ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ»
(Н И У « Б е л Г У »)

ИНСТИТУТ ИНЖЕНЕРНЫХ ТЕХНОЛОГИЙ И ЕСТЕСТВЕННЫХ НАУК

КАФЕДРА МАТЕМАТИЧЕСКОГО И ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ИНФОРМАЦИОННЫХ СИСТЕМ

ИТЕРАЦИОННЫЙ МЕТОД РЕШЕНИЯ СЛАУ НА ОСНОВЕ МОДИФИЦИРОВАННОГО СТАБИЛИЗИРОВАННОГО МЕТОДА БИСПОРЯЖЁННЫХ ГРАДИЕНТОВ С ИСПОЛЬЗОВАНИЕМ ТЕХНОЛОГИИ CUDA

Выпускная квалификационная работа
обучающегося по направлению подготовки
02.03.02 Фундаментальная информатика и информационные технологии
очной формы обучения, группы 07001401
Гребнева Максима Геннадьевича

Научный руководитель
к.т.н., ст.пр., Лихошерстный А.Ю.

БЕЛГОРОД 2018

СОДЕРЖАНИЕ

ВВЕДЕНИЕ 4
1. СОВРЕМЕННЫЕ МЕТОДЫ РЕШЕНИЯ СЛАУ 7
1.1 Основные понятия и определения 7
1.2 Практическое применение СЛАУ 9
1.3 Прямые методы решения СЛАУ 10
1.3.1 Метод Крамера 11
1.3.2 Метод Гаусса 12
1.3.3 Метод обратной матрицы 14
1.4 Итерационные методы решения СЛАУ 15
1.4.1 Метод Якоби и метод Зейделя 16
1.4.2 Метод сопряженных градиентов 20
1.5 Способы хранения разреженных матриц в памяти вычислительных устройств 22
1.5.1 Координатный формат хранения 23
1.5.2 Разреженный строчный формат 26
2. РАЗРАБОТКА МОДИФИЦИРОВАННОГО АЛГОРИТМА РЕШЕНИЯ СЛАУ МЕТОДОМ СТАБИЛИЗИРОВАННЫХ БИСОПРЯЖЕННЫХ ГРАДИЕНТОВ 28
2.1 Стабилизированный метод бисопряженных градиентов 28
2.2 Итерационный алгоритм стабилизированного метода бисопряженных градиентов 30
2.3 Использование CSR формата для хранения разреженных матриц при решении СЛАУ стабилизированным методом бисопряженных градиентов 35

3. ПРОГРАММНАЯ РЕАЛИЗАЦИЯ СТАБИЛИЗИРОВАННОГО МЕТОДА БИСОПРЯЖЕННЫХ ГРАДИЕНТОВ С ИСПОЛЬЗОВАНИЕМ ТЕХНОЛОГИИ CUDA 38
3.1 Программная реализация последовательной версии алгоритма 38
3.2 Программная реализация параллельной версии алгоритма 43
3.3 Оценка эффективности модификации итерационного алгоритма на основе проведения вычислительных экспериментов 48
ЗАКЛЮЧЕНИЕ 55
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 57
ПРИЛОЖЕНИЕ 61

ВВЕДЕНИЕ

На сегодняшний день, с активным ростом потребности в решении сложных вычислительных задач, наблюдается высокий спрос на разработку программного обеспечения, позволяющего корректно производить точные расчеты с использованием современной вычислительной техники.

Применение программного обеспечения, разработанного для автоматизации решения различных технических задач, и в частности, для произведения сложных и трудоемких вычислительных расчетов, наблюдается в строительстве, машиностроении, авиастроении, а также в системах автоматизированного проектирования.

Из известных математических проблем и задач, наибольшую потребность в оптимизации использования вычислительных ресурсов представляют задачи, в которых присутствуют многочисленные однотипные вычислительные операции, выполняемые над предельно большими массивами данных, и в частности над матрицами.

В трудоемких математических задачах, обычно, в качестве входных или промежуточных данных, выступают матрицы больших размерностей. С точки зрения условий подобных задач, матрицы представляются в виде систем алгебраических линейных уравнений (СЛАУ). Очевидно, что корректным результатом такой задачи, является нахождение решения промежуточной или входной СЛАУ.

Среди многочисленных методов решения систем алгебраических линейных уравнений, выделяется один из часто используемых и устойчивых методов – стабилизированный метод бисопряженных градиентов (англ. Biconjugate gradient stabilized method, BiCGStab).

Основное преимущество BiCGStab, помимо устойчивости, заключается в том, что данный метод, использует, в основном, стандартные вычислительные операции, производимые над векторами и матрицами,

которые в целом, можно оптимизировать, посредством их программной параллелизации.

Основной целью данной выпускной квалификационной работы является разработка и программная реализация итерационного алгоритма решения СЛАУ на основе модифицированного стабилизированного метода бисопряженных градиентов с использованием технологии NVIDIA CUDA. Поставленная цель достигается решением следующих задач:

- Обзор существующих методов решения систем линейных алгебраических уравнений;
- Разработка модифицированного алгоритма стабилизированного метода биспоряженных градиентов с экономией видеопамяти;
- Программная реализация разработанного алгоритма с использованием технологии CUDA;
- Оценка эффективности разработанного алгоритма на основе проведения вычислительных экспериментов;

Выпускная квалификационная работа включает в себя: введение, три главы, и заключение.

В первой главе производится сравнительный анализ, существующих итерационных методов предназначенных для решения систем линейных алгебраических уравнений. Подробно описываются и сравниваются оптимизированные форматы хранения разреженных матриц в памяти вычислительных устройств.

Вторая глава посвящена подробному описанию стабилизированного метода биспоряженных градиентов. Приводится описание модификаций метода, связанных со значительной оптимизацией вычислений при работе с разреженными матрицами. С использованием языка блок-схем описывается итерационный процесс предлагаемого метода.

В третьей главе описываются аспекты разработки программного средства с использованием технологии CUDA. Производится анализ результатов вычислительного эксперимента, полученных в ходе

тестирования последовательной и параллельной версии разработанного приложения с использованием данных, представленных в виде разреженных матриц различной размерности и плотности.

В заключении приводится краткий анализ полученных в ходе работы результатов. Подробно описываются итоги проделанной работы, а также преимущества и недостатки разработанного программного средства.

Выпускная квалификационная работа состоит из 78 страниц, 14 рисунков, 9 таблиц и приложения.

1. СОВРЕМЕННЫЕ МЕТОДЫ РЕШЕНИЯ СЛАУ

1.1 Основные понятия и определения

Системой линейных алгебраических уравнений или сокращенно СЛАУ, называют некоторое множество, содержащее m линейных уравнений, каждое из которых содержит n неизвестных:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots a_{2n}x_n = b_2 \\ \ldots \\ a_{n1}x_m + a_{n2}x_m + \cdots a_{mn}x_n = b_n \end{cases} \quad (1.1)$$

В матричной форме систему (1.1) можно записать в следующем виде:

$$Ax = b \quad (1.2)$$

Где A – квадратная матрица или матрица системы размерности $m \times m$ (1.3), состоящая из коэффициентов матрицы (1.4):

$$\begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1m} \\ a_{21} & a_{22} & \ldots & a_{2m} \\ \ldots & \ldots & \ldots & \ldots \\ a_{m1} & a_{m2} & \ldots & a_{mm} \end{pmatrix} \quad (1.3)$$

$$a_{ij} \ (i = \overline{1,m}, \ j = \overline{1,m}) \quad (1.4)$$

b – вектор-столбец свободных членов (в дальнейшем T – операция транспонирования):

$$b = (b_1, \ldots, b_n)^T \quad (1.5)$$

В случае если каждое значения вектора (1.5) равно нулю, то СЛАУ (1.2) будет называться однородной. Если среди значений свободных членов вектора (1.5) имеется хотя бы одно, отличное от нуля значение, то СЛАУ (1.2) считается неоднородной.

x – вектор-столбец решения системы:

$$x = (x_1, \dots, x_n)^T \tag{1.6}$$

Как правило, числовые значения вектора (1.5), как и сам вектор *x,* называют корнями системы линейных алгебраических уравнений[3]. Любая однородная СЛАУ имеет хотя бы одно решение – нулевое, или по другому, тривиальное:

$$x_1 = x_2 = \dots = x_n = 0 \tag{1.7}$$

Любая СЛАУ (1.2) будет считаться совместной, если она имеет хотя бы одно решение, в противном случае такая система считается несовместной[3].

Система линейных алгебраических уравнений (1.2) будет считаться определенной, только в том случае если она совместна, и имеет ровно одно решение[3]. В противном случае, то есть в том случае если СЛАУ (1.2) имеет бесконечное множество решений, система (1.2) будет называться неопределенной.

В общем виде матричную форму записи СЛАУ (1.2) удобно представить следующим образом:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} \tag{1.8}$$

Довольно часто для более понятного и краткого представления условий задач используют следующую форму записи СЛАУ:

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}; \quad \tilde{A} = \left(\begin{array}{cccc|c} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} & b_n \end{array}\right); \quad x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}; \ b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} \tag{1.9}$$

Где $\tilde{A}$ - расширенная матрица системы. Данную матрицу получают путем добавления к матрице системы (1.3) вектор-столбец свободных членов (1.5), при этом, разделяя матрицу и столбец вертикальной чертой.

Основной целью решения систем линейных алгебраических уравнений (1.1) является нахождения вектора-столбца решения системы(1.6), которое обращает выражение (1.2) в верное тождество[2].

1.2 Практическое применение СЛАУ

Как показывает история, за многое количество столетий, большая часть математических задач была тесно связана, как с решением различных неравенств и уравнений, так и систем состоящих из них. На текущий момент времени, большая часть историков сошлась на том, что умение решать системы линейных неравенств и уравнений зародилось еще в Древнем Вавилоне и Египте. Интересен тот факт, что в одной из первых книг посвященных математике – “Математика в девяти книгах”, были изложены основные правила решения систем линейных уравнений, с относительно малой размерностью основной матрицы системы[1].

На данный момент, системы линейных алгебраических уравнений широко используются в различных задачах экономики, физики, химии, географии и других науках[2].

Необходимость в решении СЛАУ возникает при использовании широкого класса математических моделей используемых при автоматизированном проектировании различной радиоэлектронной техники[8]. Так, например, задача рассеяния электромагнитных волн воздушными и наземными радиолокационными объектами, которая является одной из основных проблем электромагнитной теории, не обходится без вычисления СЛАУ с плотной матрицей системы.

СЛАУ и основные методы ее решения широко используются в задачах проектирования антенных систем с заданными требованиями к полю излучения в вертикальной и горизонтальной плоскости.

Как правило, задачи динамики вязкой жидкости и тепломассопереноса, описываемых при помощи разностной аппроксимации многомерных

дифференциальных эллиптических уравнений, сводятся к решению систем линейных алгебраических уравнений, основная матрица которых имеет относительно большую размерность.

Практическое применение математических матриц в перечисленных научных задачах, порождает высокий спрос на вычислительные программные средства, позволяющие автоматизировать стандартные операции, выполняемые над матрицами, и в частности решения СЛАУ, что в целом позволяет не тратить время на длительное моделирование или представление какой-либо математической задачи посредством рутинных ручных вычислений. Современные программные средства и вычислительная техника, позволяют в должной мере моделировать решения большинство различных задач.

1.3 Прямые методы решения СЛАУ

Основные методы решения систем линейных алгебраических уравнений (1.1), делятся на два типа:

- Прямые методы;
- Итерационные методы;

Чаще всего прямые методы называют точными методами[8]. К этому типу методов относятся методы, которые в предположении, что большая часть результатов вычислительных операций ведется без округления, позволяют получить точные значения неизвестных членов вектора (1.6).

Основным преимуществом таких методов является простота, универсальность, а также и конечность выполняемых вычислительных операций.

К основным недостаткам прямых методов относится в первую очередь невозможность их применения к СЛАУ, основная матрица которых, имеет относительно высокую размерность ($n > 500$). Этот весомый недостаток

обусловлен возникновением значительных погрешностей получаемых в ходе выполнения вычислительных операций.

Известными прямыми методами считаются: метод Крамера, метод Гаусса и метод обратной матрицы[3]. Перечисленные методы будут подробно описаны в пунктах 1.3.1, 1.3.2 и 1.3.3 соответственно.

1.3.1 Метод Крамера

Метод Крамера был реализован в 1740 году известным ученым-математиком Габриэлем Крамером[3]. Данный метод является одним из популярных и часто используемых прямых методов, позволяющих решать системы линейных алгебраических уравнений с основной матрицей системы различных размерностей.

Перейдем к рассмотрению самого метода. Определим стандартное условие. Пусть требуется решить СЛАУ вида (1.1), или иными словами найти такой вектор (1.6) при котором выражение (1.2) обращалось в верное тождество. Будем считать, что основная матрица системы (1.3) не вырожденная, или иными словами, ее определитель отличен от нуля:

$$det\ A \neq 0 \tag{1.10}$$

В таком случае СЛАУ (1.2) имеет одно единственное решение, которое может быть найдено посредством метода Крамера[5]. Подробнее рассмотрим сам алгоритм решения СЛАУ. В первую очередь необходимо проверить условие (1.10), вычислив определитель основной матрицы системы (1.11).

$$det\ A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mm} \end{pmatrix} \tag{1.11}$$

Согласно теореме Крамера, при выполнении условия (1.10) решение

системы (1.1) находится по формулам Крамера:

$$x_i = \frac{det\ A_i}{det\ A} \qquad (1.12)$$

Где *det Ai* определитель матрицы системы, в котором вместо i-ого столбца стоит вектор-столбец правой части (1.5), иначе данное выражение можно записать в виде:

$$\Delta_{x_1} = \begin{vmatrix} b_1 & a_{12} & \cdots & a_{1n} \\ b_2 & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ b_n & a_{n2} & \cdots & a_{nn} \end{vmatrix} \quad \Delta_{x_2} = \begin{vmatrix} a_{11} & b_1 & \cdots & a_{1n} \\ a_{21} & b_2 & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & b_n & \cdots & a_{nn} \end{vmatrix} \quad \cdots \quad \Delta_{x_n} = \begin{vmatrix} a_{11} & a_{12} & \cdots & b_1 \\ a_{21} & a_{22} & \cdots & b_2 \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & b_n \end{vmatrix} \qquad (1.13)$$

Не смотря на всю простоту и не большое количество вычислительных операций, данный метод хорошо себя показывает при решении СЛАУ, у которой размерность матрицы системы относительно не велика[3]. В случаях если количество неизвестных n в СЛАУ (1.1) относительно большое, например при $n > 200$, целесообразно использовать более точные методы.

1.3.2 Метод Гаусса

Метод Гаусса является одним из простых и часто применяемых при решении СЛАУ (1.1) прямых методов. Данный метод отличается относительно малым объемом вычислительных операций[1].

Перейдем к рассмотрению метода. Процедура нахождения корней СЛАУ сводится к последовательной реализации прямого и обратного хода[1].

Реализация прямого хода сводится к последовательному преобразованию исходной матрицы (1.3) к треугольной матрице. На первом этапе прямого хода выбирается ведущий элемент, один из коэффициентов первой либо последней строки матрицы (1.3). Уравнение с ведущим элементом делится на коэффициент a_{11} и приобретает следующий вид:

$$x_1 + \sum_{j=2}^{n} \frac{a_{1j}}{a_{11}} x_j = \frac{b_1}{a_{11}} . \tag{1.14}$$

Результат выражения выше умножается на коэффициент *a21* и вычитается из второго выражения системы (1.1). Затем уравнение (1.14) последовательно умножается a_{i1}, где $i = 3, ..., n$ (n – размерность матрицы (1.2)) и вычитается из соответствующего i-ого уравнения исходной системы. В результате помимо уравнения (1.14), мы получим систему из $n - 1$ уравнений, которое содержит $n - 1$ неизвестных:

$$\sum_{j=2}^{n} a_{ij}^{(1)} \cdot x_j = b_i^{(1)} , \quad i = 2, ..., n. \tag{1.15}$$

Последующие $(n - 1)$ шагов выполняются аналогично. В результате продолжения процедуры последовательного исключения неизвестных получим уравнение, содержащее только одно неизвестное:

$$a_{nn}^{(n-1)} \cdot x_n = b_n^{(n-1)} . \tag{1.16}$$

Таким образом, исходная система (1.1) приведена к эквивалентной системе с треугольной матрицы системы.

Далее подробно рассмотрим следующий шаг метода – обратны ход. На первом шаге обратного хода из уравнения (1.16) сразу вычисляется n-й корень системы:

$$x_n = \frac{b_n^{(n-1)}}{a_{nn}^{(n-1)}} \tag{1.17}$$

В предпоследнем шаге (n-2)-м шаге обратного хода получаем уравнение:

$$x_{n-1} + \frac{a_{n-1,n}^{(n-2)}}{a_{n-1,n-1}^{(n-2)}} \cdot x_n = b_{n-1}^{(n-2)} \tag{1.18}$$

Второй шаг обратного хода заключается в подстановке выражения (1.17) в выражение (1.18) и дальнейшие вычисление корня x_{n-1}.Далее аналогичным образом последовательно применяются уравнения, полученные ранее на каждом шаге прямого хода. Таким образом, решаем поставленную задачу, получая значения вектора (1.6).

Из достоинств данного метода следует выделить простой алгоритм, и малый объем выполняемых вычислительных операций. Однако, как и в метод Краммера, в данном методе имеется существенный недостаток, связанный с резким увеличением времени и погрешности результатов промежуточных вычислительных операций, с ростом размерности матрицы системы.

1.3.3 Метод обратной матрицы

Метод обратной матрицы является одним из самых простых прямых методов, позволяющих решать системы линейных алгебраических уравнений.

Метод базируется на вычислении обратной матрицы системы (1.2). То есть матрицы, для которой справедливо следующее выражение:

$$A \cdot A^{-1} = A^{-1} \cdot A = E \tag{1.19}$$

Где A^{-1} - обратная матрица, E – единичная матрица.

Для существования обратной матрицы, необходимо и достаточно, чтобы матрица (1.3) была не вырожденной, или иными словами выполнялось условие (1.10).

Пусть имеется матрица вида (1.3), тогда обратную к матрице (1.3) матрицу можно вычислить по формуле:

$$A^{-1} = \frac{1}{\det A}\begin{pmatrix} A_{11} & A_{12} & \dots & A_{1n} \\ A_{21} & A_{22} & \dots & A_{2n} \\ \dots & \dots & \dots & \dots \\ A_{n1} & A_{n2} & \dots & A_{nn} \end{pmatrix}^T = \frac{1}{\det A}\begin{pmatrix} A_{11} & A_{21} & \dots & A_{n1} \\ A_{12} & A_{22} & \dots & A_{n2} \\ \dots & \dots & \dots & \dots \\ A_{1n} & A_{n2} & \dots & A_{nn} \end{pmatrix} \tag{1.20}$$

Где алгебраическое дополнение к каждому элементу матрицы (1.3) находится следующим образом:

$$A_{ij} = (-1)^{i+j} M_{ij} \qquad (1.21)$$

За M_{ij} обозначим минор матрицы.

После проверки условия (1.10) и соответственного вычисления обратной матрицы, можно перейти к последнему и окончательному шагу метода. На конечном шаге находятся значения вектора (1.6), посредством умножения обратной матрицы на вектор-столбец (1.5):

$$x = A^{-1} \times b \qquad (1.22)$$

К преимуществам данного метода, безусловно, можно отнести конечность и простоту вычислительных операций. Основной недостаток данного метода заключается в трудоемкости вычисления обратной матрицы системы, а также невозможности применения данного метода для СЛАУ с большим количеством неизвестных.

1.4 Итерационные методы решения СЛАУ

Итерационные методы, или по-другому – приближенные методы, основаны на использовании повторяющегося циклического процесса, в котором большая часть выполняемых вычислительных операций, в большинстве случаев, выполняется без округления. Такие методы позволяют получить значения неизвестных членов вектора (1.6) с заранее заданной точностью, и исходя из этого, активно применяются при решении СЛАУ основная матрица которых имеет высокую размерность.

К известным итерационным методам относятся: метод Якоби, метод Зейделя, обобщенный метод минимальных невязок, а также наиболее применимый при решении реальных задач – метод бисопряженных

градиентов. Основные итерационные методы будут подробно описаны в следующих пунктах.

1.4.1 Метод Якоби и метод Зейделя

Метод Якоби, и в частности его модификация – метод Зейделя, довольно часто используется при решении различных прикладных задач[6].

Перейдем к детальному рассмотрению итерационного процесса метода Якоби. Для начала, предположим, что все диагональные элементы матрицы (1.3) отличны от нуля:

$$a_{ii} \neq 0 \quad (i = 1, 2, ..., n) \tag{1.23}$$

Тогда разрешим каждое уравнение i-е системы (1.2) относительно x_i неизвестного. В результате получим систему уравнений эквивалентную (1.1):

$$\begin{cases} x_1 = d_1 + c_{12}x_2 + c_{13}x_3 + \cdots + c_{1n}x_n \\ x_2 = d_2 + c_{22}x_2 + c_{23}x_3 + \cdots + c_{2n}x_n \\ \quad\quad\quad \cdots \\ x_n = d_n + c_{n2}x_2 + c_{n3}x_3 + \cdots + c_{nn-1}x_{n-1} \end{cases} \tag{1.24}$$

В данной системе введены следующие обозначения:

$$c_{ij} = -\frac{a_{ij}}{a_{ii}} \ (\text{при } i \neq j), \quad c_{ij} = 0 \ (\text{при } i = j), \quad \text{где } i, j = 1, 2, ..., n; \tag{1.25}$$

$$d_i = \frac{b_i}{a_{ii}}, \qquad \text{где } i = 1, 2, ..., n. \tag{1.26}$$

Определим матрицу коэффициентов C:

$$C = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix} \tag{1.27}$$

А также столбец коэффициентов D:

$$D = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{bmatrix} \quad (1.28)$$

Затем перепишем систему (1.24) в матричной форме:

$$x = D + Cx \quad (1.29)$$

Перед началом итерационного процесса примем вектор D за начальное приближение вектора $x^{(0)}$.Тогда получим первое приближение корней системы:

$$x^{(1)} = D + Cx^{(0)} \quad (1.30)$$

Приближение на k-ой итерации будет определяться следующим выражением:

$$x^{(k+1)} = D + Cx^{(k)} \quad (1.31)$$

В более подробном виде выражение (1.31) выглядит следующим образом:

$$x_i^{(0)} = d_i, \qquad x_i^{(k+1)} = d_i + \sum_{j=1}^{n} c_{ij} x_j^{(k)} \quad (1.32)$$

Таким образом, с каждой последующей итерацией значение членов вектора x будет приближаться к действительному решению системы (1.2).

Критерий окончания итерационного процесса определятся следующим выражением:

$$\left\| x^{(n+1)} - x^{n} \right\| < \varepsilon \quad (1.33)$$

Где ε - любое числовое значение. Достаточное условие сходимости для метода Якоби определено как:

$$||C|| < 1 \quad (1.34)$$

Однако практическое применение метода показывает, что из-за округления результатов промежуточных операций сходимость к точному

решению может быть нарушена. Для корректного получения решения СЛАУ посредством метода Якоби, необходимо чтобы все диагональные коэффициенты матрицы (1.3) были отличны от нуля (1.23).

В общем случае можно доказать, что условие сходимости (1.34) если для системы (1.1) справедливо следующее неравенство:

$$|a_{ii}| > \sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{ij}|, \quad i = 1, 2, \ldots, n \tag{1.35}$$

Иными словами метод Якоби даст единственное решение системы (1.1), если в каждом уравнении этой системы модуль диагонального коэффициента превышает сумму модулей основных коэффициентов при неизвестных в этом же уравнении[6].

Довольно часто при решении прикладных задач используют итерационный метод Зейделя. Данный метод является модификацией метода Якоби.

Метод Зейделя базируется на вышеописанной схеме и также используется рекуррентное соотношение (1.31). Суть модификации метода заключается в том, что при вычислении *k*-ого приближения корня x_i вектора (1.6) СЛАУ (1.2) учитываются уже вычисленные не предыдущих итерациях приближения корней $x_1, x_2, \ldots, x_{i-1}$.

Пусть исходная система (1.1) успешно преобразована в систему (1.24). Затем также как и в методе Якоби определяет вектор начальных приближений (1.30). С учетом выбранного начального приближения опишем итерационный процесс метода Зейделя:

$$x_1^{(k+1)} = d_1 + \sum_{j=1}^{n} c_{1j} x_j^{(k)} ,$$

$$x_2^{(k+1)} = d_2 + c_{21}\, x_1^{(k+1)} + \sum_{j=2}^{n} c_{2j} x_j^{(k)} ,$$

$$\ldots$$

$$x_i^{(k+1)} = d_i + \sum_{j=1}^{i-1} c_{ij} x_j^{(k+1)} + \sum_{j=i}^{n} c_{ij} x_j^{(k)} ,$$

$$\ldots$$

$$x_n^{(k+1)} = d_n + \sum_{j=1}^{n-1} c_{ij} x_j^{(k+1)} + c_{nn}\, x_n^{(k)} . \qquad (1.36)$$

Для определения условий сходимости метода Зейделя целесообразно ввести понятие нормальной системы линейных уравнений. Система (1.1) называется нормальной, если выполняются следующие условия:

1. Матрица (1.3) коэффициентов при неизвестных симметрична, т.е. $a_{ij} = a_{ji}$.
2. Квадратичная форма U, соответствующая матрице (1.2):

$$U = \sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij} x_i x_j \qquad (1.37)$$

положительно определена, или иными словами принимает неотрицательные значения при любых аргументах x_i ,при этом обращается в ноль, только в том случае, когда все аргументы x_i равны нулю.

Если матрица системы (1.3) имеет определитель отличный от нуля (1.10), то система уравнений общего вида (1.1) может быть приведена к нормальному виду. Для этого обе части матричного выражения необходимо умножить слева на транспонированную матрицу A`:

$$A'Ax = A'b \qquad (1.38)$$

Произведение квадратных матриц $A`A$ представляет собой матрицу A_N размерностью n × n. Доказано, что матрица A_N симметрическая и соответствует положительно определенной квадратичной форме[2]. Произведение $A'b = b_N$ является n-мерным вектором. Исходя из этого, выходит, что нормализация исходной системы (1.1) дает эквивалентную систему вида:

$$A_N x = b_N \tag{1.39}$$

После реализации процесса получения нормальной системы (1.39) ее можно преобразовать к системе вида (1.24)[6]. Далее, имеет место быть доказанная теорема: если СЛАУ вида (1.1) - нормальная, то итерационный процесс метода Зейделя (1.36) для эквивалентной системы (1.24) сходится к точному решению системы при любом выборе начального приближения.

Основным из преимуществ метод Зейделя над Якоби является высокая скорость сходимости итерационного процесса, что в целом позволяет использовать именно метод Зейделя для решения большинства прикладных задач. Также стоит отметить, что оба метода отлично себя показывают при решении СЛАУ с большим количеством переменных, и соответственно большой размерности основной матрицы системы. Из недостатков данных методов стоит отметить отсутствие желаемой массовости, обусловленной преимущественно строгими условиями сходимости каждого из методов.

1.4.2 Метод сопряженных градиентов

Метод сопряженных градиентов является одним из часто используемых в практике итерационным методом для решения СЛАУ основная матрица которой, имеет большую размерность.

Метод сопряженных градиентов является итерационным методом Крыловского типах[2]. Рассмотрим данный метод более подробно.

Как и прежде, имеется система линейных алгебраических уравнений (1.2). Матрицей системы (1.2) должна обладать следующим свойством:

$$A = A^T > 0 \tag{1.40}$$

Или иными словами, матрица системы (1.2) должна быть симметрично положительной матрицей. Тогда, процесс решения СЛАУ (1.2) можно представить, как минимизацию функционала:

$$(Ax, x) - 2(b, x) \to min \tag{1.41}$$

Далее можно определить алгоритм решения СЛАУ методом сопряженных градиентов. На первом шаге, как и в предыдущих итерационных методах, выбираются значения начального приближения вектора (1.6) и полагается:

$$\begin{aligned} r^0 &= b - Ax^0 \\ z^0 &= r^0 \end{aligned} \tag{1.42}$$

Далее, в общем виде, определим *k*-ую итерацию метода:

$$\begin{aligned} \alpha_k &= \frac{(r^{k-1}, r^{k-1})}{(Az^{k-1}, z^{k-1})} \\ x^k &= x^{k-1} + \alpha_k z^{k-1} \\ r^k &= r^{k-1} - \alpha_k A z^{k-1} \\ \beta_k &= \frac{(r^k, r^k)}{(r^{k-1}, r^{k-1})} \\ z^k &= r^k + \beta_k z^{k-1} \end{aligned} \tag{1.43}$$

где, x_k – вектор решения на k-ой итерации, r_k – вектор невязки на k-ой итерации, z_k – вектор сопряженного направления, a_k и b_k – коэффициенты.

Стандартным критериям останова итерационного процесса (1.43) может осуществляться либо по условию (1.33) либо по условию малости относительной невязки:

$$\frac{||r^k||}{||b||} < \varepsilon \tag{1.44}$$

Также, одним из условий прекращения итерационного процесса является установка максимального количества итераций[3].

Основным недостатком предложенного метода является не малое количество вычислительных операций на каждом k-ом приближении итерационного процесса. Однако, стоит отметить, что данный метод отлично подходит для решения СЛАУ, матрица системы которой имеет высокую размерность, что в целом позволяет использовать данный метод при решении различных прикладных задач.

1.5 Способы хранения разреженных матриц в памяти вычислительных устройств

На данный момент одной из актуальных задач в области вычислительной техники и информационных технологий, является задача рационального хранения больших объемов произвольных данных. Вне зависимости от конкретного типа данных (числовые, символьные последовательности), необходим способ хранения, а также оптимизированный алгоритм, позволяющий работать с огромным количеством универсальной информации наиболее эффективно. Как показывает практика применения подобных алгоритмов, далеко не всегда удается с успехом реализовать подобную задачу.

Очевидно, что огромной проблемой с точки зрения рационального распределения больших объемов данных в памяти вычислительных устройств, является проблема хранения разреженных данных.

В общем случае, разреженные данные, представляют собой совокупность данных состоящих преимущественно из пустых или нулевых значений. Данные такого типа занимают избыточную область памяти вычислительного устройства, что в целом является особой проблемой, в случае хранения сравнительно больших объемов информации.

Хранение в памяти компьютера разреженных данных, с точки зрения наименьшей избыточности, предполагает использование конкретного

формата хранения, в общем случае подобранного исходя из условий поставленной задачи.

Представление различных графов, карт, изображений матриц и т.д., в частности, имеющих относительно большое количество пустых либо нулевых элементов, и в общем случае подходящих под определение разреженных данных, обычно сопровождается использованием наименее избыточного формата хранения.

Корректный выбор конкретного формата данных сопровождается положительной оптимизацией в плане распределения памяти и в общем случае, улучшением производительности вычислительного устройства с точки зрения аппаратных ресурсов.

1.5.1 Координатный формат хранения

Координатный формат хранения (сокр. англ. Coordinate list – COO) является одним из самых простых способов хранения произвольной разреженной матрицы. Реализованный во множестве различных программных библиотек, данный формат, представляет собой де факто стандарт хранения, как разреженных, так и плотных матриц произвольной размерности.

Определяющим принципом данного формата, является сохранение в некотором списке координат основных ненулевых элементов матрицы (1.45).

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \tag{1.45}$$

Рассмотрим более подробно структуру хранения COO. Матрица (1.45) хранится в памяти вычислительного устройства в виде трех одномерных массивов фиксированного размера (см. рис. 1.1).

$$B = \boxed{b1}\boxed{b2}\boxed{b3} \ldots \boxed{bn}$$

Рис. 1.1. Одномерный массив фиксированной длины

где n – размерность массива. Все ненулевые элементы матрицы хранимой матрицы, т.е.:

$$a_{ij} \ (i = \overline{1,m}\,, \ j = \overline{1,m}), a_{ij} \neq 0 \quad (1.46)$$

построчно записываются в одномерный массив *VALUES*. Номер (индекс) строки матрицы (1.45), каждого ненулевого элемента матрицы (1.46) записывается в одномерный массив *ROWS*. Аналогичным образом в массив *COLS* записывается номер столбца соответствующего ненулевого элемента матрицы (1.46). Таким образом, общая структура формата COO определяется из трех основных одномерных массивов фиксированной размерности *(VALUES,ROWS,COLS)*.

В качестве примера, возьмем простую квадратную разреженную матрицу размерности 5×5 состоящую из семи ненулевых элементов (см. рис. 1.2).

0	0	2	0	0
0	-1	0	11	0
0	0	0	0	8
0	4	0	0	0
1	0	5	0	0

Рис. 1.2. Пример квадратной разреженной матрицы

В координатном формате данная матрица определена на рис. 1.3.

VALUES	2	-1	11	8	4	1	5
ROWS	1	2	2	3	4	5	5
COLS	3	2	4	5	2	1	3

Рис. 1.3. Разреженная матрица в координатном формате

В обще случае существуют две формы координатного формата: *упорядоченная форма* и *неупорядоченная форма.* Упорядоченная форма предполагает сортировку основных элементов одномерных массивов. Благодаря сортировке, обеспечивается быстрый доступ к строкам/столбцам хранимой матрицы, но в таком случае уходит довольно много времени на вставку и удаление элементов. В неупорядоченной форме наблюдается обратная ситуация: вставка и удаление элементов осуществляется довольно быстро, а на поиск необходимого элемента по заданному индексу, наоборот, уходит много времени.

Каждая из форм характеризуется своими особенностями реализации и выбирается исходя из условий решаемой задачи.

В общем случае, при использовании координатного формата для хранения в памяти вычислительных устройств разреженных матриц произвольной размерности ($m \times n$) и количеством ненулевых элементов в данной матрице – nz, затраты памяти составляют $3 \times nz$, т.е. на каждый одномерный массив *(VALUES,ROWS,COLS)* приходится по nz элементов.

Не смотря на простую реализацию, а также на поддержку данного формата большим количеством математико-ориентированных библиотек, COO формат предполагает медленный доступ к элементам матрицы и в целом является довольно избыточным с точки зрения используемой памяти. Также стоит отметить, что, преимущественно, из-за излишней избыточности, данный формат не часто применяется при решении реальных практических задач.

1.5.2 Разреженный строчный формат

Довольно часто при решении реальных практических задач, а в частности задач, в которых фигурируют разреженные матрицы, используют разреженный строчный формат (CSR – Compressed Sparse Rows). Разреженный строчный формат – это один из наиболее часто используемых способов хранения разреженных матриц произвольной размерности.

Использование CSR в качестве формата хранения разреженных матриц, при решении какой-либо прикладной задачи, сопровождается минимальными затратами памяти, а также оказывается очень удобным для выполнения основных математических операций производимых над хранимой матрицей, таких как: сложение, умножение, перестановка столбцов, строк, и транспонирование.

Рассмотрим более подробно структуру хранения разреженного строчного формата. Как и прежде, имеется некоторая разреженная матрица (1.45). Данная матрица хранится в виде трех одномерных массивов фиксированной длины. Все ненулевые элементы матрицы (1.46) записываются в одномерный массив *VALUES.* Номер столбца соответствующего ненулевого элемента, записывается в массив *COLS.* Элементы 3-его одномерного массива *POINTERS* записываются по следующим правилам: элементы i-ой строки хранятся в позициях матрицы (1.45) (их значения хранятся в массиве COLS), с номера *POINTERS[i]* по *POINTER[i+1] – 1*, если же в i-ой строке только нулевые элементы или иными словами, i-ая строка является пустой, то значение *POINTERS[i]* будет равно значению *POINTERS[i + 1].*

В качестве примера, возьмем простую квадратную разреженную матрицу размерности 5×5 состоящую из семи ненулевых элементов. В CSR, матрица (см. рис. 1.2) представлена на рис. 1.4.

VALUES	*2*	*-1*	*11*	*8*	*4*	*1*	*5*
COLS	*3*	*2*	*4*	*5*	*2*	*1*	*3*
POINTERS	*1*	*2*	*4*	*5*	*6*	*8*	

Рис. 1.4. Разреженная матрица в CSR формате

В общем случае, при использовании CSR формата для хранения в памяти вычислительных устройств разреженных матриц произвольной размерности (m×n) и количеством ненулевых элементов – *nz,* затраты памяти составляют – $2 \times nz + (n+1)$. На массивы *VALUES* и *COLS* приходится по nz элементов, а на массив *POINTERS* – n элементов.

CSR формат обеспечивает быстрый и эффективный доступ к строкам основной матрицы. Однако доступ к столбцам матрицы производится менее эффективно. Исходя из этого, предпочтительнее использовать разреженный строчный формат, в тех случаях, когда основные вычислительные операции производятся преимущественно над строками хранимой матрицы.

Разреженный строчный формат является одним из наименее избыточных и оптимизированных форматов хранения разреженных матриц, что в целом, позволяет его использовать в большинстве реальных прикладных задач, и в частности при решении систем линейных алгебраических уравнений основная матрица которых, имеет большую размерность.

2. РАЗРАБОТКА МОДИФИЦИРОВАННОГО АЛГОРИТМА РЕШЕНИЯ СЛАУ МЕТОДОМ СТАБИЛИЗИРОВАННЫХ БИСОПРЯЖЕННЫХ ГРАДИЕНТОВ

2.1 Стабилизированный метод бисопряженных градиентов

Стабилизированный метод биспоряженных градиентов (англ. Biconjugate gradient stabilized method, BiCGStab) представляет собой численный метод решения СЛАУ Крыловского типа. Данный метод был предложен известным Голландским ученым-математиком – Генриком Ван Дер Ворстом для решения систем линейных уравнений, матрица системы которых, является несимметричной. Метод является модификацией известного метода бисопряженных градиентов, который, в свою очередь, является неустойчивым, и исходя из этого, находит практическое применение гораздо реже чем BiCGStab.

Рассмотрим предложенный метод более подробно. Стабилизированный метод бисопряженных градиентов относится к классу проекционных методов, основной принцип которых заключается в поиске наилучшего приближения корней системы (1.6) в некотором подпространстве пространства R^n. В качестве пространства R^n берутся пространства Крылова, определенные посредством матрицы системы и невязкой первого приближения.

В случае решения СЛАУ с разреженной основной матрицей системы довольно часто используется *ILU* предобусловливание (не полная *LU* разложение), которое в общем случае позволяет значительно экономить память (в случае программной реализации), не учитывая нулевые элементы основной матрицы системы в предполагаемых матрицах *L* и *U,* и как результат получение выражения – $A \approx LU$.

Рассмотрим итерационный процесс метода. На первом шаге, как и в любом другом итерационном методе, необходимо выбрать начальные приближения членов вектора (1.6). Далее, в общем виде определяется первая итерация метода:

$$\begin{aligned} &\boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0 \\ &\rho_0 = \alpha = \omega_0 = 1 \\ &\boldsymbol{v}_0 = \boldsymbol{p}_0 = \boldsymbol{0} \end{aligned} \tag{2.1}$$

где, r^0 – невязка первой итерации, p^0 – базисный вектор,v_0 – вспомогательный вектор, ρ_0 – вспомогательный коэффициент, a,w_0 – вспомогательные скаляры. *k-я* итерация метода выглядит следующим образом:

$$\begin{aligned} &\rho_i = (\hat{\boldsymbol{r}}_0, \boldsymbol{r}_{i-1}) \\ &\beta = (\rho_i/\rho_{i-1})(\alpha/\omega_{i-1}) \\ &\boldsymbol{p}_i = \boldsymbol{r}_{i-1} + \beta(\boldsymbol{p}_{i-1} - \omega_{i-1}\boldsymbol{v}_{i-1}) \\ &\boldsymbol{v}_i = \boldsymbol{A}\boldsymbol{p}_i \\ &\alpha = \rho_i/(\hat{\boldsymbol{r}}_0, \boldsymbol{v}_i) \\ &\boldsymbol{h} = \boldsymbol{x}_{i-1} + \alpha\boldsymbol{p}_i \\ &\boldsymbol{s} = \boldsymbol{r}_{i-1} - \alpha\boldsymbol{v}_i \\ &\boldsymbol{t} = \boldsymbol{A}\boldsymbol{s} \\ &\omega_i = (\boldsymbol{t}, \boldsymbol{s})/(\boldsymbol{t}, \boldsymbol{t}) \\ &\boldsymbol{x}_i = \boldsymbol{h} + \omega_i\boldsymbol{s} \\ &\boldsymbol{r}_i = \boldsymbol{s} - \omega_i\boldsymbol{t} \end{aligned} \tag{2.2}$$

где, β – вспомогательный коэффициент, h,s – вспомогательные вектора, r^0 – невязка k-ой итерации, p^k – базисный вектор определяемые на каждой k-ой итерации, v_k – вспомогательный вектор определяемый на каждой k-ой итерации, ρ_k – вспомогательный коэффициент определяемый на каждой итерации, a,w_0 – вспомогательные скаляры определяемые на каждой итерации.

Критерием останова итерационного процесса (2.2) может быть стандартное условие малости относительной невязки (1.44). Довольно часто в качестве критерия останова, так же как и в методе сопряженных

градиентов, может выступать ограничение на количество итераций, а также выполнение условия (1.33).

Стабилизированный метод бисопряженных градиентов, является одним из наиболее часто используемых на практике итерационных методов решения СЛАУ. Основным преимуществом данного метода является массовость по отношению к основной матрице системы (1.3), что в целом позволяет использовать метод для решения СЛАУ практически с любыми матрицами системы за исключением нессеметричных матриц. Стоит отметить, что предложенный метод, как показывает практика, сходится намного быстрее чем методы описыемые ранее (метод Зейделя, метод сопряженных градиентов) и к тому же является устойчивым. Основной недостаток BiCGStab заключается в не малом количестве вычислительных операций за один цикл итерационного процесса (2.2), однако, в случае программной реализации метода, данный недостаток может оказаться не существенным при корректной параллелизации вычислительных операций связанных с умножением, сложением, вычитанием производмых над векторами и матрицами.

2.2 Итерационный алгоритм стабилизированного метода бисопряженных градиентов

Как было отмечено ранее, стабилизированный метод бисопряженнных градиентов является одним из часто используемых на практике методов решения систем линейных алгебраических уравнений. Такая популярность метода, в основном, обусловлена алгоритмической простотой промежуточных операций вычисляемых за один цикл итерационного процесса (умножение, сложение, векторов и матриц), что в целом позволяет без труда реализовать данный метод программно.

Рассмотрим итерационный алгоритм стабилизированного метода биспоряженных градиентов более подробно. В первую очередь необходимо

определить основные вычислительные операции, выполняемые за каждый цикл итерационного процесса (2.2) и определить их алгоритм. К этим операциям относятся: умножение, сложение векторов и матриц, скалярное произведение векторов, простые операции умножения, деления, сложения вычитания простых числовых коэффициентов. Имеет смысл рассмотреть только операции, имеющие вычислительную сложность более O(1).

Скалярное произведение векторов определяется следующей математической формулой:

$$(\vec{a}\,,\vec{b}) \tag{2.3}$$

а алгоритм вычисления скалярного произведения векторов произвольной размерности *N* представлен на рис. 2.1.

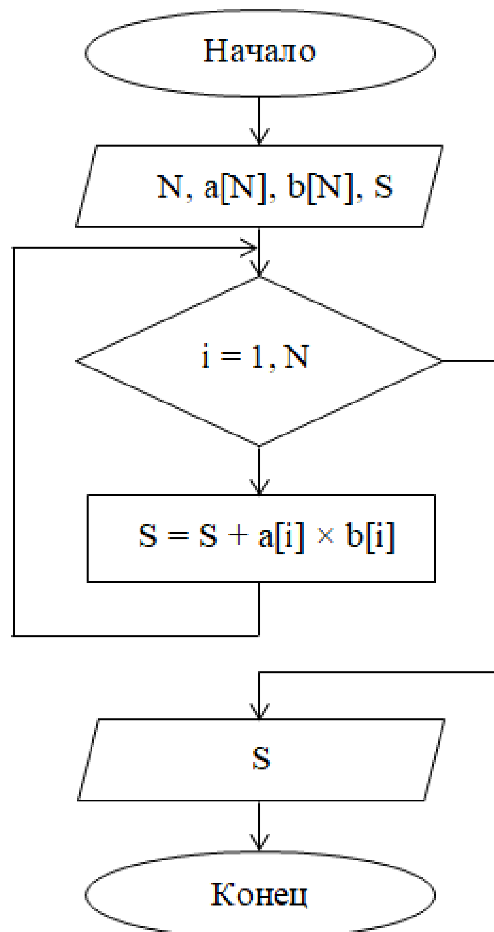

Рис. 2.1. Блок-схема алгоритма вычисления скалярного произведения векторов размерности N

Сложение/вычитание векторов определяется как сумма/разность членов двух векторов соответственно. В общем случае, алгоритм для операции сложения показан на рис. 2.2.

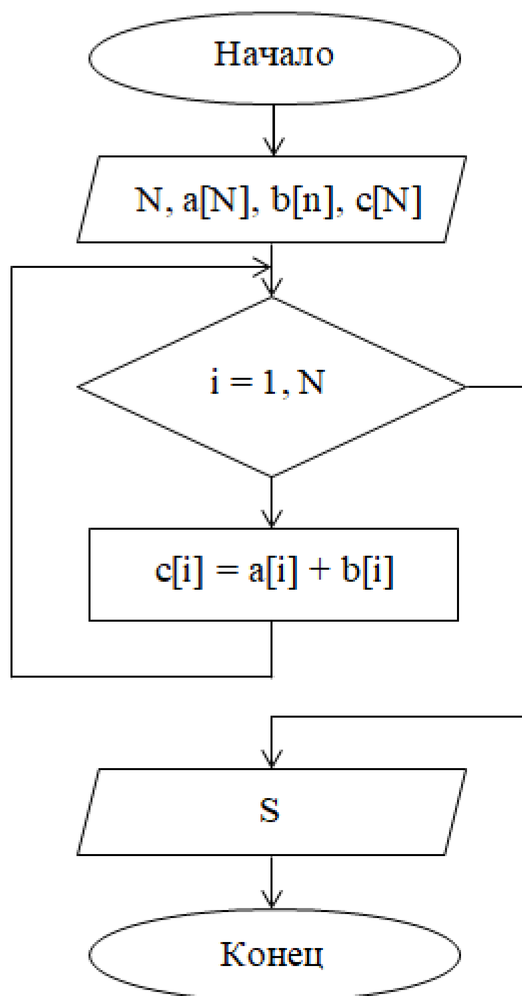

Рис. 2.2. Блок-схема алгоритма вычисления суммы векторов размерности N

Одной из самых трудоемких операций итерационного процесса (2.2) является произведение матриц. В итерационном процесса (2.2), по большей части, преобладает произведения квадратных матриц. Имеет смысл рассмотреть блок-схему операции произведения квадратных матриц (см. рис. 2.3).

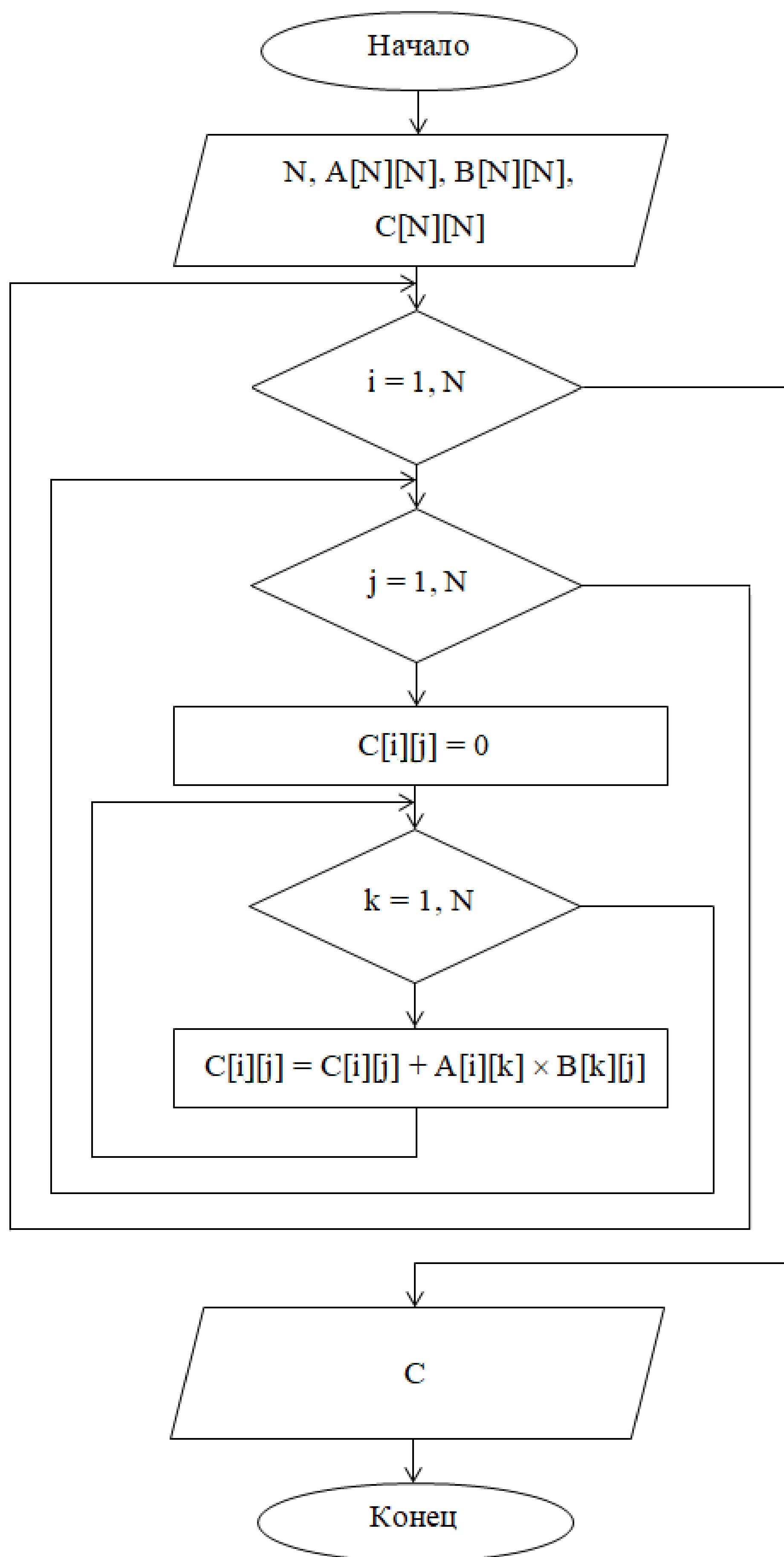

Рис. 2.3. Блок-схема алгоритма умножения квадратных матриц

Блок-схема основного итерационного процесса представлена рис. 2.4.

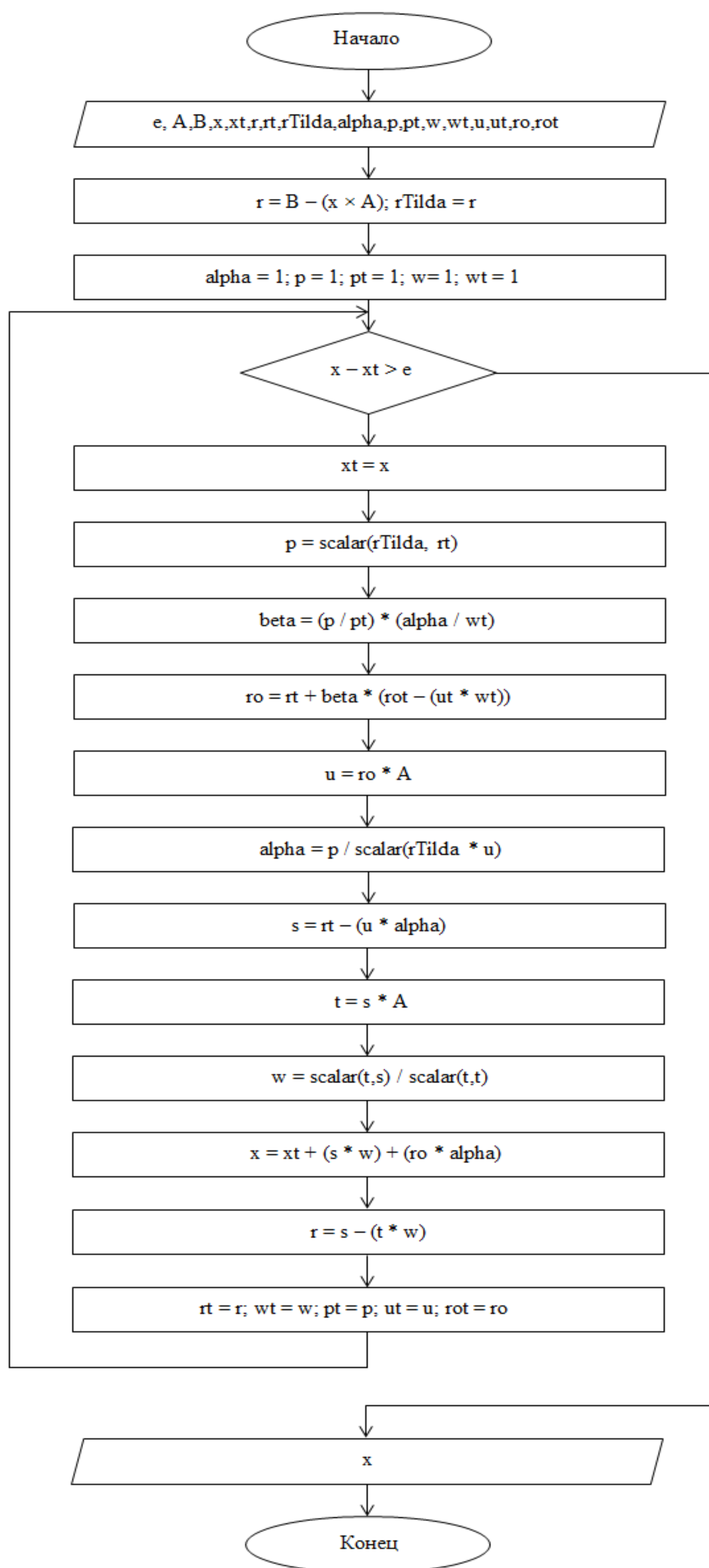

Рис. 2.4. Блок-схема итерационного алгоритма стабилизированного метода бисопряженных градиентов

где функция *scalar(a,b)* – скалярное произведение векторов a и b соответственно (2.3).

В результате подробного анализа итерационного алгоритма стабилизированного метода бисопряженных градиентов, была определена вычислительная сложность алгоритма – $O(n^2)$.

2.3 Использование CSR формата для хранения разреженных матриц при решении СЛАУ стабилизированным методом бисопряженных градиентов

Разреженный строчный формат является одним из наименее избыточных и оптимизированных форматов хранения разреженных матриц. В случае программной реализации стабилизированного метода бисопряженных градиентов, ориентированного на решения СЛАУ, основная матрица системы которых является в основном разреженной, имеет смысл повысить общую производительность итерационного алгоритма (Рис.4), за счет использования CSR формата для хранения в памяти вычислительного устройства результатов операций, выполняемых преимущественно над матрицам.

Преимущества использования разреженного строчного формата, на первый взгляд, являются очевидными. Экономия оперативной памяти вычислительного устройства, очевидным образом, положительно скажется на его производительности, особенно в случае, если система параллельно выполняет дополнительные вычислительные операции, активно использующие общую оперативную память. В некоторых случаях, из-за недостаточного объема физической памяти, корректная работа программы, которая оперирует в основном большими объемами данных, не возможна, и в лучшем случае, ее выполнение завершится ошибкой.

Для наглядности, проведем сравнение существующих форматов хранения разреженных матриц. Результаты сравнения форматов хранения разреженных матриц приведены в табл. 2.1.

Таблица 2.1

Сравнение избыточности форматов хранения разреженных матриц с 16% ненулевых элементов

Размерность разреженной матрицы	Количество памяти (МБ) выделяемой под хранение разреженной матрицы целочисленных значений для каждого формата		
	Двумерный массив	COO	CSR
1000 × 1000	3,814	1,831	1,224
5000 × 5000	95,367	45,776	30,536
12000 × 12000	549,316	263,671	175,827
20000 × 20000	1525,878	732,421	388,357

График зависимости количества выделенной памяти от размерности матрицы и выбранного формата хранения представлен на рис. 2.5.

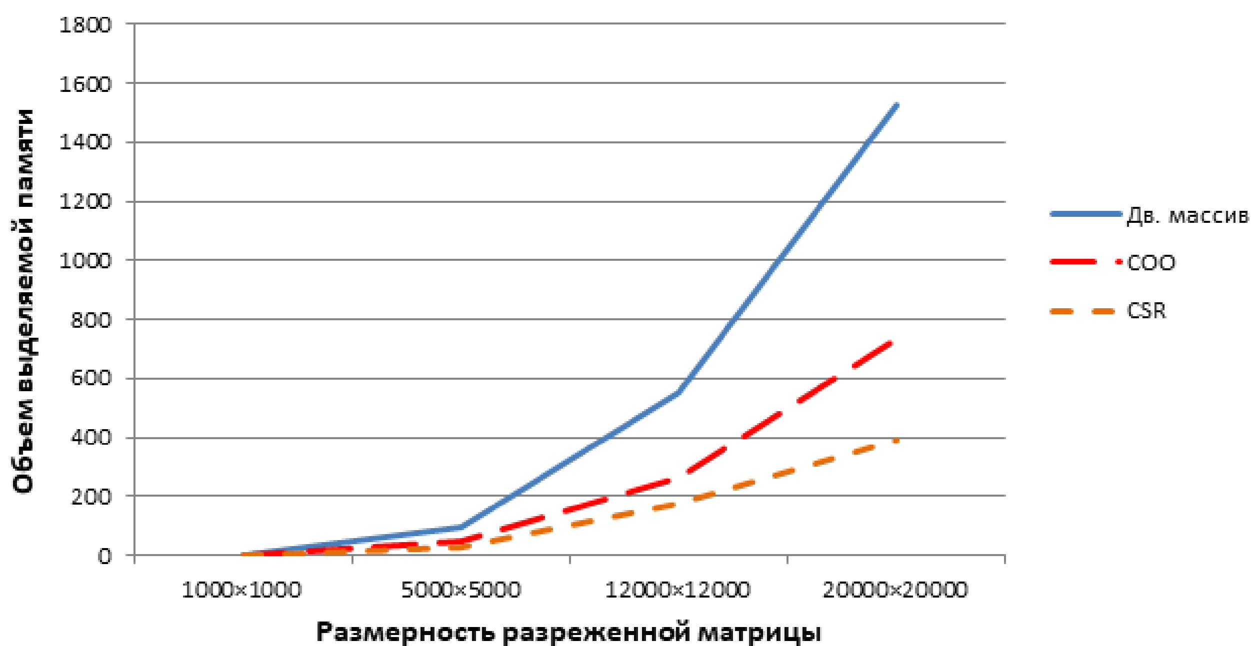

Рис. 2.5. График зависимости количества выделенной памяти от размерности матрицы и выбранного формата хранения

Исходя из полученных сравнительных данных табл. 2.1 и соответствующего ей графика, можно сделать вывод, что наименее избыточным форматом хранения разреженных матриц является разреженный строчный формат, в общем, что и следовало ожидать.

Использование CSR формата, является основной модификацией реализуемого в данной работе метода. Существенная оптимизация производительности, обусловленная применением CSR формата в качестве основного формата хранения результатов промежуточных вычислений метода, как будет показано в главе 3, в значительной степени снижает время выполнения разработанного приложения.

3. ПРОГРАММНАЯ РЕАЛИЗАЦИЯ СТАБИЛИЗИРОВАННОГО МЕТОДА БИСОПРЯЖЕННЫХ ГРАДИЕНТОВ С ИСПОЛЬЗОВАНИЕМ ТЕХНОЛОГИИ CUDA

3.1 Программная реализация последовательной версии алгоритма

На данный момент, существует большое количество математических программных библиотек написанных под широко известные и популярные языки программирования(Java,C++,Python и др.), которые в общем случае, предоставляют разработчику набор разнообразных функций инкапсулирующих в себе различные трудоемкие и в тоже время рутинные математические операции[10]. К таким операциям, в частности, можно отнести транспонирование, умножение, сложение, вычитания матриц и векторов. В случае самостоятельной реализации данных функций разработчиком, есть высокая вероятность сделать массу программных ошибок и недоработок, и как следствие, потерять не малое количество времени на их устранение, и в целом на разработку типичного функционала.

Программная реализация последовательной версии стабилизированного метода бисопряженных градиентов, подразумевает использование в качестве основного формата хранения промежуточных вычислений – CSR формат. В таком случае, как было отмечено ранее, самостоятельная программная реализация API позволяющая использовать данный формат, не имеет смысла. Большинство современных математических программных библиотек, ориентированных преимущественно на работу с матрицами и векторами, уже имеет встроенные инструменты предоставляющие разработчику простые API позволяющие работать с матрицами в CSR формате.

Одной из таких библиотек является библиотека Eigen ориентированная на язык С++. К основным преимуществам данной библиотеки относится:

- Скорость выполнения вычислительных операций;
- Поддержка основных операций над матрицами и векторам;
- Способность работать с матрицами, представленными в различных форматах (CSR,COO);
- Поддержка большинства современных C++ компиляторов (GNU C++, Intel C++ compiler MS Visual Studio);

Разработанная последовательная версия программы, использует в качестве основной математической библиотеки, библиотеку Eigen. Рассмотрим более подробно фрагменты кода последовательной версии разработанного приложения.

Данными поступающими на вход программы являются, основная матрица системы *A*, и правый вектор-столбец свободных членов *b.* В общем, входные данные формируют основную систему линейных алгебраических уравнений. Задача разработанной последовательной версии программы, заключается в решении данной СЛАУ стабилизированным методом биспоряженных градиентов, то есть нахождения вектора (1.6), и дальнейшая запись значений вектора в файл.

Входные данные считываются с файла формата COO. Посредством встроенной в C++ стандартной библиотеки файлового потокового ввода вывода *fstream,* осуществляется построчное считывание данных с файла в стандартную структуру данных типа вектор. Фрагмент кода выполняемой операции, приведен в листинге 3.1

Листинг 3.1. Построчное считывание данных с файла

```
std::vector<string> lines;
ifstream file;
file.open(filename.data());
if (file.is_open()) {
    string line;
    while(getline(file,line)) lines.push_back(line);
}
```

Данная функция принимает входной аргумент определяющий путь до конкретного файла матрицы. В качестве возвращаемого данной функции значения выступает структура данных типа вектор. Общий вид файла считываемой функцией *getAllLines()* определен на рис. 3.1.

```
    10   10   70
1       1     0.71684
2       1     2.33284
5       1     -1.74223
6       1     5.18232
8       1     2.12334
9       1     6.29384
```

Рис. 3.1. Файл матрицы в COO формате

Путь до файлов вектора *b* и матрицы *A*, как и значение переменной *epsilon*, характеризующей точность итерационного процесса, задается пользователем посредством передачи текстовых параметров командной строки, при запуске непосредственно самого приложения. Считывание переданных параметров осуществляется за счет функции представленной в листинге 3.2.

Листинг 3.2. Считывание параметров командной строки

```
void initArgs(int argc, char* argv[],
double &epsilon, string &matrixFile, string &vectorFile){
   std::map<string,string> params;
   for (int i = 0; i < argc; i++){
      if (i != argc - 1)
         if (params.count(argv[i]) == 1 & argv[i] != "")
            params[argv[i]] = argv[i + 1];
   }
   if (params["-e"].length() > 0) epsilon = atof(params["-e"].data());
   if (params["-m"].length() > 0) matrixFile = params["-m"];
   if (params["-v"].length() > 0) vectorFile = params["-v"];
}
```

В данном фрагменте, все значения переданных параметров сохраняются в соответствующих переменных, доступ к которым, в данной функции осуществляется по ссылке.

Пример Фрагмент кода создания матрицы в разреженном строчном формате, посредством библиотеки Eigen, представлен в приложении.

В данном случае создается матрица размерности 3×3, в которой определено 7 ненулевых элементов (см. рис. 3.1). Аналогичным образом происходит генерация вектора целиком состоящего из единичных числовых значений (см. листинг 3.3).

Листинг 3.3. Генерация единичного вектора

```
SparseMatrix<double,RowMajor> getOnesVector(int size){
   SparseVector<double,RowMajor> vector(1,size);
   for (int i = 0 ; i < size; i++)
      vector.coeffRef(0,i) = 1;
   return vector;}
```

Перейдем к рассмотрению итерационного алгоритма. Основной фрагмент кода итерационного процесса представлен в листинге 3.4.

Листинг 3.4. Реализация итерационного процесса BiCGStab

```
while (!isVectorsDifferenceLessThanEpsilon(x,xTemp,epsilon)){
    xTemp = x;
    p = getScalar(rTilda,rTemp);
    double beta = (p/pTemp) * (alpha / wTemp);
    SparseMatrix<double,RowMajor> tempMatrix = realPTemp - (uTemp
* wTemp);
    realP = rTemp + (tempMatrix * beta);
    u = realP * A;
    alpha = p / SparseMatrixUtil::getScalar(rTilda,u);
    SparseMatrix<double,RowMajor> s = rTemp - (u * alpha);
    SparseMatrix<double,RowMajor> t = s * A;
    w = getScalar(t,s) / getScalar(t,t);
    x = xTemp + ((s * w) + (realP * alpha)); r = s - (t * w);
    rTemp = r; alphaTemp = alpha; wTemp = w;  pTemp = p; uTemp = u;
    realPTemp = realP;
}
```

Функция *isVectorsDifferenceLessThanEpsilon()* (см. приложение) определяет критерий останова итерационного процесса (2.2). Функция *getScalar()* (см. приложение) возвращает значение вычисленного скалярного произведения.

Большая часть функций, имеющих отношение к считыванию строк с файла, распознаванию аргументов командной строки, перевода матрицы из COO в CSR формат, определены в отдельных классах. Список классов и соответствующее их назначение приведен в табл. 3.1.

Таблица 3.1

Основные классы программной реализации

Файл класса	Назначение
ArgsResolver.cpp	Определение переданных пользователем аргументов командной строки
MatrixFileUtil.cpp	Считывание данных с файла формата COO
SparseMatrixUtil.cpp	Генерирование вспомогательных векторов в CSR формате. Предоставление основных вспомогательных математических функций, которые оперируют векторами и матрицами в формате CSR
TimeUtil.cpp	Определение времени выполнения программы

Таким образом, основные функции последовательной версии разработанного приложения, логическим образом абстрагированы от основного кода итерационного процесса, что в целом позволяет динамически менять реализацию в случае внесения различных изменений.

3.2 Программная реализация параллельной версии алгоритма

Параллельная версия разработанного приложения реализована на базе библиотеки CUDA NVIDIA. CUDA представляет собой набор программных библиотек реализованных под различные языки программирования (C++, C, C#), позволяющих существенно повысить производительность вычислительных операций за счет использования графических процессоров (GPU – Graphic Processing Unit)[18].

Как было отмечено ранее, в качестве основного формата хранения разреженных матриц, был выбран CSR формат. На сегодняшний день инструментальные средства библиотеки NVIDIA CUDA позволяют эффективно реализовывать стандартные вычислительные операции над матрицами и векторам, которые в свою очередь, представлены в памяти вычислительного устройства в виде разреженного строчного формата.

Очевидным является то, что самостоятельная реализация многочисленных вычислительных математических операций над матрицами и векторами в CSR формате, с использованием GPU, является абсолютно не рациональной тратой времени и ресурсов. Задача параллелизации итерационного процесса стабилизированного метода бисопряженных градиентов, сводится к выбору библиотек от NVIDIA CUDA, позволяющих посредством реализации стандартных параллельных вычислительных операций над матрицами и векторами, ускорить промежуточные вычисления основного итерационного процесса.

Среди популярных программных библиотек, предоставляемых разработчиками NVIDIA CUDA, стоит выделить те, которые активно использовались при разработке параллельной версии программы. Используемые библиотеки, а также краткое описание их функциональности представлены в табл. 3.2.

Таблица 3.2

Используемые библиотеки NVIDIA CUDA

Название	Описание
cuBLAS	Реализация стандартных функций линейной алгебры
CUDA Math Library	Реализация стандартных математических функций
cuSPARSE	Реализация выч. операций над разреженными матрицами

Текущие версии перечисленных библиотек, реализованы на множестве языков программирования, таких как, C, C++, Python, Java. Стоит отметить, что большинство библиотек используются в известных математических пакетах, таких как MATLAB и Maple. В качестве языка программирования для реализации параллельной версии приложения был выбран язык C++.

Рассмотрим более детально основные фрагменты кода параллельной версии программы.

Как и в последовательной реализации, основным данными поступающими на вход параллельной версии программы являются, основная матрица системы *A*, и правый вектор-столбец свободных членов *b.* Матрица *A* и вектор *b,* определены в файлах формата COO, расположение которых передается программе через аргументы командной строки. Также, в качестве дополнительных аргументов передаются параметры: максимальное количество итераций – *MAX_IT,* и точность вычислений – *accuracy.* Фрагмент кода, в котором реализуется распознавание, и считывание перечисленных аргументов определен в приложении.

Перейдем к рассмотрению реализации итерационного процесса стабилизированного метода бисопряженных градиентов. Функция реализующая основной итерационный процесс,и получаемая основные данные характеризующие потенциальную систему линейных уравнений, реализована в классе *SystemEquationsResolver.* Фрагмент кода, определяющий непосредственное определение функции в заголовочном файле *SystemEquationsResolver.h* предоставлен в листинге 3.5.

Листинг 3.5. Определение основной функции итерационного процесса

```
static int bicgstabCUDA(
double *a, double *b, int *ia, int *ja, double *x, int size_a, int size_b, double accuracy, int MAX_IT, bool index_base_zero_or_one);
```

Основные параметры данной функции перечислены в табл. 3.3.

Таблица 3.3

Описание основных параметров функции bicgstabCUDA

Аргумент	Описание
a	Массив ненулевых элементов исходной матрицы
b	Правый вектор столбец
ia	Номера первых ненулевых элементов строк
ja	Номера столбцов ненулевых элементов
x	Вектор решения СЛАУ
size_a	Количество ненулевых элементов матрицы системы
size_b	Размерность вектора-столбца
accuracy	Точность вычислений
MAX_IT	Максимальное количество итераций
index_base_zero_or_one	Нумерация индексов

Основная часть код функции *bicgstabCUDA,* определяет передачу данных и соответствующие резервирование памяти из CPU в GPU, и наоборот. Фрагмент кода определяющие выделение GPU памяти для матрицы системы *A,*представлен в листинге 3.6.

Листинг 3.6. Выделение GPU памяти для основной матрицы системы

```
cudaStat = cudaMalloc ((void**)&devA, size_a*sizeof(double));
    if (cudaStat != cudaSuccess) {
        dataPointersHolder.CLEANUP("Device malloc failed\n");
   return -1;
}
```

В случае возникновения ошибки передачи данных, посредством функции *CLEANUP*, определенной в структуре *DataPointersHolder*(см. приложение)*,* происходит очищение уже выделенной под переданные параметры CPU и GPU памяти. Аналогичным образом происходит передача и освобождение данных передаваемых из GPU в CPU.

Рассмотрим подробнее фрагменты кода, реализующие промежуточные вычислительные операции посредством библиотеки cuBLAS. Фрагмент кода реализующего вычисление Евклидовой нормы вектора, посредством функции *cublasDnrm2,*определенной в библиотеке cuBLAS, представлен в листинге 3.7.

Листинг 3.7. Реализация вычисления Евклидовой нормы вектора посредством NVIDIA CUDA

```
cublasDnrm2 (cublasHandle, size_b, r, 1,&bnrm2);
```

В данном фрагменте вычисляется норма вектора *r*. Основной и часто используемой вычислительной операции производимой над векторами, является произведение векторов. Данная операция реализуется программно, и соответствующий фрагмент кода представлен в листинге 3.8.

Листинг 3.8. Реализация произведения векторов посредством NVIDIA CUDA.

```
cublasDdot (cublasHandle, size_b, r_tld, 1, r, 1, &rho);
```

В данном фрагменте вычисляется значение переменной *rho.* Основной фрагмент кода итерационного процесса определен в приложении.

Таким образом, за счет использования основных оптимизированных и ориентированных под архитектуру GPU функций библиотек cuBLAS, cuSPARSE и CUDA Math Library, значительно оптимизируется и ускоряется вычисление ведущих операций итерационного процесса, что в целом, сокращает время вычисления решения заданной системы линейных уравнений.

3.3 Оценка эффективности модификации итерационного алгоритма на основе проведения вычислительных экспериментов

Основным способом оценки эффективности, как параллельной, так и последовательной версии программной реализации, является способ измерения времени работы программы при получении на вход тестируемой программы, данных различного размера.

Данные, предназначенные для тестирования разработанной программной реализации, являются системы линейных алгебраических уравнений, в которых варьируется размерность основной матрицы системы (1.3), а также вектор-столбец свободных членов (1.5).

Вычислительный эксперимент, проводимый над разработанной параллельной и последовательной версией программы, выполнялся на персональном компьютере, на базе операционной системы Windows 10. Основными вычислительными устройствами компьютера являются, графическая видеокарта компании NVIDIA, GeForce 840M и центральный процессор Intel Core i5-3210M. Используемая видеокарта является одной из первых видеокарт компании NVIDIA, основанных на архитектуре Maxwell. Подробные характеристики данной видеокарты предоставлены в табл. 3.4.

Таблица 3.4

Технические характеристики видеокарты NVIDIA GeForce 840M

Архитектура	Maxwell
Конвейеры	384 - унифицированный
Частота GPU	1024 МГц
Скорость памяти	2000 МГц
Ширина шины данных	64 бит
Тип памяти	DDR3
Объем памяти	4096 Мбайт

Подробные характеристики двухъядерного центрального процессора Intel Core i5-3210M предоставлены в табл. 3.5.

Таблица 3.5

Технические характеристики центрального процессора Intel Core i5-3210M

Количество ядер	2
Количество потоков	4
Тактовая частота	2.5 ГГц
Максимальная тактовая частота	3.1 МГц
Кэш 1-го уровня	128 Кб
Кэш 2-го уровня	512 Кб
Кэш 3-го уровня	3072 Кб
Поддержка 64-бит	Есть

Перейдем к анализу результатов вычислительного эксперимента. Все тестовые данные, используемые в качестве входных данных, для разработанной программной реализации, отличаются друг от друга, преимущественно размерностью основной матрицы системы.

В первую очередь рассмотрим результаты вычислительного эксперимента, проводимые над последовательной версией программы (см. табл. 3.6 и рис. 3.2).

Таблица 3.6

Результаты вычислительного эксперимента проводимого над последовательной версией приложения

Размерность матрицы	Время выполнения программы (сек.)
500×500	0.964
1500×1500	2.031
3000×3000	29.413
5000×5000	41.324

Рис. 3.2. – График зависимости времени выполнения последовательной версии программы, от размерности основной матрицы системы

Рассмотрим результаты вычислительного эксперимента проводимого над параллельной версией разработанной программы (см. табл. 3.7 и рис. 3.3).

Таблица 3.7

Результаты вычислительного эксперимента проводимого над параллельной версией приложения

Размерность матрицы	Время выполнения программы (сек.)
500×500	5.235
1500×1500	6.139
3000×3000	10.923
5000×5000	12.306

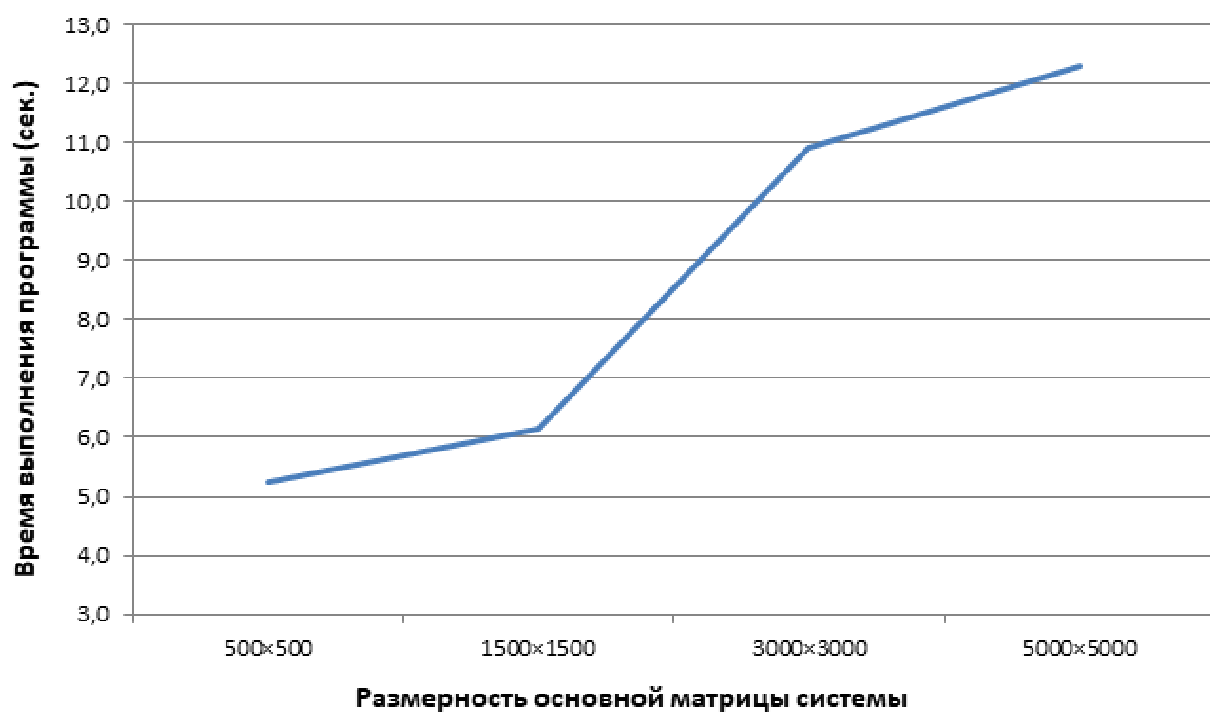

Рис. 3.3. График зависимости времени выполнения параллельной версии программы, от размерности основной матрицы системы

Общий график сравнения времени выполнения разработанной параллельной реализации с последовательной версией программы, представлен на рис. 3.4.

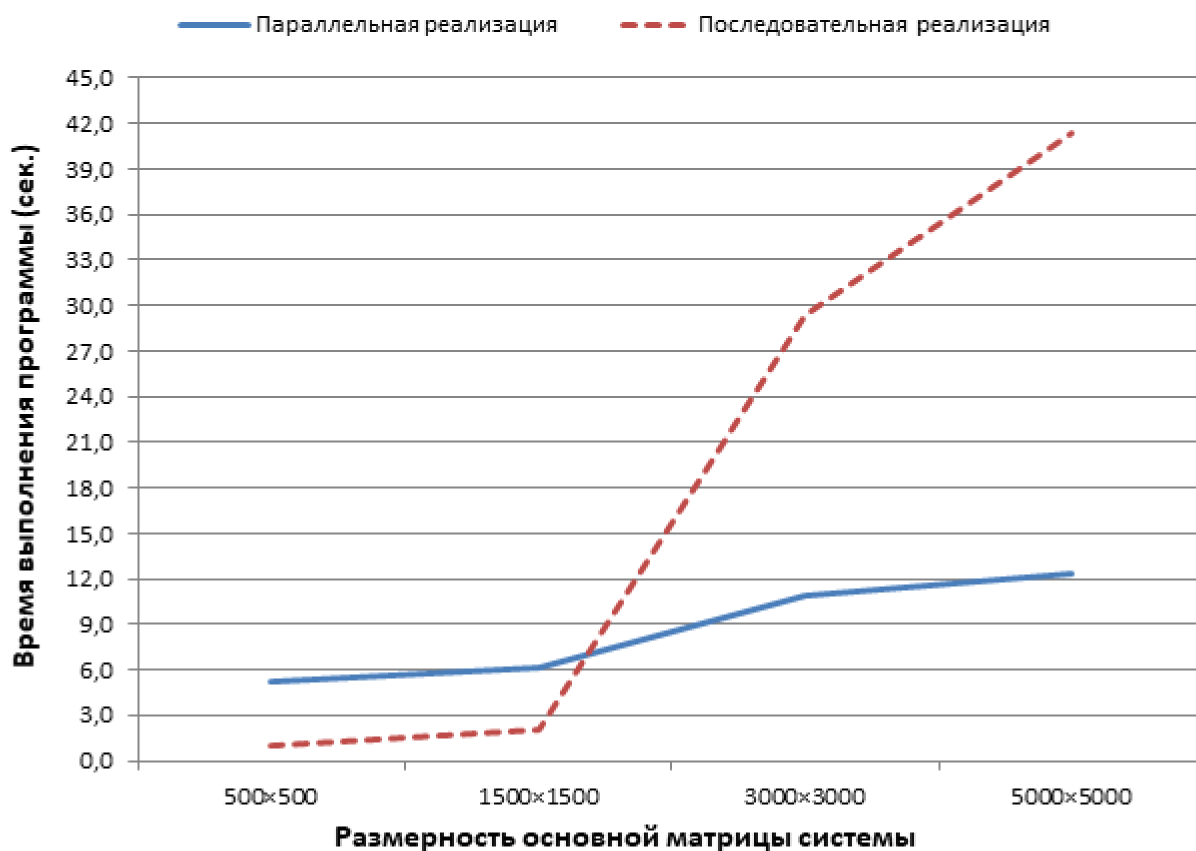

Рис. 3.4. График сравнения времени работы параллельной и последовательной версии разработанных приложений от размерности основной матрицы системы

В качестве основной количественной характеристики, определяющей качество оптимизации и производительность параллельной версии приложения, является ускорение (3.1).

$$S_{p(n)} = \frac{T_{(n)}}{T_{p(n)}} \tag{3.1}$$

Где, n – объем входных данных, $S_{p(n)}$ – ускорение, $T_{(n)}$ – время работы последовательной версии программы, $T_{p(n)}$ – время работы параллельной версии программы.

Стоит отметить, что далее, при расчетах ускорения разработанной параллельной версии программы, коэффициент n – размерность матрицы основной системы.

Результаты расчета ускорения на основе проведенных вычислительных экспериментов, а также соответствующий график, представлены в табл. 3.8 и на рис 3.5.

Таблица 3.8

Результаты расчета ускорения на основе проведенных вычислительных экспериментов

Размерность матрицы (n)	Ускорение ($S_{p(n)}$)
500×500	0.18
1500×1500	0.33
3000×3000	2.97
5000×5000	3.41

Также стоит отметить, что при использовании современных видеокарт, в частности видеокарт архитектуры Kepler, среднее ускорение, $S_{p(n)} \approx 18$.

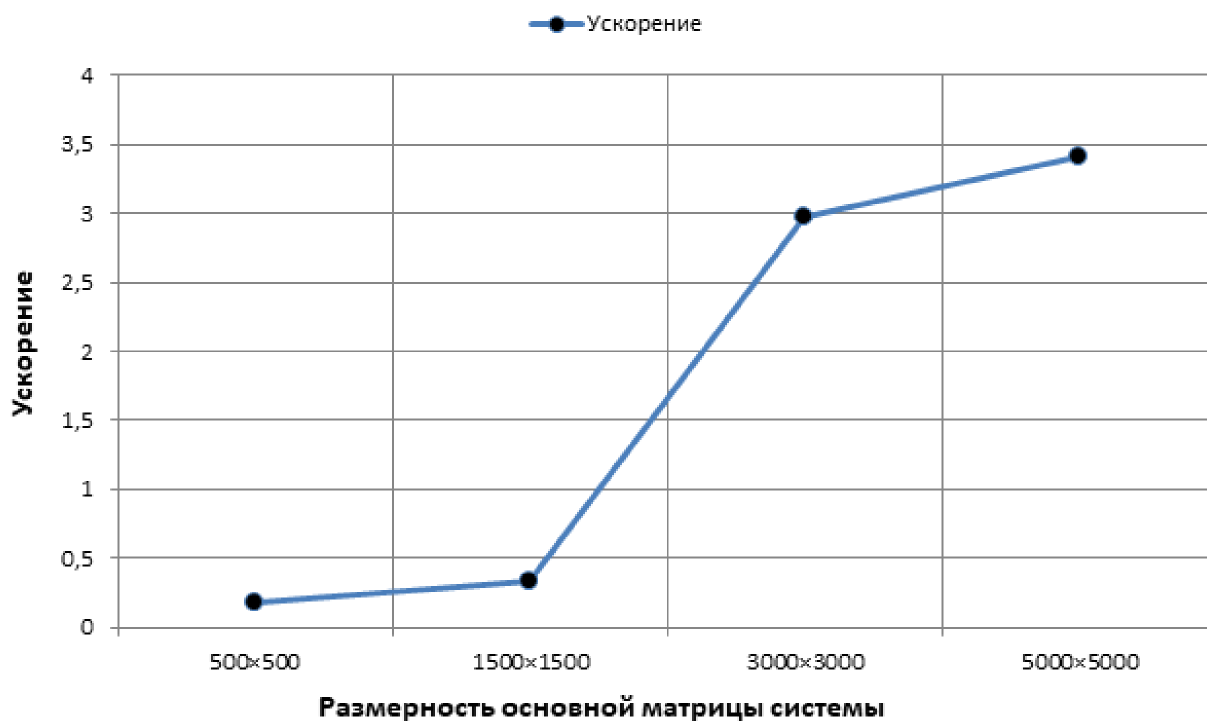

Рис. 3.5. График зависимости ускорения от размерности матрицы основной системы

Исходя из проведенных вычислительных экспериментов, можно сделать вывод, что параллельная версия разработанного приложения значительным образом оптимизирует основной вычислительный итерационный процесс стабилизированного алгоритма бисопряженных градиентов. Исходя из графика, предоставленного на рис. 3.4, можно точным образом определить, что время работы параллельной версии приложения значительно ниже, чем время работы последовательной программы с теми же данными, что в целом означает, что разработанное параллельное приложение, при очевидном совпадении полученных в ходе итерационного процесса результатов, можно использовать для решения действительных практических задач.

ЗАКЛЮЧЕНИЕ

Цель данной выпускной квалификационной работы заключалась в разработке и программной реализации итерационного алгоритма решения СЛАУ на основе модифицированного стабилизированного метода биспоряженных градиентов с использованием технологии NVIDIA CUDA.

В первой главе были проанализированы основные современные методы решения систем линейных алгебраических уравнений. Исходя из результатов полученного анализа, было выявлено, что большинство предлагаемых прямых и итерационных методов, являются не оптимизированными, с точки зрения расхода памяти.

Во второй главе был предложен итерационный алгоритм решения несимметричных систем линейных алгебраических уравнений, а также его модификации, позволяющие значительно сократить общий объем памяти, выделяемый под результаты промежуточных вычислений.

В третьей главе был подробно описан процесс реализации последовательной и параллельной версии программ, позволяющих решать системы линейных алгебраических уравнений посредством модифицированного и стабилизированного метода биспоряженных градиентов. Результаты проведенных вычислительных экспериментов, а в частности, полученное в ходе расчетов среднее ускорение равное 3, доказывают, что разработанная параллельная версия приложения значительно оптимизирует итерационный процесс разработанного метода.

Исходя из общих данных, полученных путем проведения многочисленных вычислительных экспериментов над разработанным приложением, можно сделать вывод, что параллельная версия реализованного приложения корректно выполняет задачу решения систем алгебраических уравнений, при этом минимально расходуя аппаратные ресурсы вычислительного устройства.

Стоит отметить, что использование GPU, в качестве основного инструмента повышения производительности разработанной программы, в целом, положительно скажется на скорости работы приложения, при использовании новых, актуальных и соответственно более мощных видеокарт. Стремительное совершенствование графических процессоров, предполагает значительное повышение оптимизации и скорости работы приложений разработанных, на базе технологии NVIDIA CUDA.

Таким образом, можно отметить, что разработанное программное приложение, исходя из корректности полученных результатов и очевидной скорости вычисления конечного решения с минимальными затратами памяти и в общем, рациональным расходом аппаратных ресурсов, можно использовать для решения различных прикладных задач.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Александреску А. Современное проектирование на C++ [Текст] / А. Александреску. – Москва.: Вильямс, 2015, – 336с.
2. Амосов, А.А. Вычислительные методы для инженеров [Текст] / А.А. Амосов, Ю.А. Дубинский, Н.В. Копченова. – Москва.: Высшая школа, 2014. – 672с.
3. Амосов, А.А. Вычислительные методы. Учебное пособие [Текст] / А.А. Амосов, Ю.А. Дубинский, Н.В. Копченова. – Санкт-Петербург.: Питер, 2014. – 672с.
4. Богачев, К.Ю. Основы параллельного программирования [Текст] / К.Ю. Богачев. – Москва.: БИНОМ. Лаборатория знаний, 2015 – 344с.
5. Боресков, А.В. Основы работы с технологией CUDA [Текст] / А.В. Боресков, А.А. Харламов. – Москва.: ДМК Пресс, 2010, – 232с.
6. Борзунов, С.В. Практикум по параллельному программированию [Текст] / С.В. Борзунов, С.Д. Кургалин, А.В. Флегель. – Санкт-Петербург.: Питер, 2017. – 236с.
7. Брюс Экель. Философия C++. Введение в стандартный C++ [Текст] / Брюс Экель. – Санкт-Петербург.: Питер, 2004. – 572с.
8. Бубнов, А.А. Вычислительная математика для программистов. Учебное пособие [Текст] / А.А. Бубнов, С.А. Бубнов, Е.Н. Проказникова. – Москва.: Курс, 2018, – 144с.
9. Бурова, И.Г. Алгоритмы параллельных вычислений и программирование. Курс лекций [Текст] / И.Г. Бурова, Ю.К. Демьянович. – Санкт-Петербург.: Издательство СПбГУ, 2007, – 208с.
10. Бьерн Страуструп. Программирование. Принципы и практика использования C++ [Текст] / Бьерн Страуструп. – Москва.: Вильямс, 2016. – 1328с.
11. Вабищевич, П.Н. Вычислительные технологии. Профессиональный уровень [Текст] / П.Н. Вабищевич. – Москва.: Ланад, 2017, – 352с.

12. Воеводин, В.В. Параллельные вычисления [Текст] / В.В. Воеводин, Вл.В. Воеводин. – Санкт-Петербург.: Питер, 2004. – 608с.
13. Гельфанд, И.М. Лекции по линейной алгебре [Текст] / И.М. Гельфанд. – Москва.: КДУ, 2009, – 320с.
14. Герб Саттер. Решение сложных задач на C++ [Текст] / Герб Саттер. – Москва.: Вильямс, 2017, – 400с.
15. Герб Саттер. Стандарты программирования на C++ [Текст] / Герб Саттер, А. Александреску. – Москва.: Вильямс, 2016, – 224с.
16. Гэри Маклин Холл. Адаптивный код: гибкое кодирования с помощью паттернов проектирования и принципов SOLID [Текст] / Гэри Маклин Холл. – Москва.: Вильямс, 2017, – 448с.
17. Демьянович, Ю.К. Теория распараллеливания и синхронизация [Текст] / Н.А. Демьянович, Т.О. Евдокимова. – Санкт-Петербург.: Издательство СПбГУ, 2005, – 112с.
18. Джейсон Сандрэс. Технология CUDA в примерах. Введение в программирование графических процессоров [Текст] / Джейсон Сандрэс, Эдвард Кэндрот. – Москва.: ДМК Пресс, 2011, – 232с.
19. Ездаков А.Л. Функциональное и логическое программирование [Текст] / А.Л. Ездаков. – Москва.: БИНОМ. Лаборатория знаний, 2009 – 120с.
20. Зализняк, В.Е. Численные методы. Основы научных вычислений. Учебник и практикум [Текст] / В.Е. Зализняк. – Москва.: Юрайт, 2017, – 356с.
21. Зимина, О.В. Линейная алгебра и аналитическая геометрия [Текст] / О.В. Зимина. – Ростов-на-Дону.: Феникс, 2015, – 384с.
22. Ильин, В.А. Линейная алгебра [Текст] / В.А. Ильин, Э.Г. Позняк. – Москва.: ФИЗМАТЛИТ, 2014, – 280с.
23. Калиткин, Н.Н. Численные методы [Текст] / Н.Н. Калиткин. – Санкт-Петербург.: БХВ-Петербург, 2011. – 592с.

24. Камерон Хьюз. Параллельное и распределенное программирование с использованием C++ [Текст] / Камерон Хьюз, Трейси Хьюз. – Москва.: Вильямс, 2004, – 672с.
25. Кейт Грегори. C++ AMP. Построение массивно параллельных программ с помощью Microsoft Visual C++ [Текст] / Кейт Грегори, Эйд Миллер. – Москва.: ДМК Пресс, 2012, – 412с.
26. Келвин Лин. Принципы параллельного программирования. Учебное пособие [Текст] / Келвин Лин, Лоуренс Снайдер. – Москва.: Издательство МГУ, 2013, – 388с.
27. Корнюшин, П.Н.. Численные методы [Текст] / П.Н. Корнюшин. – Владивосток.: Издательство дальневосточного универстета, 2002, – 104с.
28. Лизунова, Н.А. Матрицы и системы линейных уравнений [Текст] / Н.А. Лизунова, Шкроба С.П. – Москва.: ФИЗМАТЛИТ, 2007, – 352с.
29. Панюкова, Т.А. Численные методы [Текст] / Т.А. Панюкова. – Москва.: Либроком, 2018, – 224с.
30. Перепёлкин, Е.Е. Вычисления на графических процессорах (GPU) в задачах математической и теоритической физики [Текст] / Е.Е. Перепёлкин, Б.И. Садовников, Н.Г. Иноземцева. – Москва.: URSS, 2014, – 176с.
31. Расс Миллер. Последовательные и параллельные алгоритмы [Текст] / Расс Миллер, Лоренс Боксер. – Москва.: БИНОМ. Лаборатория знаний, 2015 – 408с.
32. Робер Седжвик. Алгоритмы на C++ [Текст] / Робер Седжвик. – Москва.: Вильямс, 2017, – 1056с.
33. Роберт К. Мартин. Чистый код. Создание анализ и рефакторинг. Библиотека программиста [Текст] / Роберт К. Мартин. – Санкт-Петербург.: Питер, 2018, – 464с.
34. Род Стивенс. Алгоритмы. Теория и практическое применение [Текст] / Род Стивенс. – Москва.: Эксмо, 2017, – 544с.

35. Самарский, А.А. Введение в численные методы [Текст] / А.А. Самарский. – Санкт-Петербург.: Лань, 2009, – 288с.
36. Скотт Майерс. Эффективный и современный С++. 42 рекомендации по использованию С++11 и С++14 [Текст] / Скотт Майерс. – Москва.: Вильямс, 2017. – 304с.
37. Скотт Миллет. Предметно-ориентированное проектирование. Паттерны, принципы и методы [Текст] / Скотт Миллет, Ник Тьюн. – Санкт-Петербург.: Питер, 2017, – 832с.
38. Фадеев, Д.К. Вычислительные методы линейной алгебры. Учебник [Текст] / Д.К. Фадеев, В.Н. Фадеева. – Санкт-Петербург.: Лань, 2009, – 736с.
39. Энтони Уильямс. Параллельное программирование на С++ в действии. Практика разработки многопоточных программ [Текст] / Энтони Уильямс. – Москва.: ДМК Пресс, 2016, – 672с.
40. Якобовский, М.В. Введение в параллельные методы решения задач. Учебное пособие [Текст] / М.В. Якобовский. – Москва.: Издательство МГУ, 2013, – 328с.

ПРИЛОЖЕНИЕ

bicgstab.h

```
#include <stdlib.h>
#include <string>
#include <vector>
#include <fstream>
#include <map>
#include <sstream>
#include <iostream>

#pragma once

using namespace std;

class SystemEquationsResolver{
	public:
		static int bicgstabCUDA(double *a, double *b, int *ia, int *ja, double *x, int size_a, int size_b, double accuracy, int MAX_IT,
						bool index_base_zero_or_one);
};

class SparseFileUtil{
   public:
      static void getMatrix(string filename, int size_a, int countRows, double *a,  int *ja, int*ia);
      static void getVector(string filename, double *b);
      static void getMatrixFileInfo(string filename, int *countNonZeroElements, int *countRows);
   private:
      static std::vector<string> getAllLines(string filename);
      static std::vector<double> split(const std::string &s, char delimiter);
      static void initMatrixInfo(string info, int *countNonZeroElements, int *countRows);
      static std::string getLineByNumber(string filename, int number);
};
```

```
class ArgsResolver{
   public:
       static void initArgs(int argc, char* argv[],
                       double &accuracy, int &maxIteration ,string &matrixFile, string
&vectorFile);
};
```

ArgsResolver.cpp

```
#include "bicgstab.h"

void ArgsResolver::initArgs(int argc, char* argv[], double &accuracy, int &maxIteration ,string
&matrixFile, string &vectorFile){
   std::map<string,string> params;

   params["-a"] = "";
   params["-ic"] = "";
   params["-m"] = "";
   params["-v"] = "";

   for (int i = 0; i < argc; i++){
       if (i != argc - 1)
           if (params.count(argv[i]) == 1 & argv[i] != "")
               params[argv[i]] = argv[i + 1];
   }

   if (params["-a"].length() > 0)
       accuracy = atof(params["-a"].data());

   if (params["-ic"].length() > 0)
       maxIteration = atof(params["-ic"].data());

   if (params["-m"].length() > 0)
       matrixFile = params["-m"];
```

```
    if (params["-v"].length() > 0)
        vectorFile = params["-v"];
}
```

SparseMatrixUtil.cpp

```
#include "SparseUtil.h"

SparseMatrix<double,RowMajor> SparseMatrixUtil::getDefaultMatrix(){
    SparseMatrix<double,RowMajor> spM(3,3);
    spM.coeffRef(0,0) = -2;
    spM.coeffRef(0,1) = 3;
    spM.coeffRef(0,2) = 1;

    spM.coeffRef(1,0) = 1;
    spM.coeffRef(1,1) = -5;
    spM.coeffRef(1,2) = 0;

    spM.coeffRef(2,0) = 0;
    spM.coeffRef(2,1) = 2;
    spM.coeffRef(2,2) = 5;
    spM.makeCompressed();
    return spM;
}

SparseMatrix<double,RowMajor> SparseMatrixUtil::getDefaultVector(){
    SparseVector<double,RowMajor> vector(1,3);
    vector.coeffRef(0,0) = 1;
    vector.coeffRef(0,1) = 10;
    vector.coeffRef(0,2) = -6;
    return vector;
}

SparseMatrix<double,RowMajor> SparseMatrixUtil::getOnesVector(int size){
```

```
    SparseVector<double,RowMajor> vector(1,size);
    for (int i = 0 ; i < size; i++)
        vector.coeffRef(0,i) = 1;
    return vector;
}

SparseMatrix<double,RowMajor> SparseMatrixUtil::getZeroVector(int size){
    SparseVector<double,RowMajor> vector(1,size);
    for (int i = 0 ; i < size; i++)
        vector.coeffRef(0,i) = 0;
    return vector;
}

bool SparseMatrixUtil::isVectorsDifferenceLessThanEpsilon(SparseMatrix<double,RowMajor>
x, SparseMatrix<double,RowMajor> tempX, double e){
    bool isAllValuesLessThanEpsilon = true;
    for (int i = 0; i < x.cols(); i++){
        if (std::abs(x.coeffRef(0,i) - tempX.coeffRef(0,i)) > e){
            isAllValuesLessThanEpsilon = false;
        }
    }
    return isAllValuesLessThanEpsilon;
}

double SparseMatrixUtil::getScalar(SparseMatrix<double,RowMajor>
a,SparseMatrix<double,RowMajor> b){
    SparseVector<double,RowMajor> scalar = a * b.transpose();
    return scalar.coeff(0,0);
}
```

SparseFileUtil.cpp

```
#include "bicgstab.h"
```

```
void SparseFileUtil::getMatrixFileInfo(string filename, int *countNonZeroElements, int *countRows){
    string matrixInfo = getLineByNumber(filename,2);
    initMatrixInfo(matrixInfo, countNonZeroElements, countRows);
}

void SparseFileUtil::initMatrixInfo(string info, int *countNonZeroElements, int *countRows){
    std::vector<double> partsOfInfo = split(info,' ');
    *countRows = (int)partsOfInfo.at(0);
    *countNonZeroElements = (int)partsOfInfo.at(2);
}

std::string SparseFileUtil::getLineByNumber(string filename, int number){
    ifstream infile(filename.data());

    string resultLine;
    if (infile.good()) {
        for (int i = 0; i < number; i++)
            getline(infile, resultLine);
    }

    infile.close();

    return resultLine;
}

void SparseFileUtil::getMatrix(string filename, int size_a, int countRows, double *a,  int *ja, int*ia){
    std::vector<string> fileLines = getAllLines(filename);

    ia[0] = 0;
    int iaCounter = 0;
    int previousIaValue = 0;
    for (int i = 2; i < fileLines.size(); i++){
        std::vector<double> currentLineInfo = split(fileLines.at(i),' ');
```

```
        int currentCol = ((int) currentLineInfo.at(0)) - 1;
        int currentRow = ((int) currentLineInfo.at(1)) - 1;
        double currentValue = currentLineInfo.at(2);

        a[i-2] = currentValue;
        ja[i-2] = currentCol;
        if (currentRow != previousIaValue){
            previousIaValue = currentRow;
            ia[previousIaValue] = iaCounter;
        }
        ++iaCounter;
    }
    ia[countRows - 1] = iaCounter;
}

void SparseFileUtil::getVector(string filename, double *b){
        std::vector<string> fileLines = getAllLines(filename);
        for (int i = 2; i < fileLines.size(); i++)
                b[i-2] = atof(fileLines.at(i).c_str());


}

std::vector<double> SparseFileUtil::split(const string &s, char delimiter){
    stringstream ss(s);
    std::string item;
    std::vector<double> elems;
    while (std::getline(ss, item, delimiter)) {
        if (item.size() > 0) elems.push_back(atof(item.c_str()));
    }
    return elems;
}

std::vector<string> SparseFileUtil::getAllLines(string filename){
    std::vector<string> lines;
    ifstream file;
```

```
    file.open(filename.data());
    if (file.is_open()) {
        string line;
        while(getline(file,line)) lines.push_back(line);
    }
    file.close();
    return lines;
}
```

TimeUtil.cpp

```
#include "SparseUtil.h"

struct timespec tstart={0,0}, tend={0,0};

void TimeUtil::start(){
    clock_gettime(CLOCK_MONOTONIC, &tstart);
}

double TimeUtil::end(){
    clock_gettime(CLOCK_MONOTONIC, &tend);
    return (((double)tend.tv_sec + 1.0e-9*tend.tv_nsec) -
            ((double)tstart.tv_sec + 1.0e-9*tstart.tv_nsec));
}
```

bicgstab.cpp

```
#include <cuda_runtime.h>
#include <cusparse_v2.h>
#include <cublas_v2.h>
#include <cusolver_common.h>
#include <cusolverSp.h>
#include <omp.h>
#include "bicgstab.h"
#include <stdio.h>
```

```
#include <string>

struct DataPointersHolder{

	double *a; double *b; int *ia; int *ja; double *x;

	double *devA, *r, *r_tld,*devX, *p, *s, *t, *v;
	int    *devIA, *devJA;

	void initGPUData(double *devA, double *r,double *r_tld,double *devX,double *p,double *s,double *t,double *v,
						int *devIA, int *devJA){
		this->devA = devA;
		this->r = r;
		this->r_tld = r_tld;
		this->devX = devX;
		this->p = p;
		this->s = s;
		this->t = t;
		this->v = v;
		this->devIA = devIA;
		this->devJA = devJA;
	}
	void initCPUData(double *a, double *b, int *ia, int *ja, double *x){
		this->a = a;
		this->b = b;
		this->ia = ia;
		this->ja = ja;
		this->x = x;
	}

	void CLEANUP(char *data){
		printf("%s",data);

		delete a,b,ia,ja,x;
```

```
			cudaFree(devA);
			cudaFree(r);
			cudaFree(r_tld);
			cudaFree(p);
			cudaFree(s);
			cudaFree(t);
			cudaFree(v);
			cudaFree(devX);
			cudaFree(devIA);
			cudaFree(devJA);
		}
};

int SystemEquationsResolver::bicgstabCUDA(double *a, double *b, int *ia, int *ja, double *x,
int size_a, int size_b, double accuracy, int MAX_IT,
			bool index_base_zero_or_one){

    double bnrm2, snrm2, error, alpha, beta, omega, rho, rho_1, resid=0;
	int size_ia = size_b + 1;

	cudaError_t cudaStat;

	cublasStatus_t cublasStatus;
    cublasHandle_t cublasHandle;

	cusparseStatus_t cusparseStatus;
    cusparseHandle_t cusparseHandle=0;
    cusparseMatDescr_t descra=0;

	cusolverSpHandle_t cusolverHandle;
    int reorder = 0;
    int singularity = 0;

	double *devA = 0, *r = 0, *r_tld = 0,*devX = 0, *p = 0, *s = 0, *t = 0, *v = 0;
```

```
	int    *devIA = 0, *devJA = 0;

	DataPointersHolder dataPointersHolder;
	dataPointersHolder.initCPUData(a,b,ia,ja,x);
	dataPointersHolder.initGPUData(devA,r,r_tld,devX,p,s,t,v,devIA,devJA);

	double time;
    time = omp_get_wtime();

	cusparseStatus= cusparseCreate(&cusparseHandle);
	if (cusparseStatus != CUSPARSE_STATUS_SUCCESS) {
		dataPointersHolder.CLEANUP("CUSPARSE Library initialization failed\n");
		return -4;
	}
    double tol = accuracy;

	cusolverSpCreate(&cusolverHandle);

	cusparseStatus = cusparseCreateMatDescr(&descra);
	if (cusparseStatus != CUSPARSE_STATUS_SUCCESS) {
		dataPointersHolder.CLEANUP("Matrix descriptor initialization failed");
		return -5;
	}



	cusparseSetMatType(descra, CUSPARSE_MATRIX_TYPE_GENERAL);

	if(index_base_zero_or_one){
		cusparseSetMatIndexBase(descra,CUSPARSE_INDEX_BASE_ONE);
	}
	else{
		cusparseSetMatIndexBase(descra,CUSPARSE_INDEX_BASE_ZERO);
	}
	cusolverSpDcsrlsvluHost(cusolverHandle, size_b, size_a, descra, a, ia, ja, b, tol, reorder,
x, &singularity);
```

```
   cusolverSpDestroy(cusolverHandle);

   cudaStat = cudaMalloc ((void**)&devA, size_a*sizeof(double));
   if (cudaStat != cudaSuccess) {
            dataPointersHolder.CLEANUP("Device malloc failed\n");
return -1;
   }

   cudaStat = cudaMalloc ((void**)&r, size_b*sizeof(double));
   if (cudaStat != cudaSuccess) {
            dataPointersHolder.CLEANUP("Device malloc failed\n");
return -1;
   }
   cudaMemset(r,0,size_b*sizeof(double));
   cudaStat = cudaMalloc ((void**)&r_tld, size_b*sizeof(double));
   if (cudaStat != cudaSuccess) {
            dataPointersHolder.CLEANUP("Device malloc failed\n");
return -1;
   }

   cudaStat = cudaMalloc ((void**)&s, size_b*sizeof(double));
   if (cudaStat != cudaSuccess) {
            dataPointersHolder.CLEANUP("Device malloc failed\n");
return -1;
   }

   cudaStat = cudaMalloc ((void**)&t, size_b*sizeof(double));
   if (cudaStat != cudaSuccess) {
            dataPointersHolder.CLEANUP("Device malloc failed\n");
return -1;
   }

   cudaStat = cudaMalloc ((void**)&p, size_b*sizeof(double));
   if (cudaStat != cudaSuccess) {
```

```
            dataPointersHolder.CLEANUP("Device malloc failed\n");
return -1;
   }

   cudaStat = cudaMalloc ((void**)&v, size_b*sizeof(double));
   if (cudaStat != cudaSuccess) {
            dataPointersHolder.CLEANUP("Device malloc failed\n");
return -1;
   }

   cudaStat = cudaMalloc ((void**)&devX, size_b*sizeof(double));
   if (cudaStat != cudaSuccess) {
            dataPointersHolder.CLEANUP("Device malloc failed\n");
return -1;
   }

   cudaStat = cudaMalloc ((void**)&devIA, size_ia*sizeof(int));
   if (cudaStat != cudaSuccess) {
            dataPointersHolder.CLEANUP("Device malloc failed\n");
return -1;
   }

   cudaStat = cudaMalloc ((void**)&devJA, size_a*sizeof(int));
   if (cudaStat != cudaSuccess) {
            dataPointersHolder.CLEANUP("Device malloc failed\n");
return -1;
   }

   cudaStat = cudaMemcpy(devA, a,
              size_a*sizeof(double),
              cudaMemcpyHostToDevice);
   if ((cudaStat != cudaSuccess)) {
            dataPointersHolder.CLEANUP("Memcpy from Host to Device failed\n");
            return -2;
   }
```

```
	cudaStat = cudaMemcpy(r, b,
			size_b*sizeof(double),
			cudaMemcpyHostToDevice);
	if ((cudaStat != cudaSuccess)) {
		dataPointersHolder.CLEANUP("Memcpy from Host to Device failed\n");
		return -2;
	}

	cudaStat = cudaMemcpy(devIA, ia,
			size_ia*sizeof(int),
			cudaMemcpyHostToDevice);
	if ((cudaStat != cudaSuccess)) {
		dataPointersHolder.CLEANUP("Memcpy from Host to Device failed\n");
		return -2;
	}

	cudaStat = cudaMemcpy(devJA, ja,
			size_a*sizeof(int),
			cudaMemcpyHostToDevice);
	if ((cudaStat != cudaSuccess)) {
		dataPointersHolder.CLEANUP("Memcpy from Host to Device failed\n");
		return -2;
	}

	cublasStatus = cublasCreate(&cublasHandle);
	if (cublasStatus != CUBLAS_STATUS_SUCCESS) {
		printf ("CUBLAS initialization failed\n");
		return -3;
	}

cudaStat = cudaMemcpy(devX, x, size_b*sizeof(double), cudaMemcpyHostToDevice);
	if ((cudaStat != cudaSuccess)) {
		dataPointersHolder.CLEANUP("Memcpy from Host to Device failed\n");
		return -2;
```

```
        }
    cudaMemset(r_tld,0,size_b*sizeof(double));
    cudaMemset(s,0,size_b*sizeof(double));
    cudaMemset(t,0,size_b*sizeof(double));

    cublasDnrm2 (cublasHandle, size_b, r, 1,&bnrm2);
    if ( bnrm2 == 0.0 ) bnrm2 = 1.0;

    double alpha1 = -1.0;
    double beta1 = 1.0;

    cusparseStatus =
cusparseDcsrmv_mp(cusparseHandle,CUSPARSE_OPERATION_NON_TRANSPOSE, size_b,
size_b, size_a,
            &alpha1, descra, devA, devIA, devJA, devX, &beta1, r);
    if (cusparseStatus != CUSPARSE_STATUS_SUCCESS) {
        printf("Error status: %d\n",cusparseStatus);
      dataPointersHolder.CLEANUP("Matrix-vector multiplication failed\n");
      return -6;
    }
    double temp,temp2,error_temp = DBL_MAX;
    omega = 1.0;
    cudaStat = cudaMemcpy(r_tld, r, size_b*sizeof(double),
                          cudaMemcpyDeviceToDevice);
    if ((cudaStat != cudaSuccess)) {
      dataPointersHolder.CLEANUP("Memcpy from r to r_tld failed\n");
      return -7;
    }
    double alpha2, beta2;
    int iter=0,flag=0;
    alpha = 1.0; rho = 1.0;
    cudaMemset(v, 0, size_b*sizeof(double));
    cudaMemset(p, 0, size_b*sizeof(double));
    for (iter = 0; iter < MAX_IT; ++iter)
      {
```

```
        cublasDdot (cublasHandle, size_b, r_tld, 1, r, 1, &rho);
    if ( rho == 0.0 ){break;}
        if ( iter > 0 ){
     beta = (rho/rho_1) * (alpha/omega);
     alpha2 = -omega;
     beta2 = 1.0;
          cublasDaxpy (cublasHandle,size_b, &alpha2, v, 1, p, 1);
                cublasDscal (cublasHandle,size_b, &beta, p, 1);
                cublasDaxpy (cublasHandle,size_b, &beta2, r, 1, p, 1);
      }
    else{
cudaStat = cudaMemcpy(p, r,size_b*sizeof(double),cudaMemcpyDeviceToDevice);
                if ((cudaStat != cudaSuccess)) {
                    dataPointersHolder.CLEANUP("Memcpy from r to p failed \n");
                    return -7;
                }
      }
        alpha1 = 1.0; beta1 = 0.0;
        cusparseStatus                                                    =
cusparseDcsrmv_mp(cusparseHandle,CUSPARSE_OPERATION_NON_TRANSPOSE,     size_b,
size_b, size_a,
                    &alpha1, descra, devA, devIA, devJA, p,  &beta1, v);
        if (cusparseStatus != CUSPARSE_STATUS_SUCCESS) {
                dataPointersHolder.CLEANUP("Matrix-vector multiplication failed\n");
                return -6;
    }
        cublasDdot (cublasHandle,size_b, r_tld, 1, v, 1,&temp);
        if ( temp == 0.0 ) {break;}
    alpha = rho / temp;
    cudaStat = cudaMemcpy(s, size_b*sizeof(double),cudaMemcpyDeviceToDevice);
    if ((cudaStat != cudaSuccess)) {
         dataPointersHolder.CLEANUP("Memcpy from r to s failed\n");
         return -7;
    }
        alpha2 = -alpha;
```

```
                cublasDaxpy (cublasHandle, size_b, &alpha2, v, 1, s, 1);
        cublasDnrm2( cublasHandle, size_b, s, 1,&snrm2);
                if ( snrm2 < accuracy ){
             cublasDaxpy (cublasHandle, size_b, &alpha, p, 1, devX, 1);
             resid = snrm2 / bnrm2;
             break;
            }
                alpha1 = 1.0; beta1 = 0.0;
                cusparseStatus                                                          =
cusparseDcsrmv_mp(cusparseHandle,CUSPARSE_OPERATION_NON_TRANSPOSE,size_b,
size_b, size_a,
                                     &alpha1,descra, devA, devIA, devJA, s, &beta1, t);
                temp = 0.0; temp2 = 0.0;
                cublasDdot (cublasHandle,size_b, t, 1, s, 1,&temp);
                cublasDdot (cublasHandle,size_b, t, 1, t, 1,&temp2);
                if(temp2 == 0.0){
                        break;
                }
                omega = temp/temp2;
                cublasDaxpy (cublasHandle,size_b, &alpha, p, 1, devX, 1);
        cublasDaxpy (cublasHandle,size_b, &omega, s, 1, devX, 1);
                alpha2 = -omega;
                cublasDaxpy (cublasHandle,size_b, &alpha2, t, 1, s, 1);
                cudaStat = cudaMemcpy(r, s, size_b*sizeof(double),
                         cudaMemcpyDeviceToDevice);
        if ((cudaStat != cudaSuccess)) {
             dataPointersHolder.CLEANUP("Memcpy from s to r failed\n");
             return -7;
        }
                cublasDnrm2( cublasHandle, size_b, r, 1,&temp);
                error = temp / bnrm2;
        if ( error <= accuracy) {break;}
        if (error > error_temp){ break;}
        if ( omega == 0.0 ) {break;}
        rho_1 = rho;
```

```
    }
        cudaStat = cudaMemcpy(x, devX, size_b*sizeof(double),cudaMemcpyDeviceToHost);
        if ((cudaStat != cudaSuccess)) {
                dataPointersHolder.CLEANUP("Memcpy from devX to x failed\n");
                return -8;
        }
        time = omp_get_wtime() - time;
        printf_s("GPU compute time: %g sec\n", time);
        cudaFree(devA);
        cudaFree(r);
        cudaFree(r_tld);
        cudaFree(p);
        cudaFree(s);
        cudaFree(t);
        cudaFree(v);
        cudaFree(devX);
        cudaFree(devIA);
        cudaFree(devJA);
        cusparseDestroyMatDescr(descra);
        cusparseStatus = cusparseDestroy(cusparseHandle);
        cusparseHandle = 0;
        if (cusparseStatus != CUSPARSE_STATUS_SUCCESS) {
                dataPointersHolder.CLEANUP("CUSPARSE Library release of resources failed\n");
                return -9;
        }
        cublasStatus = cublasDestroy(cublasHandle);
        cublasHandle = 0;
        if (cublasStatus != CUBLAS_STATUS_SUCCESS) {
                dataPointersHolder.CLEANUP("CUBLAS Library release of resources failed\n");
                return -10;
        }
  return 0;
}
```

Выпускная квалификационная работа выполнена мной совершенно самостоятельно. Все использованные в работе материалы и концепции из опубликованной научной литературы и других источников имеют ссылки на них.

«___» _________________ _____ г.

___________________________ ______________________

(подпись) *(Ф.И.О.)*