МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение
высшего образования
«Тольяттинский государственный университет»

Институт **математики, физики и информационных технологий**

Кафедра **«Прикладная математика и информатика»**

01.03.02 ПРИКЛАДНАЯ МАТЕМАТИКА И ИНФОРМАТИКА

СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ И КОМПЬЮТЕРНЫЕ ТЕХНОЛОГИИ

БАКАЛАВРСКАЯ РАБОТА

на тему: **Решение систем линейных алгебраических уравнений с использованием технологии CUDA**

Студент П.В. Макеев

Руководитель А.В. Очеповский

Допустить к защите
Заведующий кафедрой к.тех.н, доцент, А.В. Очеповский _____

«_____»_____2016 г.

Тольятти 2016

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение
высшего образования
«Тольяттинский государственный университет»
Институт **математики, физики и информационных технологий**

Кафедра **«Прикладная математика и информатика»**

УТВЕРЖДАЮ
Зав.кафедрой«Прикладная
математика и информатика»
_____А.В.Очеповский

«____»_____2016 г.

ЗАДАНИЕ
на выполнение бакалаврской работы

Студент Макеев Павел Владимирович

1. Тема Решение систем линейных алгебраических уравнений с использованием технологии CUDA
2. Срок сдачи студентом законченной выпускной квалификационной работы 24.06.2016
3. Исходные данные к выпускной квалификационной работе поддержка компьютером технологии CUDA
4. Содержание выпускной квалификационной работы (перечень подлежащих разработке вопросов, разделов)

Введение

1) Вычисления систем линейных алгебраических уравнений больших размерностей на электронно-вычислительной машине
2) Реализация решения систем линейных алгебраических уравнений
3) Анализ эффективности алгоритмов

Заключение

Литература

5. Ориентировочный перечень графического и иллюстративного материала презентация, включающая блок-схемы работы приложения, графики, диаграммы, экранные формы, демонстрирующие работоспособность программного продукта
6. Дата выдачи задания «11» января 2016 г.

Руководитель выпускной
квалификационной работы _____ А.В. Очеповский

Задание принял к исполнению _____ П.В. Макеев

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение
высшего образования
«Тольяттинский государственный университет»

Институт **математики, физики и информационных технологий**

Кафедра **«Прикладная математика и информатика»**

УТВЕРЖДАЮ
Зав.кафедрой«Прикладная
математика и информатика»
_____А.В.Очеповский

«____»_____2016 г.

КАЛЕНДАРНЫЙ ПЛАН
выполнения бакалаврской работы

Студента Макеева Павла Владимировича
по теме Решение систем линейных алгебраических уравнений с использованием технологии CUDA

Наименование раздела работы	Плановый срок выполнения раздела	Фактический срок выполнения раздела	Отметка о выполнении	Подпись руководителя
Изучение теоретических основ	20.01.2016	20.01.2016	выполнено	
Написание введения ВКР	01.02.2016	01.02.2016	выполнено	
Изучение математических основ	25.02.2016	25.02.2016	выполнено	
Написание 1 главы ВКР	23.03.2016	23.03.2016	выполнено	
Создание ориентированного графа структуры сайта ТГУ	1.04.2016	1.04.2016	выполнено	
Изменение структуры сайта	10.04.2016	10.04.2016	выполнено	
Написание 2 главы ВКР	20.04.2016	20.04.2016	выполнено	

Написание заключения ВКР	8.05.2016	8.05.2016	выполнено	
Подведение итогов, редактирование ВКР	9.05.2016	9.05.2016	выполнено	
Представление ВКР	11.05.2016	11.05.2016	выполнено	
Создание презентационного материала	11.05.2016	11.05.2016	выполнено	
Проверка на наличие заимствований (плагиата) в системе antiplagiat.ru	30.05.2016	30.05.2016	выполнено	
Предварительная защита	6.06.2016 - 11.06.2016	6.06.2016 - 11.06.2016	выполнено	
Сдача на кафедру отзыва научного руководителя и ознакомление с ним	20.06.2016	20.06.2016	выполнено	
Сдача на кафедру комплекта документов для защиты	24.06.2016	24.06.2016	выполнено	
Защита ВКР	29.06.2016	29.06.2016	выполнено	

Руководитель выпускной квалификационной работы _____ А.В. Очеповский

Задание принял к исполнению _____ П.В. Макеев

Аннотация

Актуальность работы состоит в том, что в наше время компьютерное моделирование стало неотъемлемой частью решения сложных инженерных задач. Компьютерное моделирование тесно связано с системами линейных алгебраических уравнений, являющиеся одним из основных объектов линейной алгебры. Применяясь во всех разделах аналитической геометрии, СЛАУ, помогает описывать важные геометрические плоскости и прямые.

Объект исследования: вычислительный процесс решения СЛАУ на аппаратно-программной платформе CUDA.

Предмет исследования: параллельные алгоритмы и программная реализация решения СЛАУ.

Цель: проанализировать эффективность разработанных алгоритмов решения СЛАУ с использованием программно-аппаратной архитектуры параллельных вычислений NVIDIA CUDA.

Для этого необходимо решить следующие задачи:

- проектирование параллельных алгоритмов решения СЛАУ;
- реализация алгоритмов на языке CUDA C/C++ для решения СЛАУ;
- тестирование алгоритмов решения;
- исследование эффективности разработанных алгоритмов решения СЛАУ.

Первая глава работы посвящена анализу СЛАУ и численных способов решения под ЭВМ с распараллеливанием. А также анализу технологии CUDA.

Вторая глава описывает особенности решения СЛАУ под технологию CUDA, реализацию алгоритмов решения и их тестирование.

Третья глава посвящена процессу анализа эффективности реализованных алгоритмов решения СЛАУ и их улучшению.

Выпускная квалификационная работа выполнена на сорока шести страницах, состоит из введения, трёх глав, заключения, списка литературы, состоящего из двадцати четырех литературных источников, двадцати восьми рисунков и одной таблицы.

Оглавление

Введение

Хотя задача решения систем линейных алгебраических уравнений (СЛАУ) сравнительно редко представляет самостоятельный интерес для приложений, от умения эффективно решать такое системы часто зависит сама возможность математического моделирования самых разнообразных процессов. Значительная часть численных методов решения различных, в особенности — нелинейных, задач включает в себя решение СЛАУ как элементарный шаг советующего алгоритма [11].

Несмотря на то, что общие методы решения СЛАУ широко известны с середины 17 века, развитие вычислительной техники и вызванный этим процессом переход к более сложным (трехмерным, в произвольных геометрических областях) моделям в виде систем дифференциальных уравнений в частных производных и их дискретным аналогам на неструктурированных сетках, привел к необходимости решения больших разреженных систем линейных алгебраических уравнений с матрицами нерегулярной структуры [5].

Суперкомпьютеры позволили начать решать новые задачи большого размера, приступить к реализации реалистических математических моделей сложных физических явлений и технических устройств, которые раньше нельзя было решать на маломощных последовательных компьютерах [6].

На протяжении десятков лет суперкомпьютеры добивались повышения производительности. Однако, помимо впечатляющего повышения быстродействия одного процессора, производители суперкомпьютеров увеличивали число процессоров. Самые быстрые современные суперкомпьютеры насчитывают десятки и сотни тысяч процессорных ядер, работающих согласованно [1].

Благодаря распараллеливанию алгоритмов под многоядерные ЦП увеличилась производительность вычислений. Тем не менее, повышать скорость вычислений на традиционных архитектурах становится все сложнее.

В настоящий момент, все более широко для высокопроизводительной

обработки данных применяются дополнительные ускорители, принимающие на себя часть вычислительной нагрузки, к которым относятся видео ускорители.

В отличие от предыдущих поколений GPU, в которых вычислительные ресурсы подразделялись на вершинные и пиксельные шейдеры, в архитектуру CUDA (Compute Unified Device Architecture) включен унифицированный шейдерный конвейер, позволяющий программе, выполняющей вычисления общего назначения, задействовать любые арифметически-логическое устройство (АЛУ), входящее в микросхему [1].

Все эти средства были добавлены в архитектуру CUDA с целью создать GPU, который отлично справлялся бы с вычислениями общего назначения, а не только традиционными задачами компьютерной графики.

Благодаря большому количеству ядер и других архитектурных особенностям использование GPU может значительно увеличить производительность выселений. Поэтому задача решения СЛАУ на программно-аппаратной архитектуре параллельных вычислений NVidia CUDA является **актуальной**.

Новизна работы заключается в решении СЛАУ на графическом процессоре с учетом архитектурных особенностей плат NVidia Tesla K10.

Объектом выпускной квалификационной работы является вычислительный процесс решения СЛАУ на аппаратно-программной платформе CUDA.

Предмет выпускной квалификационной работы: параллельные алгоритмы и программная реализация решения СЛАУ.

Цель выпускной квалификационной работы: проанализировать эффективность разработанных алгоритмов решения СЛАУ с использованием программно-аппаратной архитектуры параллельных вычислений NVidia CUDA.

Задачами выпускной квалификационной работы являются:

- проектирование параллельных алгоритмов решения СЛАУ;
- реализация параллельных алгоритмов на языке CUDA C/C++ для решения СЛАУ;

- тестирование алгоритмов решения;
- исследование эффективности разработанных алгоритмов решения СЛАУ.

Выпускная квалификационная работа состоит из трех глав.

Первая глава работы посвящена анализу СЛАУ и численных способов решения под ЭВМ с распараллеливанием. А также анализу суперкомпьютера ТГУ, технологии CUDA и архитектуры NVidia Kepler.

Вторая глава описывает особенности решения СЛАУ под технологию CUDA, реализацию алгоритмов решения и их тестирование.

Третья глава посвящена процессу анализа эффективности реализованных алгоритмов решения СЛАУ и их улучшению.

В заключении подводится общая оценка данных реализаций алгоритма, анализ возникших при разработке трудностей, а также перспективы применения алгоритма в различных ситуациях.

Глава 1 Вычисления систем линейных алгебраических уравнений больших размерностей на электронно-вычислительной машине

1.1 Применение систем линейных алгебраических уравнений

В настоящее время все чаще решение сложных инженерных задач не обходится без помощи компьютерного моделирования, что позволяет визуализировать задачу и представить её в естественной для пользователя, графической форме, а также автоматически переводить это описание на язык компьютера.

Деятельность инженера-исследователя все тесно связана с компьютерным моделированием. Математические исследования с визуальным представлением помогают инженерам в проектировании разных, по уровню сложности, изделий. Данная работа занимает промежуточную позицию между конструированием изделия и натурными испытаниями его прототипа.

Однако компьютерное моделирование больше не ограничивается проектированием технических изделий. Это связано как с усложнением объектов исследования, так и с ростом возможностей ЭВМ (электронно-вычислительная машина).

В различных областях человеческой деятельности, таких как, биология, медицина, социология и т.п., появляются все более сложные задачи, для решения которых приходит на помощь компьютерное моделирование.

Компьютерное моделирование тесно связано с системами линейных алгебраических уравнений, являющиеся одним из основных объектов линейной алгебры. Применяясь во всех разделах аналитической геометрии, СЛАУ, помогает описывать важные геометрические плоскости и прямые.

В работе рассмотрена задача проектирования и реализации алгоритмов решения диагональных СЛАУ, являющимися одними из самых востребованных в различных вычислительных задачах, таких как, задачи диффузии и электроразведки [8,9]. Количество диагоналей будут возрастать по мере увеличения размерности СЛАУ.

Развитие вычислительной техники и компьютерного моделирования

требует переход к все более сложным графическим моделям, привело к необходимости решения больших разреженных систем линейных алгебраических уравнений с матрицами нерегулярной структуры.

Но развитие вычислительной техники подразумевает и возрастание требований к скорости решения прежних и новых задач. Поэтому актуальность использования сверхмощных компьютеров, или проще суперкомпьютеров, получают все большую популярность. К сожалению, технологические возможности увеличения тактовой частоты, следовательно, и производительности, всего в 2 раза приводит к увеличению тепловыделения в 16 раз. До сих пор с этим удавалось справляться за счет уменьшения размеров отдельных элементов микросхем. Дальнейшая миниатюризация связана со значительными трудностями, поэтому в настоящее время рост производительности происходит в основном за счет увеличения числа параллельно работающих ядер, т.е. за счет параллелизма [2].

Скорость решения сильно зависит от тактовых частот центрального процессора(CPU) и памяти. Процессоры с высокой частотой и большим количеством интегрированной памяти обладают превосходной производительностью, но не являются массовыми из-за слишком высокой цены. Поэтому на смену CPU в решениях задач, обладающих параллелизмом по данным, все чаще стали использовать графические ускорители(GPU).

Термин GPU (Graphics Processing Unit) впервые был использован корпорацией NVIDIA для обозначения того факта, что графический ускоритель, который первоначально использовался только для ускорения трехмерной графики, стал мощным программируемым устройством, пригодным для решения более широкого класса вычислительных задач.

Первые графические ускорители лишь выполняли растеризацию (перевод треугольников в массивы пикселов) и наложения текстур. При этом обработка производилась центральным процессором, и ускоритель получал на вход уже отображенные на экран вершины. Однако именно эти очень простые задачи первые GPU умели решать намного быстрее, чем универсальный центральный

процессор, что привело к широкому распространению графических ускорителей трехмерной графики.

Современные GPU представляют собой массивно-параллельные вычислительные устройства с производительностью порядка TFLOPS (10^{12} FLOPS или 10^{12} операций в секунду), имеющие большой объем собственной высокоскоростной памяти и возможность обработки данных без участия CPU.

1.2 Общий вид систем линейных алгебраических уравнений

Численные методы линейной алгебры — это классический раздел вычислительной математики. Широкое применение вычислительного эксперимента в научной практике и развитие микропроцессорной архитектуры обуславливают постоянный интерес, как к разработке новых численных методов, так и к совершенствованию хорошо известных. Не будет преувеличением отметить задачу решения систем линейных алгебраических уравнений (СЛАУ) как наиболее массовую [1].

Нормальный вид системы линейных алгебраических уравнений:

$$\begin{pmatrix} a_1^1 x^1 + a_2^1 x^2 + \cdots + a_n^1 x^n = b^1 \\ a_1^2 x^1 + a_2^2 x^2 + \cdots + a_n^2 x^n = b^2 \\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\ a_1^m x^1 + a_2^m x^2 + \cdots + a_n^m x^n = b^m \end{pmatrix}, \quad (1.1)$$

где m – количество линейных уравнений;

n – количество переменных;

x – неизвестное число;

a – коэффициенты неизвестных числах;

b – свободные члены.

Коэффициенты при неизвестных уравнениях можно записать в виде матрицы

$$A = \begin{pmatrix} a_1^1 & a_2^1 & \cdots & a_n^1 \\ a_1^2 & a_2^2 & \cdots & a_n^1 \\ \cdots & \cdots & \cdots & \cdots \\ a_1^m & a_2^m & \cdots & a_n^m \end{pmatrix}, \tag{1.2}$$

которые называются матрицей системы. Числа, стоящие в правых частях СЛАУ, образуют столбец

$$b = \begin{pmatrix} b^1 \\ b^2 \\ \cdots \\ b^m \end{pmatrix}, \tag{1.3}$$

называемый столбцом свободных членов.

Если неизвестные $x^1, x^2, \cdots, x^n$, так же записать в виде столбца

$$x = \begin{pmatrix} x^1 \\ x^2 \\ \cdots \\ x^n \end{pmatrix}. \tag{1.4}$$

Тогда СЛАУ можно представить в матричной записи:

$$\begin{pmatrix} a_1^1 & a_2^1 & \cdots & a_n^1 \\ a_1^2 & a_2^2 & \cdots & a_n^1 \\ \cdots & \cdots & \cdots & \cdots \\ a_1^m & a_2^m & \cdots & a_n^m \end{pmatrix} \times \begin{pmatrix} x^1 \\ x^2 \\ \cdots \\ x^n \end{pmatrix} = \begin{pmatrix} b^1 \\ b^2 \\ \cdots \\ b^m \end{pmatrix}, \tag{1.5}$$

или $Ax = b$.

Если свободные члены СЛАУ равны нулю, то такая система называется однородной, в противном случае – неоднородной.

Однородная система линейных уравнений имеет вид:

$$\begin{pmatrix} a_1^1 x^1 + a_2^1 x^2 + \cdots + a_n^1 x^n = 0, \\ a_1^2 x^1 + a_2^2 x^2 + \cdots + a_n^2 x^n = 0, \\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots, \\ a_1^m x^1 + a_2^m x^2 + \cdots + a_n^m x^n = 0 \end{pmatrix}. \tag{1.6}$$

Совокупность чисел $\alpha^1, \alpha^2, \cdots, \alpha^n$ называется решением СЛАУ, если каждое уравнение системы обращается в чистовое равенство после подстановки в него чисел $\alpha^1, \alpha^2, \cdots, \alpha^n$ вместо соответствующих неизвестных $x^1, x^2, \cdots, x^n$.

Представим СЛАУ в более простом виде:

$$x^1 \times \begin{pmatrix} a_1^1 \\ a_1^2 \\ \cdots \\ a_1^m \end{pmatrix} + x^2 \times \begin{pmatrix} a_2^1 \\ a_2^2 \\ \cdots \\ a_n^m \end{pmatrix} + \cdots + x^n \times \begin{pmatrix} a_n^1 \\ a_n^2 \\ \cdots \\ a_n^m \end{pmatrix} = \begin{pmatrix} b^1 \\ b^2 \\ \cdots \\ b^m \end{pmatrix}. \quad (1.7)$$

Отсюда вытекает следующая интерпретация решения системы линейных уравнений – это совокупность коэффициентов $x^1, x^2, \cdots, x^n$, с которыми столбец свободных членов раскладывается по столбцам матрицы A системы.

Если СЛАУ имеет решение, то его называют определенным. При этом, если имеется несколько решений, то совместным. Неопределенной СЛАУ называют если решений бесконечно много. Системы, не имеющие решений, называют несовместными.

Решение систем линейных алгебраических уравнений с разреженной матрицей $Ax = b$ размера $n \times n$ является одной из стандартных наиболее часто встречающихся трудоемких вычислительных задач. Где под разреженной матрицей подразумевают матрицу с преимущественно нулевыми элементами. Чаще встречаются задачах, которые объединены одним общим свойством: в этих задачах присутствует большое количество неизвестных, связанных между собой уравнениями, однако в каждую связь входит лишь некоторые неизвестные [3,10].

Благодаря параллельным алгоритмам и развитию архитектур центральных процессоров можно существенно оптимизировать решение разреженных СЛАУ. Однако в настоящее время происходит развитие вычислений на графических процессорах, которые при определенных условиях имеют показатели производительности значительно выше, чем у центральных процессоров.

Таким образом, решение разреженных СЛАУ на высокопроизводительных многопроцессорных системах все еще является открытой проблемой вычислительной математики и требует дальнейших исследований.

1.3 Численные методы решения систем линейных алгебраических уравнений

Методы численного решения СЛАУ делятся на две большие группы:

прямые методы и итерационные методы.

Прямыми методами называются методы, позволяющие получить решение системы за конечное число арифметических операций, т.е. эквивалентными преобразованиями привести решаемую алгебраическую систему к наиболее простому виду, из которого решение находится уже непосредственно. Под простейшими подразумеваются системы с диагональными, двухдиагональными, треугольными и т.п. матрицами. Уменьшение ненулевых элементов в матрице преобразованной системы, ведет к простоте и устойчивости решения, но, с другой стороны, приведение к такому виду сложно и неустойчиво. Обычно стремятся к компромиссу между этими взаимно противоположными требованиями, и в зависимости от целей, преследуемых при решении СЛАУ, приводят её к диагональному, двухдиагональному или треугольному виду [12].

К прямым методам относятся метод Крамера, метод Гаусса, LU-метод и т.д. [5,11]. Эти методы можно назвать конечными или даже точными. Точными они являются, поскольку решение выражается в виде точных формул через коэффициенты системы.

Следует заметить, что реализация прямых методов на компьютере приводит к решению с погрешностью, т.к. все арифметические операции над переменными с плавающей точкой выполняются с округлением. В зависимости от свойств матрицы исходной системы эти погрешности могут достигать значительных величин.

Итерационные методы или методы последовательных приближений — это методы, порождающие последовательность приближений $\{x^{(k)}\}_{k=0}^{\infty}$ к искомому решению x^*, которое получается как предел

$$x^* = \lim_{k\to\infty} x^{(k)} \qquad (1.8)$$

Сходимость метода обычно подразумевает под собой, если к пределу сходится конструируемая им последовательность приближений $\{x^{(k)}\}$.

К этим методам относятся метод Зейделя, Якоби, метод верхних релаксаций и т.д.

Естественно, что практическом решении переход к пределу по $k \to \infty$ невозможен в силу конечности объёма вычислений, который возможно провести. Поэтому нахождение x^* при вычислении итерационных методов заменяется на нахождение какого-то достаточно хорошего приближения $x^{(k)}$ к x^*. Важной задачей является правильный выбор условия остановки итераций, при котором итерационный процесс завершится и будет решение $x^{(k)}$.

Вообще методы последовательных приближений возникли как уточняющие процедуры, которые позволяют решать системы уравнений за небольшое количество итераций, получив тем самым приемлемое по точности приближённое решение задачи.

Большинство итерационных методов являются самоисправляющимися, т. е. допущенная погрешность в вычислениях, при сходимости исправляется в ходе итерирования и не отражается, или отражается незначительно, на окончательном результате. Это следует из конструкции оператора перехода, в котором обычно по самому его построению присутствует информация о решаемой системе уравнений. При выполнении алгоритма эта информация на каждом шаге оказывает влияние на итерации. Это существенное отличие итерационного метода от прямого, ведь в прямых методах решения СЛАУ отталкиваясь от исходной системы, оперируем с её следствиями без обратной связи.

При программировании сходящиеся итерационные методы, при небольшом количестве итераций могут обеспечивать выигрыш по времени даже для СЛАУ общего вида. Так же итерационные процессы довольно легко программируются, так как представляют собой повторяющиеся алгоритмы, применяемые к последовательным приближениям к решению.

При решении СЛАУ с разреженными матрицами в итерационных процессах легче, чем в прямых методах, учитывать структуру нулевых и ненулевых элементов матрицы.

Также как прямые методы, так и итерационные в большинстве своем не обладают естественным параллелизмом, и их параллельная реализация либо

требует существенного изменения алгоритма, либо основывается на работе с матрицами специального вида, допускающими независимую обработку своих элементов.

1.4 Проектирование алгоритма решения систем линейных алгебраических уравнений

Квадратную $n \times n$ матрицу $A = (a_{ij})$ называют матрицей с диагональным преобладанием, если для любого $i = 1, 2, \ldots, n$ имеет место

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}|. \quad (1.9)$$

Матрицы, полностью удовлетворяющие этому определению, можно назвать матрицами со «строгим диагональным преобладанием». Нестрогое диагональное преобладание $n \times n$ матрицы $A = (a_{ij})$ будет в случае выполнения неравенств

$$|a_{ii}| \geq \sum_{j \neq i} |a_{ij}|, \quad (1.10)$$

для любого $i = 1, 2, \ldots, n$. В данном случае описывается диагональное преобладание «по строкам», однако так же существует диагональное преобладание «по столбцам», которое определяется совершенно аналогичным образом с суммированием вне диагональных элементов по столбцам.

Далее выберем метод решения диагональной СЛАУ. Как уже было сказано выше, прямые методы не учитывают высокую степень разреженности матриц, и поэтому, несмотря на их простоту и универсальность, не позволяют эффективно решать СЛАУ. Решение прямым методом может модифицировать матрицу, хранящую СЛАУ. Кроме того, выбор главного элемента предполагает перестановку столбцов и строк матрицы, из-за чего в процессе решения может получиться практически полностью заполненные СЛАУ с большими порядками, что неизбежно приведет к увеличению объема данных и последующего возрастания вычислительных затрат на реализацию. Поэтому в последнее время, для решения СЛАУ большой размерности, широко

применяются итерационные методы.

К настоящему времени существуют несколько итерационных методов. В отличии от прямых, итерационные методы как правило не меняют матрицу или меняют незначительно. Данные методы решения из-за итераций требуют большего объёма процессорного времени. Их главная вычислительная сложность заключается в многократном умножении матрицы на вектор, данная операция отлично распараллеливается, что делает итерационные методы хорошим методом решения на многоядерных GPU [9].

В нашем случае для решения СЛАУ на Tesla K10 реализуем классические итерационные методы решения СЛАУ. Выбор падает на методы Якоби и Гаусса-Зейделя.

На рисунке 1.1 изображена параллельная схема метода Якоби [3], где показано распределение матрицы A и компонент векторов b, x^0 по процессорам $P_1, P_2, \dots, P_N$. Итерационный метод Якоби строится на решении алгебраических уравнений $Ax = b$, где диагональные элементы матрицы $A = (a_{ij})$ не равны нулю, т.е. $a_{ii} \neq 0, i = 1,2, \dots, n$. Данное условие не ограничивает решение методом Якоби и других СЛАУ, которые не подходят под данное требование, поскольку в неособенной квадратной матрице A всегда есть возможность сделать диагональные элементы ненулевыми с помощью перестановки строк СЛАУ.

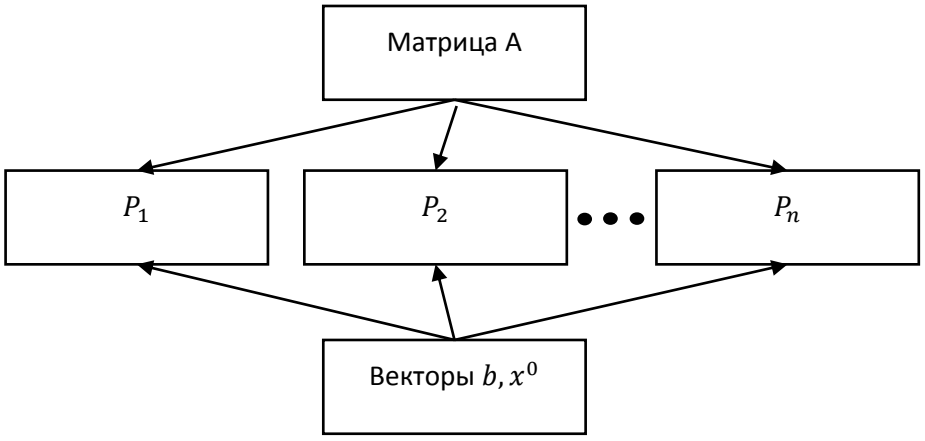

Рисунок 1.1 – Схема параллельного метода Якоби

В итерационном методе Якоби вычисления производится по инструкции

$$x_i^{(k+1)} \leftarrow \frac{1}{a_{ii}}\left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)}\right), i = 1,2, \dots, n. \quad (1.11)$$

где x – неизвестное число;

a – коэффициенты при неизвестных числах;

b – свободные члены;

k – счетчик итераций.

компоненты очередного приближения $x^{(k+1)}$ находятся последовательно одна за другой, так что к моменту вычисления -ой компоненты вектора $x^{(k+1)}$ уже найдены $x_1^{(k+1)}, x_2^{(k+1)}, \dots, x_{i-1}^{(k+1)}$.

Достаточным признаком сходимости будем считать утверждение – если в системе линейных алгебраических уравнений $Ax = b$ матрица A является квадратной и имеет диагональное преобладание, то метод Якоби для решения этой системы сходится при любом начальном приближении.

Диагональное преобладание в матрице $A = (a_{ij})$ означает, что

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}|, i = 1,2, \dots, n. \quad (1.12)$$

Следовательно,

$$\sum_{j \neq i} \left|\frac{a_{ij}}{a_{ii}}\right| < 1, i = 1,2, \dots, n. \quad (1.13)$$

что равносильно

$$\max_{1 \leq i \leq n} \left(\sum_{j \neq i} \left|\frac{a_{ij}}{a_{ii}}\right|\right) < 1. \quad (1.14)$$

Однако метод Якоби никак не использует новые значения $x^{(k+1)}$, и при вычислении любой компоненты следующего приближения всегда опирается только на вектор x^k предшествующего приближения. Если итерации сходятся к решению, то естественно ожидать, что все компоненты $x^{(k+1)}$ ближе к

искомому решению, чем x^k, а посему немедленное вовлечение их в процесс вычислений будет способствовать ускорению сходимости.

На этой идее основан итерационный метод Гаусса-Зейделя. В нём суммирование в формуле (1.11) для вычисления -ой компоненты очередного приближения $x^{(k+1)}$ разбито на две части — по индексам, предшествующим i, и по индексам, следующим за i. Первая часть суммы использует новые вычисленные значения $x_1^{(k+1)}, \dots, x_{i-1}^{(k+1)}$, тогда как вторая — компоненты $x_{i+1}^{(k+1)}, \dots, x_n^{(k+1)}$ из старого приближения. Метод Гаусса-Зейделя иногда называют также итерационным методом «последовательных смещений», а его основная идея — немедленно вовлекать уже полученную информацию в вычислительный процесс — с успехом применима и для нелинейных итерационных схем.

Опишем суть метода. Пусть есть система линейных алгебраических уравнений, чтобы получить для метода Гаусса-Зейделя матричное представление, перепишем его расчётные формулы в виде

$$a_{ii}x_i^{(k+1)} + \sum_{j=1}^{i-1} a_{ij}x_j^{(k-1)} = -\sum_{j=i+1}^{n} a_{ij}x_j^{(k)} + b_i, i = 1,2,\dots,n\,, \qquad (1.15)$$

Тогда организация вычислений будет происходить по инструкции:

$$x_i^{(k+1)} \leftarrow \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k-1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{k}\right), i = 1,2,\dots,n, \qquad (1.16)$$

Ввиду диагонального преобладания в матрице A достаточное условие сходимости СЛАУ, так же, как и у метода Якоби, будет:

$$\sum_{j\neq i}\left|\frac{a_{ij}}{a_{ii}}\right| < 1. \qquad (1.17)$$

1.5 Описание вычислителя на графическом процессоре лаборатории распределенных вычислений ТГУ

В крупнейшем учебном заведении Тольятти, Тольяттинском

Государственном университете, в 2014 году создана лаборатория распределенных вычислений.

В основе суперкомпьютера ТГУ лежит серверная платформа SuperMicro SYS-7047GR-TPRF, с двумя восьми ядерными CPU Intel Xeon E5-2670 работающей на максимальной частоте 3.3Гц. Оперативная память составляет 64 Гб высокоскоростной памяти третьего поколения.

За вычисления отвечают четыре GPU ускорителя NVIDIA Tesla K10, имея суммарно 12288 ядер графических процессоров и обеспечивающие производительность примерно в 18.32 TFLOPS в операциях с одинарной точностью.

GPU NVIDIA Tesla – это массивно параллельные ускорители, основанные на платформе параллельных вычислений NVIDIA CUDA. Графические процессоры Tesla созданы с нуля для экономичных, высокопроизводительных вычислений. NVIDIA Tesla обеспечивает наилучшую производительность и эффективность для обработки сейсмических данных, моделирования биохимии, моделирование погоды и климата, и других научных и коммерческих приложений по сравнению с системой на базе CPU. [14]

Tesla K10 основана на архитектуре третьего поколения от NVIDIA для вычислений на CUDA – Kepler. Состоящий из 7,1 млрд транзисторов, Kepler является самым быстрым и эффективным в мире высокопроизводительным процессором для вычислений на Суперэвм. Kepler разработана с нуля, чтобы максимизировать производительность вычислений с максимальной энергоэффективностью. Архитектура имеет новшества, которые делают гибридные вычисления значительно проще, более доступными и применимыми к более широкому набору приложений [13, 14].

С точки зрения стоимости и энергоэффективности основанные на архитектуре NVIDIA видеоускорители имеют высокую производительность. Архитектура NVIDIA Kepler является одной из самых быстрых, эффективных и производительных архитектур на сегодняшний день [14].

Основной единицей построения видеокарт является потоковый

мультипроцессор SM (Streaming Multiprocessor) или SMX (Next Generation Streaming Multiprocessors). Он чем-то напоминает ядро CPU, однако в архитектуре Kepler в одном SM расположены 192 вычислительных ядра CUDA core, которые и выполняют код параллельно [21].

Kepler увеличил максимальное число одновременных блоков на мультипроцессоре с 8 до 16, по сравнению со старыми архитектурами. В результате близкого размещения ядер достигается максимальное количество блоков и нитей на мультипроцессоре, что может увеличить теоретическую скорость работы архитектуры Kepler [20].

На рисунке 1.2 изображено 16 потоковых мультипроцессоров [14, 15], которые содержатся в Tesla K10.

Рисунок 1.2 – Кластеры SMX в архитектуре Kepler

На рисунке 1.3 изображена структура одного мультипроцессора [14], в каждом из 16 находятся по 192 вычислительных ядра CUDA core выполняющих код параллельно, что суммарно 3072 CUDA core. 64 DP Unit ядра исполняющие

инструкции для операндов типа double. Именно отношение DP Unit к количеству CUDA cores как 1 к 3 определяет производительность вычислений с плавающей запятой, двойной точностью как 1/3 от производительности с одинарной точностью [17,18].

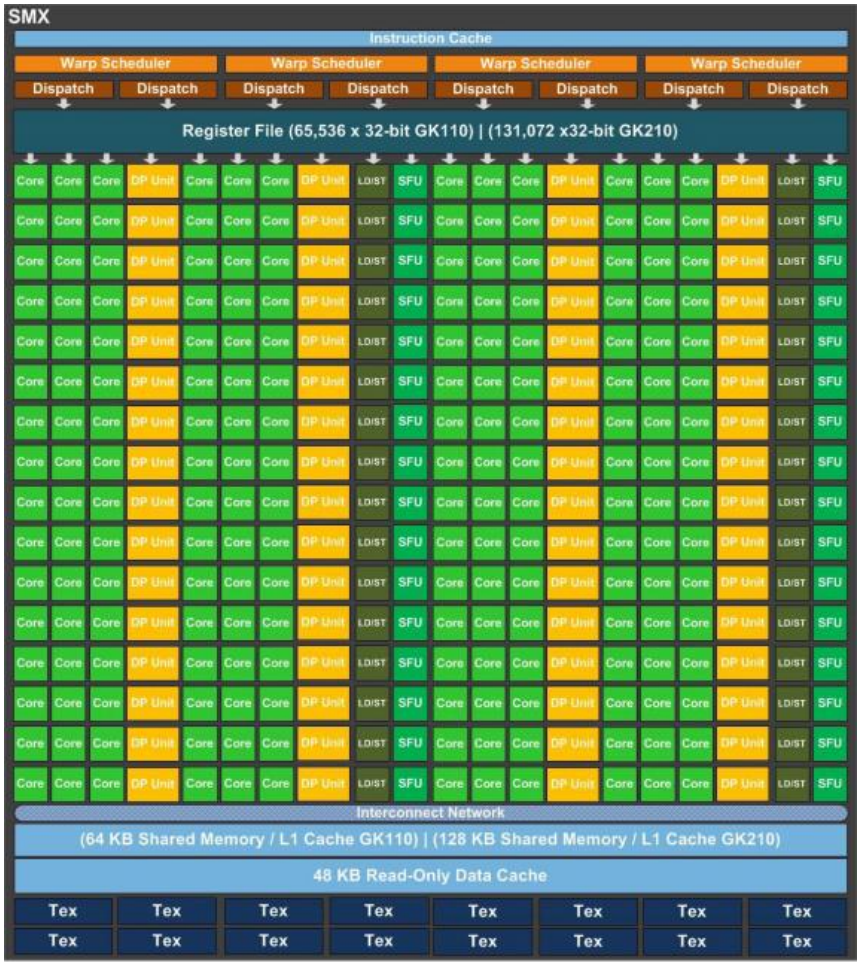

Рисунок 1.3 – Структура SMX

В NVIDIA был разработан специальный компилятор для GPU – CUDA.

Compute Unified Device Architecture — технология GPGPU (General-Purpose computing on Graphics Processing Units), позволяющая программистам реализовывать алгоритмы на упрощённом языке программирования C, C++, Fortran и OPENACC.

С точки зрения программного обеспечения, реализация CUDA представляет собой кроссплатформенную систему компиляции и исполнения программ, части которых работают на CPU и GPU. На CPU выполняется последовательная часть кода и подготовительные стадии для GPU- вычислений,

где будут одновременно выполнятся большим множеством нитей (threads). Нити являются непосредственными исполнителями вычислений, а их создание, управление и удаление является низкозатратным процессом.

У Tesla K10 быстрое переключение контекста нитей. При запуске большого количества нитей исполнения ожидание данных для одних нитей будет покрываться исполнением других. Таким образом для эффективной загрузки Tesla K10, чтобы вычислительные ресурсы не простаивали в ожидании требуемых данных, требуется запускать значительно большее количество нитей, чем количество CUDA ядер [23,24].

Нити выполняются параллельно и имеют доступ ко всей памяти графической платы. Так же из-за аппаратного ограничения на размер блока, нити собираются в два блока по 1536 нитей каждый [22].

На рисунке 1.4 изображена блочная архитектура CPU [16]. Ключевым различием между нитями в одном блоке и нитями в разных блоках является возможность синхронизации. Так глобальная синхронизация внутри кода функции ядра отсутствует, зато можно синхронизовать нити одного блока. Можно организовать взаимодействие этих нитей между собой за счет использования специальной разделяемой памяти.

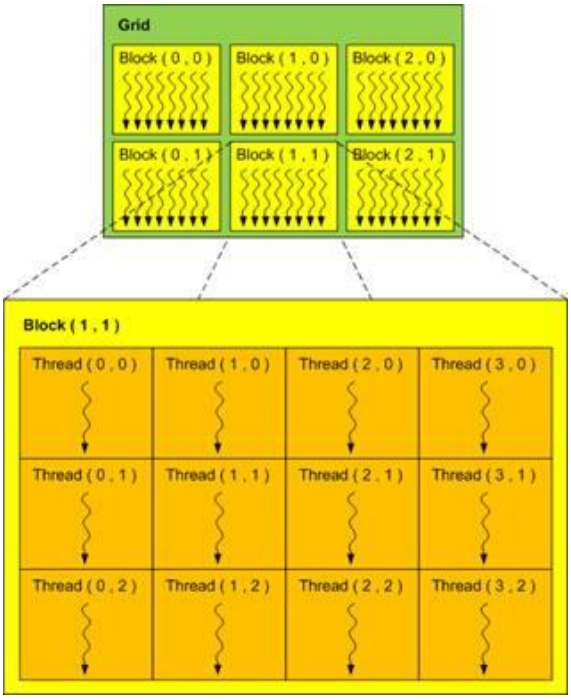

Рисунок 1.4 – Иерархия нитей в CUDA

Нити различных блоков могут находиться на разных стадиях выполнения программы. Такой метод обработки не подходит под определение SIMD «все нити одновременно выполняют одну инструкцию» и под MIMD «каждая нить выполняется независимо от других». Поэтому появилось новое определение, изображенное на рисунке 1.5 [4], SIMT (Single Instruction – Multiple Threads) – «одна инструкция и много потоков».

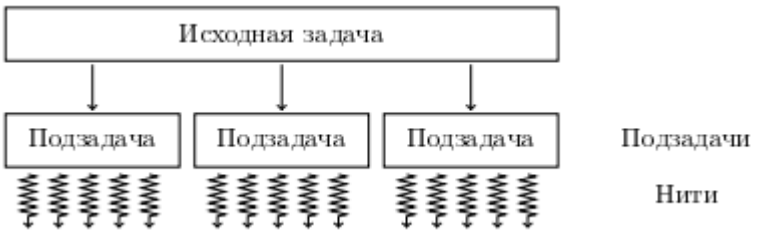

Рисунок 1.5 – Разбиение исходной задачи на набор подзадач

Блоки разделяются на группы по 32 нити. Такая группировка называется варпом [17]. Нити в варпы выделяются последовательно в соответствии с их линейным индексом в блоке. Все нити в варпах выполняют одну инструкцию одновременно. За работой варпов следят планировщики, которые на каждом цикле работы выбирает готовый к исполнению варп и запускает ее на CUDA ядрах мультипроцессора.

Все задачи, решаемые на Tesla K10, предполагают интенсивную работу с большими объемами данных. Поэтому для программирования GPU важным моментом является сбалансированность нагрузки и времени обращения в память видеокарты. Этому аспекту на этапе проектирования алгоритмов должно уделяться много внимания. Так как в память 32 блока объединены в варп, поэтому при работе с Tesla K10 необходимо соблюдать определенные правила, основанные на размере варпов [23,24].

Существует несколько типов памяти, которые показаны в таблице 1.1.

Таблица 1.1 – Типы памяти в GPU Nvidia

Тип памяти	Расположение	Чтение/Запись	Скорость работы
register (регистровая)	мультипроцессор	чтение и запись	высокая
local (локальная)	DRAM GPU	чтение и запись	низкая
shared (разделяемая)	мультипроцессор	чтение и запись	высокая
global (глобальная)	DRAM GPU	чтение и запись	низкая
constant (константная)	DRAM GPU	только чтение	высокая
texture (текстурная)	DRAM GPU	только чтение	высокая

Рассмотрим каждый тип памяти, представленный в таблице 1.1, отдельно.

Регистровая память наиболее быстрая, потому что расположена в потоковом мультипроцессоре. Доступна на чтение и на запись в рамках одной нити. У каждой нити есть свой набор 32 битовых регистров, который недоступен соседним нитям, в то числе, когда они не исполняются. Количество регистров, которые требуются нитям для исполнения уже известно по время компиляции. Все переменные, которые создаются в процессе выполнения нитей, автоматически помещаются в регистры. В этом случае часть локальных переменных будет сохранена в локальную память;

Локальная память находится в оперативной памяти(DRAM) GPU, и ограничена только объемом этой памяти, из-за чего довольно медленная, скорость работы примерно в 400 раз медленнее, чем у регистровой памяти. У каждой нити своя область локальной памяти доступной ей для чтения и для записи. Эта память может значительно снизить работу приложений, если ее использовать вместо регистровой памяти;

Разделяемая память располагается на мультипроцессоре, непосредственно в том месте, где происходят вычисления. Задержка на один – два порядка меньше, чем у глобальной памяти. Доступна на чтение и запись нитями одного блока и, следовательно, с помощью нее можно обмениваться данными внутри блока. Разделяемая память физически делит пространство с кэшем первого уровня и на эти две структуры выделяется 64 кб и может распределяться между ними. Вместе эти структуры – кэшируют медленную глобальную память. Однако разделяемая память является программно управляемым интеллектуальным кэшем. Это означает, что программист может напрямую указать какие данные должны быть помещены в разделяемую память, использование этой памяти является одной из ключевых методов оптимизации программ;

Глобальная память. Через глобальную память передаются данные между Tesla K10 и ЭВМ, на которой запускается компилятор. Процесс копирования данных идет через PCI-Express 3 x16 шины, к которым подключены Tesla K10 в серверной платформе SuperMicro SYS-7047GR-TPRF. Объем памяти Tesla K10 составляет 8192 Мб. Скорость записи на глобальную память низкая, по сравнению с копированием данных внутри GPU. Поэтому количество таких копирований желательно свести к минимуму.

Остальные два типа памяти крайне редко используются при решении математических задач, однако в кратко опишем их.

Константная память. Располагается в глобальной памяти, размер 64 килобайта. Константная память доступна только на чтение с девайса всем нитям сети и на чтение и запись с хоста.

Текстурная память представляет собой участок глобальной памяти, которая доступная только на чтение с GPU всем нитям. Память, так же, как и константная, расположена в глобальной памяти и имеет кэш.

Правильное использование различных видов памяти способно увеличить производительность вычислений. Так, к примеру, нужно минимизировать количество обращений к глобальной и локальной памяти, т.к. они медленные и

не поддерживают кэширования.

Глава 2 Реализация решения систем линейных алгебраических уравнений

2.1 Особенности решения систем линейных алгебраических уравнений на CUDA

Полный отказ от работы с CPU при разработке приложения решения СЛАУ не является возможным, так как графический процессор напрямую не имеет доступ к памяти компьютера. Поэтому прежнему за исполнение программы и постобработку данных будет отвечать CPU, а все трудоемкие вычислительные процессы будут выполняться на GPU.

Общая концепция CUDA строится в том, что GPU, так же называемый устройством или device, выступает в роли массивно-параллельного сопроцессора к GPU, называемым host. Отсюда получаем, что программа на CUDA включает работу как CPU, так и GPU, при этом весь непараллельный код, подготовка и копирование данных на устройство, задание параметров для ядра и его запуск, выполняется на CPU.

Как уже упоминалось, графический процессор напрямую не имеет доступ к памяти компьютера, однако, CPU тоже не может напрямую обращаться в память GPU. Поэтому при работе с данными обычными указателями не обойтись. Для обращения к памяти к GPU через CPU и наоборот, необходимо пользоваться специальными функциями, входящими в стандартный набор CUDA SDK.

Все стандартные математические функции из библиотек языка C, C++ поддерживаются CUDA. Однако есть особенность использования чисел с двойной точностью (double). На графических устройствах наблюдается падение производительности при использовании double, по сравнению числами с плавающей запятой (float), поэтому предпочтительнее использовать, при возможности, тип float [12].

В CUDA есть особенность использования условного оператора if. Когда выполнение программы идет по разным веткам условного оператора if, нити

одного варпа будут выполнять различные инструкции. Оказывается, нити, которые не должны исполнять данную инструкцию просто будут простаивать. В случае ветвления в начале будет выполнена одна ветвь оператора if, а затем другая. Очевидно, что в этом случае можно потерять до 50% теоретической производительности GPU. И все больше усложняется, если такое ветвление многоуровневое. Естественно если все нити варпа должны идти по одной ветви условного оператора, другая ветвь считается, не будет и в этом случае не будет потери производительности. Таким образом, в CUDA программах предпочтительно использовать те реализации алгоритмов, в которых меньше ветвления в рамках одного варпа.

Чтобы добиться максимальной производительности на архитектуре Kepler и его мультипроцессоре (SMX) требуется, чтобы ядро запускалось с полным заполнением Kepler нитями и блоками. Поэтому лучше запускать ядра со значительно большим количеством блоков, чем требуется, чтобы заполнить GPU.

При работе с CUDA можно выделить основные шаги при проектировании программы.

Подготовка памяти. Необходимо заранее распределить данные по типам памяти в GPU. Для этого в CUDA используются функции cudaMalloc, cudaMemcpy и cudaFree, аналогичные стандартным malloc, memcpy и free, но, разумеется, все операции проводятся над видеопамятью. У функции cudaMemcpy присутствует дополнительный параметр, отвечающий направление копирования информации.

Конфигурация блоков**.** Необходимо найти оптимальный баланс между размером блоков и их количеством. Отсюда можно снизить количество обращений к глобальной памяти через увеличение количества потоков и работать с быстрой разделяемой памятью, что повысит производительность. Однако и большое количество регистров выделяемых на блок может негативно сказаться на работе, ведь размер регистров фиксирован и при слишком большом количестве GPU начнет размещать данные в медленной локальной памяти.

Запуск ядра**.** Вызов ядра не отличается от вызова любой функции на языке C, только добавляется в начале объявления идентификатор __global__. Единственное существенное отличие кроется в необходимости заранее передать ему размерности сетки и блока при вызове ядра.

–получение результатов**.** После вычислений на GPU необходимо скопировать результаты выполнения программы в хост. Используется все та же функция cudaMemcpy, только с указанием обратного направления копирования (из GPU в CPU).

Освобождение памяти. И последнее действие, помогающее предотвратить утечку памяти, освободить все выделенные ресурсы.

2.2 Реализация и тестирование алгоритмов решения СЛАУ

На рисунке А.2 представлена блок-схема алгоритма решения СЛАУ методом Гаусса-Зейделя на ЭВМ. В данном алгоритме присутствует неудобный ввод данных с клавиатуры. При решении разряженных СЛАУ, матрицы содержат множество значений и поэтому обычно поступают уже заполненными для алгоритма решения СЛАУ, поэтому реализуем заполнение матрицы генератором случайных чисел. Главная диагональ всегда будет содержать положительные ненулевые значения.

Основная матрица A будет одномерной, содержащей последовательно значения свободных членов при неизвестных. Такой подход должен упросить решение и ускорить работу алгоритма на GPU.

На рисунке 2.1 изображено заполнение главной диагонали матрицы A. Так как матрица будет одномерной, то заполнять необходимо с условием $i + i * N$, где цикл до N. Таким способом, шагая каждый раз на N элементов вперед, заполним главную диагональ. Дополнительные диагонали будут заполняться похожим образом.

```
for (i = 0; i < N; i++) {
    hA[i + i*N] = rand() % 50 + 1.0f*N;
}
for (k = 1; k < Num_diag + 1; k++) {
    for (i = 0; i < N - k; i++) {
        hA[i + k + i*N] = rand() % 5;
        hA[i + (i + k)*N] = rand() % 5;
    }
}
```

Рисунок 2.1 – Заполнение главной и дополнительной диагонали

Для численного решения СЛАУ построим точное решение x матрицы A используя генератор случайных чисел, см. рисунок 2.2

```
for (i = 0; i < N; i++) {
    hX[i] = rand() % 50;
}
```

Рисунок 2.2 – Заполнение точного решения x

На рисунке 2.3 изображено заполнение столбца свободных членов $b = Ax$. Такой подход позволит генерировать СЛАУ имеющие решения, что необходимо для точного тестирования алгоритма. Естественно при численном решении будем использовать только матрицу A и столбец свободных членов b. Идея распараллеливания итерационного процесса для GPU состоит в параллельных вычислениях i компонент вектора $x_i^{(k+1)}$ при фиксированном k. Таким образом номер нити на GPU будет соответствовать номеру i.

```
for (i = 0; i < N; i++) {
    sum = 0.0f;
    for (j = 0; j < N; j++) sum += hA[j + i*N] * hX[j];
    hF[i] = sum;
}
```

Рисунок 2.3 – Заполнение столбца свободных членов

Алгоритм решения СЛАУ на GPU будет работать как на девайсе, так и на хосте. На хост будет выполняться заполнение массива A, нахождение x и b, а так же выполнятся цикл по итерациям k и запускается функция ядро ответственная за итерации. На рисунке 2.4 изображена функция ядро

выполняется на девайсе, где параллельно пересчитывает по вектору $x_i^{(k)}$ вектор $x_i^{(k+1)}$. Данный метод распараллеливания на GPU не является сложным и не требует особых знаний архитектуры GPU от программиста. Все расчеты будем проводить с двойной точностью (double).

```
__global__ void KernelGZ(double* dA, double* dF, double* dX0, double* dX1, int N) {
    double sum = 0.0f;
    for (int i = 0; i < N; i++) dX0[i] = dX1[i];
    int t = blockIdx.x * blockDim.x + threadIdx.x;
    for (int j = 0; j<t; j++) sum += dA[j + t*N] * dX1[j];
    for (int j = t + 1; j<N; j++) sum += dA[j + t*N] * dX0[j];
    dX1[t] = (dF[t] - sum) / dA[t + t*N];
}
```

Рисунок 2.4 – Функция–ядро, выполняющая метод Гаусса-Зейделя

Функция ядра вызывается на каждой итерации по начальному приближению $x_i^{(k)}$ находит следующее приближение решения $x_i^{(k+1)}$. . Каждая нить вычисляет свою компоненту вектора $x_i^{(k+1)}$.

Реализация метода Якоби не сильно отличается от метода Гаусса-Зейделя и представлена на рисунке А.1. Функция решения методом Якоби изображена на рисунке 2.5. Главное отличие состоит в процессе итераций, где вектор $x_i^{(k)}$ не будет заменяться на $x_i^{(k+1)}$, тем самым будет храниться оба вектора приближений. Вовремя перемножения матрицы А на столбец начального приближения $x_i^{(k)}$ в случае попадания на диагональный элемент матрицы происходит его запоминание тем самым экономится одно обращение в глобальную память, так как диагональный элемент используется при вычислении приближения x.

```
__global__ void KernelJacobi(double* dA, double* dF, double* dX0, double* dX1, int N) {
    double temp;
    int t = blockIdx.x * blockDim.x + threadIdx.x;
    dX1[t] = dF[t];
    for (int g = 0; g < N; g++) {
        if (t != g)
            dX1[t] -= dA[g + t*N] * dX0[g];
        else temp = dA[g + t*N];
    }
    dX1[t] /= temp;
}
```

Рисунок 2.5 – Функция–ядро, выполняющая метод Якоби

Так как оба метода вычисляют компоненты приближения по нитям, то для того чтобы ядро могло однозначно определить номер нити, а значит, и элемент данных, который нужно обработать, необходимо использовать встроенные переменные $threadIdx, blockIdx$ типа $dim3$, изображенные на рисунке 2.6. Каждая из этих переменных является трехмерным целочисленным вектором. Отсюда следует определение типа $dim3$ – вектор, принимающий значения сетки и блоков при вызове ядра.

```
int Block = (int)ceil((float)N / Thread);
dim3 Blocks(Block);
dim3 Threads(Thread);
```

Рисунок 2.6 – Целочисленные вектора блоков и нитей

Переменная `Thread` будет отвечать за количество нитей в блоках. Сами блоки будут вычисляться как округленное ближайшее целое значение деления размерности матрицы на количество нитей.

На рисунке 2.7 предоставлено отображение локальных номеров нити и блока в глобальный индекс нити [4]. Зная номер нити внутри блока, номер блока внутри сетки и размерность сетки, можно получить глобальный индекс нити:

$$Idx = blockIdx.x * blockDim.x + threadIdx.x, \quad (2.1)$$

$$Idy = blockIdx.y * blockDim.y + threadIdx.y, \quad (2.2)$$

$$Idz = blockIdx.z * blockDim.z + threadIdx.z. \quad (2.3)$$

Максимальное число нитей в блоке ограничено, так как все нити блока располагаются на одном потоковом процессоре и должны разделять ограниченное число ресурсов этого процессора. Блок нитей может содержать до 1024 нитей. Однако ядро может быть запущено на большем числе блоков. Поэтому общее число нитей будет равно произведению числа нитей в блоке и числа блоков.

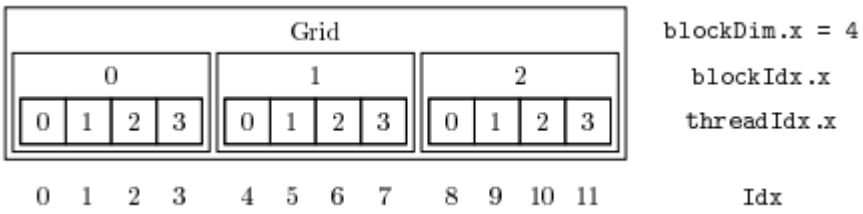

Рисунок 2.7 – Индексы нитей и блоков

Для девайса будут созданы свои переменные, в которые будут записываться данные с хоста. На рисунке 2.8 изображено выделение памяти и копирование переменных в память девайса. После вычислений на GPU переменные ответственные за начальное приближение будут возвращены на хост.

```
cudaMalloc((void**)&dA, mem_sizeA);
cudaMalloc((void**)&dF, mem_sizeX);
cudaMalloc((void**)&dX0, mem_sizeX);
cudaMalloc((void**)&dX1, mem_sizeX);
cudaMalloc((void**)&delta, mem_sizeX);
cudaMalloc((void**)&dT, mem_sizeA);
cudaMemcpy(dA, hA, mem_sizeA, cudaMemcpyHostToDevice);
cudaMemcpy(dF, hF, mem_sizeX, cudaMemcpyHostToDevice);
cudaMemcpy(dX0, hX0, mem_sizeX, cudaMemcpyHostToDevice);
cudaMemcpy(dX1, hX1, mem_sizeX, cudaMemcpyHostToDevice);
```

Рисунок 2.8 – Выделение памяти и копирование переменных в память GPU

Для проверки корректности выполнения алгоритмов решения проведем тестирование. Для тестирования реализованы выводы результатов, указанные на рисунке 2.9, в которые входит количество итераций и столбец решения x.

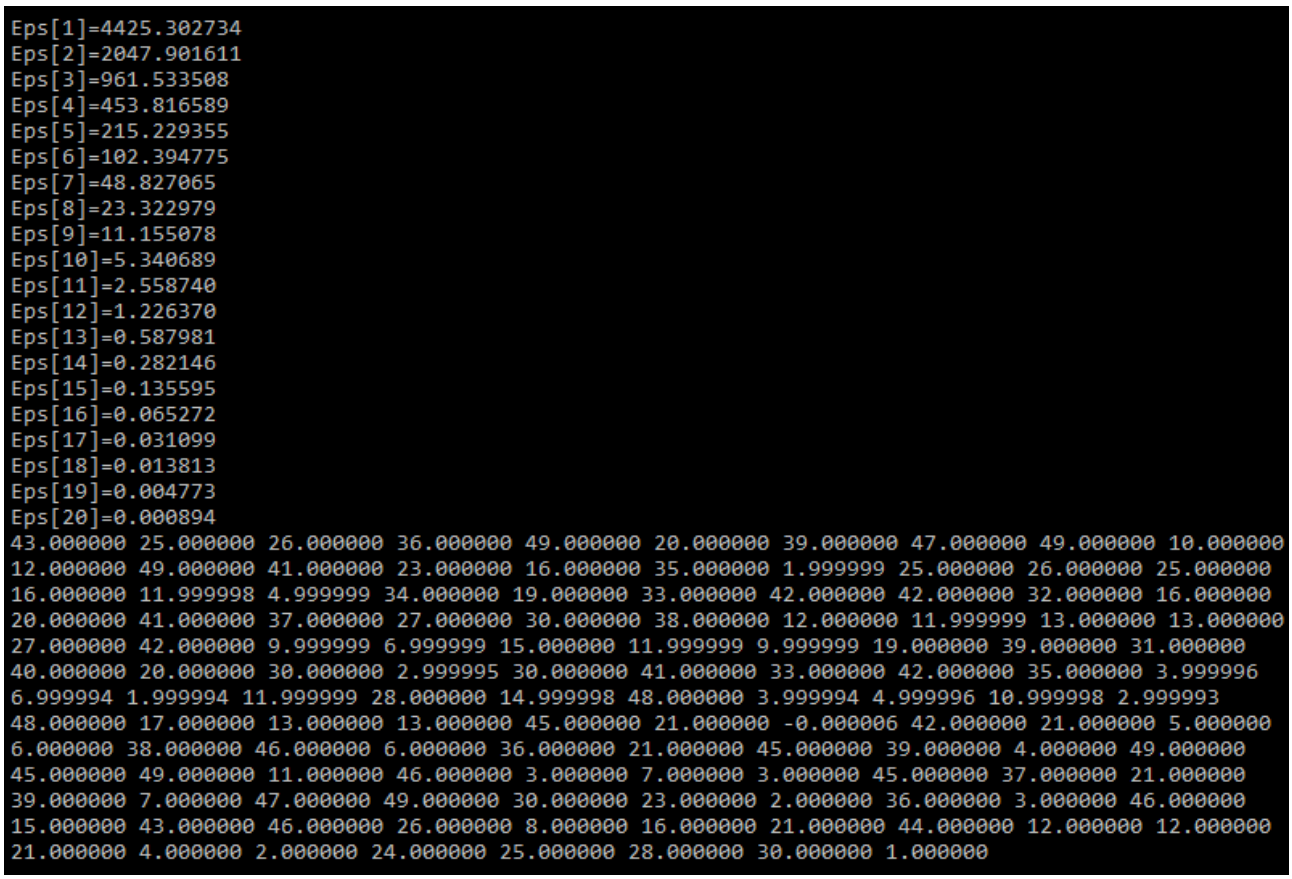

Рисунок 2.9 – Вывод результатов работы алгоритма Якоби

В сам процесс тестирования будет входить запуск программы, корректная отработка программы и вывод результатов. Размерность матрицы и количество нитей в блоке при тестировании будет меняться, чтобы протестировать алгоритмы более детально. Как видно на рисунке 2.9 метод Якоби решил СЛАУ после 20 итераций при матрице в 100×100 элементов, количество нитей в блоке не влияет на столбец решения x. Метод Гаусса-Зейделя показывает схожие показатели. Полный листинг кода алгоритмов решения СЛАУ методом Якоби и методом Гаусса-Зейделя показаны в приложении Б и В.

При тестировании алгоритмов не было выявлено критических ошибок. Оба алгоритма выполняются корректно при размерностях от 100×100 до 10000×10000 элементов и количеством нитей в 32, 64, 128, 256, 512, 1024 на блок.

Глава 3 Анализ эффективности разработанных алгоритмов

3.1 Разработка технологии анализа эффективности алгоритмов

Под эффективностью работы алгоритмов будем подразумевать скорость выполнения и общую загрузку графического процессора.

Важной характеристикой работы алгоритма на GPU является знание загруженности GPU или occupancy. Загруженностью будем называть отношение числа активных варпов на мультипроцессоре к максимальному количеству возможных активных варпов. Данный критерий покажет загруженность GPU во время выполнения алгоритмов и тем самым, насколько эффективно используются ресурсы GPU.

Следующей характеристикой анализа эффективности будет служить время выполнения приложения. Это один из самых простых и показательных методов проверки эффективности выполнения приложения.

Однако нас не устраивает общее время т.е. выполнение программы на CPU и GPU. Наша задача сравнить время только время выполнения на GPU и некие затраты на распределение ресурсов, поэтому считать будем время, затраченное на создание переменных для GPU, перенос переменных из памяти CPU в память GPU, вычисления на GPU и обратный перенос переменных из памяти GPU в память CPU.

Один из методов анализа эффективности – Visual Profiler необходимой для профилирования создаваемого приложения.

Профилирование – это выполнения приложения и сбор сведений о параметрах работы, анализируя специальные счетчики производительности, встроенные в GPU, для последующего анализа, с целью повышения эффективности и поиска узких мест. Профилирование показывает сколько времени было затрачено на выполнение каждого ядра, сколько было запущено блоков, объединялись ли операции доступа к памяти со стороны ядра, сколько расходящихся ветвей было выполнено в варпах и т.д.

При запуске профайлер просит указать пусть исполняемого файла и выбрать счетчики (counters), которые нужно отслеживать. Каждый счетчик

показывает, насколько часто происходит определенное событие. После этого программа запускается, и профайлер регистрирует значения счётчиков по ходу ее исполнения для каждого ядра в отдельности.

Когда выполнение приложения завершено, профайлер создаст таблицу по запуску каждого ядра, в которой можно сравнить значения счетчиков, и тем самым выявить узкие места в коде.

Профилирование кода можно так же запустить вручную, без использования Visual Profiler. Для этого в коде необходимо установить переменную среды CUDA_PROFILE=1. Информация после ручного профилирования будет записана в лог-файл.

Именно профайлер показывает загруженность GPU. Более высокая загруженность не всегда приравнивается к более высокой производительности. Значение, выше которого дополнительная загруженность не всегда увеличивает производительность, находится в промежутке в 50-66%. Однако низкая загруженность всегда приводит к ухудшению производительности [19].

Следующий метод использование CUDA events. Использование стандартных библиотек time языков C, C++ замедляет работу GPU. [6] По этой причине, CUDA предлагает относительно облегченную альтернативу таймеров. Для точного измерения времени выполнения различных операций на GPU воспользуемся так называемыми событиями (CUDA events). Важное значение при анализе эффективности имеет возможность точно замерять время выполнения различных операций на GPU. Событие – это объект типа cudaEvent_t, используемый для обозначения «точки» среди вызовов CUDA. Соответствующие функции позволяют создавать и уничтожать CUDA события, обозначать позиции начала и конца анализируемого фрагмента кода, проверять возникновение определённого события или ожидать его наступления, а также получать время в миллисекундах, прошедшее между двумя событиями. Каждое событие, привязанное к точке, характеризуется тем, пройдена GPU данная точка или нет.

CUDA runtime API обеспечивает точный замер времени, позволяя

приложению асинхронно записывать события в любой точке программы, запрашивать наступило ли данное событие, ждать наступления события, а так же получать интервал времени в миллисекундах между событиями.

3.2 Описание эксперимента и их анализ полученных данных

Алгоритмы запускались последовательно. Для выбора среднего значения времени каждый алгоритм повторялся 50 раз, каждый раз получая новые значения матрицы A и столбца свободных членов b.

Для более детального анализа алгоритмов использовались различные по размерам матрицы. Начиная от 1000×1000, и заканчивая 10000×10000. Такой разброс помог более детально скорость выполнения при каждой размерности матрицы.

Тестирование по времени алгоритмов Гаусса-Зейделя и Якоби показало незначительную разницу, которая объясняется не использованием новых значения $x^{(k+1)}$ методом Якоби и тем самым экономией обращений в медленную глобальную память. График тестирования показан на рисунке 3.1.

Рисунок 3.1 – Сравнение времени выполнения алгоритмов Гаусса-Зейделя и метода Якоби

Следующий этап тестирования реализаций – профилирование алгоритмов. В нашем случае загрузка GPU составила 23.5% и 24.4% для метода Гаусса-Зейделя и метода Якоби соответственно, что показано на рисунках 3.2 и 3.3. Показатель загрузки говорит о возможности поднять производительность как минимум в два раза, до 50% загрузки GPU и выше.

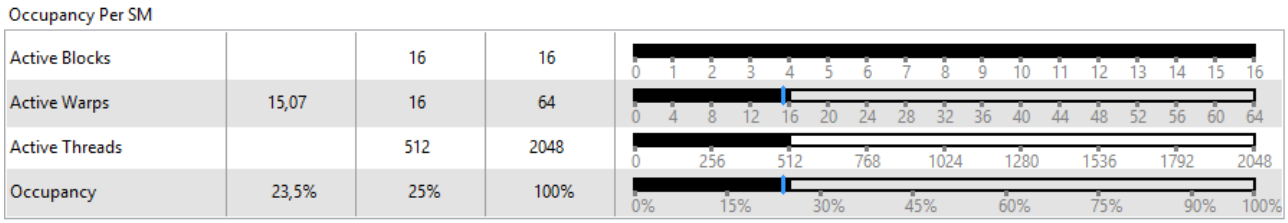

Рисунок 3.2 – Загруженность GPU при методе Гаусса-Зейделя

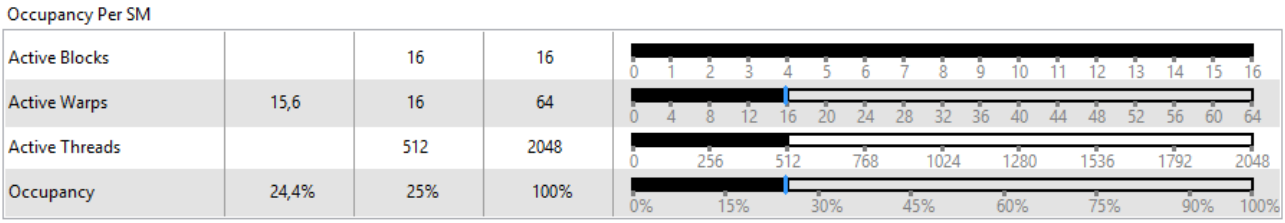

Рисунок 3.3 – Загруженность GPU при методе Якоби

В функциях обоих методов решения СЛАУ на GPU присутствует неэффективное обращение в глобальную память. Считывание матрицы A происходит из глобальной памяти в виде одномерного массива с индексацией по двум индексам j и i, где N число элементов в строке или столбце матрицы, поэтому переменная i соответствует номеру нити A так как матрица квадратная. При считывании из глобальной памяти идет обращение к варпам в этом случае, при фиксированном j, варп считывает элементы матрицы A стоящие друг от друга на N элементов. Следовательно, такое считывание из глобальной памяти идет непоследовательно одним непрерывным блоком, а с промежутками, что приводит к многочисленным обращениям в глобальную память. Отсюда увеличится время считывания и как следствие понижается производительность. Решение данной проблемы является транспонирование матрицы A. Если скопировать в глобальную память транспонированную матрицу A, тогда при

фиксированном номере *j* считывание из глобальной памяти варпом.

Идея транспонирования состоит в разбиении исходной матрицы на двумерные блоки. Каждый блок будет обрабатывать свою подматрицу см. Рисунок 3.4. Такая схема обеспечит чтение глобальной памяти с объединением запросов. Так как итерационные методы не изменяют исходную матрицу, а в нашем случае исходная станет транспонированной, то запускать функцию транспонирования есть смысл вне итерационного процесса, т.е. до вычислений приближений.

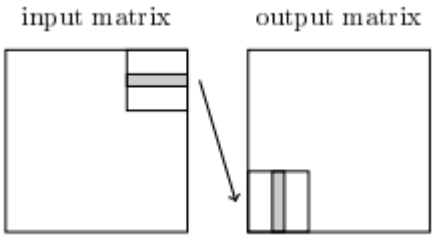

Рисунок 3.4 – Транспонирование матрицы

Полный листинг кода алгоритмов решения СЛАУ методом Якоби и методом Гаусса-Зейделя для транспонированных матриц, а также сам код транспонирования, показан в приложении Г и Д соответственно.

Для дальнейшего тестирования оптимизации методов, необходимо протестировать алгоритм транспонирования матрицы. Хоть алгоритм запускается только один раз при решении любого из двух методов, он может значительно увеличить время выполнение итерационных методов.

Во временные рамки транспонирования алгоритма не входят пересылка памяти между CPU и GPU, а также наоборот.

Анализ блочного транспонирования матрицы показывает неудовлетворяющую загрузку GPU и высокое время выполнения, поэтому для производительности используем разделяемую память. Зависимость времени работы алгоритма от размерности матрицы показано на рисунке 3.5.

Рисунок 3.5 – Время выполнения блочного транспонирования матрицы

В алгоритме находится неэффективное обращение в глобальную память, в ходе которого получаем задержки при обращении к памяти, поэтому реализуем данный метод с применением разделяемой памяти. Идея увеличения производительности заключается в том, что копирование информации из одного участка глобальной памяти в другой занимает куда больше времени, чем копирование из глобальной памяти в разделяемую, а потом назад, в глобальную. Поэтому теперь блок нитей будет считывать подматрицу, записывать ее в разделяемую память, вычислять новые индексы и записывать. Результат транспонирования представлен на рисунке 3.6.

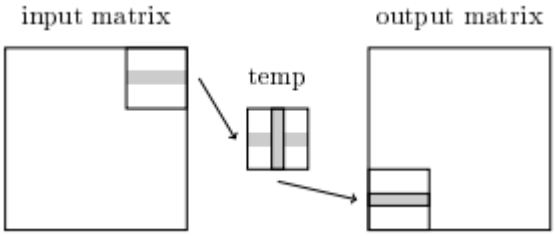

Рисунок 3.6 – Транспонирование матрицы с помощью разделяемой памяти

Дальнейшее тестирование проводилось с двумя методами: блочной и блочной с использованием разделяемой памятью, смотрите рисунок 3.7.

Рисунок 3.7 – Сравнение времени выполнения алгоритмов Гаусса-Зейделя блочной реализации и реализации блочной с разделяемой памятью

Блочная с разделяемой памятью показывает стабильно высокую производительность, а загрузка GPU, изображенная на рисунке 3.8, достигла 91.2%. При 10000×10000 элементов выигрыш составляет 2 раза, по сравнению с блочной реализацией без разделяемой памяти.

Occupancy Per SM

Active Blocks		2	16	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Active Warps	58,38	64	64	0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60 64
Active Threads		2048	2048	0 256 512 768 1024 1280 1536 1792 2048
Occupancy	91,2%	100%	100%	0% 15% 30% 45% 60% 75% 90% 100%

Рисунок 3.8 – Загруженность метода транспонирования матрицы

Следующее тестирование проводилось между методом Гаусса-Зейделя и Гаусса-Зейделя с транспонированной матрицей. Во время выполнения Гаусса-Зейделя и метода Якоби с транспонированной матрицей включается и само транспонирование, однако загрузка GPU указывается для метода отдельно.

Хотя считывание из глобальной памяти идет с учетом исправлений, т.е. варпом, производительность значительно поднять не получилось. Сказывается

работа алгоритма с новыми приближениями, из-за чего частично не срабатывает идея оптимизации. График изображен на рисунке 3.9. Данная оптимизация увеличила производительность в 2 раза при размере в 10000 × 10000 элементов. При решении разреженных матриц менее 5000 × 5000 элементов данная оптимизация является неэффективной.

Рисунок 3.9 – Сравнение времени выполнения алгоритмов Гаусса-Зейделя и Гаусса-Зейделя с транспонированной матрицей

На рисунке 3.10 изображена загруженность GPU метода Гаусса-Зейделя. Загруженность GPU по сравнению со стандартным методом возросла в 2.65 раза и составила 62.4%.

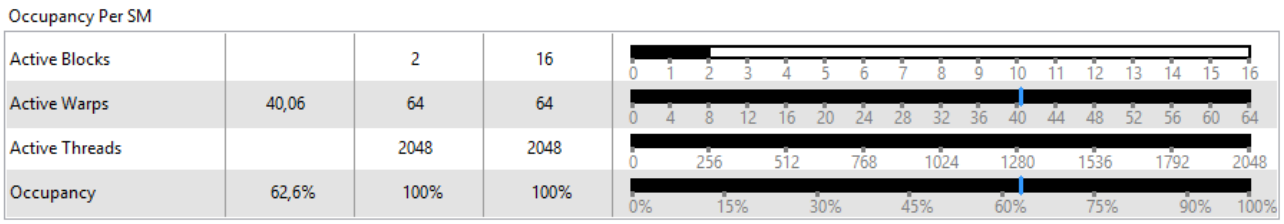

Рисунок 3.10 – Загруженность GPU метода Гаусса-Зейделя с транспонированной матрицей

Тестирование метода Якоби с транспонированной матрицей показало

существенное ускорение. График продемонстрирован на рисунке 3.11. В данной оптимизации идет считывание варпом из глобальной памяти. При 4000×4000 производительность увеличилась в 3 раза, при размере 10000×10000 в 3.4 раза.

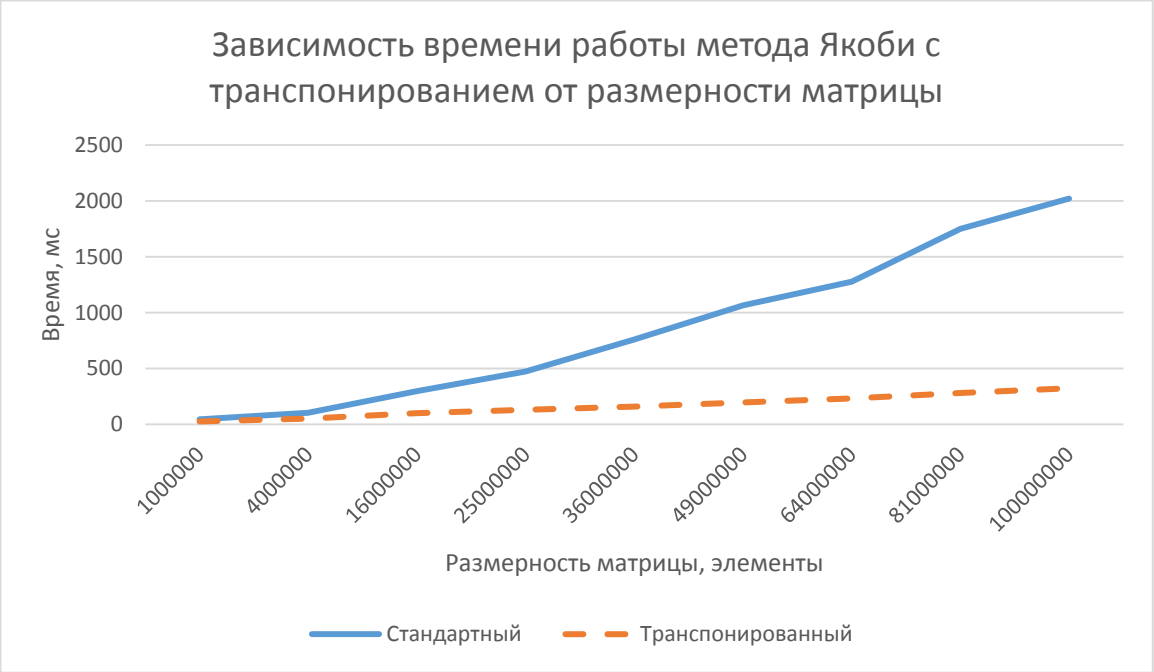

Рисунок 3.11 – Сравнение времени выполнения алгоритмов Якоби и Якоби с транспонированной матрицей

На рисунке 3.12 изображена загруженность GPU на 62.5%, как и у метода Гаусса-Зейделя с транспонированной матрицей, считается хорошим результатом [19].

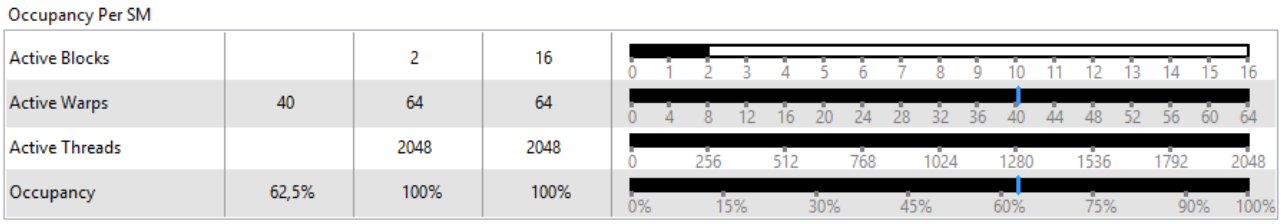

Рисунок 3.12 – Загруженность GPU метода Якоби с транспонированной матрицей

Исходя из полученных результатов, оба метода решения СЛАУ после добавления транспонированной матрицы увеличили свою эффективность в

плане загруженности GPU с 23.5% и 24.4% до 62.4% и 62.5% для методов Гаусса-Зейделя и метода Якоби соответственно. Сам метод транспонирования реализован с использованием разделяемой памяти, что помогло ему снизить скорость выполнения и загрузить GPU под 90%, что является почти максимальным результатом, который возможно получить на GPU архитектуры Kepler.

Метод Якоби после транспонирования матрицы показал лучшие результаты по сравнению с другими представленными реализациями, что показывает необходимость знания архитектуры GPU для написания алгоритмов решения СЛАУ.

Хотя результат загруженности GPU обоих методов решения СЛАУ составляет около 62%, дальнейшие исследование данных алгоритмов, возможно, позволит еще больше увеличить производительность.

Заключение

В ходе проведенной работы была изучена архитектура видео ускорителей от NVidia, были изучены и реализованы методы повышения производительности алгоритмов под графические процессоры NVidia Tesla K10, а также проведен сравнительный анализ методов решения систем линейных алгебраических уравнений.

Реализация итерационного алгоритма решения СЛАУ – метод Гаусса-Зейделя с учетом архитектуры графического ускорителя NVidia Tesla K10 показала прирост производительности, по сравнению со стандартной реализацией, в 2.2 раза при разреженной матрице 10000×10000. А загруженность процессора получилось поднять с 23.5% до 62.4%.

Следующий итерационный алгоритм, реализованный с учетом графического ускорителя NVidia Tesla K10 – метод Якоби. Данный метод так же, как и метод Гаусса-Зейделя показал увеличение производительности после оптимизации под графические процессоры. Прирост составил 3.4 раза при размере разреженной матрице в 10000×10000 элементов, по сравнению с неоптимизированной реализацией. Загруженность GPU возросла с 24.4% до 62.5%. Такой прирост производительности отлично демонстрирует необходимость знания архитектуры GPU, под который будут реализоваться алгоритмы.

Графические процессоры от NVIDIA и программно-аппаратная архитектура параллельных вычислений NVidia CUDA позволяют программистам использовать тысячи CUDA ядер для параллельных вычислений. Для простых вычислений глубокие знание архитектуры GPU не требуются. Однако только реализация программ под конкретную архитектуру конкретного GPU даст максимальную производительность.

Список использованной литературы

Учебники и учебные пособия

1. Сандерс Дж. Технология CUDA в примерах: введение в программирование графических процессоров:Пер. с англ. Слинкина А.А., научный редактор Боресков А.В./ Сандерс Дж., Кэндрот Э. – М.:ДМК Пресс, 2011. – 232с.
2. Боресков А.В. Параллельные вычисления на GPU. Архитектура и программная модель CUDA: Учеб.пособие / Боресков А.В. и др. Предисл.: Садовничий В.А. – М.: Издательство Московского университета, 2012. – 336с.
3. Малышкин В.Э. Параллельное программирование мультикомпьютеров: Учеб. Пособие. / Малышкин В.Э., Корнеев В.Д. – М.: Изд-во НГТУ, 2011. – 296с.
4. Варыгина М.П. Основы программирования в CUDA: Учеб. Пособие / Варыгина М.П. – М.: Краснояр. гос. пед. ун-т. В.П. Астафьева. – Красноярск, 2012. – 138с.

Периодические издания

5. Старовойтов С.В. Формат SKYLINE хранения и обработки разреженных матриц нерегулярной структуры для численного решения СЛАУ. / Старовойтов С.В. // Вестник РГУ им. И. Канта [Электронный ресурс]. – Электрон. журн. – 2008. – Вып. 10. – С.84-94. – Режим доступа: http://cyberleninka.ru/article/n/format-skyline-dlya-hraneniya-i-obrabotki-razrezhennyh-matrits-neregulyarnoy-struktury-dlya-chislennogo-resheniya-slau
6. Казённов А.М. Основы технологии CUDA. / Казённов А.М. // Компьютерные исследования и моделирование [Электронный ресурс]. – Электрон. журн. – 2010. – Т.2 №3. – С.295-308. – Режим доступа: http://crm.ics.org.ru
7. Фролов А.В. Еще один метод распараллеливания прогонки с использованием ассоциативности операций / Фролов А.В. // Суперкомпьютерные дни в России 2015 [Электронный ресурс]. – Электрон.

журн. – 2015. – Режим доступа: http://russianscdays.org/files/pdf/151.pdf

8. Акимова Е.Н. Параллельные алгоритмы решения СЛАУ с блочно-трехдиагональными матрицами на многопроцессорных вычислителях / Акимова Е.Н, Белоусов Д.В. // Вестник УГАТУ [Электронный ресурс]. – Электрон. журн. – 2011. – Т.15 №5 (45). – С.87-93. – Режим доступа: http://journal.ugatu.ac.ru/index.php/vestnik/article/view/411

9. Берчун Ю.В. Итерационный метод решения СЛАУ на основе механической аналогии / Берчун Ю.В., Бурков П.В., Чиркова А.С., Прокопьева С.М., Рабкин Д.Л., Лукьянов А.А.// Наука и Образование МГТУ им. Н.Э. Баумана [Электронный ресурс]. – Электрон. журн. – 2015. –№08. – С.14-31. – Режим доступа: http://technomag.bmstu.ru/doc/791351.html

10. Капорин И.Е. Массивно-параллельные предобусловленные итерационные методы решения стандартных задач линейной алгебры с разреженными матрицами большого размера [Электронный ресурс] / И.Е. Капорин, О.Ю. Милюкова, Ю.Г. Бартенев. – Электрон.дан. – Режим доступа: http://2013.nscf.ru/TesisAll/Section%205/04_2010_KaporinIE_S5.pdf

Электронные ресурсы

11. Сравнение простых и итерационных методов систем линейных алгебраических уравнений [Электронный ресурс] – Сенина А.С., Межаков А.В. [2016]. – Режим доступа: http://www.scienceforum.ru/2016/pdf/26516.pdf

12. Шарый С.П. Курс вычислительных методов [Электронный ресурс] / Шарый С.П. – Электрон.дан. – М.: Новосибирск: НГУ, 2016. – 529. – Режим доступа: http://www.ict.nsc.ru/matmod/files/textbooks/SharyNuMeth.pdf

13. NVIDIA Tesla K10 GPU ускоряет поиски залежей нефти и газа и обработку сигналов и изображений для военных [Электронный ресурс] – NVidia Corp., [2012]. – Режим доступа: http://www.nvidia.ru/object/nvidia-tesla-k10-gpu-accelerator-20120516-ru.html

14. GTC 2012: финальные подробности GK110 [Электронный ресурс] – Шиллинг А. [2012]. – Режим доступа:

http://www.hardwareluxx.ru/index.php/news/hardware/grafikkarten/22110-gtc-2012-gk110.html

15. NVIDIA представила Tesla K20 и K20X на основе GK110 [Электронный ресурс] – Шиллинг А. [2012]. – Режим доступа: http://www.hardwareluxx.ru/index.php/news/hardware/grafikkarten/23874-nvidia-tesla-k20-k20x-gk110.html

Литература на иностранном языке

16. CUDA Programming [Electronic resource]: [CUDA программирование]. – Romero. M., Urra. R. [2012]. – Mode of access: http://cuda.ce.rit.edu/cuda_overview/cuda_overview.htm

17. NVIDIA CUDA Compute Unified Device Architecture Programming Guide. Version 2.0 [Electronic resource]: [Единая вычислительная архитектура устройств. Руководство по программированию. Версия 2.0]. – Electronic data. – NVidia Corp., [2015]. – Mode of access: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

18. CUDA Toolkit 4.1 CURAND Guide [Electronic resource]: [Руководство по CUDA Toolkit 4.1 CURAND]. – Electronic data. – NVidia Corp., [2012]. – Mode of access: https://hpc.oit.uci.edu/nvidia-doc/sdk-cuda-doc/CUDALibraries/doc/CURAND_Library.pdf

19. OpenCL Best Practices Guide [Electronic resource]: [Руководство по OpenCL]. – Electronic data. – NVidia Corp., [2011]. – Mode of access: http://camlunity.ru/swap/Library/Conflux/OpenCL/NVIDIA_OpenCL_Best_Practices_Guide.pdf

20. Tuning CUDA Applications for Kepler [Electronic resource]: [Настройка CUDA приложений для архитектуры Kepler]. – Electronic data. – NVidia Corp., [2015]. – Mode of access: http://docs.nvidia.com/cuda/pdf/Kepler_Tuning_Guide.pdf

21. Dynamic Parallelism in CUDA 5.0 [Electronic resource]: [Динамический Параллелизм в CUDA 5.0]. – Electronic data. – NVidia Corp., [2012]. – Mode of access:

http://developer.download.nvidia.com/assets/cuda/docs/TechBrief_Dynamic_Parallelism_in_CUDA_v2.pdf

22. Understanding the CUDA Data Parallel Threading Model [Electronic resource]: [Общие сведения о параллельной модели потоков данных CUDA]. – Electronic data. – Technical News from The Portland Group, [2012]. – Mode of access: https://www.pgroup.com/lit/articles/insider/v2n1a5.htm

23. Fastest, Most Efficient HPC Architecture [Electronic resource]: [Самая быстрая и самая эффективная архитектура СуперЭВМ]. – NVidia Corp., [2012]. – Mode of access: http://www.nvidia.com/content/PDF/kepler/NV_DS_Tesla_KCompute_Arch_May_2012_LR.pdf

24. Fastest, Most Efficient HPC Architecture Ever build [Electronic resource]: [Самая быстрая и самая эффективная GPU архитектура когда-либо созданная]. – NVidia Corp., [2012]. – Mode of access: http://www.nvidia.com/content/tesla/pdf/NV_DS_TeslaK_Family_May_2012_LR.pdf

Приложение А

Блок-схемы вычислительных алгоритмов

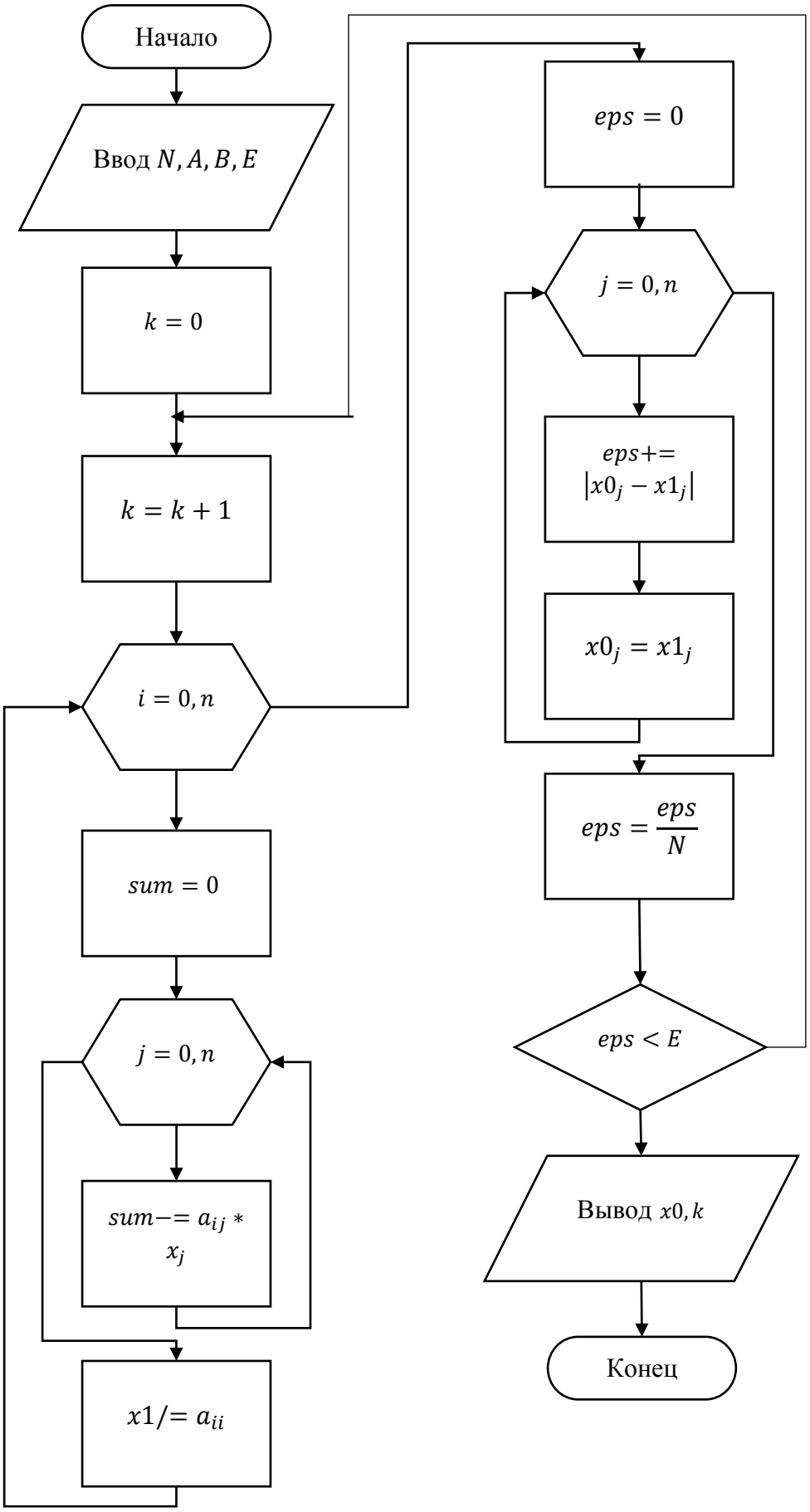

Рисунок А.1 – Блок-схема метода Якоби

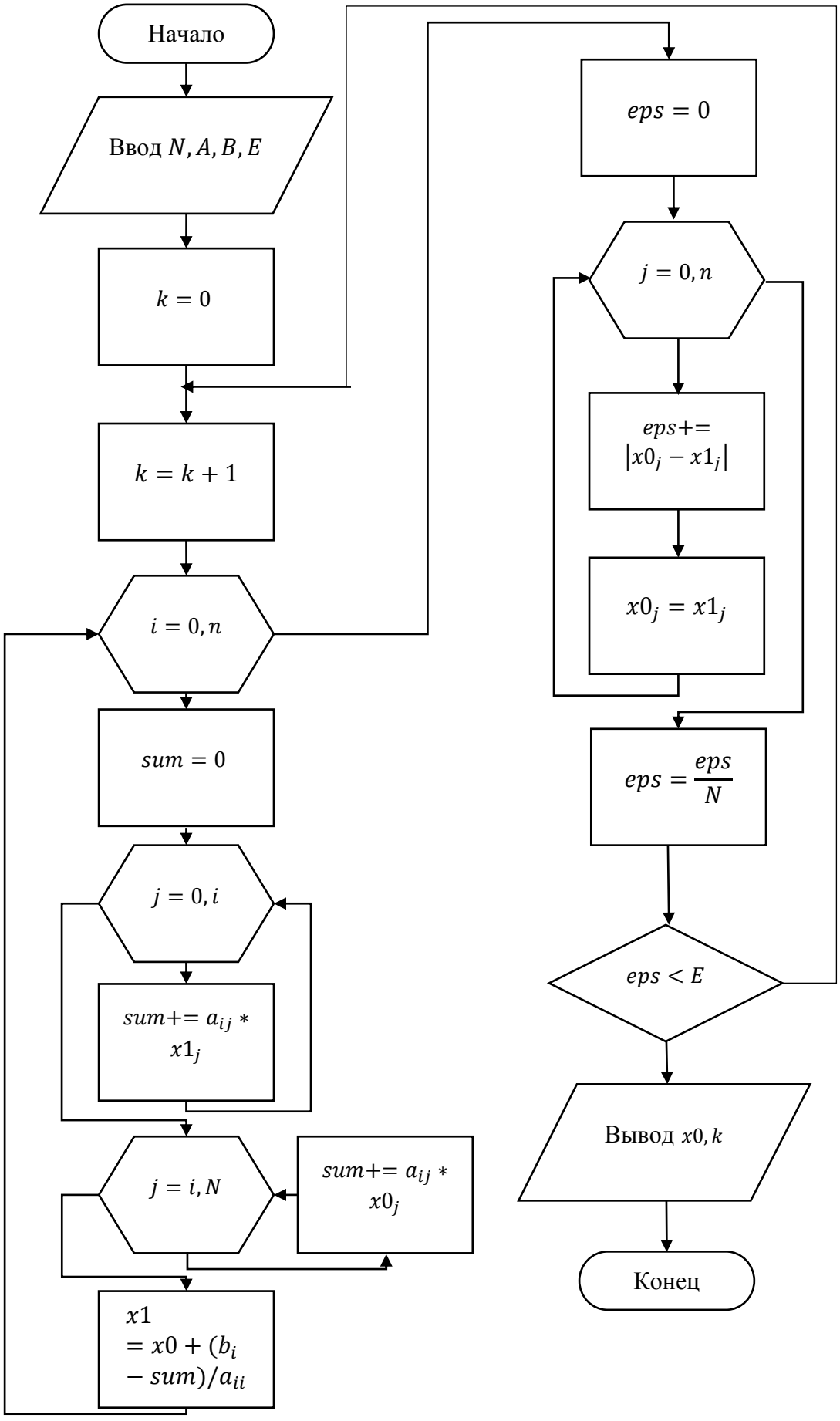

Рисунок А.2 – Блок схема метода Гаусса-Зейделя

Приложение Б

Листинг кода файла Jacobi.cu

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <device_functions.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <cstring>

#define Thread 1024

__global__ void KernelJacobi(float* deviceA, float* deviceF, float* deviceX0, float*
deviceX1, int N) {
	float temp;
	int i = blockIdx.x * blockDim.x + threadIdx.x;
	deviceX1[i] = deviceF[i];
	for (int j = 0; j < N; j++) {
		if (i != j)
			deviceX1[i] -= deviceA[j + i*N] * deviceX0[j];
		else temp = deviceA[j + i*N];
	}
	deviceX1[i] /= temp;
}

__global__ void EpsJacobi(float *deviceX0, float *deviceX1, float *delta, int N){
	int i = blockIdx.x * blockDim.x + threadIdx.x;
	delta[i] += fabs(deviceX0[i] - deviceX1[i]);
	deviceX0[i] = deviceX1[i];
}

int main() {
	srand(time(NULL));
	float *hostA, *hostX, *hostX0, *hostX1, *hostF, *hostDelta;
	float sum, eps;
	float EPS = 1.e-5;
	int N = 10000;
	float size = N*N;
	int count;
	int Block = (int)ceil((float)N / Thread);
	dim3 Blocks(Block);
	dim3 Threads(Thread);

	int Num_diag = 0.5f*(int)N*0.3f;

	float mem_sizeA = sizeof(float)*size;
	unsigned int mem_sizeX = sizeof(float)*(N);

	hostA = (float*)malloc(mem_sizeA);
	hostF = (float*)malloc(mem_sizeX);
	hostX = (float*)malloc(mem_sizeX);
	hostX0 = (float*)malloc(mem_sizeX);
	hostX1 = (float*)malloc(mem_sizeX);
	hostDelta = (float*)malloc(mem_sizeX);

	for (int i = 0; i < size; i++) {
		hostA[i] = 0.0f;
	}

	for (int i = 0; i < N; i++) {
		hostA[i + i*N] = rand() % 50 + 1.0f*N;
```

```
	}

	for (int k = 1; k < Num_diag + 1; k++) {
		for (int i = 0; i < N - k; i++) {
			hostA[i + k + i*N] = rand() % 5;
			hostA[i + (i + k)*N] = rand() % 5;
		}
	}

	for (int i = 0; i < N; i++) {
		hostX[i] = rand() % 50;
		hostX0[i] = 1.0f;
		hostDelta[i] = 0.0f;
	}

	for (int i = 0; i < N; i++) {
		sum = 0.0f;
		for (int j = 0; j < N; j++) sum += hostA[j + i*N] * hostX[j];
		hostF[i] = sum;
	}

	float *deviceA, *deviceX0, *deviceX1, *deviceF, *delta;
	for (int i = 0; i<N; i++) hostX1[i] = 1.0f;
	cudaMalloc((void**)&deviceA, mem_sizeA);
	cudaMalloc((void**)&deviceF, mem_sizeX);
	cudaMalloc((void**)&deviceX0, mem_sizeX);
	cudaMalloc((void**)&deviceX1, mem_sizeX);
	cudaMalloc((void**)&delta, mem_sizeX);
	cudaMemcpy(deviceA, hostA, mem_sizeA, cudaMemcpyHostToDevice);
	cudaMemcpy(deviceF, hostF, mem_sizeX, cudaMemcpyHostToDevice);
	cudaMemcpy(deviceX0, hostX0, mem_sizeX, cudaMemcpyHostToDevice);
	cudaMemcpy(deviceX1, hostX1, mem_sizeX, cudaMemcpyHostToDevice);
	count = 0; eps = 1.0f;
	while (eps > EPS)
	{
		count++;
		cudaMemcpy(delta, hostDelta, mem_sizeX, cudaMemcpyHostToDevice);
		KernelJacobi << < Blocks, Threads >> > (deviceA, deviceF, deviceX0, deviceX1,
N);
		EpsJacobi <<< Blocks, Threads >>> (deviceX0, deviceX1, delta, N);
		cudaMemcpy(hostDelta, delta, mem_sizeX, cudaMemcpyDeviceToHost);
		eps = 0.0f;
		for (int j = 0; j < N; j++) {
			eps += hostDelta[j]; hostDelta[j] = 0;
		}
		eps = eps / N;
	}
	cudaMemcpy(hostX1, deviceX1, mem_sizeX, cudaMemcpyDeviceToHost);
	cudaFree(deviceA);
	cudaFree(deviceF);
	cudaFree(deviceX0);
	cudaFree(deviceX1);
	free(hostA);
	free(hostF);
	free(hostX0);
	free(hostX1);
	free(hostX);
	free(hostDelta);
	return 0;
}
```

Приложение В

Листинг кода файла GaussSeidel.cu

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <device_functions.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <cstring>

#define Thread 1024

__global__ void KernelGaussSeidel(float* deviceA, float* deviceF, float* deviceX0, float*
deviceX1, int N) {
	float sum = 0.0f;
	for (int i = 0; i < N; i++) deviceX0[i] = deviceX1[i];
	int t = blockIdx.x * blockDim.x + threadIdx.x;
	for (int j = 0; j<t; j++) sum += deviceA[j + t*N] * deviceX1[j];
	for (int j = t + 1; j<N; j++) sum += deviceA[j + t*N] * deviceX0[j];
	deviceX1[t] = (deviceF[t] - sum) / deviceA[t + t*N];
}

__global__ void EpsGaussSeidel(float *deviceX0, float *deviceX1, float *delta, int N){
	int i = blockIdx.x * blockDim.x + threadIdx.x;
	delta[i] += fabs(deviceX0[i] - deviceX1[i]);
	deviceX0[i] = deviceX1[i];
}

int main() {
	srand(time(NULL));
	float *hostA, *hostX, *hostX0, *hostX1, *hostF, *hostDelta;
	float sum, eps;
	float EPS = 1.e-5;
	int N = 10000;
	float size = N*N;
	int count;
	int Block = (int)ceil((float)N / Thread);
	dim3 Blocks(Block);
	dim3 Threads(Thread);

	int Num_diag = 0.5f*(int)N*0.3f;

	float mem_sizeA = sizeof(float)*size;
	unsigned int mem_sizeX = sizeof(float)*(N);

	hostA = (float*)malloc(mem_sizeA);
	hostF = (float*)malloc(mem_sizeX);
	hostX = (float*)malloc(mem_sizeX);
	hostX0 = (float*)malloc(mem_sizeX);
	hostX1 = (float*)malloc(mem_sizeX);
	hostDelta = (float*)malloc(mem_sizeX);

	for (int i = 0; i < size; i++) {
		hostA[i] = 0.0f;
	}

	for (int i = 0; i < N; i++) {
		hostA[i + i*N] = rand() % 50 + 1.0f*N;
	}

	for (int k = 1; k < Num_diag + 1; k++) {
```

```
		for (int i = 0; i < N - k; i++) {
			hostA[i + k + i*N] = rand() % 5;
			hostA[i + (i + k)*N] = rand() % 5;
		}
	}

	for (int i = 0; i < N; i++) {
		hostX[i] = rand() % 50;
		hostX0[i] = 1.0f;
		hostDelta[i] = 0.0f;
	}

	for (int i = 0; i < N; i++) {
		sum = 0.0f;
		for (int j = 0; j < N; j++) sum += hostA[j + i*N] * hostX[j];
		hostF[i] = sum;
	}

	float *deviceA, *deviceX0, *deviceX1, *deviceF, *delta;
	for (int i = 0; i<N; i++) hostX1[i] = 1.0f;
	cudaMalloc((void**)&deviceA, mem_sizeA);
	cudaMalloc((void**)&deviceF, mem_sizeX);
	cudaMalloc((void**)&deviceX0, mem_sizeX);
	cudaMalloc((void**)&deviceX1, mem_sizeX);
	cudaMalloc((void**)&delta, mem_sizeX);
	cudaMemcpy(deviceA, hostA, mem_sizeA, cudaMemcpyHostToDevice);
	cudaMemcpy(deviceF, hostF, mem_sizeX, cudaMemcpyHostToDevice);
	cudaMemcpy(deviceX0, hostX0, mem_sizeX, cudaMemcpyHostToDevice);
	cudaMemcpy(deviceX1, hostX1, mem_sizeX, cudaMemcpyHostToDevice);
	count = 0; eps = 1.0f;
	while (eps > EPS)
	{
		count++;
		cudaMemcpy(delta, hostDelta, mem_sizeX, cudaMemcpyHostToDevice);
		KernelGaussSeidel << < Blocks, Threads >> > (deviceA, deviceF, deviceX0,
deviceX1, N);
		EpsGaussSeidel << < Blocks, Threads >> > (deviceX0, deviceX1, delta, N);
		cudaMemcpy(hostDelta, delta, mem_sizeX, cudaMemcpyDeviceToHost);
		eps = 0.0f;
		for (int j = 0; j < N; j++) {
			eps += hostDelta[j]; hostDelta[j] = 0;
		}
		eps = eps / N;
	}
	cudaMemcpy(hostX1, deviceX1, mem_sizeX, cudaMemcpyDeviceToHost);
	cudaFree(deviceA);
	cudaFree(deviceF);
	cudaFree(deviceX0);
	cudaFree(deviceX1);
	free(hostA);
	free(hostF);
	free(hostX0);
	free(hostX1);
	free(hostX);
	free(hostDelta);
	return 0;
}
```

Приложение Г

Листинг кода файла JacobiTransposition.cu

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <device_functions.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <cstring>
#include <cuda_profiler_api.h>

#define BLOCK_SIZE 32
#define Thread 1024

__global__ void KernelJacobi(float* deviceA, float* deviceF, float* deviceX0, float*
deviceX1, int N) {
	float temp;
	int i = blockIdx.x * blockDim.x + threadIdx.x;
	deviceX1[i] = deviceF[i];
	for (int j = 0; j < N; j++) {
		if (i != j)
			deviceX1[i] -= deviceA[i + j*N] * deviceX0[j];
		else temp = deviceA[i + j*N];
	}
	deviceX1[i] /= temp;
}

__global__ void EpsJacobi(float *deviceX0, float *deviceX1, float *delta, int N){
	int i = blockIdx.x * blockDim.x + threadIdx.x;
	delta[i] += fabs(deviceX0[i] - deviceX1[i]);
	deviceX0[i] = deviceX1[i];
}

__global__ void Transposition(float* deviceA, float* deviceT, int N) {
	__shared__ float sh[BLOCK_SIZE][BLOCK_SIZE + 1];
	int x = blockIdx.x * blockDim.x;
	int y = blockIdx.y * blockDim.y;
	int i = x + threadIdx.x;
	int j = y + threadIdx.y;
	sh[threadIdx.y][threadIdx.x] = deviceA[(y + threadIdx.y) * N + (x + threadIdx.x)];
	__syncthreads();
	deviceT[(x + threadIdx.y)*N + (y + threadIdx.x)] = sh[threadIdx.x][threadIdx.y];
}

int main() {
	float timerValueCPU;
	clock_t start, stop;
	srand(time(NULL));
	float *hostA, *hostX, *hostX0, *hostX1, *hostF, *hostDelta;
	float sum, eps;
	float EPS = 1.e-5;
	int N = 10000;
	float size = N*N;
	int count;
	int Block = (int)ceil((float)N / Thread);
	dim3 Blocks(Block);
	dim3 Threads(Thread);
	dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
	dim3 dimGrid((int)ceil((float)N / BLOCK_SIZE), (int)ceil((float)N / BLOCK_SIZE));

	int Num_diag = 0.5f*(int)N*0.3f;
```

```
float mem_sizeA = sizeof(float)*size;
unsigned int mem_sizeX = sizeof(float)*(N);

hostA = (float*)malloc(mem_sizeA);
hostF = (float*)malloc(mem_sizeX);
hostX = (float*)malloc(mem_sizeX);
hostX0 = (float*)malloc(mem_sizeX);
hostX1 = (float*)malloc(mem_sizeX);
hostDelta = (float*)malloc(mem_sizeX);

for (int i = 0; i < size; i++) {
	hostA[i] = 0.0f;
}

for (int i = 0; i < N; i++) {
	hostA[i + i*N] = rand() % 50 + 1.0f*N;
}

for (int k = 1; k < Num_diag + 1; k++) {
	for (int i = 0; i < N - k; i++) {
		hostA[i + k + i*N] = rand() % 5;
		hostA[i + (i + k)*N] = rand() % 5;
	}
}

for (int i = 0; i < N; i++) {
	hostX[i] = rand() % 50;
	hostX0[i] = 1.0f;
	hostDelta[i] = 0.0f;
}

for (int i = 0; i < N; i++) {
	sum = 0.0f;
	for (int j = 0; j < N; j++) sum += hostA[j + i*N] * hostX[j];
	hostF[i] = sum;
}

cudaEvent_t GPU_start, GPU_stop;
cudaEventCreate(&GPU_start);
cudaEventCreate(&GPU_stop);

float *deviceA, *deviceX0, *deviceX1, *deviceF, *delta, *deviceT;
for (int i = 0; i<N; i++) hostX1[i] = 1.0f;
cudaMalloc((void**)&deviceA, mem_sizeA);
cudaMalloc((void**)&deviceF, mem_sizeX);
cudaMalloc((void**)&deviceX0, mem_sizeX);
cudaMalloc((void**)&deviceX1, mem_sizeX);
cudaMalloc((void**)&deviceT, mem_sizeA);
cudaMalloc((void**)&delta, mem_sizeX);

cudaEventRecord(GPU_start, 0);
cudaProfilerStart();

cudaMemcpy(deviceA, hostA, mem_sizeA, cudaMemcpyHostToDevice);
cudaMemcpy(deviceF, hostF, mem_sizeX, cudaMemcpyHostToDevice);
cudaMemcpy(deviceX0, hostX0, mem_sizeX, cudaMemcpyHostToDevice);
cudaMemcpy(deviceX1, hostX1, mem_sizeX, cudaMemcpyHostToDevice);
count = 0; eps = 1.0f;

Transposition <<< dimGrid, dimBlock >>> (deviceA, deviceT, N);

while (eps > EPS)
{
	count++;
```

```
		cudaMemcpy(delta, hostDelta, mem_sizeX, cudaMemcpyHostToDevice);
		KernelJacobi <<< Blocks, Threads >>> (deviceA, deviceF, deviceX0, deviceX1,
N);
		EpsJacobi <<< Blocks, Threads >>> (deviceX0, deviceX1, delta, N);
		cudaMemcpy(hostDelta, delta, mem_sizeX, cudaMemcpyDeviceToHost);
		eps = 0.0f;
		for (int j = 0; j < N; j++) {
			eps += hostDelta[j]; hostDelta[j] = 0;
		}
		eps = eps / N;
	}
	cudaMemcpy(hostX1, deviceX1, mem_sizeX, cudaMemcpyDeviceToHost);

	cudaEventRecord(GPU_stop, 0);
	cudaEventSynchronize(GPU_stop);
	float timeGPU = 0;
	cudaEventElapsedTime(&timeGPU, GPU_start, GPU_stop);
	printf("\n GPU calculation time %f msec\n", timeGPU);

	cudaFree(deviceA);
	cudaFree(deviceF);
	cudaFree(deviceX0);
	cudaFree(deviceX1);
	cudaFree(deviceT);
	free(hostA);
	free(hostF);
	free(hostX0);
	free(hostX1);
	free(hostX);
	free(hostDelta);
	return 0;
}
```

Приложение Д

Листинг кода файла GaussSeidelTransposition.cu

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <device_functions.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <cstring>
#include <cuda_profiler_api.h>

#define BLOCK_SIZE 32
#define Thread 1024

__global__ void KernelGaussSeidel(float* deviceA, float* deviceF, float* deviceX0, float*
deviceX1, int N) {
	float sum = 0.0f;
	for (int i = 0; i < N; i++) deviceX0[i] = deviceX1[i];
	int t = blockIdx.x * blockDim.x + threadIdx.x;
	for (int j = 0; j<t; j++) sum += deviceA[t + j*N] * deviceX1[j];
	for (int j = t + 1; j<N; j++) sum += deviceA[t + j*N] * deviceX0[j];
	deviceX1[t] = (deviceF[t] - sum) / deviceA[t + t*N];
}

__global__ void EpsGaussSeidel(float *deviceX0, float *deviceX1, float *delta, int N){
	int i = blockIdx.x * blockDim.x + threadIdx.x;
	delta[i] += fabs(deviceX0[i] - deviceX1[i]);
	deviceX0[i] = deviceX1[i];
}

__global__ void Transposition(float* deviceA, float* deviceT, int N) {
	__shared__ float sh[BLOCK_SIZE][BLOCK_SIZE + 1];
	int x = blockIdx.x * blockDim.x;
	int y = blockIdx.y * blockDim.y;
	int i = x + threadIdx.x;
	int j = y + threadIdx.y;
	sh[threadIdx.y][threadIdx.x] = deviceA[(y + threadIdx.y) * N + (x + threadIdx.x)];
	__syncthreads();
	deviceT[(x + threadIdx.y)*N + (y + threadIdx.x)] = sh[threadIdx.x][threadIdx.y];
}

int main() {
	float timerValueCPU;
	clock_t start, stop;
	srand(time(NULL));
	float *hostA, *hostX, *hostX0, *hostX1, *hostF, *hostDelta;
	float sum, eps;
	float EPS = 1.e-5;
	int N = 10240;
	float size = N*N;
	int count;
	int Block = (int)ceil((float)N / Thread);
	dim3 Blocks(Block);
	dim3 Threads(Thread);
	dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
	dim3 dimGrid((int)ceil((float)N / BLOCK_SIZE), (int)ceil((float)N / BLOCK_SIZE));

	int Num_diag = 0.5f*(int)N*0.3f;

	float mem_sizeA = sizeof(float)*size;
	unsigned int mem_sizeX = sizeof(float)*(N);
```

```
	hostA = (float*)malloc(mem_sizeA);
	hostF = (float*)malloc(mem_sizeX);
	hostX = (float*)malloc(mem_sizeX);
	hostX0 = (float*)malloc(mem_sizeX);
	hostX1 = (float*)malloc(mem_sizeX);
	hostDelta = (float*)malloc(mem_sizeX);

	for (int i = 0; i < size; i++) {
		hostA[i] = 0.0f;
	}

	for (int i = 0; i < N; i++) {
		hostA[i + i*N] = rand() % 50 + 1.0f*N;
	}

	for (int k = 1; k < Num_diag + 1; k++) {
		for (int i = 0; i < N - k; i++) {
			hostA[i + k + i*N] = rand() % 5;
			hostA[i + (i + k)*N] = rand() % 5;
		}
	}

	for (int i = 0; i < N; i++) {
		hostX[i] = rand() % 50;
		hostX0[i] = 1.0f;
		hostDelta[i] = 0.0f;
	}

	for (int i = 0; i < N; i++) {
		sum = 0.0f;
		for (int j = 0; j < N; j++) sum += hostA[j + i*N] * hostX[j];
		hostF[i] = sum;
	}

	cudaEvent_t GPU_start, GPU_stop;
	cudaEventCreate(&GPU_start);
	cudaEventCreate(&GPU_stop);

	float *deviceA, *deviceX0, *deviceX1, *deviceF, *delta, *deviceT;
	for (int i = 0; i<N; i++) hostX1[i] = 1.0f;
	cudaMalloc((void**)&deviceA, mem_sizeA);
	cudaMalloc((void**)&deviceF, mem_sizeX);
	cudaMalloc((void**)&deviceX0, mem_sizeX);
	cudaMalloc((void**)&deviceX1, mem_sizeX);
	cudaMalloc((void**)&deviceT, mem_sizeA);
	cudaMalloc((void**)&delta, mem_sizeX);

	cudaEventRecord(GPU_start, 0);
	cudaProfilerStart();

	cudaMemcpy(deviceA, hostA, mem_sizeA, cudaMemcpyHostToDevice);
	cudaMemcpy(deviceF, hostF, mem_sizeX, cudaMemcpyHostToDevice);
	cudaMemcpy(deviceX0, hostX0, mem_sizeX, cudaMemcpyHostToDevice);
	cudaMemcpy(deviceX1, hostX1, mem_sizeX, cudaMemcpyHostToDevice);
	count = 0; eps = 1.0f;

	Transposition <<< dimGrid, dimBlock >>> (deviceA, deviceT, N);

	while (eps > EPS)
	{
		count++;
		cudaMemcpy(delta, hostDelta, mem_sizeX, cudaMemcpyHostToDevice);
		KernelGaussSeidel << < Blocks, Threads >> > (deviceA, deviceF, deviceX0,
deviceX1, N);
```

```
			EpsGaussSeidel << < Blocks, Threads >> > (deviceX0, deviceX1, delta, N);
			cudaMemcpy(hostDelta, delta, mem_sizeX, cudaMemcpyDeviceToHost);
			eps = 0.0f;
			for (int j = 0; j < N; j++) {
				eps += hostDelta[j]; hostDelta[j] = 0;
			}
			eps = eps / N;
		}
		cudaMemcpy(hostX1, deviceX1, mem_sizeX, cudaMemcpyDeviceToHost);

		cudaEventRecord(GPU_stop, 0);
		cudaEventSynchronize(GPU_stop);
		float timeGPU = 0;
		cudaEventElapsedTime(&timeGPU, GPU_start, GPU_stop);
		printf("\n GPU calculation time %f msec\n", timeGPU);

		cudaFree(deviceA);
		cudaFree(deviceF);
		cudaFree(deviceX0);
		cudaFree(deviceX1);
		cudaFree(deviceT);
		free(hostA);
		free(hostF);
		free(hostX0);
		free(hostX1);
		free(hostX);
		free(hostDelta);
		return 0;
}
```