Refinement with Load Balancing for Finite Element Method // Lecture Notes in Computer Science. — 2001. — Vol. 2127. — P. 266–267.