Параллельные алгоритмы метода дополнения Шура на гибридной архитектуре *

С.П. Копысов, И.М. Кузьмин, Н.С. Недожогин, А.К. Новиков

Институт механики УрО РАН

Эффективность применения метода дополнения Шура на гибридных (CPU/GPU) архитектурах зависит от распределения вычислений между центральным процессором и графическими ускорителями. Показано, что формирование матриц дополнения Шура для нескольких подобластей эффективно выполнять на GPU, а с ростом числа подобластей на CPU. Представлен параллельный алгоритм обращения матрицы при помощи решения матричной системы множеством параллельных потоков. Для решения интерфейсной системы предложен параллельный алгоритм метода сопряженных градиентов с явным предобуславливателем, позволяющий достигать существенного ускорения вычислений на нескольких GPU.

1. Введение

Блочное разложение исходной матрицы или системы уравнений на основе дополнения Шура позволяет ускорить вычисления за счет параллельного обращения и умножения подматриц меньшего размера, чем у исходной матрицы [1,2]. Первоначально метод дополнения Шура [3] рассматривался как метод декомпозиции области для вычислений на системах с ограниченными вычислительными мощностями. В настоящее время, вычисление дополнения Шура чаще рассматривают, как гибридный метод решения систем линейных алгебраических уравнений (СЛАУ) [4, 5], сочетающий преимущества как прямых, так и итерационных методов, и учитывающий различные архитектуры параллельных вычислительных систем. На его основе строятся эффективные предобуславливатели [6].

Появление программного обеспечения для вычислений общего назначения на графические устройствах (GPGPU) позволило на ряде задач, в том числе вычислительной линейной алгебры, получать ускорение вычислений в десятки и сотни раз, по сравнению с центральным процессором (CPU). Тысячи потоков (нитей) GPU могут эффективно выполнять одновременно большое число простых арифметических операций, что характерно для мультипликативных и аддитивных операций с векторами и матрицами. Вместе с тем, последовательные операции и ветвления, характерные для разложения матриц на треугольные множители, выполняются на GPU медленнее, чем ядрами CPU. Кроме того, графический ускоритель обладает существенно меньшим объемом собственной оперативной памяти, чем типичный современный вычислительный модуль, что приводит к необходимости использования в расчетах нескольких GPU, в том числе связанных вычислительной сетью.

Таким образом, эффективное применение графических ускорителей, тем более нескольких, для метода дополнения Шура связано с декомпозицией матриц и определением алгоритмов, которые более эффективно выполняются на CPU или GPU. Переход к параллельным вычислениям на нескольких графических ускорителях предполагает использование нескольких технологий параллельного программирования: CUDA — для вычислений на одном GPU, OpenMP — для распараллеливания вычислений между несколькими GPU внутри одного вычислительного модуля и MPI — для передачи данных при вычислениях на кластере с гибридной архитектурой.

В работе решаются системы уравнений, получаемые при конечно-элементом решении трехмерных задач теории упругости и использующие разделение расчетной сетки при формировании блочной структуры матриц системы. В данном методе можно реализовать два

*Работа выполнена в рамках программы Президиума РАН №18 при поддержке УрО РАН (проект 12-П-1-1005) и РФФИ (грант №11-01-00275-а, 12-07-31114-мол-а).

уровня распараллеливания: первый уровень связан с разделением вычислений между подобластями [7, 9], второй с параллельной реализацией методов, которые используются для формирования внутри отдельной подобласти. Помимо этого, распараллеливанию подлежит и решение системы для дополнения Шура (интерфейсной системы). В представленной работе рассматривается второй уровень распараллеливания, для которого предложены алгоритмы формирования матриц дополнения Шура, а также решение интерфейсной системы уравнений с использованием нескольких графических ускорителей.

2. Ресурсная эффективность метода дополнения Шура

Рассмотрим использование построения дополнения Шура, как один из вариантов методов декомпозиции области в виде алгоритма метода подструктур [3]. Для обеспечения независимости вычислений в отдельных подобластях и последующего их взаимодействия, все узлы области делятся на два множества: внешних и внутренних узлов. Неизвестные перемещения области рассматриваются в виде суперпозиции двух составляющих. Первая составляющая — перемещения, вызванные внешними силами при закреплении границ в подобластях. Перемещения каждой подобласти определяются из уравнений, включающих неизвестные, связанные только с данной подобластью. Вторая составляющая — перемещения, вызванные смещениями границ подобласти с исключенными внутренними узлами.

Пусть область Ω разбита на n_Ω непересекающихся подобластей:

$$\Omega = \Omega_1 \bigcup \Omega_2 \bigcup \ldots \bigcup \Omega_{n_\Omega}, \quad \text{где} \quad \Omega_i \bigcap \Omega_j = \emptyset, \quad \Gamma_B = \bigcup_{i=1}^{n_\Omega} \partial\Omega_i \setminus \partial\Omega. \tag{1}$$

Разделение на подобласти наследуется от процесса разделения дуального графа расчетной сетки $G(V, E) = \bigcup_{i=1}^{n_\Omega} G_i(V_i, E_i)$, здесь множество вершин графа V — это множество конечных элементов расчетной сетки, множество ребер графа E — множество смежных конечных элементов, $V_i \subset V$ — множество конечных элементов, образующих подобласть. Далее полагается, что все подграфы $G_i(V_i, E_i)$ связные, в противном случае система уравнений (2) для подобласти Ω_i распадается на несвязанные системы уравнений.

Узлы расчетной сетки образуют множество $\hat{V}$ и условно разделяются на внешние $\hat{V}_{B_i}$ — принадлежат границе области и внутренние $\hat{V}_{I_i}$ — связанные с узлами подобласти сетки, соответствующей подграфу $G_i(V_i, E_i)$. Из множества внешних узлов выделяются интерфейсные $\hat{V}_{C_i} \subset \hat{V}_{B_i}$, связанные с узлами из других подобластей.

Для каждой подобласти Ω_i строятся системы уравнений, причем степени свободы, связанные с внутренними и внешними (граничными) узлами, разделяются:

$$\begin{pmatrix} A^i_{II} & A^i_{IB} \\ A^i_{BI} & A^i_{BB} \end{pmatrix} \begin{pmatrix} u^i_I \\ u^i_B \end{pmatrix} = \begin{pmatrix} f^i_I \\ f^i_B \end{pmatrix}, \tag{2}$$

где индексы I, B относятся к внутренним и граничным степеням свободы.

Система для интерфейсных узлов определяется как

$$S_{BB}\tilde{u}_B = \tilde{f}_B, \tag{3}$$

здесь $S_{BB} = \sum_i^{n_\Omega} \left(A^i_{BB} - A^i_{BI} A^{i^{-1}}_{II} A^i_{IB} \right)$ — матрица граничных жесткостей или дополнение Шура для подобласти i, вектор $\tilde{f}_B = \sum_i^{n_\Omega} \left(f^i_B - A^i_{BI} A^{i^{-1}}_{II} f^i_I \right)$ — вектор правых частей.

Как правило для расчетов задач используется усовершенствованный алгоритм метода подструктур. В его основе лежит использование свойств невырожденности и положительной определенности матриц подобластей. Для этих матриц существует разложение Холесского $A_{II} = L_{II} L^T_{II}$, где L — нижняя треугольная матрица с положительными диагональными элементами. Использование разложения Холесского значительно сокращает вычислительные затраты и используемый объем оперативной памяти.

Рассмотрим реализацию последовательного алгоритма вычисления дополнения Шура ($n_\Omega > n_p$ и число процессоров $n_p = 1$) и вычислительные затраты, связанные с каждым шагом выполнения (после шага алгоритма показано число необходимых операций с учетом симметрии матриц, причем операция сложения и умножения принимается как одна). Здесь

Алгоритм 1 Последовательный вариант дополнения Шура:

1: Выполним разложение Холесского матрицы A_{II} для соответствующей подобласти(индекс i опущен)
$$A_{II} = L_{II} L_{II}^T. \qquad \left(n_I^3/6\right)$$
2: Вычисляем вспомогательные переменные
$$A'_{IB} = A_{II}^{-1} A_{IB}. \qquad \left(n_B \cdot n_I^2\right)$$
3: Сформируем матрицы граничной жесткости подобластей
$$S_{BB} = A_{BB} - A_{BI} A'_{IB}. \qquad \left((n_I \cdot n_B^2)/2\right)$$
4: Формируем вектор правой части
$$\tilde{f}_B = f_B - A_{BI} A_{II}^{-1} f_I. \qquad (n_I \cdot n_B)$$
5: Собираем и решаем систему уравнений
$$S_{BB} \tilde{u}_B = \tilde{f}_B. \qquad (k(M^{-1} S_{BB}) \leq C(1 + \log(H/h)))$$
6: Определяем неизвестные для внутренних узлов
$$u_I = A_{II}^{-1} f_I - A'_{IB} \tilde{u}_B. \qquad \left(n_I^2\right).$$

$n_I = m \cdot |\hat{V}_{I_k}|$, $n_B = m \cdot |\hat{V}_{B_k}|$ — число внутренних и граничных степеней свободы, m — число степеней свободы в узле сетки, h — шаг сетки, H – размер подобласти, M — предобуславливатель для дополнения Шура. На пятом шаге **Алгоритма 1** приведена оценка обусловленности матрицы S_{BB}, а не вычислительная сложность, которая зависит от выбора метода решения системы уравнений. В данной работе, в качестве метода решения СЛАУ, использовался метод сопряженных градиентов с различными предобуславливателями. Матрицы S_{BB}, A_{II} положительно определены и симметричны. Отметим, что порядок матрицы S_{BB} значительно меньше, чем исходной матрицы, но достаточно большой, поэтому для системы с дополнением Шура применяются итерационные методы решения и компактные схемы хранения.

Для обеспечения эффективной работы с матрицами используются различные схемы хранения. Матрицы S_{BB}, A_{II} являются симметричными, поэтому возможно хранение только её части (верхний или нижний треугольник с диагональю).

Формат хранения (DCSR), используемый при вычислениях в данной работе, представляет массив, состоящий из списка упакованных строк, реализован в системе конечно-элементного анализа FEStudio [7, 8]. Каждая строка матрицы представляет структуру, состоящую из двух массивов. Первый массив хранит значения ненулевых элементов, второй — столбцовые индексы этих элементов. Размер каждого из массивов для строки i равен числу ненулевых элементов Nnz_i и меняется динамически по мере необходимости.

Предложенный формат DCSR является разновидностью распространенного формата хранения разреженных матриц — формат CSR (Compressed Sparse Row Storage Format). Для матрицы A, в формате CSR, выделяются три одномерных массива, в которых хранятся: ненулевые значения $\{a_{ij} \,|\, a_{ij} \neq 0\}$, $1 \leqslant i, j \leqslant Nnz$, их столбцовые индексы $\{j \,|\, a_{ij} \neq 0\}$, $1 \leqslant i, j \leqslant Nnz$ и позиции элементов a_{i1}, $1 \leqslant i \leqslant N + 1$ в двух первых массивах. Здесь N — размерность матрицы, $Nnz = \sum_{i=1}^{N} Nnz_i$ — число ненулевых элементов в матрице, Nnz_i — число ненулевых элементов в i-строке матрицы.

Сравним ресурсоёмкость предложенных схем хранения матриц по следующим параметрам: занимаемая память, алгоритмическая сложность доступа к элементу и его добавления. Оценка показывает, что алгоритмические сложности доступа к элементу $\mathcal{O}(2N_\beta + 1)$ и его добавления для ленточного формата составляет $\mathcal{O}((2N_\beta + 1)N)$, для DCSR —

$\mathcal{O}(Nnz^*)$ и $\mathcal{O}(Nnz^*)$ соответственно, а для формата CSR — $\mathcal{O}(Nnz^*)$ и $\mathcal{O}(Nnz)$, где $Nnz^* = \max_{i=1}^{N}\{Nnz_i\}$, $N_\beta = \max_{i=1}^{N}\{\max_{j=1}^{N}\{j-i \,|\, a_{ij} \neq 0\}\}$ — так называемая полуширина ленты, зависящая от нумерации неизвестных и уравнений. Необходимый объем памяти для ленточного формата — $8(2N_\beta + 1)N$, для формата DCSR — $\sum_{i=1}^{N}(12Nnz_i + 4)$, для формата CSR — $12Nnz + 4(N + 1)$ байт. В этом случае полагается, что при хранении вещественной величины используется 8 байт памяти, и 4 байта — для целого числа. В симметричном случае, величины $2N_\beta + 1$ в оценках сложности алгоритма (и приведенной далее частоты доступа) заменяются на $N_\beta + 1$, а Nnz и Nnz_i относятся к элементам треугольной матрицы.

Отметим, что формат DCSR более удобен, чем CSR при выполнении треугольного разложения матриц A_{II}, когда в строках появляются новые ненулевые элементы. В этом случае изменение в одной из строк не требует смещения всех последующих элементов в массивах, как в формате CSR. Поэтому процедура добавления нового элемента имеет меньшую алгоритмическую сложность $\mathcal{O}(Nnz^*)$, чем в формате CSR — $\mathcal{O}(Nnz)$. Кроме того, из представленных форматов, необходимый для DCSR объем памяти — $\sum_{i=1}^{N}(12Nnz_i + 4)$ байт, является минимальным. Преимущества ленточного формата, по сложности доступа и добавления элемента, компенсируются необходимым объемом памяти $8(2N_\beta + 1)N$ и частотой доступа к элементам матрицы $\mathcal{O}((2N_\beta + 1)N)$, вместо $\mathcal{O}(Nnz)$ для форматов DCSR и CSR.

Как показали эксперименты [9], схема хранения оказывает существенное влияние на время вычислений. Отметим, что время формирования S_{BB} с матрицами в ленточном формате составляет порядка 80% общего времени вычислений. Переход на формат DCSR для матриц A_{BB}, A_{IB}, A_{II} для одной и той же задачи дал сокращение затрат в четыре раза. Таким образом, в последовательном варианте, наиболее эффективным, с точки зрения ресурсоемкости и алгоритмической сложности, представляется использовать метод дополнения Шура с разложением Холесского матрицы A_{II} и хранением матриц в сжатом формате.

Обработка строк матрицы, хранящейся в формате DCSR, на графическом ускорителе (GPU) потребует для каждой строки: выделения памяти на GPU, копирования строки и далее, в ядре CUDA, выполнения вычислений над строкой и возвращения результата в оперативную память или кэш CPU. Таким образом, потребуется $2N$ выделений памяти и копирований массивов размера Nnz_i, вместо выделения памяти и копирования двух массивов размера Nnz, поэтому, для вычислений на GPU, матрица переводится в CSR формат.

3. Параллельная эффективность метода дополнения Шура

В методе декомпозиции области можно выделить два направления распараллеливания процесса решения. Первое связано с методами явного деления на подобласти, где параллельные вычисления осуществляются на уровне области, то есть число ветвей параллельного алгоритма равно числу подобластей. Как показывает практика, такой подход дает значительное ускорение, если каждый процессор решает в своей подобласти свою подзадачу. Второе направление — неявные методы внутри подобластей. В каждой подобласти различные задачи могут быть решены наиболее эффективным способом.

Рассмотрим **Алгоритм 1**, исходя из введенных вариантов распараллеливания. *Шаги 1–4, 6* выполняются для каждой подобласти независимо, что говорит о естественном параллелизме, который обеспечивает первый уровень распараллеливания — на уровне области.

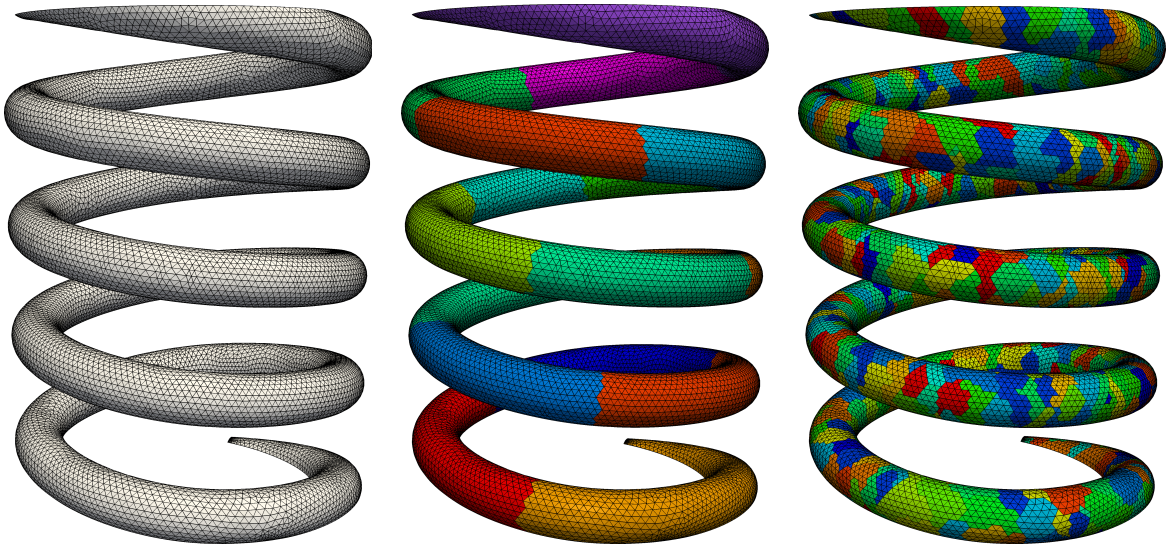
Наиболее трудоёмкими шагами алгоритма являются: процессы формирования матрицы дополнения Шура — *Шаги 1–3*, вектора правых частей — *Шаг 4*, а также решение системы — *Шаг 5*. Вычисления внутри каждой подобласти выполняются независимо от других подобластей, значит становится возможной их параллельная реализация.

При реализации параллельных алгоритмов неизбежно возникает проблема балансировки вычислительной нагрузки. На четвертом шаге **Алгоритма 1** формируются локальные матрицы дополнения Шура S_{BB}^i, независимо для каждой из подобластей. Далее из S_{BB}^i строится система уравнений с глобальной матрицей дополнения Шура (3), которая не мо-

жет быть решена, пока S^i_{BB} не сформируются всеми параллельными процессами i.

Таким образом, источников неравномерности нагрузки может быть несколько. Одним из них является неравномерное распределение количества подобластей на вычислительные узлы — этот недостаток легко исключить, разделив область на количество подобластей кратное количеству используемых вычислительных узлов. Другой источник — неравномерное распределение узлов сетки и конечных элементов по подобластям. В этом случае неравномерность исключается заданием дополнительных условий при разделении сетки.

Сбалансированное распределение вычислительной нагрузки при выполнении *Шагов 1–4* зависит не от общего числа узлов $|\hat{V}_i|$ в подобластях Ω_i, а от числа внутренних $|\hat{V}_{I_i}|$ и внешних $|\hat{V}_{B_i}|$ узлов в подобластях. Вместе с тем, сетки для трехмерных областей с развитой поверхностью (пружины, тонкие пластины и оболочки) содержат большое число конечных элементов, в которых все узлы, принадлежат границе расчетной области. Разделение дуальных графов таких сеток и выполнение условия $|V_i| \approx |V_j|, \forall i \neq j$ приводит к подобластям Ω_i: $|\hat{V}_{I_i}| = 0$. Наглядным примером является задача моделирования напряженно-деформированного состояния пружины, рассматриваемая в данной работе.



а) б) в)

Рис. 1. Сетка: а) исходная; б)разделенная на 16 подобластей; в)разделенная на 1024 подобласти двухуровневым разделением.

Геометрия пружины аппроксимируется неструктурированной расчетной сеткой (см. рисунок 1а) с ячейками в виде тетраэдров, число ячеек $|V| = 174264$, число узлов $|\hat{V}| = 40743$. Для получения подобластей $\Omega_i : |\hat{V}_{I_i}| > 0$, дуальный граф полагался взвешенным $G(V, E, W)$, с множеством весов $W = \{w_k\}$. Если хотя бы один из узлов конечного элемента не лежит на $\partial\Omega$, то вес соответствующей вершины графа $w_k = 3$, в другом случае $w_k = 1$.

Проведенные вычислительные эксперименты с различным числом подобластей ($n_\Omega = 16, 32, \ldots, 1024$) показали, что увеличение числа подструктур приводит к увеличению размера матрицы дополнения Шура. Отметим, что число подобластей увеличилось в 64 раза, а размер матрицы S_{BB} только в 1.44 раза ($N = 66030$ в случае $n_\Omega = 16$ и $N = 95523$ — для $n_\Omega = 1024$). Вместе с тем, число ненулевых элементов матрицы дополнения Шура уменьшилось в 18 раз (с $Nnz \approx 2.7 \cdot 10^8$ в случае 16-ти, до $Nnz \approx 1.5 \cdot 10^7$ — для 1024-х подобластей), как следствие, уменьшилась её заполненность с 6.11% до 0.16%. В среднем, примерно каждый шестнадцатый элемент в строке S_{BB} является ненулевым (4041 из 66030), при $n_\Omega = 16$ подобластей (см. рисунок 1 б), и примерно каждый шестисотый (154 из 95523) — при разделении на 1024 подобласти (см. рисунок 1 в).

Важно отметить, что размер системы (3) меньше размера исходной конечно-элементной системы (в рассмотренных случаях в 1.4–2.0 раза), тогда как заполненность матрицы S_{BB} на один-два порядка больше, чем заполненность глобальной матрицы жесткости (0.03%).

4. Формирование матрицы дополнения Шура

Одной из самых затратных операций формирования дополнения Шура является обращение матрицы A_{II}. Обычно, методами нахождения обратной матрицы являются плохо распараллеливаемые прямые методы, например метод обращения на основе LL^T-разложения.

Рассматриваемый в работе алгоритм вычисления обратной матрицы состоит из решений матричной системы вида $A_{II}X = E$, где E — единичная матрица $n_I \times n_I$. Система эффективно решается на GPU предобусловленным алгоритмом сопряженных градиентов (см. следующий параграф). Если в правой части системы вместо матрицы E брать A_{IB}, то её решением будет матрица $A'_{IB} = A_{II}^{-1} A_{IB}$. Такое представление позволяет заменить операции обращения матрицы и матричного произведения на решение n_B систем, каждая из которых решается независимо, что позволяет использовать одновременно несколько GPU. Дополнение Шура или матрица граничных жесткостей вычисляется по соотношениям (3), где $A_{BB} \in \mathbb{R}^{n_B \times n_B}$, $A_{II} \in \mathbb{R}^{n_I \times n_I}$ $A_{BI} \in \mathbb{R}^{n_B \times n_I}$, $A_{IB} \in \mathbb{R}^{n_I \times n_B}$.

Пусть $\tilde{n}_B^i$ — количество столбцов матрицы A_{BB} пересылаемых на i-ый GPU, $\tilde{A}_{BB}^i \in \mathbb{R}^{n_B \times \tilde{n}_B^i}$ матрица, состоящая из столбцов матрицы A_{BB} с номерами от $\sum_{j=0}^{i} \tilde{n}_B^j$ до $\sum_{j=0}^{i+1} \tilde{n}_B^j$, где i — номер GPU. Каждый графический ускоритель решает $\tilde{n}_B^i$ систем $A_{II}a^k = a_{IB}^k$, здесь a_{IB}^k — k-ый столбец матрицы A_{IB}, $k \in \left[\sum_{j=0}^{i} \tilde{n}_B^j, \sum_{j=0}^{i+1} \tilde{n}_B^j\right]$, а $A'_{IB} = \{a^1, a^2 \dots a^{n_B}\}$. В реализации подхода независимого решения систем уравнений на нескольких графических ускорителях используется технология OpenMP. Для этого создаются несколько нитей (число которых равно количеству доступных устройств GPU), определяется номер нити и каждой назначается графический ускоритель с тем же номером.

Ниже представлен **Алгоритм 2**, реализующий этот подход. На каждом GPU после

Алгоритм 2 Параллельный алгоритм формирования дополнения Шура на каждой подобласти Ω_i (индекс i опущен):

1: Решаем систему $A_{II}A'_{IB} = A_{IB}$; {матрицы хранятся на GPU в CSR формате, решение — по столбцам}
2: $S_{BB} = A_{BB} - A_{BI}A'_{IB}$; {$S_{BB}$ и результат произведения матриц — построчно, A_{BB} — CSR формате}
3: $\tilde{f}_B = f_B - A_{BI}x$; {$x$ — решение системы $A_{II}x = f_I$}
4: Формируем и решаем: $S_{BB}\tilde{u}_B = \tilde{f}_B$; {$S_{BB}$ формируется в DCSR на CPU, а затем копируется в CSR на GPU }
5: Определяем $u_I = x - A'_{IB}\tilde{u}_B$; {$x$ и A'_{IB} вычислены ранее, и хранятся на CPU}.

решения систем остаётся матрица $(A'_{IB})^i \in \mathbb{R}^{n_I \times n_B^i}$, состоящая из столбцов матрицы A'_{IB} с номерами от $\sum_{j=0}^{i} \tilde{n}_B^j$ до $\sum_{j=0}^{i+1} \tilde{n}_B^j$. Это позволяет без дополнительных коммуникаций выполнить оставшиеся операции (произведение и разность матриц, см. соотношения (3)) для каждой матрицы $(A'_{IB})^i$ независимо на нескольких GPU, которые используются при вычислении матриц S_{BB}^i. Далее формируется глобальная матрица дополнения Шура S_{BB} (*Шаг* 4 в **Алгоритме 2**), состоящая из локальных S_{BB}^i, принадлежащих i-подобласти.

Локальные матрицы дополнения Шура S_{BB}^i хранятся в несжатом формате. Матрица S_{BB}, после формирования из локальных S_{BB}^i, преобразуется из несжатого к тому формату, в котором будет наиболее удобно с ней работать на GPU, например CSR.

В таблице 1 приведены затраты времени на параллельное формирование S_{BB}, в зависимости от n_Ω и числа GPU. Во второй колонке представлены данные для последовательного алгоритма, выполняемого на центральном процессоре (**Алгоритм 1**).

Таблица 1. Время формирования дополнения Шура, мин. : сек.

n_Ω	CPU	1 GPU	2 GPU	4 GPU	6 GPU	8 GPU
16	26:23.6	31:52.6	19:25.5	13:09.9	10:57.6	09:49.4
32	08:00.6	19:33.9	11:01.8	06:41.7	05:14.0	04:26.5
64	02:25.1	11:18.8	06:18.2	03:44.8	02:54.7	02:29.0
128	—	—	03:57.2	02:25.8	01:58.5	01:42.5
256	—	—	03:14.0	02:01.0	01:37.7	01:26.0
512	—	—	02:48.3	01:45.7	01:26.0	01:15.4
1024	00:18.7	03:53.4	02:24.0	01:32.2	01:17.5	01:08.0

Ускорение параллельного алгоритма (**Алгоритм 2**), реализованного на GPU, по отношению к CPU, определим как $s(n_p)_{CPU} = t_{CPU}/t(n_p)_{GPU}$, где t_{CPU} — время выполнения последовательного алгоритма, реализованного на CPU, а $t(n_p)_{GPU}$ — время выполнения параллельной реализации на n_p GPU. Аналогичным образом введём ускорение $s(n_p)_{GPU} = t(1)_{GPU}/t(n_p)_{GPU}$. В рассмотренных вариантах максимальное ускорение составляет $s(8)_{GPU} = 2.7$ и достигается при числе подобластей $n_\Omega = 16$, при этом в каждой подобласти находится в среднем примерно по 11000 ячеек сетки. Формирование матрицы дополнения Шура на двух GPU приводит к ускорению $s(2)_{GPU} > 1.5$, в зависимости от числа подобластей. Использование восьми графических ускорителей даёт наименьшее время выполнения для реализации на GPU, но, для задач с небольшим числом ячеек в каждой подобласти (< 2500), CPU-реализация оказывается эффективнее. В этом случае, при решении большого числа систем уравнений малой размерности для каждой подобласти требуются время на копирование данных между GPU и CPU, которое становится больше, чем время решения систем. В рамках одного GPU (при использовании нескольких) затраты на решение сокращаются, но не покрывают затрат на инициализацию и копирование.

Полученные данные позволяют применять **Алгоритм 2** для обеспечения более сбалансированного использования GPU и CPU в гибридных вычислительных системах.

5. Решение интерфейсной системы уравнений на GPU

Матрица дополнения Шура $S_{BB} \in \mathbb{R}^{N\times N}$ (далее $S = [s_{ij}]$) имеет порядок и обусловленность меньше, чем у исходной матрицы и является симметричной, положительно определённой и разреженной. Решение интерфейсной системы (3) и систем уравнений из предыдущего параграфа выполняются предобусловленным методом сопряженных градиентов.

В большинстве работ, посвященных реализации итерационных методов на GPU, рассматривается предобуславливатель Якоби или его блочный аналог. Оптимальным выбором для вычислений на GPU представляются предобуславливатели, в которых считается, что известна аппроксимация обратной матрицы системы $\bar{M} \approx S^{-1}$ [10]. Тогда дополнительные операции, связанные с предобуславливанием, сводятся к произведению $z_{k+1} = \bar{M}r_{k+1}$.

В данной работе рассматриваются методы сопряжённых градиентов с диагональным предобуславливателем (DIAG), предобуславливанием по методу симметричной последовательной верхней релаксации (SSOR) и предобуславливателем, построенным масштабированием системы (DIP). К сожалению, в наших экспериментах предобуславливатели, основанные на аппроксимации обратной матрицы с использованием дополнения Шура, не показали

сравнимую эффективность вычислений на рассматриваемых матрицах.

Диагональный предобуславливатель имеет вид:

$$\bar{M} = M^{-1} = \{\bar{m}_{ij}\}_{ij=1}^{N}, \quad \bar{m}_{ij} = \begin{cases} s_{ij}^{-1}, & \text{если } i = j; \\ 0, & \text{в остальных случаях.} \end{cases}$$

Определим предобуславливатель SSOR следующим образом. Пусть матрица системы представима как $S = L + D + L^T$, тогда $M = KK^T$, $K = \frac{1}{\sqrt{2-\omega}}\tilde{D}(I + \tilde{D}^{-1}L)\tilde{D}^{-1/2}$, или $K^{-1} = \sqrt{2-\omega}\tilde{D}^{1/2}((I + \tilde{D}^{-1}L))^{-1}\tilde{D}^{-1}$, где $0 < \omega < 2$, $\tilde{D} = (1/\omega)D$.

Вычислим приближенно K^{-1} ограничиваясь первым членом в разложении

$$K^{-1} \approx \bar{K} = \sqrt{2-\omega}\tilde{D}^{-1/2}(I - L\tilde{D}^{-1}). \tag{4}$$

Тогда предобуславливатель SSOR примет вид $\bar{M} = \bar{K}^T\bar{K}$. Если положить $\omega = 1$ в (4) получим предобуславливатель вида $\bar{K} = D^{-1/2}(I - LD^{-1})$ и соответственно $\bar{M} = D^{1/2}\bar{K}\bar{K}^T D^{1/2}$. При построении предобуславливателя DIP масштабируем исходную систему

$$\tilde{S} = D^{-1/2}SD^{-1/2}; \quad \tilde{f} = D^{-1/2}f; \quad \tilde{u} = D^{1/2}u;$$

тогда предобуславливатель примет вид $\bar{M} = (I - \tilde{L}^T)(I - \tilde{L})$.

Решение интерфейсной системы методом сопряжённых градиентов распараллелено с помощью технологии CUDA для вычисления на GPU. 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 (см. **Алгоритм 3**). После завершения работы метода сопряжённых градиентов, массив u, в котором хранится приближение вектора решения, копируется в память CPU.

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, которая кэшируется, что даёт более быстрый доступ и уменьшает временные затраты. Для вы-

Алгоритм 3 Алгоритм метода сопряжённых градиентов с предобуславливателем:

1: $S, \bar{M} \in \mathbb{R}^{N\times N}$ {$\bar{M}$ формируется на GPU, матрицы хранятся в CSR формате}
2: $u, r, p, q, z \in \mathbb{R}^N$ {Вектора хранятся в памяти GPU, копии на CPU нет}
3: $r_0 \leftarrow f$ {копирование векторов осуществляется с помощью cublasDcopy}
4: $u_0 \leftarrow 0$ {инициализация выполняется на GPU}
5: $z_0 \leftarrow \bar{M}r_0$ {выполняется на GPU}
6: $p_0 \leftarrow z_0$ {копирование векторов осуществляется с помощью cublasDcopy}
7: $\rho_0 \leftarrow (r_0, z_0)$ {здесь и далее $(\cdot,\cdot) = \sum\limits_P (\cdot,\cdot)^{(P)}$}
8: **while** $||r_i||_2/||b||_2 > \varepsilon$ **do**
9: $q_i \leftarrow Sp_i$ {выполняется на GPU}
10: $\alpha_i \leftarrow (r_i, z_i)/(q_i, p_i)$ {вычисляется с помощью функции cublasDdot}
11: $u_{i+1} \leftarrow u_i + \alpha_i p_i$ {операция выполняется с помощью функции cublasDasxpy}
12: $r_{i+1} \leftarrow r_i - \alpha_i q_i$ {операция выполняется с помощью функции cublasDasxpy}
13: $z_{i+1} \leftarrow \bar{M}r_{i+1}$ {выполняется на GPU}
14: $\rho_{i+1} \leftarrow (r_{i+1}, z_{i+1})$ {вычисляется с помощью функции cublasDdot}
15: $\beta_{i+1} \leftarrow \rho_{i+1}/\rho_i$
16: $p_{i+1} \leftarrow z_{i+1} + \beta_{i+1}p_i$ {последовательное использование функций cublasDscal и cublasDasxpy}
17: **end while**

числения каждой координаты вектора результата, используется от 2-х до 32-х потоков, в

зависимости от разреженности матрицы. Вычисление предобуславливателя выполняется либо на GPU, либо на CPU, в зависимости от его типа.

Предобуславливатель SSOR вычисляется на центральном процессоре. Разреженность матрицы K равна разреженности матрицы S и поэтому, выгоднее её так же хранить в компактном формате. Теоретически, метод сопряжённых градиентов решает систему не более, чем за N итераций. Пусть система решается за $\tilde{N}$ итераций, значит при вычислении $\bar{K}^T\bar{K}$ требуется N матрично-векторных произведений, плюс $\tilde{N}$ матрично-векторных произведений $z_{k+1} = \bar{M}r_{k+1}$ для решения системы методом сопряжённых градиентов. Если не вычислять $\bar{M}$ явно, а произведение $z_{k+1} = \bar{M}r_{k+1}$ заменить на два последовательных друг за другом матрично-векторных произведения $\tilde{z}_{k+1} = \bar{K}r_{k+1}$ и $z_{k+1} = \bar{K}^T\tilde{z}_{k+1}$, то нам потребуется всего $2\tilde{N}$ матрично-векторных произведений, что меньше, чем $N + \tilde{N}$ при явном нахождении матрицы $\bar{M}$. Поэтому, в реализации метода сопряжённых градиентов с SSOR предобуславливателем, произведение $z_{k+1} = \bar{M}r_{k+1}$ заменено на $z_{k+1} = \bar{K}^T\bar{K}r_{k+1}$.

Масштабирование системы и вычисление предобуславливателя DIP выполняется на GPU. Как и в случае с SSOR, разряженности матриц совпадают, поэтому они хранятся в разряженном формате. Произведение $z_{k+1} = \bar{M}r_{k+1}$ также заменяем на два матрично-векторных произведения $y = (I - \tilde{L})r_{k+1}$ и $z_{k+1} = (I - \tilde{L}^T)y$, выполняемых на GPU.

Применение предобуславливателя SSOR сокращает число итераций при решении системы в полтора раза, что позволяет покрыть затраты на его создание и на дополнительное матрично-векторное произведение. В результате чего, нет выигрыша по времени вычислений в сравнении с диагональным предобуславливателем. Аналогичные результаты получены для систем разрывного метода Галёркина [10]. Отличие в свойствах матрицы S и матриц систем разрывного метода Галёркина, позволило значительно уменьшить число итераций при использовании предобуславливателя DIP, которое стало примерно равным с SSOR.

Для решения интерфейсной системы (3) реализован блочный вариант **Алгоритма 3**, использующий несколько GPU. При разделении на блоки, матрица S системы (3) представлялась графом, имеющим число вершин равное размерности S, где s_{ij} — элемент матрицы расположенный в i-ой строке и j-м столбце. Каждой вершине графа матрицы S, на основе вычисленного разделения многоуровневым алгоритмом [11], 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).

При помощи полученного разделения, в каждом блоке формируется несколько матриц S_k, где k — номер блока. В каждом блоке выделяется несколько типов матриц: $S_k^{[i,i]}$ — матрица, элементы которой связывают внутренние вершины; $S_k^{[i,b]}$, $S_k^{[b,i]}$ — матрицы, связывающие внутренние вершины с граничными; $S_k^{[k,m]}$ — матрица, связывающая граничные вершины k-го блока с граничными вершинами m-го блока. Здесь $k, m \in [1, N]$, где N — число блоков. Используя эти обозначения, матрица S примет вид

$$S = \begin{pmatrix} S_1^{[i,i]} & S_1^{[i,b]} & 0 & 0 & \cdots & 0 & 0 \\ S_1^{[b,i]} & S_1^{[1,1]} & 0 & S_1^{[1,2]} & \cdots & 0 & S_1^{[1,N]} \\ 0 & 0 & S_2^{[i,i]} & S_2^{[i,b]} & \cdots & 0 & 0 \\ 0 & S_2^{[2,1]} & S_2^{[b,i]} & S_2^{[2,2]} & \cdots & 0 & S_2^{[2,N]} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & S_N^{[i,i]} & S_N^{[i,b]} \\ 0 & S_N^{[N,1]} & 0 & S_N^{[N,2]} & \cdots & S_N^{[b,i]} & S_N^{[N,N]} \end{pmatrix}.$$

При матрично-векторном произведении $q = Sp$ на каждом GPU вычисляются два вектора:

$$q_b^k = S_k^{[b,i]} p_k + \sum_{m=1}^{m \leqslant N} S_k^{[k,m]} p_b^m; \qquad q_k = S_k^{[i,i]} p_k + S_k^{[i,b]} p_b^k, \tag{5}$$

где k — номер GPU. Это позволяет снизить затраты связанные с обменом между блоками на каждой итерации метода сопряженных градиентов, так как для выполнения последующих операций с векторами требуется обмен q_b^k, которые при минимизации границ имеют размер гораздо меньший, по сравнению с размером q.

Решение СЛАУ методом сопряженных градиентов требует меньше временных затрат в алгоритме, использующем GPU (см. таблицу 2), при этом ускорение $s(1)_{CPU} = 72$, при разделении на 16 подобластей, и $s(1)_{CPU} = 94$ при разделении на 1024 подобласти.

Таблица 2. Время решения интерфейсной системы, часы : мин. : сек.

n_Ω	CPU	1 GPU	2 GPU	4 GPU	6 GPU	8 GPU
16	> 8:00:00	0:15:12	0:07:01	0:03:25	0:04:41	0:04:24
32	> 8:00:00	0:16:23	0:10:30	0:07:10	0:05:17	0:04:34
64	5:01:07	0:04:47	0:02:54	0:01:38	0:01:22	0:01:12
128	—	—	0:02:07	0:01:18	0:01:06	0:01:00
256	—	—	0:01:46	0:01:10	0:01:01	0:00:56
512	—	—	0:01:25	0:01:03	0:00:56	0:00:53
1024	1:48:22	0:01:09	0:01:16	0:00:59	0:00:54	0:00:52

Использование нескольких GPU даёт максимальные ускорения $s(8)_{CPU} = 251$ для $n_\Omega = 64$ и $s(8)_{GPU} = 3.5$ для $n_\Omega = 16$. С увеличением числа подобластей количество ненулевых элементов, в полученной матрице дополнения Шура, уменьшается — это приводит к тому, что эффективность использования нескольких GPU, для решения интерфейсной системы, снижается. Так, например, при разделении на 1024 подобласти $s(8)_{GPU} = 1.3$. *Шаг 6* в **Алгоритме 1**, отвечающий за нахождения решений на внутренних узлах, также выполняется на GPU. Произведение $A_{II}^{-1} f_I$ уже вычислено на *Шаге 4*, матрица A'_{IB} получена на *Шаге 2*. Поэтому, необходимо выполнить лишь две операции — матрично-векторного произведения и разности векторов, которые выполняются на GPU.

В ходе выполнения вычислительных экспериментов минимальные значения суммарного времени формирования и решения системы для дополнения Шура (3) получены при $n_\Omega = 1024$ (см. таблицы 1 и 2). При выполнении этих шагов только центральным процессором потребовался 1 час 48 мин. В случае использования только одного GPU, для формирования и решения СЛАУ, затраты сократились в 22 раза. Минимальное время вычислений на графических ускорителях получено на восьми GPU — $t(8)_{GPU} = 2$ мин. Формирование системы (3) на центральном процессоре и решение на одном графическом ускорителе выполнено за полторы минуты. Наименьшие суммарные затраты потребовались при формировании системы на CPU и её решении на восьми GPU.

Система (3) решалась также на гибридном кластере, оснащенном графическими ускорителями. Следующие варианты рассмотрены: 8 процессов MPI, выполняемые в одном модуле; по 4 процесса в двух модулях; по 2 процесса в четырех модулях и по одному процессу

в 8 модулях. Коммуникации обеспечивались коллективными функциями, которые вызывались одним из потоков OpenMP, запускаемых внутри каждого процесса MPI.

Таблица 3. Время решения (3) на нескольких вычислительных модулях, мин. : сек.

n_Ω	Nnz/N	1×8	2×4	4×2	8×1
16	266826696/66030	5:21	4:28	3:28	3:53
64	90931750/70620	1:18	1:12	1:11	1:35
128	58592886/75732	1:08	1:00	0:59	1:26
256	38066007/81753	1:06	0:57	1:06	1:27
512	23955996/88248	1:03	0:55	1:08	1:42
1024	14749329/95523	1:05	0:54	0:54	1:50

Результаты экспериментов показывают, что затраты на решение (3) существенно зависят от распределения вычислений между CPU и GPU при выполнении матрично-векторного произведения. В случае $n_\Omega = 16$ (таблица 3) основные затраты приходятся на слагаемое $\hat{q}_k^b = \sum\limits_{m=1, m\neq k}^{m<n_p} S_k^{[b_k,b_m]} p_m^b$ из (5), которое вычисляется на CPU. Основной причиной этого являются большие размеры блоков $S_k^{[b_k,b_m]}$ и неравномерное распределение их между параллельными процессами/потоками. В подобных случаях можно перенести эту операцию на графический ускоритель. Приведенные в таблице 3 результаты показывают, что размещение вычислений по разным вычислительным модулям привело к ускорению вычислений в 1.2–1.5 раза, вызванное скорее всего конкуренцией за кэш-память CPU.

Более равномерное разделение граничных вершин и много меньшие (на два, три порядка) размеры $S_k^{[b_k,b_m]}, m \neq k$, при $n_\Omega = 1024$, приводят к эффективному вычислению $\hat{q}_k^b$ на CPU, а в результате — к доминированию затрат на операции над векторами (см. **Алгоритм 3**). В этом случае обращения к функции `cublasDdot` при вычислении скалярных величин составляют около 90% времени вычислений (варианты 1×8, 2×4, 4×2).

Ускорение вычислений при увеличении числа подобластей связано, в основном, с увеличением разреженности систем за счет уменьшения Nnz и увеличения N. В результате уменьшается общее число арифметических операций, приходящихся на матрично-векторные произведения при решении системы (3).

Распределение блоков матрицы между вычислительными модулями, при помощи обмена сообщениями MPI также позволило снять ограничение (объем памяти ускорителей в пределах одного вычислительного модуля) на размер решаемой интерфейсной системы (3).

6. Заключение

Метод дополнения Шура позволяет распределить вычисления между CPU и графическими ускорителями при решении СЛАУ на гибридных вычислительных системах, чем обеспечивает их более сбалансированное и эффективное использование.

Полученные результаты показывают, что при использовании метода дополнения Шура оптимальный выбор алгоритма формирования матрицы дополнения Шура зависит от размера подобластей/подматриц, на которые делится расчётная сетка или матрица коэффициентов системы. Если в одной подобласти находится относительно небольшое число ячеек

сетки (< 5000) или неизвестных (< 1500 — для внутренних и < 2500 — для граничных), то, для формирования матрицы дополнения Шура, эффективнее использовать прямые методы нахождение обратных матриц, и, как следствие, задействовать только CPU. В другом случае наиболее эффективны итерационные алгоритмы, использующие для вычислений несколько графических ускорителей.

Интерфейсная система уравнений дополнения Шура эффективно решается на нескольких GPU итерационными методами, в этом случае время решения сокращается в десятки и сотни раз.

Вычислительные эксперименты выполнены на вычислительных модулях гибридного кластера «Уран» ИММ УрО РАН, которые содержат два процессора Intel Xeon E5675 и восемь графических ускорителей NVIDIA Tesla M2090.

Литература

1. Haynsworth E.V. On the Shur Complement // Basel Mathematical Notes. – 1968. – №20. – 17 p.

2. Фаддеев Д.К., Фаддеева В.Н. Вычислительные методы линейной алгебры. – М. Физматгиз, 1960. – 656 с.

3. Przemieniecki J.S. Theory of Matrix Structural Analysis. – New York: McGaw-Hill, 1968. – 480 p.

4. Giraud L., Haidar A., Saad Y. Sparse approximations of the Schur complement for parallel algebraic hybrid solvers in 3D // Numerical Mathematics. – 2010. – V.3 – P. 276-294.

5. Rajamanickam S., Boman E.G., Heroux M.A. ShyLU: A Hybrid-Hybrid Solver for Multicore Platforms // IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS), 21-25 May 2012. – P. 631-643.

6. Корнеев В.Г., Енсен С. Эффективное предобуславливание методом декомпозиции области для p-версии с иерархическим базисом // Известия вузов. Математика. – 1999. – Т. 444, №5. – С. 37–56.

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

8. Kopysov S.P., Krasnopyorov I.V., Novikov A.K., Rychkov V.N. Parallel Distributed Object-Oriented Framework for Domain Decomposition // Domain Decomposition Methods in Science and Engineering. – Springer, 2005. – V. 40. — P. 605-614.

9. Копысов С.П. Оптимальное разделение области для параллельного метода подструктур // Сб. трудов Пятого Всероссийского семинара «Сеточные методы для решения краевых задач и приложения». – Казань: Изд-во КГУ, 2004. – С. 121–124.

10. 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. – №3. – С. 137-147

11. Karypis G., Kumar V. Parallel multilevel k-way partitioning scheme for irregular graphs // SIAM Rev. 1999. – V.41, №2. – P. 278 – 300.