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»**

02.03.03 МАТЕМАТИЧЕСКОЕ ОБЕСПЕЧЕНИЕ
И АДМИНИСТРИРОВАНИЕ ИНФОРМАЦИОННЫХ СИСТЕМ
ТЕХНОЛОГИЯ ПРОГРАММИРОВАНИЯ

БАКАЛАВРСКАЯ РАБОТА

на тему: Реализация параллельных вычислений на основе программно-аппаратного стека CUDA

Студент М.Н. Тюряев ____________________

Руководитель Н.И. Лиманова ____________________

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 Тюряев Максим Николаевич

1. Тема: Реализация параллельных вычислений на основе программно-аппаратного стека CUDA

2. Срок сдачи студентом законченной выпускной квалификационной работы 19.06.2016.

3. Исходные данные к выпускной квалификационной работе: библиотеки программно-аппаратного стека CUDA, набор уравнений для проверки эффективности алгоритма, объектно-ориентированный язык Java.

4. Содержание выпускной квалификационной работы (перечень подлежащих разработке вопросов, разделов):

1 Анализ платформы CUDA для решения поставленной задачи

1.1 Сравнение характеристик центрального и графического процессоров в параллельных расчётах

1.2 Параллельные вычисления на GPU

1.3 Методы параллельного программирования на CUDA

1.4 Постановка краевой задачи

2 Методы решения поставленной задачи

2.1 Уравнение теплопроводности

2.2 Метод конечных разностей

2.3 Метод Гаусса для решения системы линейных уравнений

3 Разработка программной системы

3.1 Разработка проекта по реализации параллельных вычислений на основе программно-аппаратного стека CUDA

3.2 Разработка и архитектуры системы и программная реализация системы

3.3 Результаты вычислительных экспериментов

5. Ориентировочный перечень графического и иллюстративного материала: графики, демонстрация работы алгоритма на реальном наборе данных.

6. Дата выдачи задания « 11 » января 2016 г.

Руководитель выпускной квалификационной работы	______________________	Н.И. Лиманова
Задание принял к исполнению	______________________	М.Н. Тюряев

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 Тюряев Максим Николаевич
по теме: Реализация параллельных вычислений на основе программно-аппаратного стека CUDA

Наименование раздела работы	Плановый срок выполнения раздела	Фактический срок выполнения раздела	Отметка о выполнении	Подпись руководителя
Поиск и исследование литературы по теме выпускной квалификационной работы работы	19.02.2016	19.02.2016	выполнено	
Написание первой главы	11.02.2016	11.02.2016	выполнено	
Написание второй главы	25.03.2016	25.03.2016	выполнено	
Реализация параллельных вычислений на основе программно-аппаратного стека CUDA	9.04.2016	9.04.2016	выполнено	
Написание третьей главы	18.04.2016	18.04.2016	выполнено	
Представление выпускной квалификационной работы на кафедру	16.05.2016	16.05.2016	выполнено	
Предварительная защита	25.05.2016	25.05.2016	выполнено	
Сдача пояснительной записки ВКР	19.06.2016	19.06.2016	выполнено	

Руководитель выпускной
квалификационной работы ___________________ Н.И. Лиманова

Задание принял к исполнению ______________________ М.Н. Тюряев

Аннотация

Тема: Реализация параллельных вычислений на основе программно-аппаратного стека CUDA

Актуальность темы данной работы заключается в увеличении быстродействие программы за счёт использования технологии CUDA.

Целью данной бакалаврской работы является реализация параллельных вычислений на основе программно-аппаратного стека CUDA.

Для достижения поставленной цели в ходе работы были поставлены и решены следующие задачи:

– проанализировать программно-аппаратный стек CUDA;

– проанализировать основные методы параллельных вычислений на программно-аппаратном стеке CUDA;

– разработать программную систему, реализующую параллельные вычисления на программно-аппаратном стеке CUDA;

– провести вычислительные эксперименты и проанализировать полученные результаты.

Работа состоит из введения, трех глав, заключения, списка использованной литературы и приложений.

В работе использованы современные системы и технологии: технология параллельных вычислений на основе программно-аппаратного стека CUDA, Java.

Первая глава посвящена анализу предметной области, обзору и обоснованию выбора средств для разработки.

Вторая глава посвящена методом решения поставленной задачи.

Третья глава посвящена реализации и вычислительным экспериментам.

В заключении сформулированы основные выводы, которые были сделаны в процессе написания бакалаврской работы, описаны результаты практической реализации проекта.

Работа состоит из введения, трёх разделов, включающих 19 рисунков, 2

таблицы и 65 формул, заключения, списка литературы из 20 источников и 6 приложений. Общий объем работы – 45 страниц (без приложений).

Оглавление

Введение ... 3
Глава 1 Анализ платформы CUDA для решения поставленной задачи ... 5
1.1 Сравнение характеристик центрального и графического процессоров в параллельных расчётах ... 5
1.2 Параллельные вычисления на GPU ... 7
1.2.1 GPU-ускорители и принцип их работы ... 8
1.2.2 Разновидности памяти ... 10
1.3 Методы параллельного программирования на CUDA ... 12
1.4 Расширения языка C для работы с CUDA ... 16
1.5. Постановка краевой задачи ... 18
Глава 2 Методы решения поставленной задачи ... 20
2.1 Уравнение теплопроводности ... 20
2.2 Метод конечных разностей ... 23
2.3 Метод Гаусса для решения системы линейных уравнений ... 26
Глава 3 Разработка программной системы ... 30
3.1 Выбор языка программирования и среды разработки ... 30
3.2 Разработка и архитектуры системы и программная реализация системы. 31
3.2.1 Общее описание ... 31
3.2.2 Модель краевой задачи ... 32
3.2.3 Реализация конечно-разностного метода ... 32
3.2.4 Реализация метода Гаусса для решения СЛАУ ... 35
3.3 Результаты вычислительных экспериментов ... 41
Заключение ... 43
Список используемой литературы ... 44
Приложение А Код класса CPUMain ... 46
Приложение Б Код класса CudaInformationMatrixTask ... 47
Приложение В Код класса CudaMatrixStringTask ... 48
Приложение Г Код класса GAUSCounter ... 49
Приложение Д Код класса GPUMain ... 51
Приложение Е Код класса ThreadStep ... 57

Введение

Превращение персональных компьютеров в маленькие специализированные суперкомпьютеры уже давно освоено человечеством. Одним из ответвлений данной идеи являлись транспьютеры. При всей их результативности и производительности, они морально устарели из-за чего фокус внимания вновь сместился в сторону стандартизированных процессоров и домашних компьютерных систем. Хотя, стоит отметить, что похожие устройства есть и сейчас — это всякие специализированные ускорители. Но, в итоге, из-за узкой сферы их использования заметного распространения подобные ускорители не получили.

В последнее время, благодаря развитию игростроя, эстафета параллельных вычислений перешла к массовому рынку, так или иначе связанному с трёхмерными играми. Уникальные устройства с многоядерными процессорами для параллельных векторных вычислений, используемых в 3D-графике, получают высокую пиковую производительность, которая универсальным процессорам не под силу.

Следующей ласточкой в гонке производительности стала возможность задействовать графические процессоры в вычислениях. Благодаря возможности распараллеливания стало возможным применение видеокарт не по их прямому назначению. Образцом этой технологии является программно-аппаратная архитектура CUDA (от англ. Compute Unified Device Architecture), которую разработала компания Nvidia. CUDA реализует аппаратный параллелизм, основываясь на принципах вычислений SIMD (от англ. Single Instruction Multiple Data), позволяющих применять одинаковые команды одновременно к большому количеству данных.

Целью представленной работы является реализация параллельных вычислений на основе программно-аппаратного стека CUDA. Для ее достижения необходимо решить следующие задачи:

– анализ программно-аппаратного стека CUDA;

– анализ основных методов параллельных вычислений на программно-аппаратном стеке CUDA;

– разработка программной системы, реализующей параллельные вычисления на программно-аппаратном стеке CUDA;

– проведение вычислительных экспериментов и анализ полученных результатов.

Глава 1 Анализ платформы CUDA для решения поставленной задачи

1.1 Сравнение характеристик центрального и графического процессоров в параллельных расчётах

В связи с ограничением роста частот в стандартных процессорах и их высоким электропотреблением, разработчики вынуждены искать программные или технические альтернативы. Одной из них стало увеличение числа ядер. В результате увеличение производительности, зачастую, происходит за счет увеличения числа ядер на одной основе. Их основная задача – поддержка приложений на основе множественного потока команд и данных. Ядра работают независимо друг от друга, выполняя различные задания для разных процессов.

Специальные векторные возможности ради четырехкомпонентных и бинарных векторов возникли в многоцелевых процессорах из-за возросших запросов графических приложений, в первую очередь. Непосредственно по этой причине для поставленных задач использование GPU дешевле, так как они первоначально выполнены специально для них.

К примеру, в видеочипах NVIDIA основной блок — это мультипроцессор с восемью - десятью ядрами и сотнями ALU в целом, несколькими тысячами регистров и незначительным количеством делимой совокупной памяти. Ключевым составляющим видеокарты является быстрая глобальная память с допуском к ней критической массы мультипроцессоров, включающих в себя локальную память и специализированную память для констант. Основополагающие отличия архитектур графических процессоров и центральных.

1. Ядра CPU спроектированы для выполнения инструкций идущих последовательно и в один поток с максимальной производительностью.

2. Ядра GPU спроектированы для скорейшего выполнения большого числа одновременных потоков-инструкций.

3. Универсальные процессоры оптимизированы в целях достижения высочайшей производительности единичного потока директив, которые обрабатывают числа с плавающей точкой и целочисленные. В данном случае осуществляется случайный доступ к памяти.

4. Выполнение как можно большего количества инструкций одновременно – основная цель для большей части разработчиков CPU. Для этого, начиная с процессоров Intel Pentium, возникло суперскалярное выполнение, обеспечивающее выполнение пары инструкций за такт, а Pentium Pro отличился внеочередным выполнением инструкций. Однако, у параллельного выполнения последовательного потока инструкций есть определённые базовые ограничения и ростом количества исполнительных конструкций кратного увеличения скорости никак не достичь.

5. Видеочипы изначально имеют элементарную и распараллеленную деятельность, на входе принимая группу полигонов, выполняя все без исключения требуемые процедуры, и выводя на выход пиксели.

6. Отличие CPU и GPU находится в принципах доступа к памяти. В GPU всё связанно и предсказуемо — когда из памяти читается пиксель текстуры, то через некоторое время придёт время и для соседних пиксель. Да и при записи то же — пиксель заносится во фреймбуфер, и через несколько тактов будет записываться расположенный рядом с ним.

7. Работа с памятью у GPU и CPU также различна. На видеокартах применяется память, которая намного быстрее памяти на CPU, что даёт видеочипам намного большую пропускную способность памяти, что тоже очень важно для параллельных расчётов, которые управляют огромными потоками данных.

8. Универсальным процессорам приходится платить за свою универсальность. Причиной чего их огромное количество транзисторов и площадь типа тратятся на буфер команд, аппаратное предсказание ветвления и крайне высокие объемы кэш – памяти. Все эти аппаратные блоки нужны для ускорения исполнения немногочисленных потоков команд. Видеочипы тратят

транзисторы на массивы исполнительных блоков, управляющие потоками блоки, разделяемую память небольшого объёма и контроллеры памяти на несколько каналов.

9. Есть большое количество различий также и в поддержке многопоточности. CPU выполняет 1-2 потока вычислений на одно процессорное ядро, а видеочипы имеют возможность поддерживать до 1024 потоков на каждый мультипроцессор, которых в чипе несколько штук. А для переключения с одного потока на другой, CPU затрачивает сотни тактов, в то время как GPU переключает несколько потоков за один такт.

10. Помимо этого, центральные процессоры используют SIMD блоки для векторных вычислений, а видеочипы применяют SIMT (несколько потоков с одной инструкцией) для скалярной обработки потоков. SIMT не требует того, чтобы разработчик преобразовывал данные в векторы, и разрешает произвольные ветвления в потоках.

Подводя итог можно сказать, что в отличие от новейших универсальных CPU, видеочипы изначально задуманы для параллельных вычислений с большим количеством арифметических операций. Намного большее число транзисторов GPU работает по прямому назначению — обработке массивов данных, и не управляет исполнением малочисленных последовательных вычислительных потоков [1,18].

1.2 Параллельные вычисления на GPU

В связи с информатизацией общества, производительные вычисления тесно вошли в жизнь людей. Строительство, инженерия, химия, биотехнологии, везде, где есть возможность автоматизации и необходимость в вычислении множества значений и констант. Но необходимо не допускать застоя, поэтому соответствующие технологии и инструменты бесконечно изменяются и преобразуются в сторону наибольшей эффективности. Современное оборудование позволяет создать вещи, подобные симуляции кроны деревьев, погодных эффектов и даже самой Солнечной системы. Нынешнее поколение

суперкомпьютеров обладает мощностью, способной обработать около 1016 операций типа float. Безусловно, подобной мощи с лихвой хватает для решения ежедневных проблем. Естественно, человек всегда будет стремиться к большему, но современные материалы и технологии не позволяют достичь абсолюта, поэтому нам приходиться довольствоваться тем, что есть [2].

Из-за того, что графические процессоры изначально были спроектированы для поддержки большего числа ядер, чем стандартные центральные, теоретически и практически они обладают большим пулом возможностей. Достаточно организовать им нужное программное сопровождение для достижения требуемых и удовлетворительных результатов. Ускорение расчета может достигать 400% от номинала. И это на стандартном коде, а не заточенном специально под графические процессоры. Рассмотрим здесь некоторые примеры ускорений при использовании синтетического кода на GPU против SSE-векторизованного кода на CPU:

– флуоресцентная микроскопия: в 12 раз по сравнению с расчетом на CPU (12x);
– молекулярная динамика (non-bonded force calc): 8-16x;
– электростатика (прямое и многоуровневое суммирование Кулона): 40-120x и 7x.

1.2.1 GPU-ускорители и принцип их работы

Технология CUDA – мощное решение для повседневных задач. Технологиями языка Си была достигнута реализация организации доступа к большинству запрашиваемых инструкций графического ускорителя с целью управления его памятью. В данный момент поддерживаются практически все мультиядерные графические процессоры фирмы Nvidia.

Графический процессор основан на так называемой архитектуре SIMT. Технология CUDA основана на использовании специализированных функций, которые доступны ядрам. Они выполняются параллельно на графическом процессоре в качестве различных потоков. Таким образом мы можем

установить связь ядро – аналоговая функция. Потоки разделены по ядрам и выполняются с использованием своих индивидуальных инструкций и локальной памяти.

Группировка потоков одинакового размера и последующее их распределение по различным мультипроцессорам позволяет нам извлечь максимальную эффективность от каждого из ядер. Ограничение числа потоков в цепи можно уточнить при помощи функций API. Основой является качество взаимодействия потоков внутри общего блока данных, принадлежащих отдельному ядру. Оно позволяет нам эффективно взаимодействовать с памятью, добиваясь синхронизации. Не стоит так же забывать про глобальную память и атомарные операции. Их мы тоже можем задействовать в нашем графическом процессоре [8].

На аппаратном уровне потоки блока собираются в варпы по 32 элемента, в которых все потоки параллельно исполняют одни и те же инструкции. Важным моментом этой процедуры в том, что потоки выполняют одинаковые команды, но каждая со своим набором данных. Следовательно если внутри варпа происходит действие, то все нити варпа исполняют все появляющиеся при этом ветви. По этой причине операции ветвления могут негативно сказываться на производительности – отличные пути не могут выполняться параллельно. В свою очередь, блоки потоков объединяются в решетки блоков потоков. Также нужно отметить, что взаимодействие потоков из разных блоков во время работы ядра усложнено: отсутствуют явно выделенные инструкции синхронизации, взаимодействие допустимо через глобальную память и использование атомарных функций. Пример иерархии потоков приведен на рисунке 1.1 [18].

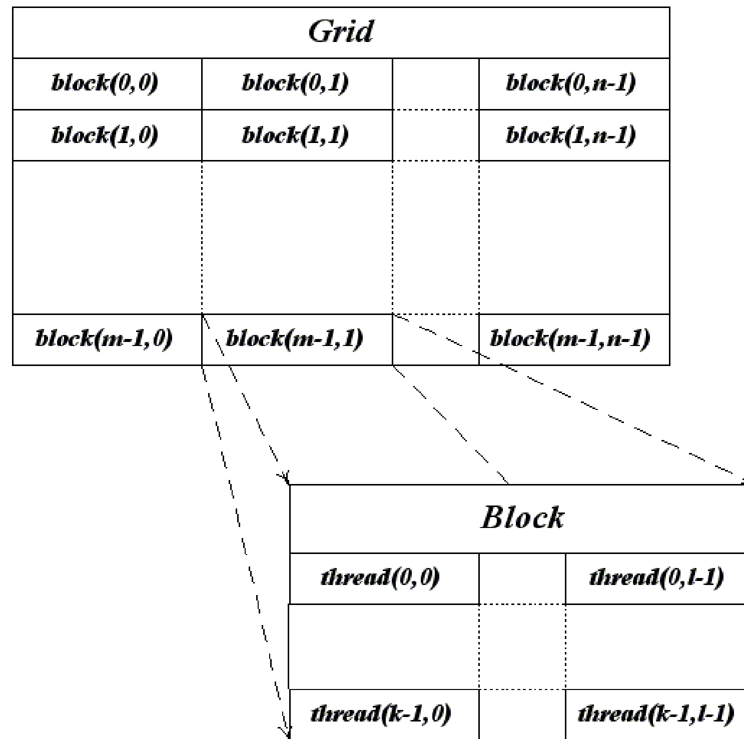

Рисунок 1.1 – Иерархия потоков CUDA

Все потоки внутри каждого блока имеют свои координаты, которые доступны через встроенную переменную threadIdx. В тоже время, координаты блока потоков внутри решетки определяются встроенной переменной blockIdx.

1.2.2 Разновидности памяти

Нам необходимо не забывать про наличие различных типов памяти и их взаимодействие с системой в целом. Это ключевые требования, если мы хотим обеспечить быстродействие для нашей системы. В традиционных центральных процессорах изначально заложено большое количество кэш-памяти.

В CUDA для GPU существует несколько различных типов памяти, доступных нитям, сильно различающихся между собой. Это показано в таблице 1.1.

Таблица 1.1 – Типы памяти в CUDA

Тип памяти	Доступ	Уровень выделения	Скорость работы
регистры (registers)	R/W	per-thread	высокая (onchip)
Local	R/W	per-thread	низкая (DRAM)
Shared	R/W	per-block	высокая (on-chip)
Global	R/W	per-grid	низкая(DRAM)
Constant	R/O	per-grid	высокая(on chip L1 cache)
Texture	R/O	per-grid	высокая(on chip L1 cache)

Основой всего является глобальная память. Она представляет из себя хранилище емкостью от 64 до 8 гигабайт и высочайшей пропускной способности. К сожалению, платой являются задержки в обработке. Так же из-за увеличения размера мы не в состоянии кэшировать ее, но инструкции и указатели на память помогут нам преодолеть данную проблему.

Следом за глобальной идет локальная. Представляя доступ только однопоточному процессору она в добавок еще и достаточно медленная, но все же быстрее глобальной.

Разделяемая память представляет собой 16-килобайтный блок памяти с общим доступом для всех потоковых процессоров в мультипроцессоре. Скорость работы этой памяти гораздо выше представленных выше экземпляров, но она не является рекордсменом в этой области. Обеспечение взаимодействия потоков и возможность «прямого» управления делают ее важным винтиком в нашей вычислительной машине. В составе ее преимуществ так же находятся снижение задержек и сокращение количества обращений к глобальной памяти [18].

Память констант – область памяти объемом 64 килобайта, доступная только для чтения всеми мультипроцессорами. Она кэшируется по 8 килобайт на каждый мультипроцессор. Она медленная, на уровне локальной.

Текстурная память есть блок памяти, открытый всем мультипроцессорам. Выборка данных осуществляется при помощи текстурных блоков видеочипа,

поэтому предоставляются возможности линейной интерполяции данных без дополнительных затрат. Кэшируется по 8 килобайт на каждый мультипроцессор. Медленная, как глобальная.

Собственно, ни для кого не секрет, что глобальная, локальная и остальные виды памяти есть ответвления от локальной видеопамяти видеокарты. И различаются они только алгоритмами кэширования и моделью доступа. Центральный процессор может читать и записывать исключительно в глобальную, константную и текстурную память только при помощи функций копирования.

1.3 Методы параллельного программирования на CUDA

Современные графические процессоры прошли длинный путь от машин для обработки и рендеринга графики до современных массовопараллельных процессоров, способных выполнять полный спектр вычислительных задач. Самые передовые игровые графические процессоры уже достигли производительность в 8.988 TFLOPS5.

С появлением первых полноценных графических процессоров от компании NVIDIA они привлекли внимание не только разработчиков компьютерных игр, но и учёных, которые обратили внимание насколько эффективны GPU в работе над вычислениями с плавающей точкой. Благодаря учёным из Стэндфордского университета в 2003 году появилось расширение языка Си, которое позволило работать с параллельными структурами данных на графическом процессоре. Компания NVIDIA не осталась в стороне, и уже в 2006 году была анонсированная разработка программно-аппаратной технологии CUDA, которая позволила писать программы для графических процессоров на высокоуровневом языке программирования и задействовать GPU как процессор общего назначения.

В основе архитектуры CUDA лежит масштабируемый массив потоковых мультипроцессоров (Streaming Multiprocessors, дальше SM). Такой мультипроцессор способен обрабатывать параллельно сотни нитей. Для

управления работой такого массива потоков была разработана уникальная архитектура – Single-Instruction, Multiple-Thread.

SIMT – это подход к параллельным вычислениям при котором несколько потоков выполняют одни и те же операции на разных данных.

Технология CUDA строится на следующем принципе работы: GPU или по-другому device — устройство является массивно-параллельным сопроцессором для CPU или host, где выполняется последовательный код. Параллельный код, применяемый для параллельных вычислений производится на GPU несколько раз на нескольких параллельных нитях, а функции, которые работают параллельно на всех нитях графического процессора являются ядрами kernel. Такие функции являются расширенными функциями языка C [8].

По принципу технологии CUDA CPU и GPU взаимодействуют для обеспечения работы ядра, что включает в себя также создание и прекращение рабочего цикла ядра.

Существует три понятия, которые лежат в основе идеи расширения языка C для его применении, воплощенном в технологии CUDA:

– разделение памяти;
– барьерная синхронизация;
– иерархия групп нитей.

Выстроим структуру работы нитей в соответствии с полученными понятиями: grid, высший уровень — это массив одномерных или двумерных блоков (blocks) в зависимости от ситуации. Из отличительных особенностей сетки заметна ее непредсказуемость очерёдного запуска блоков и отсутствие общей памяти между блоками. Сетка характерна всем нитям, которые исполняют полученные функции (ядро). Все блоки на данном верхнем уровне представляют из себя массив нитей, который к тому же может быть одномерным, двумерным или трехмерным. Нитей в составе блока может быть до 512. Блоки в сетке имеют свои адреса, индексы и являются соразмерными. Свой индекс имеет и нить блоков.

Процессы, соединенные в блоки, содержат внутри себя общий объем

памяти (shared memory) и синхронное исполнение.

При получении мультипроцессором одного или нескольких блоков нитей для выполнения функции, эти блоки разделяются на группы - варпы, которые состоят из 32 нитей. Каждый варп получает данные о своей очереди выполнения через варп scheduler. Разделение на варпы происходит изначально от схемы работы компьютерной графикой, где при ее обработке использовалось считывание больших пакетов, данных и распределение его между нитями. При считывании непоследовательных данных, не объединенных в общий массив созданные нити практически не используются и процесс не организован.

Поэтому следует тщательно отслеживать распределение данных в памяти, оптимизировать его. Необходимо учитывать и то, что нити разных из варпов могут быть на разных уровнях выполнения программы, а нити из одного варпа выполняются все в одно время [7].

Для выполнения программы каждому блоку данных соответствует один мультипроцессор, где блок функционирует отдельно от остальных блоков. Нити в рамках блока могут использоваться на одном мультипроцессоре одновременно, также возможно использование на одном мультипроцессоре нескольких отдельных блоков. Такая обработка данных на мультипроцессоре является последовательной, блоки на нем обрабатываются поочередно.

В зависимости от используемой системы и ее ресурсов выполнение работы блоков может происходить по-разному, например, в произвольном порядке, а также может выполняться последовательно или одновременно.

Масштабируемость системы происходит благодаря коммуникативным ограничениям между нитями, которые построены путем организации различных видов памяти и соответствующими им уровнями доступа.

У каждой нити есть свой личный объем памяти, так называемая локальная память local memory. Сообщение между нитями в рамках одного блока происходит благодаря разделяемой памяти shared memory, где сообщение происходит с довольно низкой задержкой.

Global memory — глобальная общая память, является объемным

участком памяти и передает информацию с высокой задержкой. Тем не менее глобальная память доступна из CPU и является по сути единственными каналом сообщения между GPU и CPU. На рисунке 1.2 [7] представлена организация уровней памяти в CUDA.

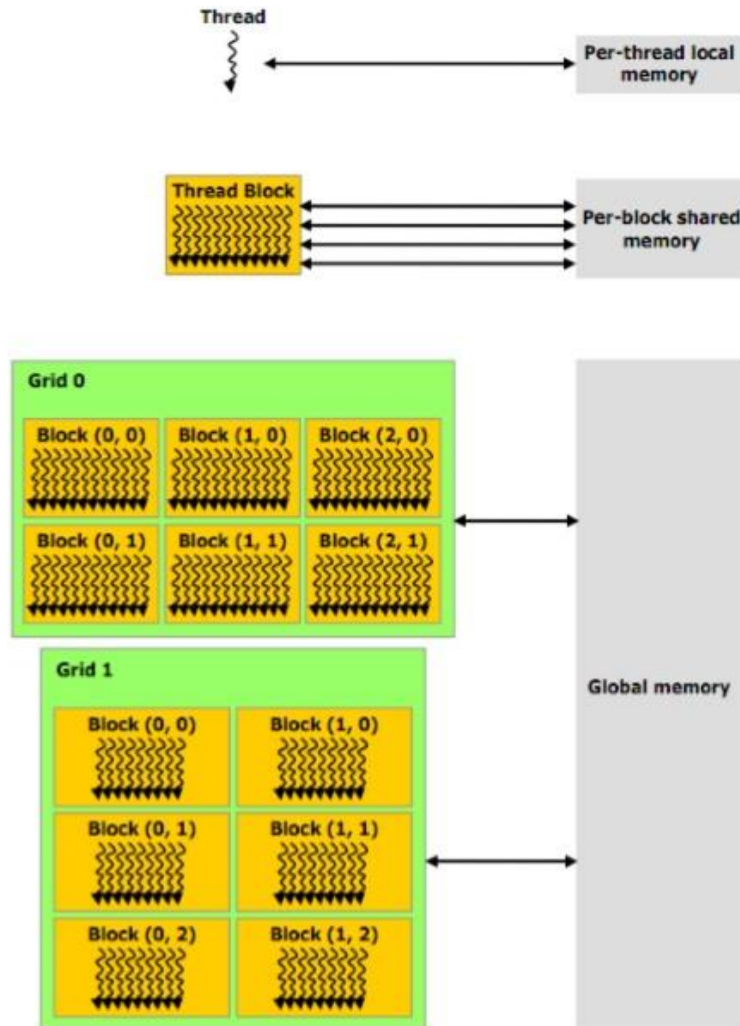

Рисунок 1.2 – Организация уровней памяти

Существует также два дополнительных вида памяти — текстурная память и константная память. Они доступны для всех нитей программы, но открыты лишь для чтения. Также они являются оптимизированными для некоторых отдельных форматов памяти, обладающих своей спецификой.

Взаимодействие нитей между собой происходит благодаря встроенной функции _syncthreads(). Также с помощью нее происходит барьерная синхронизация, характерная для CUDA. С помощью этой функции обозначают

точки синхронизации внутри ядер. Также барьерная синхронизация запрещает нитям выполнять команды, пока все нити не будут выполнять эту функцию.

1.4 Расширения языка C для работы с CUDA

Программы для CUDA используют так называемый «расширенный» язык программирования Си, который требует компиляцию с помощью специализированных команд nvcc.

Используемые в CUDA расширения языка Си состоят из:

– спецификаторов функций, демонстрирующих, местоположение выполняемой функции и место, где она первоначально находилась до вызова;

– спецификаторы, задающие тип памяти, задействованный для определенных переменных;

– директива, необходимая при запуске ядра, которая задает данные и иерархию потоков;

– интегрированные переменные, в которых содержится требуемая информация о текущем потоке;

– функция runtime, содержащая необходимые дополнительные сведения.

В таблице 1.2 представлены спецификаторы функций для работы в CUDA.

Таблица 1.2 – Спецификаторы функций в CUDA

Спецификатор	Выполняется на	Может вызываться из
__device__	Device	device
__global__	Device	host
__host__	Host	host

Стоит помнить, что спецификаторы __host__ и __device__ позволяют использовать себя совместно, что, в свою очередь, означает выполнение соответствующей функции как на графическом процессоре, так и на центральном. При этом необходимый код может быть автоматически

сгенерирован компилятором. Спецификаторы __global__ и __host__ не могут быть использованы вместе [6,19].

Спецификатор __global__ обозначает ядро и соответствующая функция должна возвращать значение типа void.

На функции, выполняемые на GPU (__device__ и __global__) накладываются следующие ограничения:

– невозможно взять их адрес, если это не __global__ функция;
– исключена поддержка рекурсии;
– нет поддержки static-переменных внутри функций;
– исключена поддержка переменного числа входных аргументов.

Для идентификации местоположения переменных в памяти графического процессора нам следует использовать спецификаторы - __device__,__constant__ и __shared__. Следует помнить, что их использование имеет свои собственные ограничения:

– эти спецификаторы не могут быть применены к полям структуры (struct или union);
– соответствующие переменные могут использоваться только в пределах одного файла, их нельзя объявлять как extern;
– запись в переменные типа __constant__ может осуществляться только при помощи центрального процессора, с использованием специальных функций;
– __shared__ переменные не могут инициализироваться при объявлении.

Обращение к компонентам вектора идет по именам - x, y, z и w. Для создания значений-векторов заданного типа служит конструкция вида make_<typeName>.

В данной версии отключена поддержка векторных покомпонентных операций. Это означает, что мы физически не сможем просто сложить два вектора при помощи операции «сложение» - нам придется делать для

отдельных компонент.

Также для задания размерности служит тип dim3, основанный на типе uint3, но обладающий нормальным конструктором, инициализирующим все не заданные компоненты единицами.

Для запуска ядра на графическом процессоре используется следующая конструкция: kernelName<<<Dg,Db,Ns,S>>> (args).

Здесь kernelName это имя соответствующей __global__ функции, Dg - переменная типа dim3, задающая размерность и размер grid'a (в блоках), Db - переменная типа dim3, задающая размерность и размер блока Ns - переменная типа size_t, задающая дополнительный объем shared-памяти, которая должна быть динамически выделена, S - переменная типа cudaStream_t задает поток, в котором должен произойти вызов, по умолчанию используется поток 0. Через args обозначены аргументы вызова функции kernelName.

Также в язык C добавлена функция __syncthreads, осуществляющая синхронизацию всех нитей блока. Управление из нее будет возвращено только тогда, когда все нити данного блока вызовут эту функцию. Т.е. когда весь код, идущий перед этим вызовом, уже выполнен. Эта функция очень удобная для организации бесконфликтной работы cshared-памятью[18].

Также CUDA поддерживает все математические функции из стандартной библиотеки C, однако с точки зрения быстродействия лучше использовать их float-аналоги - например sinf. Кроме этого CUDA предоставляет дополнительный набор математических функций обеспечивающих более низкую точность, но заметно более высокое быстродействие чем sinf, powf и т.п.

1.5. Постановка краевой задачи

Примером краевой задачи является двухточечная краевая задача для обыкновенного дифференциального уравнения второго порядка.

$$y'' = f\,(x, y, y') \tag{1.1}$$

с граничными условиями, заданными на концах отрезка $[a, b]$.

$$y(a) = y_0, \tag{1.2}$$

$$y(b) = y_1 \tag{1.3}$$

Следует найти такое решение $y(x)$ на этом отрезке, которое принимает на концах отрезка значения y_0, y_1. Если функция $f\,(x, y, y')$ линейна по аргументам y,y', то это линейная краевая задача, в противном случае – нелинейная [3].

Кроме граничных условий называемых граничными условиями первого рода, используются еще условия на производные от решения на концах - граничные условия второго рода:

$$y'(a) = \hat{y}_0, \tag{1.4}$$

$$y'(b) = \hat{y}_1 \tag{1.5}$$

или линейная комбинация решений и производных - граничные условия третьего рода:

$$\alpha y(a) + \beta y'(a) = \hat{y}_0, \tag{1.6}$$

$$\delta y(b) + \gamma y'(b) = \hat{y}_1 \tag{1.7}$$

где α, β , δ , γ - такие числа, что $|\alpha| + |\beta| \neq 0, |\delta| + |\gamma| \neq 0$.

Глава 2 Методы решения поставленной задачи

2.1 Уравнение теплопроводности

Уравнение теплопроводности является уравнением параболического типа и имеет вид:

$$\frac{\partial U}{\partial t} = a^2 \frac{\partial^2 U}{\partial x^2}, \qquad (2.1)$$

где $U(x,t)$ - температура стержня в точке x в момент времени t, a - связано с коэффициентами теплоемкости и теплопроводности. Рассмотрим однородный стержень длины l, теплоизолированный с боков и достаточно тонкий, чтобы в любой момент времени температуру во всех точках поперечного сечения можно было считать одинаковой.

Для нахождения единственного решения уравнения теплопроводности необходимо присоединить начальные и граничные условия к уравнению. Для задач такого типа задается всего одно начальное условие, а именно, начальная температура в начальный момент времени [9]. Пусть,

$$\frac{\partial U}{\partial t} = a^2 \frac{\partial^2 U}{\partial x^2}, 0 < x < l, 0 < t \leq T \qquad (2.2)$$

Краевые условия:

$$\begin{cases} U(x,0) = \varphi(x),\ 0 \leq x \leq l \\ U(0,t) = U(l,t) = 0,\ 0 \leq t \leq T \end{cases} \qquad (2.3)$$

где $\varphi(x)$ -начальное распределение температуры в стержне.

Края стержня прикреплены в термостате. В этом случае тепловая энергия стержня не сохраняется, из-за того что система не изолирована.

Найдём решение в виде произведения двух функций

$$U(x,t) = X(x) * T(t), \qquad (2.4)$$

где $X(x)$ - функция только переменного x, а $T(t)$ - функция только переменного t.

$$\frac{\partial (XT)}{\partial t} = a^2 \frac{\partial^2 (XT)}{\partial x^2} \qquad (2.5)$$

$$XT' = a^2TX''$$
$$\frac{T'}{a^2T} = \frac{X''}{X} = \lambda = const \tag{2.6}$$

где $\lambda = const$, вследствие того, что левая часть равенства зависит только от t, а правая часть зависит только от x. Следует то, что

$$\begin{cases} T' - \lambda a^2 T = 0 \\ X'' - \lambda X = 0 \end{cases} \tag{2.7}$$

Граничные условия (2.3) дают: $X(0) = 0$, $X(l) = 0$, тогда

$$\begin{cases} T' - \lambda a^2 T = 0 \\ X'' - \lambda X = 0 \\ X(0) = X(l) = 0 \end{cases} \tag{2.8}$$

Необходимо определить знак λ.

1 случай: Пусть $\lambda > 0$.

Рассмотрим уравнение (2.4):

$$T' - \lambda a^2 T=0. \tag{2.9}$$

Характеристическое уравнение имеет вид:

$$q - \lambda a^2 = 0,$$
$$q = \lambda a^2, \quad . \tag{2.10}$$
$$T(t) = Ce^{\lambda a^2 t},$$

Рассмотрим уравнение (2.5):

$$X'' - \lambda X = 0. \tag{2.11}$$

Характеристическое уравнение имеет вид:

$$q_1^2 - \lambda = 0,$$
$$q_1 = \pm\sqrt{\lambda}, \tag{2.12}$$
$$X(x) = Ae^{-\sqrt{\lambda x}} + Be^{\sqrt{\lambda x}},$$

Это решение не подходит, потому что если $t \to \infty$, то и $e^{\lambda a^2 t} \to \infty$, из-за чего нарушается второй закон термодинамики, а точнее происходит передача энергии от холодного к горячему. Обоснуем это математически, подставляя начальные условия (2.6) в (2.7):

$$\begin{gathered} A + B = 0, \\ A = -B, \\ Ae^{-\sqrt{\lambda l}} - Ae^{\sqrt{\lambda l}}, \\ e^{-\sqrt{\lambda l}} = e^{\sqrt{\lambda l}}, \end{gathered} \tag{2.13}$$

Следовательно $A = 0$ или $\lambda = 0$, но в этом случае мы получаем тривиальное решение и не можем удовлетворять начальное условие. Вследствие этого, при $\lambda = 0$ уравнение (2.1) имеет лишь нулевое решение.

2 случай**:** Пусть $\lambda = 0$, тогда

$T' = 0$, следовательно, $T = C$.

$X'' = 0$, следовательно, $X = Ax + B$.

Подставив краевые условия

$U(0, t) = A * 0 + B = 0$, получаем $B = 0$.

Как результат получим нулевое решение $A = 0$, следовательно $\lambda = 0$ не подходит.

3 случай**:** Пусть $\lambda < 0$ и $\lambda = -p^2$, тогда

$$X'' + p^2 X = 0. \tag{2.14}$$

Характеристическое уравнение имеет вид:

$$\begin{gathered} k^2 + p^2 = 0, \\ k = \pm ip, \end{gathered} \tag{2.15}$$

Общее решение может быть записано так:

$$X = A \cos px + B \sin px, \tag{2.16}$$

Подставим краевые условия.

$$\begin{gathered} X(0) = A = 0, \\ X(l) = \text{B} \sin pl = 0, \end{gathered} \tag{2.17}$$

Получим

$$p_n = \frac{\pi n}{l}, \tag{2.18}$$

Существуют нетривиальные решения уравнения (2.5), равные

$$X_n(x) = \sin \frac{\pi n}{l}, \tag{2.19}$$

Данным значениям p_n соответствуют решения уравнения (2.4)

$T_n(t) = A_n e^{-p_n^2 a^2 t}$, (2.20)

где C_n - неопределенный пока коэффициент.

$$U(x,t) = \sum_{n-1}^{\infty} A_n e^{-p_n^2 a^2 t} \sin\frac{\pi n x}{l} = \sum_{n-1}^{\infty} A_n e^{-(\frac{\pi n}{l})^2 a^2 t} \sin\frac{\pi n x}{l}, \quad (2.21)$$

Общее решение.

Удовлетворим начальным условиям (2.2):

$$U(x,0) = \varphi(x) = \sum_{n-1}^{\infty} A_n \sin\frac{\pi n x}{l}. \quad (2.22)$$

Для выполнения данного начального условия нужно взять в качестве A_n коэффициент Фурье:

$$A_n = \frac{2}{l}\int_0^l \varphi(x)\sin\frac{\pi n x}{l}\,dx. \quad (2.23)$$

Чтобы получить ответ нужно подставить указанный коэффициент в общее решение задачи [9].

2.2 Метод конечных разностей

В качестве примера решения методом конечных разностей (МКР) одномерных уравнений второго порядка рассмотрим решение задачи теплопроводности в одномерном случае [3].

Пусть нужно найти функцию T(x), удовлетворяющую дифференциальному уравнению $k\frac{d^2T}{dx^2} + Q(x) = 0$ на отрезке $0 < x < L$.

Граничные условия задания соответствуют заданию на всех концах отрезка температуры $\bar{T}$ или потока $\bar{q}$.

Для нахождения ответа задачи МКР отрезок, на котором ищется функция, разделяется определенным числом равностоящих точек с координатами $x_l (l = 0,1 \ldots L)$. При этом $x_0 = 0, x_L = L, x_{l+1} - x_l = \Delta x$.

Затем для всех внутренних точек сетки составляется конечно-разностный аналог исходного дифференциального уравнения на основе применения полученных выше формул конечно-разностного представления второй производной. В типичном узле x_l это уравнение будет иметь вид

$$\frac{k(T_{l+1}-2T_l+T_{l-1})}{\Delta x^2} = -Q_l. \qquad (2.24)$$

Когда на двух краях отрезка заданы значения температуры $\bar{T}_0$ и $\bar{T}_L$, то в первом и конечном узлах уравнения не формулируются, а во втором и предпоследнем уравнениях полученные значения температуры переносятся в правую часть.

$$\begin{aligned} -T_0 + 2T_1 - T_2 &= Q_1 * \frac{\Delta x^2}{k}, \\ -T_L + 2T_{L-1} - T_{L-2} &= Q_{L-1} * \frac{\Delta x^2}{k}. \end{aligned} \qquad (2.25)$$

В итоге, в узлах сеточной области неизвестных значений T_l, находится система $L-1$ алгебраических уравнений (АУ) относительно $L-1$

$$\begin{aligned} 2T_1 - T_2 &= \frac{Q_1\Delta x^2}{k} + \bar{T}_0, \\ -T_1 + 2T_2 - T_3 &= Q_2\Delta x^2/k, \\ -T_2 + 2T_3 - T_4 &= \frac{Q_3\Delta x^2}{k}, \\ &\vdots \\ -T_{L-2} + 2T_{L-1} &= \frac{Q_{L-1}\Delta x^2}{k} + \bar{T}_L. \end{aligned} \qquad (2.26)$$

Найденную систему уравнений в матричном представлении можно сформулировать как

$$[K]\{\varphi\} = \{R\}, \qquad (2.27)$$

где $\{\varphi\}$ - вектор неизвестных узловых величин температур $\{\varphi\}^2 = (T_1, T_2, \dots, T_{L-1})$

$[K]$ – матрица системы

$$[K] = \begin{vmatrix} 2 & -1 & 0 & 0 & 0 & . & & & \\ -1 & 2 & -1 & 0 & 0 & \cdot & & & \\ 0 & -1 & 2 & -1 & 0 & \cdot & & & \\ . & . & . & . & . & . & . & . & . \\ & & & & & . & -1 & 2 & -1 \\ & & & & & . & 0 & -1 & 2 \end{vmatrix} \qquad (2.28)$$

$\{R\}$ – вектор правых частей системы

$$\{R\} = \begin{vmatrix} \frac{Q_1 \Delta x^2}{k} + \bar{T}_0 \\ \frac{Q_2 \Delta x^2}{k} \\ \dots \\ \frac{Q_{L-1} \Delta x^2}{k} + \bar{T}_L \end{vmatrix} \qquad (2.29)$$

Также заметим, что матрица $[K]$ становится симметричной и узколенточной (трехдиагональной).

В итоге, начальная задача нахождения неизвестных непрерывных функций замещается задачей решения системы алгебраических уравнений относительно дискретных значений $T_1 \dots T_{L-1}$. То есть, МКР дает информацию о значениях функций в узлах сеточной области, но не дает информации о значениях функций между точками, т.е. дифференциальное уравнение аппроксимируется только в конечном числе точек.

Также рассмотрим случай, в котором поток тепла задан на одном из концов отрезка (например, при $x = L$)

$$k \frac{dT}{dx} = -\bar{q} \quad \text{при} \quad x = L. \qquad (2.30)$$

В данном случае, так как значение температуры T_L оказывается неизвестным для определения всех узловых температур к системе (2.12) необходимо прибавить еще одно уравнение. Это уравнение можно заполучить, расписав в конечных разностях уравнение (2.16) в узле $x = L$.

Если аппроксимировать производную разностью назад, то уравнение примет вид

$$\frac{T_L - T_{L-1}}{\Delta x} = -\bar{q}/k. \qquad (2.31)$$

Прибавляя уравнение (2.17) к системе (2.12) имеем систему L уравнений, нужных для нахождения L неизвестных значений температур [3].

Найденная этим способом система имеет существенный недостаток, связанным с различным порядком аппроксимации решения внутри области $(0(\Delta x^2))$и на границе $(0(\Delta x))$.

$$\begin{aligned} 2T_1 - T_2 &= \frac{Q_1 \Delta x^2}{k} + \bar{T}_0, \\ -T_1 + 2T_2 - T_3 &= \frac{Q_2 \Delta x^2}{k}, \\ -T_2 + 2T_3 - T_4 &= \frac{Q_3 \Delta x^2}{k}, \\ &\vdots \\ -T_{L-2} + 2T_{L-1} - T_L &= \frac{Q_{L-1} \Delta x^2}{k}, \\ -T_{L-1} + T_L &= -\frac{\bar{q} \Delta x}{k}. \end{aligned} \tag{2.32}$$

Полученного недостатка можно избежать, если прибавить к системе (2.12) уравнение (2.11) в узле $x = L$ и использовать для условия аппроксимацию производной в центральных разностях. В итоге система алгебраических уравнений примет вид

$$\begin{aligned} 2T_1 - T_2 &= \frac{Q_1 \Delta x^2}{k} + T_0, \\ -T_1 + 2T_2 - T_3 &= \frac{Q_2 \Delta x^2}{k}, \\ &\cdots\cdots\cdots \\ -T_{L-2} + 2T_{L-1} - T_L &= \frac{Q_{L-1} \Delta x^2}{k}, \\ -T_{L-1} + 2T_L - T_{L+1} &= \frac{Q_L \Delta x^2}{k}, \\ -T_{L-1} + T_{L+1} &= -\frac{2\bar{q} \Delta x}{k}, \end{aligned} \tag{2.33}$$

где T_{L+1}– значение температуры в законтурной точке.

2.3 Метод Гаусса для решения системы линейных уравнений

Для решения систем линейных алгебраических уравнений (СЛАУ) очень хорошо подходит метод Гаусса. У этого метода есть множество преимуществ по сравнению с другими методами решения СЛАУ:

- первое преимущество состоит в том, что предварительно нет необходимости исследовать систему уравнений на совместность;
- вторым преимуществом является то, что методом Гаусса можно решать системы линейных уравнений в которых количество уравнений не соответствует числу неизвестных переменных и определитель основной матрицы не нулевой, а не только СЛАУ, в которых число уравнений совпадает

с количеством неизвестных переменных и основная матрица системы невырожденная;

– третьим преимуществом является то, что при очень малом количестве вычислений, метод Гаусса приводит к результату.

Рассмотрим систему линейных алгебраических уравнений (СЛАУ) относительно n неизвестных x_1 , x_2 , ..., x_n:

$$\begin{cases} a_{11}x_1 + & a_{12}x_2 + & \cdots & + a_{1n}x_n = & b_1 \\ a_{21}x_1 + & a_{22}x_2 + & \cdots & + a_{2n}x_n = & b_2 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n1}x_1 + & a_{n2}x_2 + & \cdots & + a_{nn}x_n = & b_n \end{cases} \qquad (2.34)$$

Когда у системы есть решения, её называют совместной, а если нет, то она называется несовместной. Для решения методом Гаусса будут рассматриваться только системы в которых количество уравнений будет совпадать с количеством неизвестных, а матрица системы имеет определитель не равный нулю. Такие системы всегда являются совместными.

Введём матрицы:

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & & a_{2n} \\ \vdots & & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}, \ \tilde{A} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} & b_1 \\ a_{21} & a_{22} & & a_{2n} & b_2 \\ \vdots & & \ddots & \vdots & \\ a_{n1} & a_{n2} & \cdots & a_{nn} & b_n \end{pmatrix}, \qquad (2.35)$$

где A – матрица коэффициентов системы, $\tilde{A}$ – расширенная матрица коэффициентов.

Элементарными преобразованиями системы линейных уравнений называются:

– перестановка двух любых её уравнений местами;

– умножение всех членов любого уравнения системы на любое число, не равное нулю;

– почленное сложение её двух любых уравнений.

Понятно, что любому элементарному преобразованию системы соответствует аналогичное элементарное преобразование строчек её расширенной матрицы, и наоборот. Следовательно, элементарные

преобразования системы сводятся к соответствующим преобразованиям строчек её расширенной матрицы.

При элементарных преобразованиях система переходит в равносильную систему. Это значит, что каждое решение одной из этих систем является решением другой, и наоборот (или обе эти системы несовместимы).

Целью элементарных преобразований является приведение расширенной матрицы системы к так называемой ступенчатой форме, которая в случае, когда матрица системы невырождена, имеет вид

$$\tilde{A}_1 = \begin{pmatrix} \bar{a}_{11} & \bar{a}_{12} & \dots & \bar{a}_{1n} & \bar{b}_1 \\ 0 & \bar{a}_{22} & & \bar{a}_{2n} & \bar{b}_2 \\ \vdots & & \ddots & \vdots & \\ 0 & 0 & \cdots & \bar{a}_{nn} & \bar{b}_n \end{pmatrix}, \qquad (2.36)$$

причем элементы на диагонали $\bar{a}_{jj} \neq 0$, $j = 1,2,\dots,n$. Для приведения системы к такому виду найдём строку, в которой первый коэффициент не равен нулю (она существует, т.к. иначе первый столбец матрицы A содержит одни нули и, значит, $\det A = 0$, а мы предположили, что матрица A невырождена). Переставим её на первое место, поменяв с первой (если первый элемент первой строки ненулевой, этот шаг можно пропустить). Затем при всех $j = 2,\dots,n$ вычтем из j-й строки ($j = 3,\dots,n$), предварительно домножив на $\frac{\bar{a}_{j2}}{\bar{a}_{12}}$. После этого во втором столбце ниже диагонального элемента окажутся нули. Повторив аналогичную операцию ещё n-3 раза, добьёмся, что и в третьем, четвёртом, …, (n-1)-м столбцах ниже диагонали будут нули, то есть матрица примет ступенчатую форму [15].

Полученная матрица будет являться расширенной матрицей новой ступенчатой системы линейных уравнений:

$$\begin{cases} a_{11}x_1 + & a_{12}x_2 + & \cdots & + a_{1n}x_n = & b_1 \\ & a_{22}x_2 + & \cdots & + a_{2n}x_n = & b_2 \\ & & \cdots & \cdots & \cdots \\ & & & a_{nn}x_n = & b_n \end{cases}, \qquad (2.37)$$

причем все $\bar{a}_{jj} \neq 0$, $j = 1,2, \dots, n$. Такая ступенчатая система имеет ровно одно решение. Чтобы его найти, из последнего уравнения находим x_n, подставляем его в предыдущее уравнение, находим x_{n-1} и т.д.

Глава 3 Разработка программной системы

3.1 Выбор языка программирования и среды разработки

В результате анализа технологии разработки и алгоритмов решения задач была выбрана краевая задача для уравнения теплопроводности, которая решается конечно-разностным методом и параллельным метод Гаусса, а также была выбрана библиотека Rootbeer для запуска системы на GPU и ускорения её работы.

Java – язык программирования с объектной ориентированностью, выбран для того чтобы реализовать нашу систему. Корпорация Sun Microsystems разработала этот язык, а в дальнейшем продала его корпорации Oracle. Java-приложения чаще всего транслируются в специальный байт-код, что позволяет им запускаться на разнообразных виртуальных Java-машинах не завися от архитектуры компьютера. Это делает приложения кроссплатформенными.

Главные преимущества языка Java:

- возможность использования объектно-ориентированного программирования;
- кроссплатформенность;
- многопоточность, является одной из сильных сторон Java, ей всегда удаётся с легкостью манипулировать несколькими потоками;
- улучшенная система безопасности, в рамках которой выполнение программы полностью контролируется виртуальной машиной;
- открытый исходный код.

IntelliJ IDEA — выбранная интеллектуальная среда для разработки приложений на платформе Java, она быстро развивается и предлагает много возможностей. Программа содержит мощные инструменты для программирования, среди них проверка регулярных выражений и синтаксиса, анализ и автоматическое завершение кода, сильные инструменты рефакторинга кода. Поддерживаемые языки программирования IntelliJ IDEA – Java, Groovy, Clojure, Scala, Kotlin. Особенные потенциалы программы улучшают

производительность работы благодаря поддерживанию огромного числа каркасов, шаблонов и технологий [17].

Благодаря дизайну данной среды, программист концентрируется на продуктивную работу и разработку функциональности, а в это время IntelliJ IDEA выполняет разнообразные повторяющиеся операции.

Для запуска Java приложения на GPU была выбрана библиотека Rootbeer. Данная библиотека зарекомендовала себя хорошо со стороны сообщества habrahabr и им же была рекомендована для промышленной разработки, а также является открытым исходным проектом, позволяющим пересобирать выполняемые файлы обычных Java приложений в результате чего будет получен запускаемый файл выполняющийся на GPU. Мною была собрана собственная версия этой библиотеки, конкретно под мою систему.

3.2 Разработка и архитектуры системы и программная реализация системы

3.2.1 Общее описание

Разработанная система содержит две основные сущности: библиотека для использования CUDA в Java – Rootbeer и модуль RunGPU. Библиотека Rootbeer1 переработана для работы на более новых видеокартах и драйверах для использования CUDA.

Модуль RunGPU состоит из 2 частей: GPUMain и CPUMain, GPUMain запускает параллельный алгоритм на графическом процессоре используя Rootbeer, а CPUMain выполняет этот же алгоритм на центральном процессоре.

Общая архитектура системы представлена на рисунке 3.1.

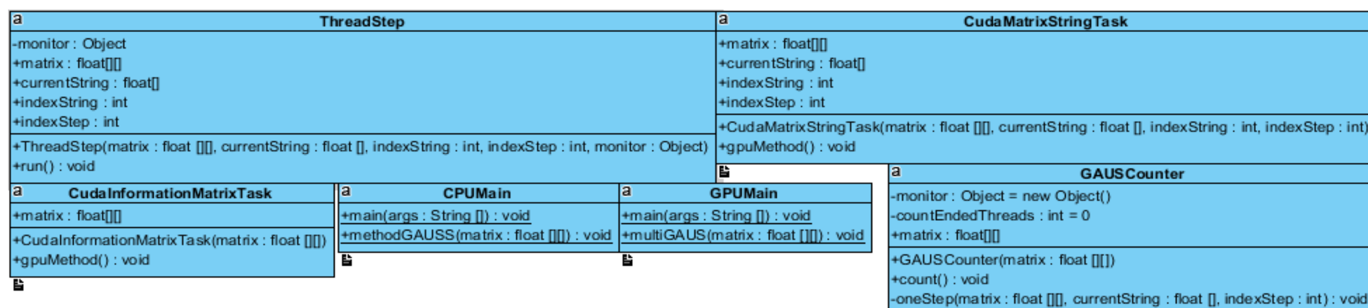

Рисунок 3.1 – Архитектура системы

3.2.2 Модель краевой задачи

Общее представление краевой задачи в разработанной системе представлено на основе формулы уравнения

$$y'' + p(x)\,y' + q(x)\,y = f\,(x), \tag{3.1}$$

$$y(a) = y_0\,, y(b) = y_1, \tag{3.2}$$

и показано на рисунке 3.2.

```
int N;    //Количество разбиений отрезка;
float a; //Значение начала отрезка;
float b; //Значение конца отрезка;
float h; //Шаг разбиения.
float y0;//Значение на левой границе;
float f; //Чему равно ДУ;
float q; //Значение константы q;
h = Math.abs(b - a) / N;
```

Рисунок 3.2 – Представление краевой задачи

3.2.3 Реализация конечно-разностного метода

Введем разностную сетку на отрезке $[a,b]$ $\Omega^{(h)} = \{x_k = x_0 + hk\}, k = 0,1,\dots.,N, h = |b - a|\,/\,N$. Решение задачи будем искать в виде сеточной функции $y^{(h)} = \{y_k, k = 0,1,\dots,N\}$, предполагая, что решение существует и единственно. Введем разностную аппроксимацию производных следующим образом:

$$y'_k = \frac{y_{k+1} - y_{k-1}}{2h} + O(h^2); \tag{3.3}$$

$$y''_k = \frac{y_{k+1} - 2y_k + y_{k-1}}{h^2} + O(h^2); \tag{3.4}$$

Подставляя аппроксимации производных получим систему уравнений для нахождения y_k,

$$\begin{cases} y_0 = y_a \\ \frac{y_{k+1} - 2y_k + y_{k-1}}{h^2} + p(x_k)\frac{y_{k+1} - y_{k-1}}{2h} + q(x_k)y_k = f(x_k), k = 1, N - 1\,, \\ y_N = y_b \end{cases} \tag{3.5}$$

а реализация в коде представлена на рисунке 3.3.

```
     float[] x = new float[N - 1];
float[] p = new float[N - 1];// Объявили массив под значения шагов x.
for (int i = 0; i < N - 1; i++) { // Заполняем массив ранее объявленный массив.
     x[i] = a;
     p[i] = x[i]; // Заполнение константы p, при p(x)=x.
     a = a + h;
}
```

Рисунок 3.3 – Получение системы уравнений

Приводя подобные и учитывая, что при задании граничных условий первого рода два неизвестных уже фактически определены, получим систему линейных алгебраических уравнений с матрицей коэффициентов а2 [5,10,11], реализация в коде показана на рисунках 3.4-3.6.

```
float[][] A = new float[N][N];
float[][] B = new float[N][N];
float[][] C = new float[N][N];
float[][] F = new float[N][N];

int n = N - 1;

float[] b2 = new float[n];
float[] x2 = new float[n];
float d, s;
float[][] a2 = new float[n][n];
// Рассчитываем коэффициенты в СЛАУ.
A[N - 1][N - 2] = 1; // Третья строчка системы из условия задачи.
B[N - 1][N - 1] = (-1 - 2 * h); // Третья строчка системы из условия задачи.
```

Рисунок 3.4 – Реализация получения системы линейных уравнений 1

```
for (int i = 1; i < N - 1; i++) {
    if (i == 1) {
        B[i][i] = (-2 - h * h * q);
        C[i][i + 1] = (1 + (p[i] * h / 2));
    } else if (i == N - 1) {
        C[i][i + 1] = (1 + (p[i] * h / 2));
    } else {
        A[i][i - 1] = (1 - (p[i] * h / 2));//
Коэффициенты формул
        B[i][i] = (-2 - h * h * q);
        C[i][i + 1] = (1 + (p[i] * h / 2));
    }
}

// Рассчитываем правую часть значений (F) после равенства
в СЛАУ.

for (int i = 1; i < N; i++) {
    if (i == 1) {
        F[i][i] = (h * h * f - (1 - (p[i] * h / 2)) *
y0);
    } else {
        F[i][i] = (h * h * f);
    }
}
```

Рисунок 3.5 – Реализация получения системы линейных уравнений 2

```
// Обрабатываем полученные данные из массивов методом
Гаусса.
for (int i = 1; i <= N - 1; i++) { // Передали массив F.
    b2[i - 1] = F[i][i];
}
// Заносим необходимые элементы из A, B, C в один общий
массив.
int i = 1;
int j = 0;
while (i < N - 1) {
    a2[i][j] = A[i + 1][j + 1];
    i++;
    j++;
}
i = 0;
j = 0;
while (i < N - 1) {
    a2[i][j] = B[i + 1][j + 1];
    i++;
    j++;
}
i = 0;
j = 1;
while (i < N - 2) {
    a2[i][j] = C[i + 1][j + 1];
    i++;
    j++;
}
```

Рисунок 3.6 – Реализация получения системы линейных уравнений 3

3.2.4 Реализация метода Гаусса для решения СЛАУ

В данной системе реализован параллельный метод Гаусса, а так же реализован обратный ход. Для выполнения параллельных вычислений на CPU используется метод methodGAUSS из класса CPUMain, принимающий на вход двумерный массив, элементы которого коэффициенты СЛАУ и правая часть СЛАУ.

Рассмотрим механизм данного метода на рисунке 3.7.

```
new GAUSCounter(matrix).count();
```

Рисунок 3.7 – Механизм метода Гаусса в реализованной системе

Для получения матрицы треугольного вида используется класс GAUSCounter, который представлен на рисунке 3.8.

```
final Object monitor = new Object();
int countEndedThreads = 0;
float[][] matrix;
```

Рисунок 3.8 – Представление GAUSCounter в системе

Поле monitor используется для синхронизации шагов вычитания строк из матрицы, он занимается в начале итерации цикла и освобождается в конце итерации цикла, после того, как все процессы созданные в данной итерации завершились. Поле countEndedThreads - подсчёт выполненных процессов в одной итерации цикла. Параметр matrix принимает коэффициенты СЛАУ и правую часть СЛАУ. Конструктор класса принимает единственный параметр matrix.

Приведением матрицы к треугольному виду занимается метод на рисунке 3.9.

```
public void count(){
    for (int i = 0; i < matrix.length - 1; i++) {
        synchronized (monitor) {
            oneStep(matrix, matrix[i], i);
            while (true) {
                countEndedThreads++;
                if (countEndedThreads == matrix.length -
1 - i) {
                    break;
                }
                try {
                    monitor.wait(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
        countEndedThreads = 0;
    }
}
```

Рисунок 3.9 – Реализация приведения матрицы к треугольному виду

Алгоритм работы метода заключается в следующем. Запускается цикл, одна итерация которого применяет на матрицу одну итерацию метода Гаусса, запускает цикл while который проверяет занятость monitor и подсчитывает количество выполненных процессов.

Выполнением одной итерации метода Гаусса занимается класс ThreadStep, который представлен на рисунке 3.10.

```
final Object monitor;
float[][] matrix;
float[] currentString;
int indexString;
int indexStep;
```

Рисунок 3.10 – Представление класса ThreadStep

Поле monitor используется для синхронизации шагов вычитания строк из матрицы, он занимается в начале итерации цикла и освобождается в конце итерации цикла, после того, как все процессы созданные в данной итерации завершились. Параметр matrix принимает коэффициенты СЛАУ и правую часть СЛАУ. Конструктор класса принимает единственный параметр matrix. indexString – номер строки из которой нужно вычесть, indexStep – номер элемента который требуется обнулить. currentString- строка которую нужно вычесть из matrix.

Данный класс является наследником класса Thread, который является представлением в языке Java процесса.

С помощью метода oneStep из класса GAUSSCounter, выполняется цикл в котором создаётся объект процесса ThreadStep в котором происходит алгоритм вычитания строки из матрицы и поиск элемента который нужно обнулить, затем процесс запускается, это показано на рисунке 3.11.

```
float znam = matrix[indexString][indexStep];
if (znam != 0.0f) {
    float d = currentString[indexStep] / znam;
    for (int j = 0; j < matrix[indexString].length; j++)
{
        matrix[indexString][j] *= d;
        matrix[indexString][j] -= currentString[j];
    }
}
synchronized (monitor) {
    monitor.notifyAll();
}
```

Рисунок 3.11 – Представление метода oneStep

По индексу indexStep берётся элемент который требуется обнулить, и берётся элемент по индексу indexStep из строки которую нужно вычесть, между ними находится общий множитель, в результате чего одно из выражений становится равным предыдущему. После разности элемент с индексом indexStep становится равным нулю, а так же после выполнения всех операций, monitor освобождается[12,14,16].

В результате выполнения метода count класса GAUSSCounter мы получим матрицу треугольного вида, затем с помощью обратного хода получим искомые значения, это представлено на рисунке 3.12.

```
float[] results = new float[matrix.length];
int i = 0;
int j = 0;
for (i = matrix.length - 1; i >= 0; i--) {
    float sum = 0.0f;
    for (j = matrix[i].length - 2; j > i; j--) {
        sum += matrix[i][j] * results[j];
    }
    results[i] = (matrix[i][matrix[i].length - 1] - sum)
/ matrix[i][j];
    System.out.println(String.format("X%d = %f %s", i,
results[i], "\t"));
}
```

Рисунок 3.12 – Нахождение и вывод искомых значений

На рисунке 3.13 показана матрица приведённая к треугольному виду и вывод решения СЛАУ в консоль.

```
-2,020408   1,081633   0,000000   0,000000   0,000000    0,000000    0,000000    -0,877551
0,000000    3,413208   -2,429030  0,000000   0,000000    0,000000    0,000000    0,786746
0,000000    0,000000   -5,250688  4,188937   0,000000    0,000000    0,000000    -0,631600
0,000000    -0,000000  0,000000   7,760905   -6,578449   0,000000    0,000000    0,390189
0,000000    0,000000   0,000000   0,000000   -11,289682  9,926740    0,000000    -0,029217
0,000000    0,000000   0,000000   0,000000   0,000000    16,371582   -14,742998  -0,502062
0,000000    0,000000   0,000000   0,000000   0,000000    0,000000    -6,306179   1,170290

X6 = -0,185578
X5 = -0,197784
X4 = -0,171319
X3 = -0,094940
X2 = 0,044547
X1 = 0,262202
X0 = 0,574714
```

Рисунок 3.13 – Вывод решения в консоль

Для запуска данного механизма на GPU используется библиотека Rootbeer.

Для запуска разработанного алгоритма нам потребуется две сущности - интерфейс Kernel которые показаны на рисунке 3.14.

```
public interface Kernel {
    void gpuMethod();
}
```

Рисунок 3.14 – Сущность интерфейса Kernel

и класс Rootbeer и его метод run представленный на рисунке 3.15.

```
public void run(List<Kernel> work) {
    Context context = this.createDefaultContext();
    ThreadConfig thread_config =
this.getThreadConfig(work, context.getDevice());

    try {
        context.setThreadConfig(thread_config);
        context.setKernel((Kernel)work.get(0));
        context.setUsingHandles(true);
        context.buildState();
        context.run(work);
    } finally {
        context.close();
    }

}
```

Рисунок 3.15 – Метод run класса Rootbeer

Данный метод принимает параметром список интерфейсов Kernel, передаёт их в сущность context и выполняет методы gpuMethod интерфейса Kernel на GPU[20].

Метод methodGAUSS переработан способом показанным на рисунке 3.16.

```
public static void multiGAUS(final float[][] matrix) {
    System.out.println("start_=======================");
    List<Kernel> tasks = new ArrayList<Kernel>();
    for (int i = 0; i < matrix.length; i++) {
        for (int j = i + 1; j < matrix.length; j++) {
            tasks.add(new CudaMatrixStringTask(matrix,
matrix[i], j, i));
        }
    }
    Rootbeer rootbeer = new Rootbeer();
    rootbeer.run(tasks);
    List<Kernel> informationTasks = new
ArrayList<Kernel>();
    informationTasks.add(new
CudaInformationMatrixTask(matrix));
    Rootbeer informationRootbeer = new Rootbeer();
    informationRootbeer.run(informationTasks);
    System.out.println("end_=======================");
}
```

Рисунок 3.16 – Реализация метода methodGAUSS

Процессы (Thread) заменены на реализацию интерфейса Kernel используемый и понимаемый библиотекой Rootbeer на GPU. В данном случае это два класса: CudaMatrixStringTask и CudaInformationMatrixTask. CudaMatrixStringTask использует алгоритм из процессов создаваемых в методе runThread из класса CPUMain. CudaInformationMatrixTask выполняет операции для обратного хода.

3.3 Результаты вычислительных экспериментов

Для оценки эффективности разработанной системы были проведены вычислительные эксперименты. В ходе экспериментов сравнение параллельных алгоритмов проводилось по нескольким характеристикам: время выполнения на GPU, время выполнения на CPU при больших и малых входных данных. В экспериментах использовался процессор Intel(R) Core(TM)i3-4150 CPU @

3.5GHz и видеокарта NVIDIA GeForce GTX 750 Ti 1Gb. Результаты экспериментов представлены на рисунке 3.17.

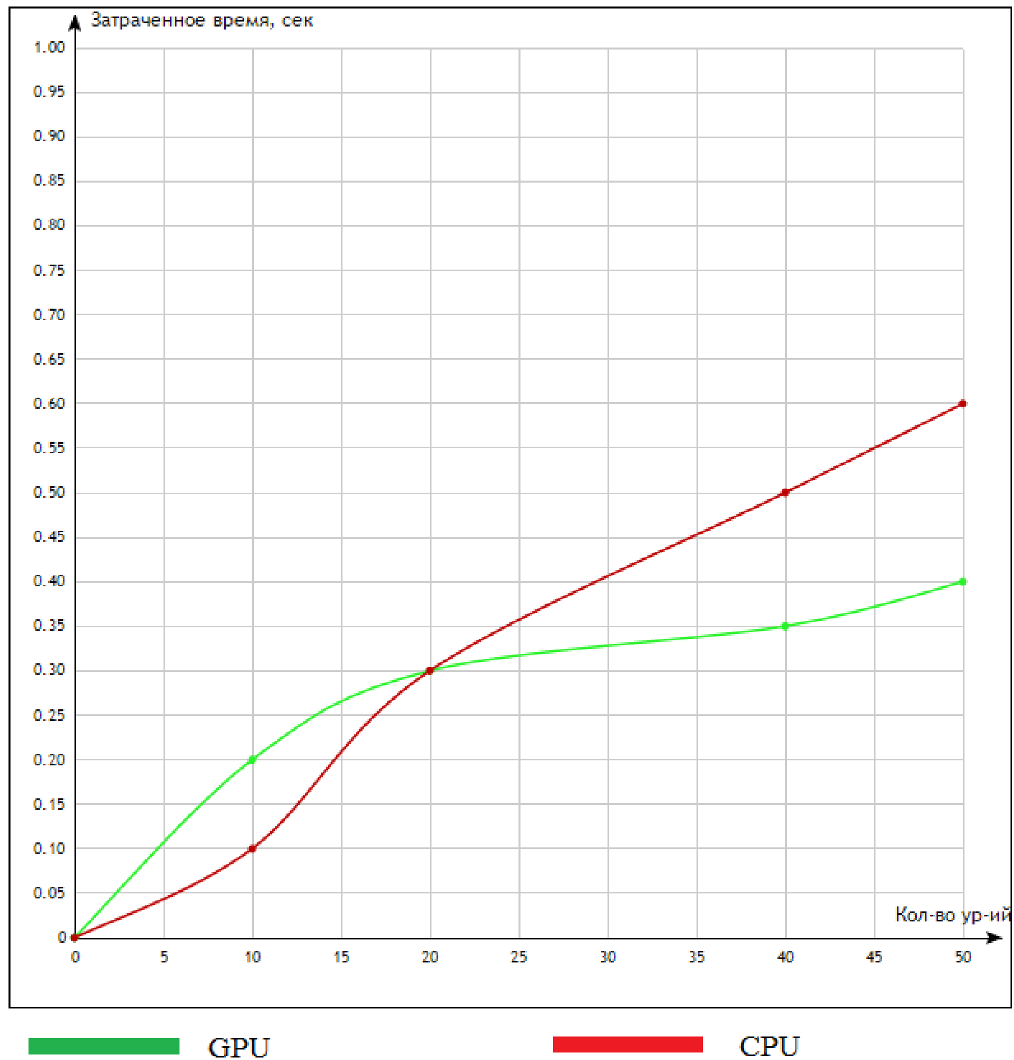

Рисунок 3.17 – Сравнение быстродействия программы на GPU и CPU

В результате проведённых экспериментов видно, что программа на CPU выполняется быстрее при малом количестве данных, но с увеличением количества данных, программа на GPU выполняется заметно быстрее, это связано с тем, что ядра CPU созданы для исполнения одного потока последовательных инструкций с максимальной производительностью, а ядра GPU спроектированы для быстрого выполнения большого числа одновременных потоков-инструкций.

Заключение

Целью данной выпускной квалификационной работы была реализация параллельных вычислений на основе программно-аппаратного стека CUDA.

Цель данной работы была достигнута благодаря изучению теоретических основ и современных направлений развития по выбранной тематике, рассмотренных в первой главе. Анализ методов решения поставленной задачи был произведён во второй главе, а разработка и проведение вычислительных экспериментов в третьей главе. Непосредственную помощь в этом оказало прохождение практики в Центре Новых Информационных технологий ТГУ.

Для достижения поставленной цели были проанализированы и подробно изучены основные технологии и решения, используемые при программировании на CUDA.

Решение было разработано в среде разработки IntelliJ IDEA с применением объектно-ориентированного языка Java и библиотеки Rootbeer.

Практическая значимость работы несомненна, потому что была существенно ускорена работа программы благодаря использованию технологии CUDA.

В результате разработки, было выявлено, что программа запущенная на GPU с использованием программно-аппаратного стека CUDA выполняется быстрее с увеличением количества введённых данных, но на малом количестве данных проигрывает по скорости программе на CPU.

Список используемой литературы

1. Боресков А. В. Основы работы с технологией CUDA/ А. В. Боресков, А. А. Харламов – М.: ДМК-Пресс, 2016. – 232 с.

2. Боресков А. В. Параллельные вычисления на GPU. Архитектура и программная модель CUDA: Учебное пособие/ А. В. Боресков, А. А. Харламов, Н. Д. Марковский, Д. Н. Микушин, Е. В. Мортиков, А. А. Мыльцев, Н. А. Сахарный, В. А. Фролов – М.: Издательство Московского Университета, 2012. – 336 с.

3. Виленкин И. В. Высшая математика: интегралы по мере; дифференциальные уравнения; ряды/ И. В. Виленкин, В. М. Гробер, О. В. Гробер – Ростов-на-Дону: Феникс, 2011. – 302 с.

4. Лизунова Н. А. Матрицы и системы линейных уравнений/ Н. А. Лизунова, С. П. Шкроба – М.: Физмалит, 2007. – 352 с.

5. Макграт М. Программирование на Java для начинающих/ М. Макграт – М.: Эксмо-Пресс, 2016. – 192 с.

6. Прата С. Язык программирования C++. Лекции и упражнения/ С. Прата – М.: Вильямс, 2015. – 1248 с.

7. Рутш Г. CUDA Fortran для инженеров и научных работников. Рекомендации по эффективному программированию/ Г. Рутш, М. Фатика – М.: ДМК-Пресс, 2014. – 364 с.

8. Сандерс Д. Технология CUDA в примерах. Введение в программирование графических процессов/ Д. Сандерс, Э. Кэндрот – М.: ДМК-Пресс, 2015. – 232 с.

9. Формалев В. Ф. Теплопроводность анизотропных тел. Аналитические методы решения задач/ В. Ф. Формалев – М.: Физматлит, 2015. – 312 с.

10. Фриман Э. Изучаем программирование на JavaScript/ Э. Фриман, Э. Робсон – СПб.: Питер, 2016. – 640 с.

11. Хорстманн К. С. Java SE 8. Базовый курс/ К. С. Хорстманн – М.:

Вильямс, 2015. – 464 с.

12. Шилдт Г. Java 8. Полное руководство/ Г. Шилдт – М.: Вильямс, 2015. – 1376 с.

13. Эдвардс Ч. Г. Дифференциальные уравнения и краевые задачи. Моделирование и вычисление с помощью Mathematica, Maple/ Ч. Г. Эдвардс, Д. Э. Пенни – М.: Вильямс, 2016. – 1104 с.

14. Эккель Б. Философия Java/ Б. Эккель – СПб.: Питер, 2015. – 1168 с.

15. Юрченко Е. Математика. Линейные уравнения и линейные выражения. Системы линейных уравнений/ Е. Юрченко, Л. Б. Слуцкий М.: Айрис-пресс, 2007. – 48 с.

16. Benjamin J. Evans, Java in a Nutshell. – 6th edition, O'Reilly Media, 2014.

17. Hudson O. A. Getting started with IntelliJ IDEA – 1st edition, Packt Publishing, 2013.

18. Malik M. Z. CUDA 4.1 Programming Cookbook – 1st edition, Packt Publishing, 2016.

19. Modak B. K. C++ Application Development with Code:: Blocks. – 1st edition, Packt Publishing, 2013.

20. Richard M. Reese, Java 7 New Features Cookbook – 1st edition, Packt Publishing, 2012.

Приложение А

Код класса CPUMain

```
public class ThreadStep extends Thread {
    private final Object monitor;
    public float[][] matrix;
    public float[] currentString;
    public int indexString;
    public int indexStep;
    public ThreadStep(float[][] matrix, float[] currentString, int indexString, int
indexStep, Object monitor) {
        this.matrix = matrix;
        this.currentString = currentString;
        this.indexString = indexString;
        this.indexStep = indexStep;
        this.monitor = monitor;
    }
    @Override
    public void run() {
        float znam = matrix[indexString][indexStep];
        if (znam != 0.0f) {
            float d = currentString[indexStep] / znam;
            for (int j = 0; j < matrix[indexString].length; j++) {
                matrix[indexString][j] *= d;
                matrix[indexString][j] -= currentString[j];}
        }
        synchronized (monitor) {
            monitor.notifyAll();
        }
    }
}
```

Приложение Б

Код класса CudaInformationMatrixTask

```
import org.trifort.rootbeer.runtime.Kernel;
public class CudaInformationMatrixTask implements Kernel {
   public float[][] matrix;
   public CudaInformationMatrixTask(float[][] matrix) {
      this.matrix = matrix;
   }
   @Override
   public void gpuMethod() {
      for (int i = 0; i < matrix.length; i++) {
         for (int j = 0; j < matrix[i].length; j++) {
            System.out.print(String.format("%f %s", matrix[i][j], "\t"));
         }
         System.out.println();
      }
      System.out.println();
      float[] results = new float[matrix.length];
      int i = 0;
      int j = 0;
      for (i = matrix.length - 1; i >= 0; i--) {
         float sum = 0.0f;
         for (j = matrix[i].length - 2; j > i; j--) {
            sum += matrix[i][j] * results[j];
         }
         results[i] = (matrix[i][matrix[i].length - 1] - sum) / matrix[i][j];
         System.out.println(String.format("X%d = %f %s", i, results[i], "\t"));
      }
   }
}
```

Приложение В

Код класса CudaMatrixStringTask.

```
import org.trifort.rootbeer.runtime.Kernel;

public class CudaMatrixStringTask implements Kernel {

   public float[][] matrix;
   public float[] currentString;
   public int indexString;
   public int indexStep;
   public CudaMatrixStringTask(float[][] matrix, float[] currentString, int
indexString, int indexStep) {
      this.matrix = matrix;
      this.currentString = currentString;
      this.indexString = indexString;
      this.indexStep = indexStep;
   }
   @Override
   public void gpuMethod() {
      float znam = matrix[indexString][indexStep];
      if (znam != 0.0f) {
         float d = currentString[indexStep] / znam;
         for (int j = 0; j < matrix[indexString].length; j++) {
            matrix[indexString][j] *= d;
            matrix[indexString][j] -= currentString[j];
         }
      }
   }
}
```

Приложение Г

Код класса GAUSCounter.

```
public class GAUSCounter {

    private final Object monitor = new Object();
    private int countEndedThreads = 0;
    public float[][] matrix;

    public GAUSCounter(float[][] matrix) {
        this.matrix = matrix;
    }

    public void count() {
        for (int i = 0; i < matrix.length - 1; i++) {
            synchronized (monitor) {
                oneStep(matrix, matrix[i], i);
                while (true) {
                    countEndedThreads++;
                    if (countEndedThreads == matrix.length - 1 - i) {
                        break;
                    }
                    try {
                        monitor.wait(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
            countEndedThreads = 0;
        }
```

```
    }

    private void oneStep(final float[][] matrix, final float[] currentString, final int indexStep) {
        for (int j = indexStep + 1; j < matrix.length; j++) {
            new ThreadStep(matrix, currentString, j, indexStep, this.monitor).start();
        }
    }
}
```

Приложение Д

Код класса GPUMain.

```
import org.trifort.rootbeer.runtime.Kernel;
import org.trifort.rootbeer.runtime.Rootbeer;

import java.util.ArrayList;
import java.util.List;

/**
 * Created by VakaVaka on 08.06.2016.
 */
public class GPUMain {

    public static void main(String[] args) {

        int N = 5;
        float a = 0, b = 1, h, y0 = 1, f = 0, q = 1;

        /*
        "Введите значение начала отрезка:";
        a;
        "Введите значение конца отрезка:";
        b;
        "Введите количество разбиений отрезка:";
        N;
        "Введите значение на левой границе:";
        y0;
        "Введите чему равно ДУ:";
        f;
        "Введите значение константы q";
```

```
q;
*/

h = Math.abs(b - a) / N; // Шаг разбиения.
N++; // Иначе грозит выход за границы массива.

float[] x = new float[N - 1];
float[] p = new float[N - 1];// Объявили массив под значения шагов x.

for (int i = 0; i < N - 1; i++) { // Заполняем массив ранее объявленный
массив.
    x[i] = a;
    p[i] = x[i]; // Заполнение константы p, при p(x)=x.
    a = a + h;
}

float[][] A = new float[N][N];
float[][] B = new float[N][N];
float[][] C = new float[N][N];
float[][] F = new float[N][N];

int n = N - 1;

float[] b2 = new float[n];
float[] x2 = new float[n];
float d, s;
float[][] a2 = new float[n][n];

// Рассчитываем коэффициенты в СЛАУ.
```

```
A[N - 1][N - 2] = 1; // Третья строчка системы из условия задачи.
B[N - 1][N - 1] = (-1 - 2 * h); // Третья строчка системы из условия задачи.

for (int i = 1; i < N - 1; i++) {
   if (i == 1) {
      B[i][i] = (-2 - h * h * q);
      C[i][i + 1] = (1 + (p[i] * h / 2));
   } else if (i == N - 1) {
      C[i][i + 1] = (1 + (p[i] * h / 2));
   } else {
      A[i][i - 1] = (1 - (p[i] * h / 2));// Коэффициенты формул
      B[i][i] = (-2 - h * h * q);
      C[i][i + 1] = (1 + (p[i] * h / 2));
   }
}

// Рассчитываем правую часть значений (F) после равенства в СЛАУ.

for (int i = 1; i < N; i++) {
   if (i == 1) {
      F[i][i] = (h * h * f - (1 - (p[i] * h / 2)) * y0);
   } else {
      F[i][i] = (h * h * f);
   }
}

// Обрабатываем полученные данные из массивов методом Гаусса.

for (int i = 1; i <= N - 1; i++) { // Передали массив F.
   b2[i - 1] = F[i][i];
```

```
}

// Заносим необходимые элементы из A, B, C в один общий массив.

int i = 1;
int j = 0;
while (i < N - 1) {
    a2[i][j] = A[i + 1][j + 1];
    i++;
    j++;
}
i = 0;
j = 0;
while (i < N - 1) {
    a2[i][j] = B[i + 1][j + 1];
    i++;
    j++;
}
i = 0;
j = 1;
while (i < N - 2) {
    a2[i][j] = C[i + 1][j + 1];
    i++;
    j++;
}
for (i = 0; i < N - 1; i++) { // Проверка.
    for (j = 0; j < N - 1; j++) {
        System.out.print(a2[i][j] + "\t");
    }
    System.out.println();
```

```
}
System.out.println("\n a2.2 ");
System.out.println();

float a3[][] = new float[a2.length][a2[0].length + 1];
for (int index = 0; index < a3.length; index++) {
    for (int jindex = 0; jindex < a3[index].length - 1; jindex++) {
        a3[index][jindex] = a2[index][jindex];
    }
    a3[index][a2.length] = b2[index];
}
// Метод Гаусса.
for (int k = 0; k < n; k++) // Прямой ход.
{
    for (j = k + 1; j < n; j++) {
        d = a2[j][k] / a2[k][k]; // формула (1)
        for (i = k; i <= n - 1; i++) {
            a2[j][i] = a2[j][i] - d * a2[k][i]; // формула (2)
        }
        b2[j] = b2[j] - d * b2[k]; // формула (2), только для расширения
    }
}
for (int k = n - 1; k >= 0; k--) // Обратный ход.
{
    d = 0;
    for (j = k; j < n; j++) {
        s = a2[k][j] * x2[j]; // формула (3)
        d = d + s; // сумма
    }
    x2[k] = (b2[k] - d) / a2[k][k]; // формула (3)
```

```
        }

        // Вывод решения в виде таблицы всех значений.

        System.out.println("Решение СЛАУ: ");
        for (i = 0; i < n; i++) {
            System.out.println("x[" + i + "]=" + x2[i] + " ");
        }

        multiGAUS(a3);
    }

    public static void multiGAUS(final float[][] matrix) {
        System.out.println("start_========================");
        List<Kernel> tasks = new ArrayList<Kernel>();
        for (int i = 0; i < matrix.length; i++) {
            for (int j = i + 1; j < matrix.length; j++) {
                tasks.add(new CudaMatrixStringTask(matrix, matrix[i], j, i));
            }
        }
        Rootbeer rootbeer = new Rootbeer();
        rootbeer.run(tasks);
        List<Kernel> informationTasks = new ArrayList<Kernel>();
        informationTasks.add(new CudaInformationMatrixTask(matrix));
        Rootbeer informationRootbeer = new Rootbeer();
        informationRootbeer.run(informationTasks);
        System.out.println("end_========================");
    }
}
```

Приложение Е

Код класса ThreadStep.

```
public class ThreadStep extends Thread {
   private final Object monitor;
   public float[][] matrix;
   public float[] currentString;
   public int indexString;
   public int indexStep;
   public ThreadStep(float[][] matrix, float[] currentString, int indexString, int
indexStep, Object monitor) {
      this.matrix = matrix;
      this.currentString = currentString;
      this.indexString = indexString;
      this.indexStep = indexStep;
      this.monitor = monitor;}
   @Override
   public void run() {
      float znam = matrix[indexString][indexStep];
      if (znam != 0.0f) {
         float d = currentString[indexStep] / znam;
         for (int j = 0; j < matrix[indexString].length; j++) {
            matrix[indexString][j] *= d;
            matrix[indexString][j] -= currentString[j];
         }
      }
      synchronized (monitor) {
         monitor.notifyAll();
      }
   }
}
```