Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: «Архитектура вычислительных систем»

|  |  |
| --- | --- |
|  | *К защите допустить* |
|  | И.о. заведующего кафедрой информатики |
|  | \_\_\_\_\_\_\_\_\_\_\_\_ С. И. Сиротко |

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту

на тему

**СрАВНЕНИЕ СКОРОСТИ ЧИСЛЕННОГО РЕШЕНИЯ КРАЕВОЙ ЗАДАЧИ НА *cuda*-ЯДРАХ ВЕРСИИ 11.6 И ПРОЦЕССОРЕ *INTEL* *I*5 1135*G*7**

БГУИР КП 1-40 04 01 01 001

|  |  |
| --- | --- |
|  | |
| Студент  Руководитель  Нормконтроль | К.А. Тимофеев  А.С. Калиновская  А.С. Калиновская |

Минск 2022

СОДЕРЖАНИЕ

Введение5

1 Архитектура вычислительной системы6

2 Платформа программного обеспечения10

3 Теоретическое обоснование разработки программного продукта14

4 Проектирование функциональных возможностей программы 23

5 Анализ полученных результатов26

Заключение29

Список литературных источников31

Приложение А (обязательное) Листинг программного кода32

Приложение Б (обязательное) Функциональная схема алгоритма, реализующего программное средство66

Приложение В (обязательное) Блок схема алгоритма, реализующего программное средство67

Приложение В (обязательное) Графики сравнения68

Приложение Д (обязательное) Ведомость документов69

**ВВЕДЕНИЕ**

На сегодняшний день прогресс в производстве и разработке процессоров достиг небывалых высот. Существует огромное количество как процессоров общего назначения для массового пользователя, так и узконаправленных аппаратных платформ для выполнения конкретных производственных задач. Каждая аппаратная платформа обладает собственной архитектурой, поэтому сравнивать платформы стоит в контексте выполнения определенных программ, т.к. универсальные показатели не всегда могут дать правильное представление о вычислительных качествах системы.

Численные методы являются важной частью решения многих прикладных задач. Сводя решение сложных аналитических задач к арифметическим операциям, они позволяют быстро и эффективно моделировать многие физические процессы реального мира с малой погрешностью.

В данной курсовой работе будет произведено сравнение скорости численного решения краевой задачи на *CUDA*-ядрах версии 11.6 на графическом сопроцессоре *NVidia GeForce GTX* 1050*Ti* и процессоре *Intel Core* *i*5 1135*G*7. При чем измерение скорости на процессоре будет производиться на двух вариантах реализации алгоритма: с учетом многопоточности и в режиме одного потока.

Программы, реализующие вычислительный алгоритм будут реализованы на языке программирования *C*++.

В итоге будут получены графики производительности процессоров, отображающие время, необходимое для численного решения краевой задачи при различных условиях. При значительной разнице результатов можно будет сделать более обобщенные выводы о том, какую аппаратную платформу стоит использовать в подобных задачах.

Результаты данной работы могут быть полезными для разработчиков и исследователей, которые занимаются численным моделированием и нуждаются в выборе наиболее подходящей платформы для своих вычислений.

**1 АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНОЙ СИСТЕМЫ**

**Структура и архитектура вычислительной системы процессора *Intel Core* *i*5 1135*G*7:** процессор *Intel* *Core i5 1135G7* является представителем 11 поколения процессоров компании Intel. Данное поколение использует архитектуру *Tiger Lake*, которая является развитием *Ice Lake*. *Tiger Lake* имеет улучшенный 10нм техпроцесс, переработанную архитектуру ядер с улучшениями в дизайне кэш-памяти и улучшенное встроенное графическое ядро *Intel Iris Xe*. В архитектуру встроен 4-ядерный цифровой сигнальный процессор для работы со звуком [1]. Также все процессоры данной архитектуры поддерживают *AVX-*инструкции. На рисунке 1.1 показана схема данной архитектуры [2].

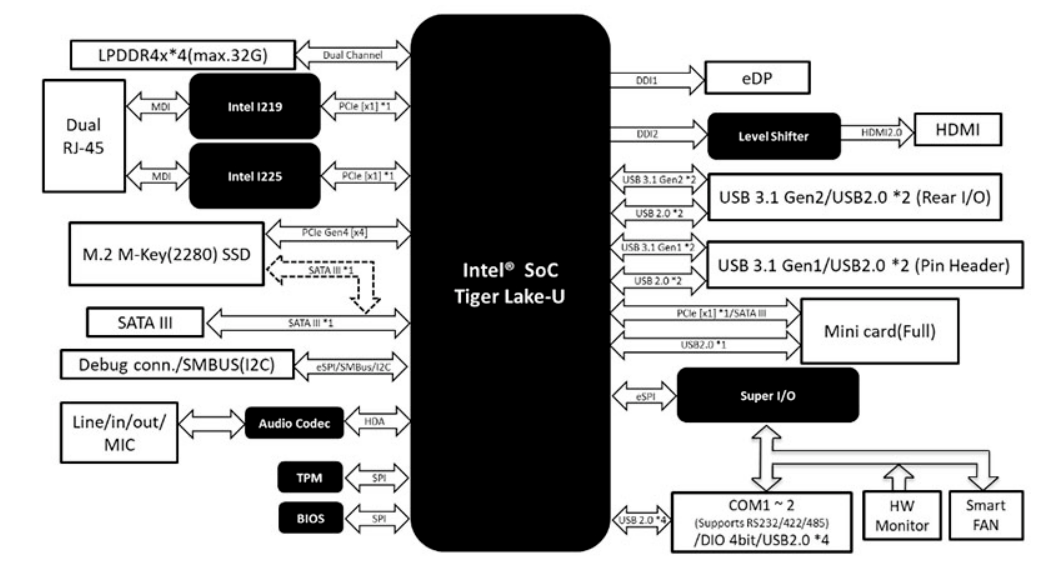


Рисунок 1.1 – Схема процессора архитектуры *Tiger Lake*

Конкретно *Intel Core i5 1135G7* имеет 4 ядра, которые при помощи технологии *hyper-threading* могут работать в режиме 8-ми потоков. Так как этот процессор разработан в первую очередь для ноутбуков, то он имеет пониженное тепловыделение от 12 до 28 *W* (для сравнения *Intel Core i5-10400F*, спроектированный для стационарных компьютеров, имеет тепловыделение в 65 *W*). Процессор обладает встроенным графическим ядром и работает с 64-битными инструкциями [3].

Остальные характеристики представлены в таблице 1.1.

Таблица 1.1 – Характеристики процессора *Intel Core i5 1135G7*

|  |  |
| --- | --- |
| Тактовая частота | 0.9 – 2.4 ГГц |
| Количество ядер/потоков | 4/8 |
| Скорость шины | 4 GT/s |
| Кэш первого уровня | 128+192 Кбайт |
| Кэш второго уровня | 5 Мбайт |
| Максимальный объем памяти | 64 Гб |
| Поддерживаемые типы памяти | DDR4-3200, LPDDR4x-4267 |

**Структура и архитектура вычислительной системы *CUDA*-ядер:** *CUDA* (Compute Unified Device Architecture) – это платформа, разработанная компанией *NVIDIA*, которая позволяет использовать возможности графических процессоров (*GPU*) для повышения производительности параллельных вычислений. Вместо традиционного подхода, когда CPU выполняет все вычисления, *CUDA* позволяет распределить нагрузку и использовать параллельные возможности *GPU* для выполнения большого количества вычислительных операций одновременно. Подобные вычисления нашли применение во многих областях, включая научные и инженерные расчеты, машинное обучение, обработку изображений, обработку сигналов и финансовые моделирования.

Одной из причин популярности *CUDA* является его высокая производительность, благодаря возможности выполнения тысяч параллельных задач на *GPU* одновременно. Среди особенностей *CUDA* можно выделить возможность написания программ на языке программирования *C* или *C++*, который совместим с *GPU*. Это позволяет разработчикам использовать уже существующие навыки и инструменты, необходимые для создания приложений, и расширить их для работы с параллельными вычислениями. Однако, есть и некоторые ограничения при использовании *CUDA*. Прежде всего, не все алгоритмы и задачи могут быть эффективно распараллелены и исполнены на *GPU*. Кроме того, использование *CUDA* требует наличия совместимой графической карты от *NVIDIA*, что может быть ограничением для некоторых пользователей. Еще одним фактором является сложность разработки и отладки программ для параллельных вычислений, требующая дополнительных знаний и умений.

*CUDA*-ядра используются в научных исследованиях, в машинном обучении, в обработке изображений, видео, в криптографии.

Одно *CUDA*-ядро имеет модули как для работы с целыми числами, так и для работы с числами с плавающей запятой. Схема одного *CUDA-*ядра представлена на рисунке 1.2 [3].

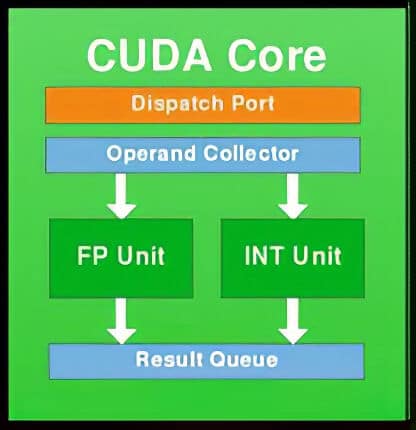


Рисунок 1.2 – Схема *CUDA*-ядра

*CUDA*-ядра имеют блоки общей памяти, но все ядра способны выполнять лишь одну инструкцию. Таким образом, графические сопроцессоры *CUDA*-ядер архитектурно спроектированы для обработки массивов данных одинаковыми операциями.

Используемый в данной работе сопроцессор *NVidia GeForce GTX 1050ti* использует архитектуру *Pascal.* В ней *CUDA*-ядра объединены в блоки обработки (*Processing blocks*). Пара из блоков обработки составляет поточный мультипроцессор (*Streaming Multiprocessor*). В каждом мультипроцессоре есть области памяти для инструкций и так называемая *Shared Memory* – область памяти, к которой могут обратиться все ядра данного стримингового мультипроцессора [4].

Пара из стриминговых мультипроцессоров образует кластер обработки текстур (*Texture Processing Claster*), в то время как несколько *TPC* образуют кластер обработки графики (*Graphic Processing Unit*). В архитектуре *Pascal* на один кластер обработки текстур приходится 128 *CUDA*-ядер [5].

В графическом сопроцессоре сопроцессор *NVidia GeForce GTX* 1050*ti* находится два кластера обработки графики, и каждый состоит из трех кластеров обработки текстур. Таким образом, при 128 *CUDA-*ядрах в одном кластере обработки текстур, данная видеокарта обладает 768 *CUDA*-ядрами. Схема графического сопроцессора приведена на рисунке 1.3 [6].

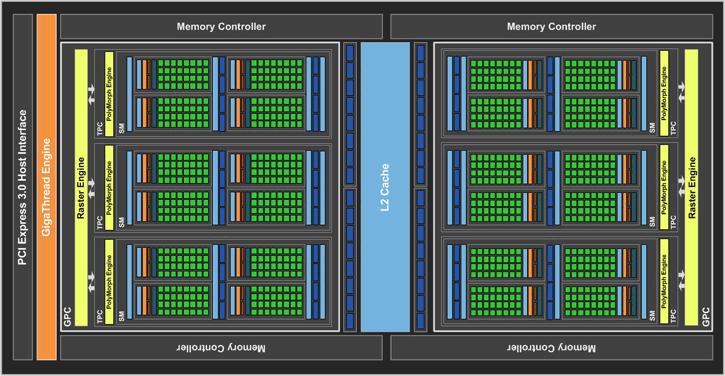


Рисунок 1.3 – Архитектура *Pascal* в видеокарте *NVidia GeForce GTX 1050ti*

**Обоснование выбора вычислительной системы**: данные аппаратные платформы были выбраны для сравнения так как они принципиально различны: процессор от *intel*  спроектирован для выполнения максимально возможного числа задач с максимальной скоростью и не большим числом исполняемых на процессоре потоков, в то время как архитектура сопроцессоров с *CUDA*-ядрами подразумевает использование этих сопроцессоров для параллельной однотипной обработки больших объемов данных.

**2 ПЛАТФОРМА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ**

**Инструменты разработки на платформе CUDA:** *CUDA* – платформа параллельных вычислений на графических процессорах, разработанная компанией *NVIDIA*. Она позволяет использовать возможности *GPU* для ускорения вычислений и повышения производительности различных задач, таких как научные исследования, анализ данных, глубокое обучение и другие. Основной элемент платформы – язык программирования *CUDA C/C++*, который является расширением стандартного языка *C/C++*. Он позволяет разрабатывать высокопроизводительные приложения, используя специальные конструкции, которые выполняются параллельно на множестве вычислительных потоков на графическом процессоре.

*CUDA* поддерживает различные типы *GPU*, начиная с архитектуры *Tesla* и вплоть до современных моделей. На платформе также доступны различные инструменты разработки, включая компиляторы, отладчики, профилировщики и библиотеки, которые упрощают разработку и оптимизацию приложений, весь этот комплекс объединяется под названием *CUDA Toolkit*.

Как можно видеть на рисунке 2.1, *CUDA* обеспечивает два *API*:

* высокоуровневый *API*: *CUDA Runtime API*;
* низкоуровневый *API*: *CUDA Driver API*.

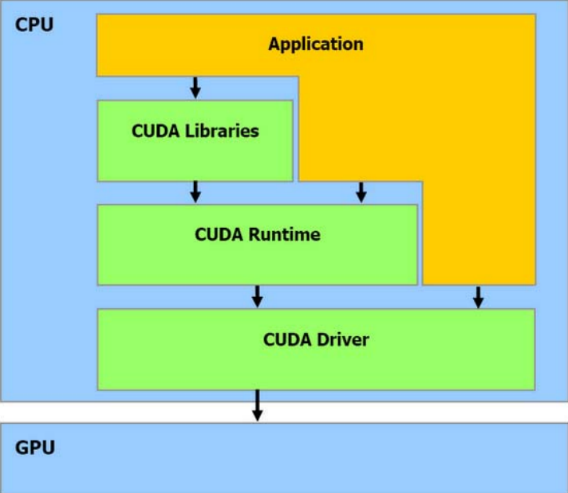


Рисунок 2.1 – схема программной модели *CUDA*

Поскольку высокоуровневый *API* реализован над низкоуровневым, каждый вызов функции уровня *Runtime* разбивается на более простые инструкции, которые обрабатывает *Driver API*. Необходимо обратить внимание, что два *API* взаимно исключают друг друга: программист может использовать один или другой *API*, но смешивать вызовы функций двух *API* не получится. Вообще, термин «высокоуровневый *API*» относителен. Даже *Runtime API* таков, что многие сочтут его низкоуровневым; впрочем, он всё же предоставляет функции, весьма удобные для инициализации или управления контекстом.

С *Driver API* работать ещё сложнее; для запуска обработки на *GPU* потребуется больше усилий. С другой стороны, низкоуровневый *API* более гибок, при необходимости предоставляя программисту дополнительный контроль.

Библиотека *CUDA Math Library* – это проверенный высокоточный набор стандартных математических функций. Доступная для любого приложения *CUDA* C или *CUDA* C++, просто добавив «*#include math.h*» в исходный код, библиотека *CUDA Math Library* гарантирует, что ваше приложение получит преимущества от высокопроизводительных математических процедур, оптимизированных для каждой архитектуры графического процессора *NVIDIA* [7].

**Средство профилирования *NVIDIA* *Visual Profiler:*** *NVIDIA* *Visual Profiler* – это графический инструмент профилирования, который отображает хронологию загрузки *CPU* и *GPU* во время работы вашего приложения. Программа автоматически анализирует *GPU*-ядра и помогает определить возможности для оптимизации.

В *Visual Profiler* можно одновременно открыть несколько временных шкал на разных вкладках. На рисунке 2.2 показана шкала для приложения *CUDA*.

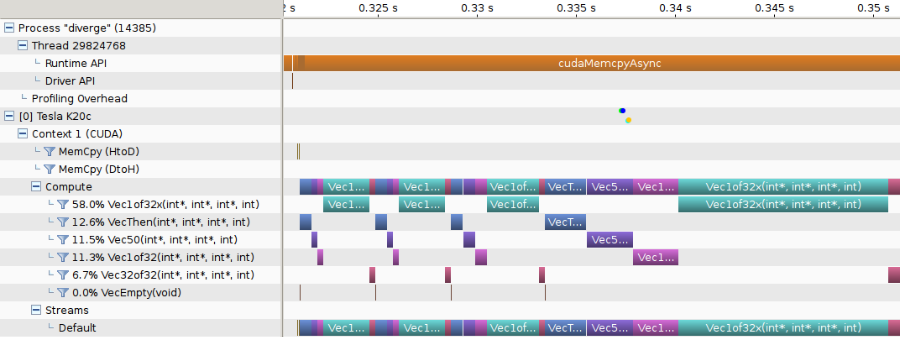


Рисунок 2.2 – Пользовательский интерфейс *NVIDIA* *Visual Profiler*

В верхней части находятся горизонтальные отметки времени, прошедшего с начала профилирования приложения. В левой части отображены единицы исполнения: процесс (*process*), потоки (*thread*), *GPU* (*device*), контексты (*context*), ядра (*kernel*), стримы (*stream*) и т.д. В центре показаны строки, отражающие активность отдельных элементов. Каждая строка отображает интервалы времени между началом и окончанием каких-либо процессов. Например, строки напротив ядер показывают время начала и окончания выполнения этого ядра.

*Analysis view* отображает результаты анализа приложения. Доступны два режима: управляемый и неуправляемый. В управляемом режиме система проводит несколько этапов анализа, чтобы помочь понять слабые места в производительности и указать на возможности оптимизации приложения. В неуправляемом режиме можно самостоятельно запустить необходимые этапы и изучить их результаты. На рисунке 2.3 показан вид управляемого анализа. В левой части находятся пошаговые инструкции, которые помогут проанализировать и оптимизировать ваше приложение. Правая часть показывает подробные результаты и аналитическую инфографику.

Неуправляемый анализ содержит список доступных процессов, каждый из которых можно запустить вручную и увидеть результат.

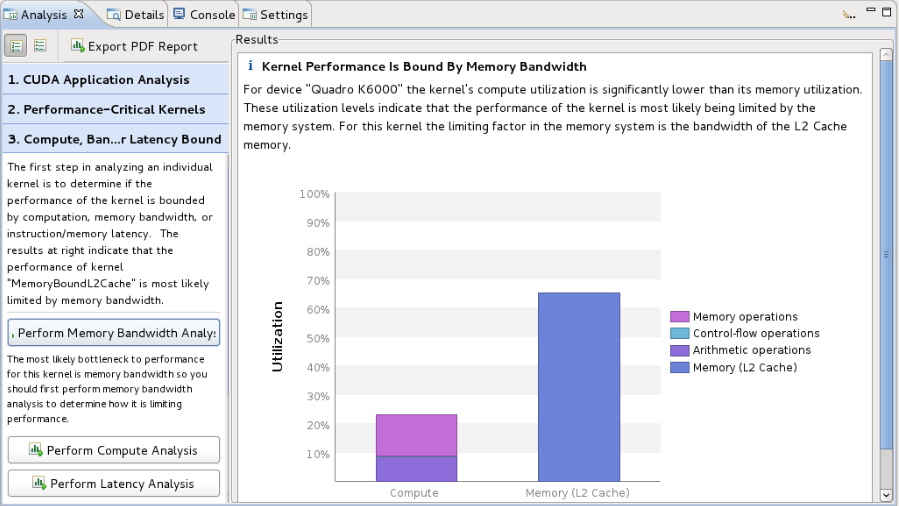


Рисунок 2.3 – Вид управляемого режима

*GPU Details View* показывает таблицу с информацией о каждом копировании памяти (*memcpy*) и запуске ядра (*kernel*) в профилируемом приложении. Для ядер в столбцах показаны соответствующие метрики и события (см. рисунок 2.4).

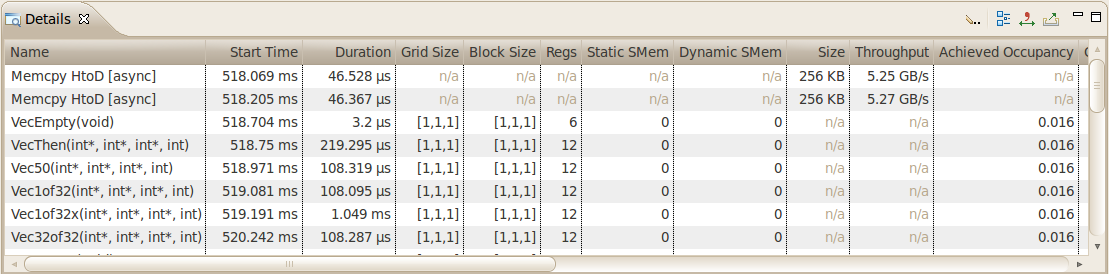


Рисунок 2.4 – Пользовательский интерфейс *GPU Details View*

**Операционная система *Ubuntu:*** *Ubuntu –* это операционная система, использующая ядро *Linux*. При выполнении программ на *CPU* эта операционная система дает ряд преимуществ: профилировщик производительности *htop*, позволяющий посмотреть нагрузку на процессор для каждого приложения, и малое количество фоновых процессов. Малое количество фоновых процессов системы позволяет демонстрировать более высокую и стабильную производительность, чем на операционных системах семейства *windows*.

***OpenMP***: Для программирования многопоточных программ будет использоваться *OpenMP*. *OpenMP –* открытый стандарт для распараллеливания программ на языках *C*, *C*++ и *Fortran*. Даёт описание совокупности директив компилятора, библиотечных процедур и переменных окружения, которые предназначены для программирования многопоточных приложений на многопроцессорных системах с общей памятью. Для использования *OpenMP* необходимо добавить директивы в код программы, указывающие компилятору, какие участки кода можно выполнять параллельно. При компиляции с поддержкой *OpenMP*, компилятор автоматически генерирует код, который может выполняться параллельно на нескольких ядрах процессора. Это позволяет ускорить выполнение программы и повысить ее производительность [10].

Выбранные платформы программного обеспечения позволят изучить программы с точки зрения использования возможностей аппаратных платформ, благодаря чему можно будет сделать выводы об производительности программ, эффективности использования ресурсов и целесообразности их использования для решения практических задач.

**3 ТЕОРЕТИЧЕСКОЕ ОБОСНОВАНИЕ РАЗРАБОТКИ ПРОГРАММНОГО ПРОДУКТА**

**Теоретические сведения:** дифференциальное уравнение второго порядка имеет вид:

,

где – непрерывные на некотором отрезке функции.

Пусть заданы условия

(2)

Решением краевой задачи для ДУ второго порядка является такая функция , которая удовлетворяет условиям (1) [10].

**Используемый метод решения:** существует множество методов численного решения краевой задачи, однако наиболее интересным для изучения можно считать интегральный метод наименьших квадратов.

В методе наименьших квадратов конечное численное решение представляется в виде суммы некоторых многочленов с разными коэффициентами :

При этом важно задать многочлены так, чтобы первый многочлен удовлетворял краевым условиям, а остальные в крайних точках обращались в ноль. Таким образом за значения в крайних точках отвечает только первый многочлен, в то время как остальные пытаются соответствовать позволяют вычислить значение решения в остальных точках на исследуемом промежутке.

Сумма (3) подставляется в выражение (1):

Далее вводится понятие невязки – разницы между левой и правой частью выражения (4).

Далее на интервале из условий (2) вычисляется интеграл квадрата невязки

После этого вычисляются производные по каждому коэффициенту и приравниваются к нулю.

Таким образом минимизируется разница между аналитическим решением и численным в виде суммы многочленов с коэффициентами. Производная равная нулю показывает, что интеграл квадрата невязки не зависит от коэффициента . Продифференцировав по каждому коэффициенту, мы получим систему линейных алгебраических уравнений.

Таким образом в конце решение сведется к обычному решению СЛАУ, что является стандартной изученной операцией над матрицами, однако в процессе вычисления квадрата невязки будут применены нестандартные методы обработки матриц, так как матрицы в данных операциях будут представлять многочлены.

**Описание алгоритма:** на вход в функцию с основным алгоритмом поступают функции в виде некоторых вычисляемых объектов, число многочленов в функциональном ряду, для которых мы ищем коэффициенты, вызываемый объект *memb*, который позволяет генерировать члены функционального ряда, число отрезков на изучаемом промежутке и сам заданный промежуток.

Первым шагом алгоритма является численное вычисление интеграла квадрата невязки (6). Для этого необходимо вычислить матрицы, описывающие значения постоянных коэффициентов при в квадрате невязки. Для приведения данной задачи к вычисляемому виду проведем следующие действия. Допустим имеется ряд многочленов При численном интегрировании нам необходимо вычислить значение коэффициентов при в выбранных точках промежутка. Подставляя в интересующие нас точки мы получим . Запишем полученное выражение в виде матрицы , где , а остальные члены равны 0. Таким образом, на данном шаге матрица представляет собой запись выражения. Для получения матрицы, описывающей квадрат невязки, создадим новую матрицу , в которой . Данная матрица также описывает многочлен, но в отличии от предыдущей элементы данной матрицы описывают постоянные коэффициенты при произведении двух , то есть чтобы узнать полученный в квадрате невязки коэффициент при произведении достаточно сложить элементы матрицы . Однако данное правило не действует на , так как элемент полноценно описывает интересующий нас коэффициент.

Проведя данный набор операций на заданном количестве точек, мы сможем численно посчитать интеграл, который будет представлять сумму матриц и описывать полученный многочлен по тем же правилам, что и матрица , так как этот интеграл получается простыми арифметическими операциями над матрицами: умножение на число и сложение.

Так как к первому многочлену коэффициент искать не нужно, потому что он обязан соответствовать краевым условиям, то его столбцы и строки нужно трактовать как произведение коэффициента исключительно на элемент матрицы, то есть представляет собой свободный коэффициент, а и – коэффициенты не при произведении двух , номера которых определяются строкой и столбцом элемента, а коэффициенты при . в таком случае представляет свободный коэффициент.

Далее в используемом методе необходимо произвести дифференцирование по каждому . Используя правила дифференцирования и особенности записи коэффициентов многочленов в матрице, достаточно будет просто обратить в ноль все элементы матрицы, у которых номер строки или столбца не является номером , по которому производится дифференцирование. Из полученных таким образом матриц получаются уравнения системы.

Получив систему линейных алгебраических уравнений, решаем ее методом Гаусса и получаем искомые коэффициенты. Умножив на них многочлены из ряда, получаем численное решение заданной системы.

Базовая схема алгоритма приведена в приложении В.

**Программная модель *CUDA***: В *CUDA* используются следующие основные термины и отношения между ними:

1. Хост (*host*) – центральный процессор, управляющий выполнением программы.
2. Устройство (*device*) — видеоадаптер, выступающий в роли сопроцессора центрального процессора.
3. Грид (*grid*) – объединение блоков, которые выполняются на одном устройстве.
4. Блок (*block*) – объединение тредов, которое выполняется целиком на одном *SM*. Имеет свой уникальный идентификатор внутри грида.
5. Тред (*thread*, поток, нить) – единица выполнения программы. Имеет свой уникальный идентификатор внутри блока.
6. Варп (*warp*) – 32 последовательно идущих треда, выполняется физически одновременно.
7. Ядро (*kernel*) – параллельная часть алгоритма, выполняется на гриде.

На центральном процессоре (хосте) выполняются только последовательные части алгоритма программы, подготовка и копирование данных на устройство, задание параметров для ядра и его запуск. Параллельные части алгоритма оформляются в ядра, которые выполняются на большом количестве тредов на устройстве [11].

Для разработки программ на *CUDA* были выпущены расширения для языка *C*, компилятор *NVCC* для сборки таких программ и расширение *\*.cu* для файлов, содержащих *CUDA* инструкции.   
 Обобщая вышесказанное, *CUDA* – расширение языков программирования *C/C++* с аннотациями для отличия от кода хоста, а также аннотациями для типов памяти данных, которые существуют на графическом процессоре. Синтаксис подобных функций похож на обычные функции языка *C*, однако позволяет дополнительно указать матрицу потоков *GPU*, которые должны выполнить вызываемую задачу. В течение своего жизненного цикла хост-процесс может отправлять множество параллельных задач графического процессора.

К расширениям языка *C* относятся:

* спецификаторы для функций и переменных (см. таблицы 3.1, 3.2);
* новые встроенные типы данных;
* встроенные в ядро переменные;
* директива для запуска ядра из *C* кода.

Таблица 3.1 – Спецификаторы функций в CUDA

|  |  |  |
| --- | --- | --- |
| Спецификатор | Выполнение осуществляет | Вызов осуществляет |
| \_\_*device*\_\_ | устройство | Устройство |
| \_\_*global*\_\_ | устройство | Хост |
| \_\_*host*\_\_ | хост | Хост |

Спецификаторы *\_\_host\_\_* и *\_\_device\_\_* могут применяться одновременно для задания функций, выполняющихся и на хосте, и на устройстве (обе версии кода создаются компилятором)

Спецификатор *\_\_global\_\_* применяется при задании ядра (дополнительно передается несколько специфических переменных). Функции с данным спецификатором могут возвращать только void. Спецификатор применяется исключительно обособленно.

Таблица 3.2 – Спецификаторы переменных в CUDA

|  |  |  |  |
| --- | --- | --- | --- |
| Спецификатор | Нахождение | Место доступа | Режим доступа |
| \_\_*device*\_\_ | устройство | устройство | Чтение |
| \_\_*constant*\_\_ | устройство | устройство и хост | чтение и запись |
| \_\_*shared*\_\_ | устройство | блок | чтение и запись |

Спецификатор *\_\_device\_\_* является аналогом *const* на *CPU*.

Запись в память со спецификатором *\_\_constant\_\_* может быть осуществлена только из специальных функций в *CPU*.

Спецификатор *\_\_shared\_\_* применяется для задания разделяемой памяти и не может быть инициализирован при объявлении. Как правило, требует явной синхронизации тредов внутри одного блока после любой операции с такой памятью.

Для удобства копирования и доступа к данным в *CUDA* было добавлено несколько векторных типов. Типы *(u)char, (u)int, (u)short, (u)long* могут быть 1-,2-,3-,4-мерными векторами, *long long* и *double* – только 1-,2-мерными. Для их создания используется функция *make*\_(тип)(размерность), например, *make\_int2*(*1, 2*). Также существует тип *dim3*, имеющий привычный для синтаксиса *C++* конструктор, позволяющий определить компоненты вектора лишь частично (при этом недостающим компонентам будет присвоено значение 1). Этот тип данных используется для задания параметров ядра.

В *CUDA* имеются несколько особых переменных, которые существуют внутри каждого вычислительного ядра. Эти переменные позволяют отличать один тред от другого:

1. *dim3* *gridDim* – содержит в себе информацию о конфигурации грида при запуске ядра;
2. *uint3* *blockIdx* – координаты текущего блока внутри грида;
3. *dim3* *blockDim* – размерность блока при запуске ядра;
4. *unit3* *threadIdx* – координаты текущего треда внутри блока;
5. *int* *warpSize* – размер варпа.

Для Запуска ядра используются специальные директивы параметров запуска ядра и передачи ему аргументов.

Объявление функции ядра с параметром *params*:

*\_\_global\_\_ void kernel\_name(params)*;

Запуск ядра: *kernel\_name<<<grid, block, mem, stream>>>(params)*, где

*dim3 grid* – размер грида для запуска, может быть 1-, 2-,3-мерным, а максимальное значение по каждому измерению (2147483647, 65535, 65535).

*dim3 block* – размер блока при запуске. Блок может быть 1-, 2-, 3-мерым, причем максимальное значение по каждому измерению (1024, 1024, 64), а максимальное число тредов 1024 [8].

*size\_t mem* – количество памяти на блок, которая выделяется для данного запуска под динамическое использование внутри ядра.

*cudaStream\_t stream* – описание потока, в котором запускается ядро

В *CUDA* есть два уровня *API*: низкоуровневый *driver-API* и высокоуровневый *runtime-API*. *Runtime-API* реализован через *driver-API*.

*Runtime-API* обладает меньшей гибкостью, но более удобен для написания программ. Оба *API* не требуют явной инициализации, и для использования дополнительных типов и других расширений языка *С* не требуется подключать дополнительные заголовочные файлы. Все функции *driver-API* начинаются с приставки *cu*, все функции *runtime-API* начинаются с приставки *cuda*. Практически все функции из обоих *API* возвращают значение типа *t\_cudaError*, которое принимает значение *cudaSuccess* в случае успеха.

Функции в *API* делятся на синхронные и асинхронные. Синхронные запуски являются блокирующими, асинхронные же позволяют выполнять другие операции во время их выполнения. Процессы копирования, запуска ядра, инициализация памяти могут быть асинхронными.

При использовании асинхронных вызовов необходимо всегда помнить о синхронизации перед использованием их результатов.

Основные функции *runtime-API*:

*char\* cudaGetErrorString*(*cudaError\_t*) – функция возвращает описание ошибки по её числовому коду.

1. *cudaError\_t cudaGetLastError*() – определение последней произошедшей ошибки, часто используется после вызова ядра для проверки правильности его работы.
2. *cudaError\_t cudaThreadSynchronize*() – команда синхронизации, необходимая после любого асинхронного вызова.
3. *cudaError\_t cudaGetDeviceCount*(*int*\*) – функция, определяющая количество устройств, доступных для вычисления под *CUDA*.
4. *cudaError\_t cudaGetDeviceProperties* (*cudaDeviceProp\* props, int deviceNo*) – функция, определяющая параметры конкретного устройства по его номеру. В частности, функция позволяет определить *ComputeCapability* из полей *major* и *minor* структуры *cudaDeviceProp*.

**Работа с памятью на *GPU*:** Физически память *GPU* разделяется на *DRAM* (*dynamic random access memory* – динамическая память с произвольным доступом) и на мультипроцессорную, размещенную непосредственно на самом графическом процессоре.

Иерархически память делится на 6 типов:

* регистры;
* локальная;
* разделяемая;
* глобальная;
* текстурная;
* константная.

Наиболее простым видом памяти является регистровая память. Каждый потоковый мультипроцессор содержит 8192 или 16384 32-битовых регистра. На этапе компиляции они распределяются между тредами блока, тем самым влияя на количество исполняемых блоков одним мультипроцессором. Каждый тред получает в свое монопольное пользование некоторое количество регистров, которые доступны как на чтение, так и на запись (*read*/*write*). Тред не имеет доступа к регистрам других тредов, но свои регистры доступны на протяжении выполнения данного ядра.

Поскольку регистры расположены непосредственно в потоковом мультипроцессоре, то они обладают максимальной скоростью доступа. Если имеющихся регистров не хватает, то для размещения локальных данных (переменных) треда используется так называемая локальная память, размещенная в *DRAM*. Поэтому доступ к локальной памяти характеризуется очень высокой латентностью – от 400 до 600 тактов.

Локальная память – это область в глобальной памяти, выделенная компилятором для хранения локальных значений потоков. Локальная память используется для хранения локальных данных потоков в случае нехватки регистров или объявления локальных массивов внутри ядра без ключевого слова \_\_*shared*\_\_.

Следующим типом памяти является разделяемая (*shared*) память. Эта память расположена непосредственно в мультипроцессоре, но она выделяется на уровне блоков – каждый блок получает в свое распоряжение одно и то же количество разделяемой памяти. Всего каждый мультипроцессор содержит 16 кбайт разделяемой памяти, и от того, сколько разделяемой памяти требуется блоку, зависит количество блоков, которое может быть запущено на одном мультипроцессоре. Разделяемая память обладает очень большой латентностью, доступна всем тредам блока, как на чтение, так и на запись.

Глобальная память – это обычная *DRAM*-память, которая выделяется при помощи специальных функций на *СPU*. Все треды сетки могут читать и писать в глобальную память. Поскольку глобальная память расположена на центральном процессоре, а не на графическом, то она обладает высокой латентностью (от 400 до 600 тактов).

Ниже в таблице 3.3 описаны вышеуказанные типы памяти с их характеристиками.

Таблица 3.3 – Характеристика типов памяти *GPU*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Тип памяти | Наименование характеристики | | | |
| Расположение | Кэширование | Доступ | Время жизни |
| Регистры | Мультипроцессор | нет | *r/w* | Тред |
| Локальная | *DRAM* | нет | *r/w* | Тред |
| Разделяемая | Мультипроцессор | нет | *r/w* | Блок |
| Глобальная | *DRAM* | нет | *r/w* | Выделяется *CPU* |
| Текстурная | *DRAM* | да | *r/o* | Выделяется *CPU* |
| Константная | *DRAM* | да | *r/o* | Выделяется *CPU* |

Как уже упоминалось ранее, для выделения памяти на устройстве можно использовать функцию *cudaMalloc*, а для освобождения – *cudaFree*. Рассмотрим еще некоторые очень полезные функции для управления памятью.

Первая из них это *cudaMallocPitch* – функция рекомендуется для выделения памяти под двумерный массив для того, чтобы память была соответствующим образом выровнена. Следовательно, будет уверенность в улучшении производительности при доступе к памяти и при копировании между двумерными массивами или другими регионами памяти устройства.

Под массивы память можно выделить функцией *cudaMallocArray* и освободить функцией *cudaFreeArray*. Функция *cudaMallocArray* требует описание формата созданного массива, созданное функцией *cudaCreateChannelDesc.*

Функция *cudaGetSymbolAddress* используется, чтобы получить адрес переменной, расположенной в глобальной памяти. Размер выделенной памяти доступен через функцию *cudaGetSymbolAddress*.

Существуют различные функции для копирования регионов памяти, например, для копирования двумерного массива используется функция *cudaMemcpy2DtoArray*.

Каждый поток копирует свою порцию данных входного массива *hostPtr* в массив *inputDevPtr* в памяти устройства, *inputDevPtr* обрабатывается на устройстве посредством вызова *myKernel*, и копирует результат *outputDevPtr* назад в *hostPtr*.

Используемый алгоритм сводит такие операции высокого уровня как взятие интеграла и производной от суммы многочленов с коэффициентами к простой обработке двумерных массивов, которыми в своей сути являются матрицы. А возможности программной модели *CUDA* позволяют перенести любые операции для обработки данных, написанные на знакомом многим разработчикам языке *C*++, на графический сопроцессор, не ограничивая разработчика в возможностях работы с сопроцессором.

**4 ПРОЕКТИРОВАНИЕ ФУНКЦИОНАЛЬНЫХ ВОЗМОЖНОСТЕЙ ПРОГРАММЫ**

Программное обеспечение, разрабатываемое в рамках данного курсового проекта, представляет собой программы численного решения краевой задачи на процессоре и на графическом сопроцессоре (*GPU*) с использованием архитектуры *CUDA.* Каждая программа имеет две версии: с учетом возможностей распараллеливания вычислений и нет. В приложении А приведен исходный код программы.

Программа решают установленные в них уравнения и засекают время, потраченное на решение конкретного уравнения при заданных параметрах решения. В качестве параметров решения выступают количество многочленов в функциональном ряду, для которых необходимо найти коэффициенты, и количество точек, на которых программа будет вычислять численное значение интеграла.

**Структура программы:** программы разработаны на языке программирования *C*++. Программы состоят и трех классов для описания матриц (класс *Matrix*), используемых многочленов (класс *Polynom*) и функционального ряда многочленов (класс *FuncRow).* В силу особенностей программы, отличия между программами будут только в основном файле *Source.cpp,* а также в интерфейсе класса *Matrix* и реализации его методов.

**Основные управляющие функции**: Программа начинается с объявления условий решаемых дифференциальных уравнений: отрезка, на котором происходит решение, значений искомой функции в крайних точках данного отрезка, количества членов в функциональном ряду и условий дифференциального уравнения. Условия заданы помощи лямбда-функций, которые как функциональные объекты передаются в функцию *dotest.* Эта функция выводит номер решаемого уравнения, время выполнения основного алгоритма, вызываемого после записи текущего времени в миллисекундах. После вывода времени, функция вызывает функцию *ShowData,* вычисляющую норму невязки между фактическим решением тестового примера и полученным приближенным решением.

**Другие функции в *Source.cpp*:**  остальные функции в главном файле служат для разграничивания кода и повышения его читаемости. Функция *IntSimpson* вычисляет интеграл при помощи метода Симпсона. Функция *Anis* является генератором многочленов. Функция *Mult* возвращает матрицу, описывающую произведение многочленов переданных матриц. *DiffSquare* служит для получения матрицы производной по номеру переданного коэффициента, а функция *backMove*, принимая треугольную матрицу, возвращает решение описываемого ей СЛАУ.

**Класс *Polynom*:**  класс *Polynom*описывает используемые в решении многочлены. Главным его полем является приватное поле *\_polynom*. Оно является статическим массивом длины 3, описывающем коэффициенты при соответствующих степенях. Так как массив статический и используется для всех степеней, есть поле *\_base*, для записи степени первого элемента в массиве *\_polynom.* Получить степень текущего многочлена можно при помощи метода *getPow()*.Также имеется метод *IncreasePow()* для повышения степени многочлена. Имеются перегруженные операторы деления, умножения, сложения, обращения по индексу, конструкторы копирования и ­*rvalue*-конструктор. Важными для программы являются методы *Diff()* и *Count().* Первый возвращает многочлен, являющийся производной от текущего, в то время как второй вычисляет значение многочлена в точке.

**Класс *FuncRow*:** класс *FuncRow* служит для инкапсуляции массива многочленов, представляющих функциональный ряд. Он позволяет обратиться к конкретному многочлену через оператор индексации, а также позволяет вычислить значение функционального ряда в точке при помощи метода *Count()*. Стоит отдельно рассмотреть конструктор данного класса. Он принимает количество членов в данном ряду и функциональный объект, возвращающий объект класса *Polynom* по переданному номеру в последовательности. Это позволяет задать функцию-генератор многочленов и создавать функциональные ряды с любым количеством многочленов. В деструкторе данного класса очищается созданный динамический массив многочленов.

**Класс *Matrix*:** класс *Matrix* инкапсулирует двумерный массив и предоставляет методы для обработки матриц. Реализации методов зависят от типа исполняемой программы: на *GPU*или на*CPU*, с распараллеливанием вычислений и без. Однако интерфейс класса остается почти неизменным. Класс имеет конструктор без параметров, конструктор копирования, *rvalue-*конструктор и соответствующие операторы присваивания. Особенно полезным являются *rvalue*-конструктор и оператор присваивания по *rvalue*. Благодаря возможностям языка *С++* и механизму *rvalue***-**ссылок, позволяющему отличить временные объекты, созданные для передачи своего значения другим, можно избежать большого числа вызовов операторов и конструкторов копирования, которые при больших размерностях матрицы серьёзно сказываются на производительности [12]. Класс инкапслуирует значения количества строк и столбцов матрицы. Класс обладает перегруженными операторами сложения для двух матриц, а также деления и умножения матрицы на число. Данные операторы позволяют сделать частые операции над матрицами естественной частью кода [13]. Для решения слау в классе предусмотрен метод *ToUpTriangle()*, приводящий матрицу к треугольной, что необходимо для решения СЛАУ методом Гаусса. Данный метод работает на матрицах, у которых число столбцов на единицу больше чем число строк, так как в последнем столбце хранятся свободные члены. В обычной реализации класса на *CPU* и одном рабочем потоке все операции с матрицами организованы через циклы *for* для перебора всех значений.

**Класс *Matrix* в программе с максимальным использованием *GPU*:**в данной реализации класса максимально используются возможности *CUDA.* В данной реализации есть 2 ключевых отличия. Первое отличие состоит в том что матрица представляет собой не двумерный массив, как было в предыдущих реализациях, а одномерный массив. Второе отличие –одномерный массив матрицы хранится в видеопамяти, поэтому некоторые функции из основной программы пришлось перенести в данную реализацию класса *Matrix***,** в которой можно обеспечить наиболее удобный доступ к элементам массива. Все операции по работе с матрицами происходят в *kernel*-функциях *CUDA* для параллельной обработки данных. При вызове *kernel****-***функций используется количество блоков и потоков равное количеству строк и колонок матрицы соответственно. На больших значениях это обеспечит оптимальную производительность, а также упростит адресацию внутри самой функции.

**Класс *Matrix* в программе с минимальным использованием *GPU*:** в данной реализации класса *Matrix* в *kernel*-функцияхколичество блоков и потоков равно 1 для использования минимально возможного количества *CUDA*-ядер. Все операции с проходом по массиву реализованы через циклы.

**5 АНАЛИЗ ПОЛУЧЕННЫХ РЕЗУЛЬТАТОВ**

**Зависимость времени выполнения от количества точек:** алгоритм принимает два параметра: количество членов в аппроксимирующем ряде и количество точек для вычисления интеграла. В таблице 5.1 представлено среднее время решения четырех краевых задач с количеством многочленов равным 1000 и с изменяющимся количеством точек.

Таблица 5.1 – Зависимость скорости решения ДУ от количества точек для расчета интеграла

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Среднее время решения ДУ, мс | | | |
| Количество точек | *CPU* | | *GPU* | |
| *Multi Thread* | *One Thread* | *Multi Thread* | *One Thread* |
| 50 | 1670 | 2610 | 2025 | 65185 |
| 100 | 1670 | 2595 | 1934 | 65507 |
| 200 | 1666 | 2584 | 1997 | 65619 |
| 500 | 1673 | 2596 | 1986 | 64405 |
| 700 | 1674 | 2629 | 1855 | 64944 |
| 1000 | 1686 | 2597 | 1971 | 65252 |

Как видно из данной таблицы и из графиков 1 и 2 приложения Г изменение количества точек практически никак не влияет на время выполнения программы ни на *CPU*, ни на *GPU*.

Для дальнейших замеров скорости решения будет использоваться количество точек равное 200.

**Результаты на процессоре *Intel*:** на графике 4 приложении Г представлен график зависимости среднего времени решения уравнения от количества многочленов. Время представлено в миллисекундах, на оси *х* указано количество членов.

На рисунке изображена зависимость времени выполнения программы в миллисекундах от количества многочленов в аппроксимирующем ряде. Замеры времени решения выполнялись при количестве точек равном 200.

На рисунке видно, что вычисления в распараллеленном алгоритме всегда быстрее.

**Результаты на *GPU***: на графическом сопроцессоре ситуация аналогична. Это можно увидеть на графике 3 приложения Г.

Однако стоит заметить, что разница во времени выполнения многопоточной и однопоточной программы на *GPU* значительно больше.

Это объясняется значительным сокращением используемых ядер в однопоточной программе.

**Промежуточные результаты:** на графике 3 приложения Г видно, что время выполнения на *GPU* в режиме одного потока сильно отстает от остальных программ. Поэтому на графике 4 представлена та же информация, только без результатов программы на *GPU* в однопоточном режиме. Как видно на графике 4, *GPU* в многопоточном режиме производит вычисления медленней, чем *CPU* в однопоточном и многопоточном режимах. Это объясняется недостаточным размером матрицы, при котором траты времени на организацию работы *GPU* все еще велики и не позволяют алгоритму отработать быстрее.

**Результаты при большом количестве членов в последовательности:** алгоритм, использующийся в данном проекте содержит операции над матрицами, связанные с перебором всех значений матрицы (сложение матриц, умножение матрицы на число). Так же в конце данного алгоритма необходимо решить систему линейных алгебраических уравнений. Это значит, что на больших значениях количества членов в функциональном ряде, от которого зависят размеры матрицы, идет работа с большим количеством данных. Поэтому справедливо ожидать, что графический сопроцессор, спроектированный для параллельной обработки больших массивов данных, справится быстрее чем процессор в однопоточном режиме. На графиках 5 и 6 приложения Г видно, что программа для *GPU*отработала быстрее только программы для *CPU* в однопоточном режиме. Высокая тактовая частота и технология *hyper-threading* позволила процессоре в многопоточном режиме отработать быстрее, чем *GPU*. На рисунке 5.1 показаны результаты профилирования в *NVIDIA visual profiler*.

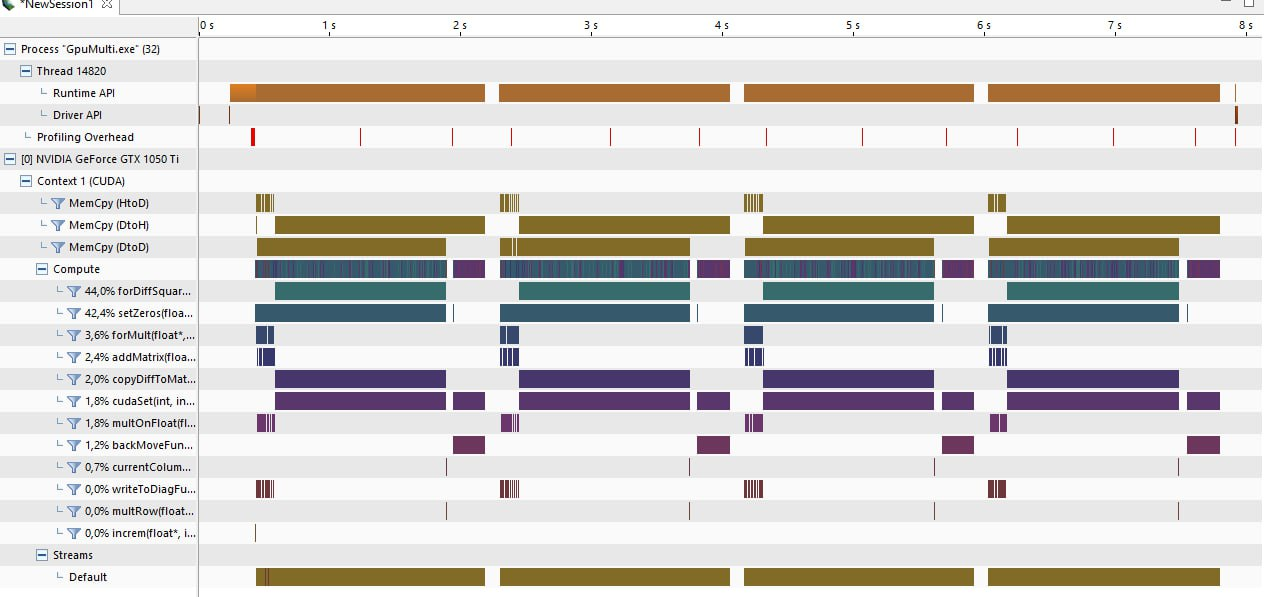


Рисунок 5.1 – Результаты профилирования

Исходя из этих данных, можно сказать что основную нагрузку на *GPU* обеспечивают опреации копирования и обращения в ноль элементов матрицы при инициализации. Это связяано с большим количеством создаваемых матриц.

**Точность решения:** предыдущие подпункты описывали лишь скорость решения краевой задачи при различном количестве многочленов, однако не брали в расчёт точность решений и оптимальность количества многочленов для численного решения. На рисунке 5.2 видно, что средняя норма отклонения от аналитического решения начинает увеличиваться, начиная с количества многочленов 50.

Рисунок 5.2 – График зависимости средней нормы невязки от количества многочленов

При большом количестве многочленов точность может значительно упасть вследствие особенностей записи чисел с плавающей точкой и возведений этих чисел в большие степени (вплоть до 1000-ой степени). На графике видно, что наилучшая точность достигается при количестве многочленов, меньшем 50. Таким образом, большие размерности матриц для данного класса задач излишни.

**ЗАКЛЮЧЕНИЕ**

В результате были изучены технологии *CUDA*, необходимые для параллельных вычислений. было проведено сравнение скорости решения краевой задачи на *CUDA*-ядрах и на процессоре *intel core* *i*5 1135*G*7. Был разработан алгоритм решения краевой задачи без использования сторонних математических библиотек. Были разработаны программы для численного решения краевой задачи разработанным алгоритмом, учитывающие возможности двух рассматриваемых архитектур.

В рамках проекта были разработаны программы, учитывающие возможности распараллеливания вычислений, и программы, работающие в режиме одного потока. *CUDA***-**ядра, вследствие особенностей архитектуры, оказались быстрее в режиме многопоточности при параллельной однотипной обработке больших массивов данных, в частности матриц размером 1000 строк на 1000 колонок. Однако для рассматриваемого класса задач подобные размерности излишни вследствие возрастающей ошибки вычислений на числах с плавающей точкой. На меньших размерах организация работы на графическом сопроцессоре и меньшая тактовая частота ядер не позволяют проводить вычисления быстрее процессора даже в режиме одного потока. Так же неэффективным будет использование программы для *GPU*, задействующую один поток выполнения. Неэффективность такого использования объясняется малой мощностью одного отдельно взятого *CUDA*-ядра.

Таким образом 4-ядерный процессор с 8 вычислительными потоками является более подходящим выбором для быстрого решения подобного класса задач. Стоит заметить, что, хоть вычисления на 8 потоках и происходят значительно быстрее, даже в режиме одного потока процессор способен достаточно быстро решать краевые задачи с достаточной точностью.

Для рассматриваемого класса задач было найдено оптимальное число многочленов для аппроксимирующего ряда – число многочленов не должно превышать 50, так как при больших временных затратах на вычисления решение теряет в точности.

**СПИСОК ЛИТЕРАТУНЫХ ИСТОЧНИКОВ**

[1] Tiger Lake: архитектура процессоров Intel Core одиннадцатого поколения [Электронный ресурс]. – Режим доступа: https://3dnews.ru/1028957/tiger-lake-arhitektura-protsessorov. – Дата доступа 02.10.2023.

[2] Intel Tiger Lake 10nm CPUs All Set To Roar on 2nd September, Official Launch & Pre-Release Architectural Presentation Confirmed [Электронный ресурс]. – Режим доступа: https://wccftech.com/intel-tiger-lake-10nm-cpus-all-set-to-roar-on-2nd-september-official-launch-pre-release-architectural-presentation-confirmed/. – Дата доступа 02.10.2023.

[3] Nvidia CUDA Cores Explained: How are they different? [Электронный ресурс]. – Режим доступа: <https://www.techcenturion.com/nvidia-cuda-cores/>. – Дата доступа 02.10.2023.

[4] История потоковых мультипроцессоров Nvidia [Электронный ресурс]. – Режим доступа: <https://habr.com/ru/articles/501108/>.– Дата доступа 03.10.2023.

[5] Гайд по Pascal: разбираемся в видеокартах NVIDIA 2016 года [Электронный ресурс]. – Режим доступа: https://habr.com/ru/articles/373023/. – Дата доступа 03.10.2023.

[6] MSI GeForce GTX 1050 & 1050 Ti Gaming X Review [Электронный ресурс]. – Режим доступа: https://www.guru3d.com/review/msi-geforce-gtx-1050-and-1050-ti-gaming-x-review/page-6/. – Дата доступа 03.10.2023.

[7] CUDA Toolkit Documentation 12.3 Update 1 [Электронный ресурс]. – Режим доступа: <https://docs.nvidia.com/cuda/>. – Дата доступа 10.10.2023.

[8] CUDA: Как работает GPU [Электронный ресурс]. – Режим доступа: <https://habr.com/ru/articles/54707/>. – Дата доступа 15.10.2023.

[9] OpenMP-4.0-C.pdf [Электронный ресурс]. – Режим доступа: https://www.openmp.org/wp-content/uploads/OpenMP-4.0-C.pdf. – Дата доступа 15.10.2023.

[10] § О24. Краевые задачи [Электронный ресурс]. – Режим доступа: http://w.ict.nsc.ru/books/textbooks/akhmerov/ode\_unicode/s-24/s-24.html. – Дата доступа 27.10.2023.

[11] Программная модель CUDA [Электронный ресурс]. – Режим доступа: https://indico-hlit.jinr.ru/event/6/attachments/104/128/talk2\_cuda\_basics

.pdf. – Дата доступа 27.10.2023.

[12] Ссылки r-value [Электронный ресурс]. – Режим доступа: https://ravesli.com/urok-190-ssylki-r-value/. – Дата доступа 13.11.2023

[13] Введение в перегрузку операторов [Электронный ресурс]. – Режим доступа: <https://ravesli.com/urok-130-vvedenie-v-peregruzku-operatorov/>. – Дата доступа 13.11.2023

ПРИЛОЖЕНИЕ А

(обязательное)

Листинг программного кода

Листинг кода

*Polynom.h:*

#pragma once

#include <string>

#include <cmath>

#include <algorithm>

float myround(float f, float prec);

class Polynom {

private:

//int \_length;

float \_pol[3] = { 0,0,0 };

int \_base;

public:

int getPow() const;

Polynom(int pow);

Polynom(const Polynom& p);

Polynom(Polynom&& p);

Polynom(float\* k);

Polynom();

~Polynom();

float Count(float x);

Polynom Diff();

operator std::string() const;

float& operator[](unsigned int i);

Polynom& operator+=(Polynom& other);

Polynom operator+(Polynom& other);

Polynom& operator-=(Polynom& other);

Polynom operator-(Polynom& other);

Polynom& operator\*=(float l);

Polynom operator\*(float l);

Polynom& operator/=(float l);

Polynom operator/(float l);

Polynom& IncreasePow(int additionalPow);

Polynom& operator=(Polynom& other);

Polynom& operator=(Polynom&& other);

};

*Polynom.cpp:*

#include "Polynom.h"

float myround(float f, float prec) {

return floorf(f \* prec) / prec;

}

float prec = 100;

const int SIZE = 3;

int Polynom::getPow() const {

return \_base + 2;

}

Polynom::Polynom(int pow) :\_base(pow - 2) {

for (int i = 0; i < SIZE; ++i) \_pol[i] = 0;

}

Polynom::Polynom(const Polynom& p) :Polynom(p.getPow()) {

for (int i = 0; i < SIZE; ++i) {

\_pol[i] = p.\_pol[i];

}

//\_base = p.\_base;

}

Polynom::Polynom(Polynom&& p) {

for (int i = 0; i < SIZE; ++i) {

\_pol[i] = p.\_pol[i];

}

\_base = p.\_base;

}

Polynom::Polynom(float\* k) :Polynom(2) {

for (int i = 0; i < SIZE; ++i) \_pol[i] = k[i];

}

Polynom::Polynom() {

\_base = 0;

}

Polynom::~Polynom() {

//delete[] \_pol;

}

float Polynom::Count(float x) {

//if ( == 0) return 0;

if (x == 0) {

if (\_base <= 0) return \_pol[0 - \_base];

else return 0;

}

float sum = 0;

float h = powf(x, \_base + 2);

for (int i = SIZE - 1; i >= 0 && i + \_base >= 0; --i, h /= x) {

if (h != INFINITY && h != NAN)

sum += h \* \_pol[i];

}

return sum;

}

Polynom Polynom::Diff() {

if (\_base <= -2) return 0;

Polynom df(getPow() - 1);

for (int i = 0; i < SIZE; ++i) {

df.\_pol[i] = \_pol[i] \* (i + \_base);

}

return df;

}

Polynom::operator std::string() const {

std::string str = " ";

for (int i = SIZE - 1; i > 0; --i) {

if (myround(\_pol[i], prec) == 0) continue;

if (myround(\_pol[i], prec) > 0 && i != SIZE - 1) str += "+";

if (abs(myround(\_pol[i], prec)) != 1) str += std::to\_string(abs(myround(\_pol[i], prec)));

else {

if (myround(\_pol[i], prec) < 0) str += '-';

}

str += "x";

if (i + \_base != 1) str += std::to\_string(i + \_base);

}

if (\_base >= 0) {

if (myround(\_pol[0], prec) != 0) {

if (myround(\_pol[0], prec) > 0) str += "+";

str += std::to\_string(myround(\_pol[0], prec));

}

}

return str;

}

float& Polynom::operator[](unsigned int i) {

return \_pol[i - \_base];

}

Polynom& Polynom::operator+=(Polynom& other) {

//int size = std::max(getPow(), other.getPow());

//float\* arr = new float(size + 1);

float arr[SIZE];

for (int i = 0; i < SIZE; ++i) \_pol[i] += other[i];

return \*this;

}

Polynom Polynom::operator+(Polynom& other) {

Polynom ret(\*this);

return ret += other;

}

Polynom& Polynom::operator-=(Polynom& other) {

for (int i = 0; i < SIZE; ++i) \_pol[i] -= other[i];

return \*this;

}

Polynom Polynom::operator-(Polynom& other) {

Polynom ret(\*this);

return ret += other;

}

Polynom& Polynom::operator\*=(float l) {

for (int i = 0; i < SIZE; ++i) \_pol[i] \*= l;

return \*this;

}

Polynom Polynom::operator\*(float l) {

Polynom ret(\*this);

return ret \*= l;

}

Polynom& Polynom::operator/=(float l) {

for (int i = 0; i < SIZE; ++i) \_pol[i] /= l;

return \*this;

}

Polynom Polynom::operator/(float l) {

Polynom ret(\*this);

return ret /= l;

}

Polynom& Polynom::IncreasePow(int additionalPow) {

// TODO: вставьте здесь оператор return

\_base += additionalPow;

return \*this;

}

Polynom& Polynom::operator=(Polynom& other) {

for (int i = 0; i < SIZE; ++i) \_pol[i] = other.\_pol[i];

\_base = other.\_base;

return \*this;

}

Polynom& Polynom::operator=(Polynom&& other) {

for (int i = 0; i < SIZE; ++i) \_pol[i] = other.\_pol[i];

\_base = other.\_base;

return \*this;

}

*FuncRow.h:*

#pragma once

#include "Polynom.h"

#include <functional>

typedef std::function<Polynom(int i)> Generator;

class FuncRow {

private:

Polynom\* \_polynoms;

int numb;

public:

FuncRow(int numb, Generator f);

FuncRow();

FuncRow(FuncRow&& frow);

FuncRow(const FuncRow& frow);

~FuncRow();

FuncRow& operator=(FuncRow&& frow);

FuncRow& operator=(FuncRow& frow);

float Count(float x);

operator std::string() const;

operator Polynom() const;

Polynom& operator[](int i) const;

};

*FuncRow.cpp:*

#include "FuncRow.h"

FuncRow::FuncRow(int numb, Generator f) {

this->numb = numb;

\_polynoms = new Polynom[numb];

for (int i = 0; i < numb; ++i) {

\_polynoms[i] = f(i);

}

}

FuncRow::FuncRow() {

\_polynoms = nullptr;

numb = 0;

}

FuncRow::FuncRow(FuncRow&& frow) {

\_polynoms = frow.\_polynoms;

frow.\_polynoms = nullptr;

numb = frow.numb;

}

FuncRow::FuncRow(const FuncRow& frow) {

\_polynoms = new Polynom[frow.numb];

for (int i = 0; i < frow.numb; ++i) {

\_polynoms[i] = frow.\_polynoms[i];

}

numb = frow.numb;

}

FuncRow& FuncRow::operator=(FuncRow&& frow) {

delete[] \_polynoms;

\_polynoms = frow.\_polynoms;

numb = frow.numb;

frow.\_polynoms = nullptr;

return \*this;

}

FuncRow& FuncRow::operator=(FuncRow& frow) {

delete[] \_polynoms;

\_polynoms = new Polynom[frow.numb];

numb = frow.numb;

for (int i = 0; i < numb; ++i) \_polynoms[i] = frow.\_polynoms[i];

return \*this;

}

FuncRow::~FuncRow() {

delete[] \_polynoms;

}

float FuncRow::Count(float x) {

float sum = 0;

for (int i = 0; i < numb; ++i) {

sum += \_polynoms[i].Count(x);

}

return sum;

}

FuncRow::operator Polynom() const {

Polynom P(0);

for (int i = 0; i < numb; ++i) {

P += \_polynoms[i];

}

return P;

}

Polynom& FuncRow::operator[](int i) const {

return \_polynoms[i];

}

FuncRow:: operator std::string() const {

std::string str = "";

for (int i = 0; i < numb; ++i) str += (std::string)\_polynoms[i] + " + ";

str += "0";

return str;

}

*Matrix.h:*

#pragma once

#include <string>

#include <iostream>

#include <thread>

class Matrix {

private:

int \_columns;

int \_rows;

static int count;

int id;

public:

float\*\* arr = nullptr;

int getColumns() {

return \_columns;

}

int getRows() {

return \_rows;

}

Matrix(int rows, int columns);

Matrix(const Matrix& other);

Matrix(Matrix&& other);

Matrix();

Matrix& operator=(const Matrix& other);

~Matrix();

Matrix& operator+= (const Matrix& other);

Matrix operator+ (const Matrix& other);

Matrix& operator\*= (float l);

Matrix operator\* (float l);

Matrix& operator /= (float l);

Matrix operator/ (float l);

Matrix& operator=(Matrix&& other);

void MultiplyRow(int row, float l);

void PlusRows(int row1, int row2);

void MinusRows(int row1, int row2);

void swapLines(int line1, int line2);

void ToUpTriangle();

operator std::string() const;

};

*ProcMulti Matrix.cpp:*

#include "Matrix.h"

#include <cmath>

int Matrix::count = 0;

float prec1 = 1000000;

Matrix::Matrix(int rows, int columns) : \_rows(rows), \_columns(columns) {

id = count;

++count;

arr = new float\* [\_rows];

float\*\* arrTmp = this->arr;

int cols = \_columns;

int rowst = \_rows;

#pragma omp parallel shared(arrTmp, cols, rowst) //num\_threads(8)

{

#pragma omp for

for (int i = 0; i < rowst; ++i) {

arrTmp[i] = new float[cols];

for (int j = 0; j < cols; ++j) {

arrTmp[i][j] = 0;

}

}

}

}

Matrix::Matrix(const Matrix& other) :Matrix(other.\_rows, other.\_columns) {

float\*\* arrTmp = this->arr;

int cols = \_columns;

int rows = \_rows;

float\*\* otharr = other.arr;

#pragma omp parallel shared(arrTmp, rows, cols, otharr) //num\_threads(8)

{

#pragma omp for

for (int i = 0; i < rows; ++i) {

for (int j = 0; j < cols; ++j) {

arrTmp[i][j] = otharr[i][j];

}

}

}

}

Matrix::Matrix(Matrix&& other) {

id = count;

++count;

arr = other.arr;

\_columns = other.\_columns;

\_rows = other.\_rows;

other.arr = nullptr;

}

Matrix::Matrix() :\_rows(0), \_columns(0) {

id = count;

++count;

arr = nullptr;

}

Matrix& Matrix::operator=(const Matrix& other) {

for (int i = 0; i < \_rows; ++i) {

delete[] arr[i];

}

delete[] arr;

\_rows = other.\_rows;

\_columns = other.\_columns;

arr = new float\* [\_rows];

float\*\* arrTmp = this->arr;

int cols = \_columns;

int rows = \_rows;

float\*\* otharr = other.arr;

#pragma omp parallel shared(arrTmp, rows, cols, otharr) //num\_threads(8)

{

#pragma omp for

for (int i = 0; i < rows; ++i) {

arr[i] = new float[cols];

for (int j = 0; j < cols; ++j)

arrTmp[i][j] = otharr[i][j];

}

}

return \*this;

}

Matrix::~Matrix() {

//std::cout << "deleting matrix " << id << "\n";

if (arr == nullptr) return;

for (int i = 0; i < \_rows; ++i) {

delete[] arr[i];

}

delete[] arr;

}

Matrix& Matrix::operator+=(const Matrix& other) {

float\*\* otharr = other.arr;

float\*\* arrTmp = this->arr;

int cols = \_columns;

int rows = \_rows;

#pragma omp parallel shared(arrTmp, otharr, rows, cols) //num\_threads(8)

{

#pragma omp for

for (int i = 0; i < rows; ++i) {

for (int j = 0; j < cols; ++j) {

arrTmp[i][j] += otharr[i][j];

}

}

}

return \*this;

}

Matrix Matrix::operator+(const Matrix& other) {

Matrix ret(\*this);

return ret += other;

}

Matrix& Matrix::operator\*=(float l) {

float\*\* arrTmp = this->arr;

int cols = \_columns;

int rows = \_rows;

#pragma omp parallel shared(arrTmp, l, rows, cols) //num\_threads(8)

{

#pragma omp for

for (int i = 0; i < rows; ++i) {

for (int j = 0; j < cols; ++j) {

arrTmp[i][j] \*= l;

}

}

}

return \*this;

}

Matrix Matrix::operator\*(float l) {

Matrix ret(\*this);

return ret \*= l;

}

Matrix& Matrix::operator/=(float l) {

float\*\* arrTmp = this->arr;

int cols = \_columns;

int rows = \_rows;

#pragma omp parallel shared(arrTmp, l, rows, cols) //num\_threads(8)

{

#pragma omp for

for (int i = 0; i < rows; ++i) {

for (int j = 0; j < cols; ++j) {

arrTmp[i][j] /= l;

}

}

}

return \*this;

}

Matrix Matrix::operator/(float l) {

Matrix ret(\*this);

return ret /= l;

}

Matrix& Matrix::operator=(Matrix&& other) {

int i = 0;

int j = 0;

for (int i = 0; i < \_rows; ++i) {

delete[] arr[i];

}

delete[] arr;

arr = other.arr;

\_columns = other.\_columns;

\_rows = other.\_rows;

other.arr = nullptr;

return \*this;

}

void Matrix::MultiplyRow(int row, float l) {

float\*\* arrTmp = this->arr;

int cols = \_columns;

#pragma omp parallel shared(arrTmp, row, l, cols)

{

#pragma omp for

for (int i = 0; i < cols; ++i) arrTmp[row][i] \*= l;

}

}

void Matrix::PlusRows(int row1, int row2) {

float\*\* arrTmp = this->arr;

int cols = \_columns;

int rows = \_rows;

#pragma omp parallel shared(arrTmp, row1, row2, cols) //num\_threads(8)

{

#pragma omp for

for (int i = 0; i < cols; ++i) arrTmp[row1][i] += arrTmp[row2][i];

}

}

void Matrix::MinusRows(int row1, int row2) {

float\*\* arrTmp = this->arr;

int cols = \_columns;

int rows = \_rows;

#pragma omp parallel shared(arrTmp, row1, l, cols) //num\_threads(8)

{

#pragma omp for

for (int i = 0; i < cols; ++i) arrTmp[row1][i] -= arrTmp[row2][i];

}

}

void Matrix::swapLines(int line1, int line2) {

float\* temp = arr[line1];

arr[line1] = arr[line2];

arr[line2] = temp;

}

void Matrix::ToUpTriangle() {

int min = \_rows < \_columns ? \_rows : \_columns;

for (int i = 0; i < min; ++i) {

MultiplyRow(i, 1 / arr[i][i]);

for (int j = i + 1; j < \_rows; ++j) {

float c = arr[j][i];

if (c == 0) continue;

MultiplyRow(i, c);

MinusRows(j, i);

MultiplyRow(i, 1 / c);

}

}

}

Matrix::operator std::string() const {

std::string ans = "";

for (int i = 0; i < \_rows; i++) {

for (int j = 0; j < \_columns; ++j)

ans += std::to\_string((floorf(arr[i][j] \* prec1) / prec1)) + " ";

ans += "\n";

}

return ans;

}

*ProcMulti: Source.cpp:*

#include <iostream>

#include "FuncRow.h"

#include "Matrix.h"

#include <functional>

#include <omp.h>

//#include <thread>

const float PI = 3.1415926f;

typedef std::function<Matrix(float x)> CoeffMatrixCounter;

typedef std::function<float(float)> Function;

struct Piece {

float a;

float b;

};

struct Cond {

float y1;

float y2;

};

int getYcount = 0;

float prec2 = 10000000;

Generator Anis(Cond cond, Piece piece) {

return [cond, piece](int i)->Polynom {

if (i == 0) {

float k = (cond.y2 - cond.y1) / (piece.b - piece.a);

float c = cond.y1 - k \* piece.a;

float arr[] = { c,k,0 };

return Polynom(arr);

}

float arr2[3] = { piece.a \* piece.b, -(piece.a + piece.b), 1 };

Polynom polly(arr2);

return polly.IncreasePow(i - 1);

};

}

Matrix IntSimpson(CoeffMatrixCounter f, int numbOfPieces, Piece piece) {

float h = (piece.b - piece.a) / numbOfPieces;

float x = piece.a + h;

Matrix s = f(piece.a) + f(piece.b);

Matrix SumOfEven(s.getRows(), s.getColumns());

Matrix SumOfUneven(s.getRows(), s.getColumns());

Matrix\* soe = &SumOfEven;

Matrix\* sou = &SumOfUneven;

#pragma omp parallel shared(soe, sou, f, numbOfPieces) private(x) num\_threads(2)

{

float x = piece.a + h + omp\_get\_thread\_num() \* h;

for (int i = omp\_get\_thread\_num() + 1; i < numbOfPieces; i += 2) {

if (i % 2 == 0) (\*soe) += f(x) \* 2;

else (\*sou) += f(x) \* 4;

x += 2 \* h;

}

}

s += SumOfEven + SumOfUneven;

return s \* (h / 3);

}

Matrix Mult(Matrix& m1, Matrix& m2) {

Matrix M(m1.getRows(), m1.getColumns());

Matrix\* \_M = &M;

Matrix\* \_m1 = &m1;

Matrix\* \_m2 = &m2;

int cols = m1.getColumns();

int rows = m1.getRows();

#pragma omp parallel shared(\_M, \_m1, \_m2, cols, rows)

{

#pragma omp for

for (int i = 0; i < cols; ++i)

for (int j = 0; j < rows; ++j)

\_M->arr[i][j] = \_m1->arr[i][i] \* \_m2->arr[j][j];

}

return M;

}

Matrix DiffSquare(Matrix& m, int variable) {

Matrix M(m);

int rows = M.getRows();

int cols = M.getColumns();

Matrix\* \_M = &M;

#pragma omp parallel shared(\_M, rows, cols, variable)

{

#pragma omp for

for (int i = 0; i < rows; ++i) {

if (i == variable) continue;

for (int j = 0; j < cols; ++j) {

if (j == variable) continue;

\_M->arr[i][j] = 0;

}

}

}

return M;

}

float\* backMove(Matrix& slau) {

float\* x = new float[slau.getColumns() - 1];

for (int i = slau.getRows() - 1; i >= 0; --i) {

int k = i + 1;

float d = slau.arr[i][slau.getColumns() - 1];

for (; k < slau.getRows(); ++k) d -= slau.arr[i][k] \* x[k];

x[i] = d;

}

return x;

}

FuncRow MnkInt(Function p, Function q, Function f, int numbOfMembers, Generator memb, int numbOfPieces,

Piece piece) {

FuncRow frow(numbOfMembers, memb);

//std::cout << std::string(frow) << "\n\n\n";

auto lambda =

[numbOfMembers, frow, p, q, f](float x)->Matrix {

static int counter = 0;

Matrix A(numbOfMembers, numbOfMembers);

//std::cout << std::string(frow) << "\n\n";

for (int i = 0; i < numbOfMembers; ++i) {

A.arr[i][i] += frow[i].Diff().Diff().Count(x);

//std::cout << i << "th number " << A.arr[i][i] << " ";

A.arr[i][i] += frow[i].Diff().Count(x) \* p(x);

//std::cout << A.arr[i][i] << " ";

A.arr[i][i] += frow[i].Count(x) \* q(x);

//std::cout << A.arr[i][i] << "\nHis pol " << std::string(frow[i]) << "\n";

}

A.arr[0][0] -= f(x);

auto M = Mult(A, A);

//std::cout << "A" << counter <<" = " << "\n";

//std::cout << std::string(A) << "\n\n\n";

++counter;

return M;

};

Matrix intgr = IntSimpson(lambda, numbOfPieces, piece);

//std::cout << std::string(intgr) << "\n\n\n";

Matrix matr(numbOfMembers - 1, numbOfMembers);

int nom = numbOfMembers;

Matrix\* \_matr = &matr;

Matrix\* \_intgr = &intgr;

#pragma omp parallel shared(nom, \_matr, \_intgr)

{

#pragma omp for

for (int i = 1; i < nom; ++i) {

Matrix df = DiffSquare(\*(\_intgr), i);

for (int j = 1; j < nom; ++j) \_matr->arr[i - 1][j - 1] = df.arr[i][j] + df.arr[j][i];

\_matr->arr[i - 1][nom - 1] = -(df.arr[i][0] + df.arr[0][i]);

}

}

//std::cout << "До треуголирования\n" << std::string(matr) << "\n\n";

matr.ToUpTriangle();

//std::cout << "После треуголирования\n" << std::string(matr) << "\n\n";

float\* coeffs = backMove(matr);

for (int i = 1; i < numbOfMembers; ++i) frow[i] \*= coeffs[i - 1];

delete[] coeffs;

return frow;

}

void ShowDataTest(FuncRow frow, Function ans, Piece piece, int numbOfPoints) {

float h = (piece.b - piece.a) / numbOfPoints;

float x = piece.a;

float\* delt = new float[numbOfPoints + 1];

for (int i = 0; i <= numbOfPoints; ++i, x += h) {

delt[i] = abs(frow.Count(x) - ans(x));

/\*std::cout << myround(x, prec2) <<

" Полученное решение : " << myround(frow.Count(x), prec2) <<

" Ответ: " << myround(ans(x), prec2) << " Невязка: " <<

myround(delt[i], prec2) << "\n";\*/

}

float Norm = 0;

for (int i = 0; i < numbOfPoints + 1; ++i) if (Norm < delt[i]) Norm = delt[i];

delete[] delt;

std::cout << "Норма невязки: " << Norm << "\n";

}

void dotest(Function p\_t, Function q\_t, Function f\_t, Function lambda, Piece piece,

Cond cond, int i, int numbOfMembers, int numbOfPoints) {

std::cout << "Интегральный МНК Тестовый пример " << i << "\n";

auto beg = clock();

FuncRow frow = MnkInt(p\_t, q\_t, f\_t, numbOfMembers, Anis(cond, piece), 30, piece);

auto end = clock();

std::cout << "Время выполнения " << end - beg << " миллисекунд" << "\n";

ShowDataTest(frow, lambda, piece, numbOfPoints);

}

int main() {

setlocale(0, "");

int i = 1;

Function p\_t = [](float x)->float { return 0; };

Function q\_t = [](float x) { return 1; };

Function f\_t = [](float x)->float { return x \* x + 3 \* x - 7; };

Function lambda = [](float x)->float { return x \* x + 3 \* x - 9; };

Piece piece\_t = { 0, 1 };

Cond cond\_t = { -9, -5 };

int N = 1000;

const int numbOfMembers = N;

const int numbOfPoints = N;

dotest(p\_t, q\_t, f\_t, lambda, piece\_t, cond\_t, i, numbOfMembers, numbOfPoints);

Function e3 = [](float x)->float {return expf(3 \* x); };

p\_t = [](float x)->float {return x \* x + 6; };

q\_t = [](float x)->float {return pow(2, -x); };

f\_t = [e3](float x)->float {return 9 \* e3(x) + 3 \* (x \* x + 6) \* e3(x) + pow(2, -x) \* e3(x); };

lambda = [](float x)->float {return expf(3 \* x); };

piece\_t = { 0, 1 };

cond\_t = { 1, expf(3) };

i = 2;

dotest(p\_t, q\_t, f\_t, lambda, piece\_t, cond\_t, i, numbOfMembers, numbOfPoints);

p\_t = [](float x)->float {return 0; };

q\_t = [](float x)->float {return 1; };

f\_t = [](float x)->float {return 2 \* x - PI; };

lambda = [](float x)->float {return 2 \* x - PI + PI \* cos(x); };

cond\_t = { 0, lambda(1) };

piece\_t = { 0, 1 };

i = 3;

dotest(p\_t, q\_t, f\_t, lambda, piece\_t, cond\_t, i, numbOfMembers, numbOfPoints);

p\_t = [](float x)->float {return 2; };

q\_t = [](float x)->float {return 1; };

f\_t = [](float x)->float {return 0; };

lambda = [](float x)->float {return expf(-x) + x \* expf(-x); };

piece\_t = { 0, 1 };

cond\_t = { lambda(0), lambda(1) };

i = 4;

dotest(p\_t, q\_t, f\_t, lambda, piece\_t, cond\_t, i, numbOfMembers, numbOfPoints);

}

*ProcUnoptimized:Matrix.cpp:*

#include "Matrix.h"

#include <cmath>

int Matrix::count = 0;

float prec1 = 1000000;

Matrix::Matrix(int rows, int columns) : \_rows(rows), \_columns(columns) {

id = count;

++count;

arr = new float\* [\_rows];

for (int i = 0; i < \_rows; ++i) {

arr[i] = new float[\_columns];

for (int j = 0; j < \_columns; ++j)

arr[i][j] = 0;

}

}

Matrix::Matrix(const Matrix& other) :Matrix(other.\_rows, other.\_columns) {

//std::cout << "def copy constructor from " << other.id << " to " << id << "\n";

for (int i = 0; i < other.\_rows; ++i) {

for (int j = 0; j < other.\_columns; ++j) {

arr[i][j] = other.arr[i][j];

}

}

}

Matrix::Matrix(Matrix&& other) {

id = count;

//std::cout << "rv copy constructor from " << other.id << " to " << id << "\n";

++count;

arr = other.arr;

\_columns = other.\_columns;

\_rows = other.\_rows;

other.arr = nullptr;

}

Matrix::Matrix() :\_rows(0), \_columns(0) {

id = count;

++count;

arr = nullptr;

}

Matrix& Matrix::operator=(const Matrix& other) {

//std::cout << "def copy operator from " << other.id << " to " << id << "\n";

for (int i = 0; i < \_rows; ++i) {

delete[] arr[i];

}

delete[] arr;

\_rows = other.\_rows;

\_columns = other.\_columns;

arr = new float\* [\_rows];

for (int i = 0; i < \_rows; ++i) {

arr[i] = new float[\_columns];

for (int j = 0; j < \_columns; ++j)

arr[i][j] = other.arr[i][j];

}

return \*this;

}

Matrix::~Matrix() {

//std::cout << "deleting matrix " << id << "\n";

if (arr == nullptr) return;

for (int i = 0; i < \_rows; ++i) {

delete[] arr[i];

}

delete[] arr;

}

Matrix& Matrix::operator+=(const Matrix& other) {

for (int i = 0; i < \_rows; ++i) {

for (int j = 0; j < \_columns; ++j) {

arr[i][j] += other.arr[i][j];

}

}

return \*this;

}

Matrix Matrix::operator+(const Matrix& other) {

Matrix ret(\*this);

return ret += other;

}

Matrix& Matrix::operator\*=(float l) {

for (int i = 0; i < \_rows; ++i) {

for (int j = 0; j < \_columns; ++j) {

arr[i][j] \*= l;

}

}

return \*this;

}

Matrix Matrix::operator\*(float l) {

Matrix ret(\*this);

return ret \*= l;

}

Matrix& Matrix::operator/=(float l) {

for (int i = 0; i < \_rows; ++i) {

for (int j = 0; j < \_columns; ++j) {

arr[i][j] /= l;

}

}

return \*this;

}

Matrix Matrix::operator/(float l) {

Matrix ret(\*this);

return ret /= l;

}

Matrix& Matrix::operator=(Matrix&& other) {

int i = 0;

int j = 0;

delete[] arr;

arr = other.arr;

arr[i][i] = arr[j][j];

\_columns = other.\_columns;

\_rows = other.\_rows;

other.arr = nullptr;

return \*this;

}

void Matrix::MultiplyRow(int row, float l) {

for (int i = 0; i < \_columns; ++i) arr[row][i] \*= l;

}

void Matrix::PlusRows(int row1, int row2) {

for (int i = 0; i < \_columns; ++i) arr[row1][i] += arr[row2][i];

}

void Matrix::MinusRows(int row1, int row2) {

for (int i = 0; i < \_columns; ++i) arr[row1][i] -= arr[row2][i];

}

void Matrix::swapLines(int line1, int line2) {

float\* temp = arr[line1];

arr[line1] = arr[line2];

arr[line2] = temp;

}

void Matrix::ToUpTriangle() {

int min = \_rows < \_columns ? \_rows : \_columns;

for (int i = 0; i < min; ++i) {

if (arr[i, i] == 0) break;

int maxRow = i;

for (int j = i + 1; j < \_rows; ++j)

if (arr[j, i] > arr[maxRow, i] && arr[j, i] != 0) maxRow = j;

swapLines(i, maxRow);

MultiplyRow(i, 1 / arr[i][i]);

for (int j = i + 1; j < \_rows; ++j) {

float c = arr[j][i];

if (c == 0) continue;

MultiplyRow(i, c);

MinusRows(j, i);

MultiplyRow(i, 1 / c);

}

}

}

Matrix::operator std::string() const {

std::string ans = "";

for (int i = 0; i < \_rows; i++) {

for (int j = 0; j < \_columns; ++j)

ans += std::to\_string((floorf(arr[i][j] \* prec1) / prec1)) + " ";

ans += "\n";

}

return ans;

}

*ProcUnoptimized:Source.cpp:*

#include <iostream>

#include "FuncRow.h"

#include "Matrix.h"

#include <functional>

const float PI = 3.1415926f;

typedef std::function<Matrix (float x)> CoeffMatrixCounter;

typedef std::function<float(float)> Function;

struct Piece {

float a;

float b;

};

struct Cond {

float y1;

float y2;

};

float prec2 = 10000000;

Matrix\* GetY(CoeffMatrixCounter f, Piece piece, int numbOfPieces) {

float h = (piece.b - piece.a) / numbOfPieces;

float x = piece.a;

Matrix\* Y = new Matrix[numbOfPieces + 1];

for (int i = 0; i < numbOfPieces + 1; ++i) {

Y[i] = f(x);

x += h;

}

return Y;

}

Generator Anis(Cond cond, Piece piece) {

return [cond, piece](int i)->Polynom {

if (i == 0) {

float k = (cond.y2 - cond.y1) / (piece.b - piece.a);

float c = cond.y1 - k \* piece.a;

float arr[] = { c,k,0 };

return Polynom(arr);

}

float arr2[3] = { piece.a \* piece.b, -(piece.a + piece.b), 1 };

Polynom polly(arr2);

return polly.IncreasePow(i - 1);

};

}

Matrix IntSimpson(CoeffMatrixCounter f, int numbOfPieces, Piece piece) {

float h = (piece.b - piece.a) / numbOfPieces;

float x = piece.a + h;

Matrix s = f(piece.a) + f(piece.b);

for (int i = 1; i < numbOfPieces; ++i) {

if (i % 2 == 0) s += f(x) \* 2;

else s += f(x) \* 4;

x += h;

}

return s \* h / 3;

}

Matrix Mult(Matrix& m1, Matrix& m2) {

Matrix M(m1.getRows(), m1.getColumns());

for (int i = 0; i < m1.getColumns(); ++i)

for (int j = 0; j < m1.getColumns(); ++j)

M.arr[i][j] = m1.arr[i][i] \* m2.arr[j][j];

return M;

}

Matrix DiffSquare(Matrix& m, int variable) {

Matrix M(m);

for (int i = 0; i < m.getRows(); ++i) {

if (i == variable) continue;

for (int j = 0; j < m.getColumns(); ++j) {

if (j == variable) continue;

M.arr[i][j] = 0;

}

}

return M;

}

float\* backMove(Matrix& slau) {

float\* x = new float[slau.getColumns() - 1];

for (int i = slau.getRows() - 1; i >= 0; --i) {

int k = i + 1;

float d = slau.arr[i][slau.getColumns() - 1];

for (; k < slau.getRows(); ++k) d -= slau.arr[i][k] \* x[k];

x[i] = d;

}

return x;

}

FuncRow MnkInt(Function p, Function q, Function f, int numbOfMembers, Generator memb, int numbOfPieces,

Piece piece) {

FuncRow frow (numbOfMembers, memb);

auto lambda =

[numbOfMembers, frow, p, q, f](float x)->Matrix{

static int counter = 0;

Matrix A (numbOfMembers, numbOfMembers);

for (int i = 0; i < numbOfMembers; ++i) {

A.arr[i][i] += frow[i].Diff().Diff().Count(x);

A.arr[i][i] += frow[i].Diff().Count(x) \* p(x);

A.arr[i][i] += frow[i].Count(x) \* q(x);

}

A.arr[0][0] -= f(x);

auto M = Mult(A, A);

++counter;

return M;

};

Matrix intgr = IntSimpson(lambda, numbOfPieces, piece);

Matrix matr(numbOfMembers - 1, numbOfMembers);

for (int i = 1; i < numbOfMembers; ++i) {

Matrix df = DiffSquare(intgr, i);

for (int j = 1; j < numbOfMembers; ++j) matr.arr[i - 1][j - 1] = df.arr[i][j] + df.arr[j][i];

matr.arr[i - 1][numbOfMembers - 1] = -(df.arr[i][0] + df.arr[0][i]);

}

matr.ToUpTriangle();

float\* coeffs = backMove(matr);

for (int i = 1; i < numbOfMembers; ++i) frow[i] \*= coeffs[i - 1];

delete[] coeffs;

return frow;

}

void ShowDataTest(FuncRow frow, Function ans, Piece piece, int numbOfPoints) {

//std::cout << ("Полученный ряд:\n");

//std::cout << (std::string)frow;

float h = (piece.b - piece.a) / numbOfPoints;

float x = piece.a;

float\* delt = new float[numbOfPoints + 1];

for (int i = 0; i <= numbOfPoints; ++i, x += h) {

delt[i] = abs(frow.Count(x) - ans(x));

/\*std::cout << myround(x, prec2) <<

" Полученное решение : " << myround(frow.Count(x), prec2) <<

" Ответ: " << myround(ans(x), prec2) << " Невязка: " <<

myround(delt[i], prec2) << "\n";\*/

}

float Norm = 0;

for (int i = 0; i < numbOfPoints + 1; ++i) if (Norm < delt[i]) Norm = delt[i];

delete[] delt;

std::cout << "Норма невязки: "<< Norm << "\n";

}

void dotest(Function p\_t, Function q\_t, Function f\_t, Function lambda, Piece piece,

Cond cond, int i, int numbOfMembers, int numbOfPoints) {

std::cout <<"Интегральный МНК Тестовый пример " << i << "\n";

auto beg = clock();

FuncRow frow = MnkInt(p\_t, q\_t, f\_t, numbOfMembers, Anis(cond, piece), 30, piece);

auto end = clock();

std::cout << "Время выполнения " << end - beg << " миллисекунд" << "\n";

ShowDataTest(frow, lambda, piece, numbOfPoints);

}

int main() {

setlocale(0, "");

int i = 1;

Function p\_t = [](float x)->float { return 0; };

Function q\_t = [](float x) { return 1; };

Function f\_t = [](float x)->float { return x \* x + 3 \* x - 7; };

Function lambda = [](float x)->float { return x \* x + 3 \* x - 9; };

Piece piece\_t = { 0, 1 };

Cond cond\_t = {- 9, -5};

int N = 10;

const int numbOfMembers = N;

const int numbOfPoints = N;

dotest(p\_t, q\_t, f\_t, lambda, piece\_t, cond\_t, i, numbOfMembers, numbOfPoints);

Function e3 = [](float x)->float {return expf(3 \* x); };

p\_t = [](float x)->float {return x \* x + 6; };

q\_t = [](float x)->float {return pow(2, -x); };

f\_t = [e3](float x)->float {return 9 \* e3(x) + 3 \* (x \* x + 6) \* e3(x) + pow(2, -x) \* e3(x); };

lambda = [](float x)->float {return expf(3 \* x); };

piece\_t = { 0, 1 };

cond\_t = { 1, expf(3)};

i = 2;

dotest(p\_t, q\_t, f\_t, lambda, piece\_t, cond\_t, i, numbOfMembers, numbOfPoints);

p\_t = [](float x)->float {return 0; };

q\_t = [](float x)->float {return 1; };

f\_t = [](float x)->float {return 2 \* x - PI; };

lambda = [](float x)->float {return 2 \* x - PI + PI \* cos(x); };

cond\_t = { 0, lambda(1) };

piece\_t = { 0, 1 };

i = 3;

dotest(p\_t, q\_t, f\_t, lambda, piece\_t, cond\_t, i, numbOfMembers, numbOfPoints);

p\_t = [](float x)->float {return 2; };

q\_t = [](float x)->float {return 1; };

f\_t = [](float x)->float {return 0; };

lambda = [](float x)->float {return expf(-x) + x \* expf(-x); };

piece\_t = { 0, 1 };

cond\_t = { lambda(0), lambda(1) };

i = 4;

dotest(p\_t, q\_t, f\_t, lambda, piece\_t, cond\_t, i, numbOfMembers, numbOfPoints);

}

*GPUMulti: Matrix.cu:*

#include "Matrix.h"

#include <cmath>

#define SIZE \_rows \* \_columns \* sizeof(float)

#define ERRORCHECKF(op) auto err = op; if(err != cudaSuccess) std::cout << cudaGetErrorString(err) << "\n";

#define ERRORCHECK(op) err = op; if(err != cudaSuccess) std::cout << cudaGetErrorString(err) << "\n";

int Matrix::count = 0;

float prec1 = 1000000;

\_\_global\_\_ void setZeros(float\* arr, int \_rows, int \_columns) {

int i = blockIdx.x;

int j = threadIdx.x;

arr[i \* \_columns + j] = 0;

}

Matrix::Matrix(int rows, int columns) : \_rows(rows), \_columns(columns) {

ERRORCHECKF( cudaMalloc(&arr, \_rows \* \_columns \* sizeof(float)));

setZeros << <\_rows, \_columns >> > (arr, \_rows, \_columns);

cudaDeviceSynchronize();

}

Matrix::Matrix(const Matrix& other) :Matrix(other.\_rows, other.\_columns) {

//cudaMalloc(&arr, SIZE);

ERRORCHECKF(cudaMemcpy(arr, other.arr, SIZE, cudaMemcpyDeviceToDevice));

}

Matrix::Matrix(Matrix&& other) {

arr = other.arr;

\_columns = other.\_columns;

\_rows = other.\_rows;

other.arr = nullptr;

}

Matrix::Matrix() :\_rows(0), \_columns(0) {

}

Matrix& Matrix::operator=(const Matrix& other) {

ERRORCHECKF(cudaFree(arr));

ERRORCHECK(cudaMalloc(&arr, SIZE));

ERRORCHECK(cudaMemcpy(arr, other.arr, SIZE, cudaMemcpyDeviceToDevice));

return \*this;

}

Matrix::~Matrix() {

cudaFree(arr);

}

\_\_global\_\_ void cudaGet(int i, int j, float\* arr, float\* ref, int \_rows, int \_columns) {

//cudaMemcpy(ref,&(arr[i \* \_columns + j]), sizeof(float), cudaMemcpyDeviceToHost);

}

float Matrix::get(int i, int j) const{

float ref = 0.1f;

cudaMemcpy(&ref, &(arr[i \* \_columns + j]), sizeof(float), cudaMemcpyDeviceToHost);

//cudaGet << <1, 1 >> > (i, j, arr, &ref, \_rows, \_columns);

//ERRORCHECKF(cudaDeviceSynchronize());

return ref;

}

\_\_global\_\_ void cudaSet(int i, int j, float\* arr, float value, int \_rows, int \_columns) {

arr[i \* \_columns + j] = value;

}

void Matrix::set(int i, int j, float value) {

cudaSet << <1, 1 >> > (i, j, arr, value, \_rows, \_columns);

cudaDeviceSynchronize();

}

\_\_global\_\_ void addMatrix(float\* dst, float\* src, int \_rows, int \_columns) {

int i = blockIdx.x \* \_columns + threadIdx.x;

dst[i] += src[i];

}

Matrix& Matrix::operator+=(const Matrix& other) {

addMatrix <<<\_rows, \_columns >>> (arr, other.arr, \_rows, \_columns);

cudaDeviceSynchronize();

return \*this;

}

Matrix Matrix::operator+(const Matrix& other) {

Matrix m (\*this);

return m += other;

}

\_\_global\_\_ void multOnFloat(float\* src, float l, int \_rows, int \_columns) {

src[blockIdx.x \* \_columns + threadIdx.x] \*= l;

}

Matrix& Matrix::operator\*=(float l) {

multOnFloat<<<\_rows, \_columns>>>(arr, l, \_rows, \_columns);

cudaDeviceSynchronize();

return\* this;

}

Matrix Matrix::operator\*(float l) {

Matrix m(\*this);

return m \*= l;

}

\_\_global\_\_ void divOnFloat(float\* src, float l, int \_rows, int \_columns) {

src[blockIdx.x \* \_columns + threadIdx.x] /= l;

}

Matrix& Matrix::operator/=(float l) {

divOnFloat <<<\_rows, \_columns >>> (arr, l, \_rows, \_columns);

cudaDeviceSynchronize();

return\*this;

}

Matrix Matrix::operator/(float l) {

Matrix m(\*this);

return m /= l;

}

Matrix& Matrix::operator=(Matrix&& other) {

cudaFree(arr);

arr = other.arr;

\_columns = other.\_columns;

\_rows = other.\_rows;

other.arr = nullptr;

return \*this;

}

\_\_global\_\_ void multRow(float\* src, int row, float l, int rows, int columns) {

int i = row \* columns + threadIdx.x;

src[i] \*= l;

}

void Matrix::MultiplyRow(int row, float l) {

multRow << <1, \_columns >> > (arr, row, l,\_rows, \_columns);

cudaDeviceSynchronize();

}

\_\_global\_\_ void plusRows(float\* src, int rowDst, int rowSrc, int rows, int columns) {

int i = rowDst \* columns + threadIdx.x;

int i1 = rowSrc \* columns + threadIdx.x;

src[i] += src[i1];

}

void Matrix::PlusRows(int row1, int row2) {

plusRows << <1, \_columns >> > (arr, row1, row2, \_rows, \_columns);

cudaDeviceSynchronize();

}

\_\_global\_\_ void minusRows(float\* src, int rowDst, int rowSrc, int rows, int columns) {

int i = rowDst \* columns + threadIdx.x;

int i1 = rowSrc \* columns + threadIdx.x;

src[i] -= src[i1];

}

void Matrix::MinusRows(int row1, int row2) {

minusRows << <1, \_columns >> > (arr, row1, row2, \_rows, \_columns);

cudaDeviceSynchronize();

}

\_\_global\_\_ void swapRows(float\* src, int row1, int row2, int rows, int columns) {

int i = row1 \* columns + threadIdx.x;

int i1 = row2 \* columns + threadIdx.x;

float tmp = src[i];

src[i] = src[i1];

src[i1] = tmp;

}

void Matrix::swapLines(int line1, int line2) {

swapRows << <6, \_columns/6 >> > (arr, line1, line2, \_rows, \_columns);

cudaDeviceSynchronize();

}

\_\_global\_\_ void currentColumnToZero(int current, float\* arr, int \_rows, int \_columns) {

int i = blockIdx.x + current + 1;

int j = threadIdx.x + current;

float f = arr[current \* \_columns + j];

f \*= arr[i \* \_columns + current];

arr[i \* \_columns + j] -= f;

}

void Matrix::ToUpTriangle() {

for (int i = 0; i < \_rows; ++i) {

MultiplyRow(i, 1 / get(i, i));

if(i != \_rows - 1) currentColumnToZero << <\_rows - i - 1, \_columns - i >> > (i, arr, \_rows, \_columns);

cudaDeviceSynchronize();

//std::cout << "�� ������\n" << std::string(\*this) << "\n\n";

}

}

\_\_global\_\_ void backMoveFunc(float\* x, float\* arr, int current, int \_rows, int \_columns) {

int i = threadIdx.x;

x[i] += x[current] \* arr[i \* \_columns + current];

}

float\* Matrix::backMove(){

float\* x\_d;

cudaMalloc(&x\_d, sizeof(float) \* \_rows);

setZeros << <1, \_rows >> > (x\_d, \_rows, \_columns);

cudaDeviceSynchronize();

//float\* xux = new float[\_rows];

//cudaMemcpy(xux, x\_d, sizeof(float) \* \_rows, cudaMemcpyDeviceToHost);

//for (int g = 0; g < \_rows; ++g) std::cout << "x" << g << " = " << xux[g] << " ";

//std::cout << "\n";

//delete[] xux;

for (int i = \_rows - 1; i >= 0; --i) {

if (i == \_rows - 1) {

cudaSet << <1, 1 >> > (0, i, x\_d, get(i, \_columns - 1), \_rows, \_columns);

}

else {

float sum = 0.1f;

cudaMemcpy(&sum, &(x\_d[i]), sizeof(float), cudaMemcpyDeviceToHost);

//std::cout << i << " col - 1 " << get(i, \_columns - 1) << "\n";

//std::cout << i << " sum " << sum << "\n";

cudaSet << <1, 1 >> > (0, i, x\_d, get(i, \_columns - 1) - sum, \_rows, \_columns);

cudaDeviceSynchronize();

}

if(i != 0 ) backMoveFunc << <1, i >> > (x\_d, arr, i, \_rows, \_columns);

cudaDeviceSynchronize();

//float\* xux = new float[\_rows];

//cudaMemcpy(xux, x\_d, sizeof(float) \* \_rows, cudaMemcpyDeviceToHost);

//for(int g = 0; g < \_rows; ++g) std::cout << "x" << g << " = " << xux[g] << " ";

//std::cout << "\n";

//delete[] xux;

}

float\* x = new float[\_rows];

cudaMemcpy(x, x\_d, sizeof(float) \* \_rows, cudaMemcpyDeviceToHost);

return x;

}

Matrix::operator std::string() const {

std::string ans = "";

for (int i = 0; i < \_rows; i++) {

for (int j = 0; j < \_columns; ++j)

ans += std::to\_string((floorf(get(i,j) \* prec1) / prec1)) + " ";

ans += "\n";

}

return ans;

}

\_\_global\_\_ void forMult(float\* dst, float\* arr1, float\* arr2, int \_rows, int \_columns) {

int i = blockIdx.x;

int j = threadIdx.x;

dst[i \* \_columns + j] = arr1[i \* \_columns + i] \* arr2[j \* \_columns + j];

}

Matrix Matrix::Multiply(const Matrix& other) {

Matrix dst(\_rows, \_columns);

forMult <<<\_rows, \_columns >>> (dst.arr, this->arr, other.arr, \_rows, \_columns);

cudaDeviceSynchronize();

return dst;

}

\_\_global\_\_ void forDiffSquare(float\* arr, int current, int \_rows, int \_columns) {

int i = blockIdx.x;

int j = threadIdx.x;

if(i != current && j != current)

arr[i \* \_columns + j] =0;

}

Matrix Matrix::DiffSquare(int variable) {

Matrix ret(\*this);

forDiffSquare<<<\_rows,\_columns>>>(ret.arr, variable, \_rows, \_columns);

cudaDeviceSynchronize();

return ret;

}

\_\_global\_\_ void increm(float\* arr, int i, int j, float inc, int \_rows, int \_columns) {

arr[i \* \_columns + j] += inc;

}

void Matrix::Increase(int i, int j, float inc){

increm<<<1, 1 >> > (arr, i, j, inc, \_rows, \_columns);

cudaDeviceSynchronize();

}

\_\_global\_\_ void copyDiffToMatrixFunc(float\* arr, float\* diffArr, int variable, int \_rows, int \_columns) {

int j = threadIdx.x + 1;

arr[(variable - 1) \* \_columns + j - 1] =

diffArr[(variable) \* \_columns + j] + diffArr[(j) \* \_columns + variable];

}

void Matrix::CopyDiffToMatrix(Matrix& diff, int variable){

copyDiffToMatrixFunc<<<1, \_rows>>>(arr, diff.arr, variable, \_rows, \_columns);

cudaDeviceSynchronize();

set(variable - 1, \_columns - 1, -(diff.get(variable, 0) + diff.get(0, variable)));

}

*GPUMulti: kernel.cu:*

#include <iostream>

#include "FuncRow.h"

#include "Matrix.h"

#include <functional>

const float PI = 3.1415926f;

typedef std::function<Matrix(float x)> CoeffMatrixCounter;

typedef std::function<float(float)> Function;

//места для распараллеливания: гетУ вычисления матриц f(x)

//интеграл симпсона суммирование матриц

// в мульте возможно разбить на зоны

// в дифсквере сомнительно но можно попробовать

struct Piece {

float a;

float b;

};

struct Cond {

float y1;

float y2;

};

float prec2 = 10000000;

Matrix\* GetY(CoeffMatrixCounter f, Piece piece, int numbOfPieces) {

float h = (piece.b - piece.a) / numbOfPieces;

float x = piece.a;

Matrix\* Y = new Matrix[numbOfPieces + 1];

for (int i = 0; i < numbOfPieces + 1; ++i) {

Y[i] = f(x);

x += h;

//std::cout << std::string(Y[i]) << "\n\n\n";

}

return Y;

}

Generator Anis(Cond cond, Piece piece) {

return [cond, piece](int i)->Polynom {

if (i == 0) {

float k = (cond.y2 - cond.y1) / (piece.b - piece.a);

//k\*(x - piece.a) + cond.y1

float c = cond.y1 - k \* piece.a;

float arr[] = { c,k,0 };

//auto pol = Polynom(arr);

//std::cout << std::string(pol) << "\n";

return Polynom(arr);

}

float arr2[3] = { piece.a \* piece.b, -(piece.a + piece.b), 1 };

Polynom polly(arr2);

//Polynom pol(i - 1);

//pol[i - 1] = 1;

//std::cout << (std::string)(pol \* polly) << "\n";

return polly.IncreasePow(i - 1);

};

}

Matrix IntSimpson(CoeffMatrixCounter f, int numbOfPieces, Piece piece) {

float h = (piece.b - piece.a) / numbOfPieces;

//Matrix\* Y = GetY(f, piece, numbOfPieces);

float x = piece.a;

//Matrix\* Y = new Matrix[numbOfPieces + 1];

Matrix s = f(x) + f(piece.b);

x += h;

//std::cout << std::string(s) << "\n\n";

for (int i = 1; i < numbOfPieces; ++i) {

if (i % 2 == 0) s += f(x) \* 2;

else s += f(x) \* 4;

x += h;

}

return s \* (h / 3);

}

Matrix Mult(Matrix& m1, Matrix& m2) {

return m1.Multiply(m2);

}

Matrix DiffSquare(Matrix m, int variable) {

return m.DiffSquare(variable);

}

FuncRow MnkInt(Function p, Function q, Function f, int numbOfMembers, Generator memb, int numbOfPieces,

Piece piece) {

FuncRow frow(numbOfMembers, memb);

//std::cout << std::string(frow) << "\n\n\n";

auto lambda =

[numbOfMembers, frow, p, q, f](float x)->Matrix {

static int counter = 0;

Matrix A(numbOfMembers, numbOfMembers);

//std::cout << std::string(frow) << "\n\n";

for (int i = 0; i < numbOfMembers; ++i) {

A.Increase(i,i, frow[i].Diff().Diff().Count(x));

A.Increase(i, i, frow[i].Diff().Count(x) \* p(x)); A.Increase(i, i, frow[i].Count(x) \* q(x));

}

A.Increase(0,0, -f(x));

auto M = Mult(A, A);

++counter;

return M;

};

Matrix intgr = IntSimpson(lambda, numbOfPieces, piece);

Matrix matr(numbOfMembers - 1, numbOfMembers);

for (int i = 1; i < numbOfMembers; ++i) {

Matrix df = DiffSquare(intgr, i);

matr.CopyDiffToMatrix(df, i);

}

matr.ToUpTriangle();

float\* coeffs = matr.backMove();

for (int i = 1; i < numbOfMembers; ++i) frow[i] \*= coeffs[i - 1];

delete[] coeffs;

return frow;

}

void ShowDataTest(FuncRow frow, Function ans, Piece piece, int numbOfPoints) {

float h = (piece.b - piece.a) / numbOfPoints;

float x = piece.a;

float\* delt = new float[numbOfPoints + 1];

for (int i = 0; i <= numbOfPoints; ++i, x += h) {

delt[i] = abs(frow.Count(x) - ans(x));

}

float Norm = 0;

for (int i = 0; i < numbOfPoints + 1; ++i) if (Norm < delt[i]) Norm = delt[i];

delete[] delt;

std::cout << "Норма невязки: " << Norm << "\n";

}

void dotest(Function p\_t, Function q\_t, Function f\_t, Function lambda, Piece piece,

Cond cond, int i, int numbOfMembers, int numbOfPoints) {

std::cout << "Интегральный МНК Тестовый пример " << i << "\n";

auto beg = clock();

FuncRow frow = MnkInt(p\_t, q\_t, f\_t, numbOfMembers, Anis(cond, piece), 30, piece);

auto end = clock();

std::cout << "Время выполнения " << end - beg << " миллисекунд" << "\n";

ShowDataTest(frow, lambda, piece, numbOfPoints);

}

int main() {

setlocale(0, "");

Matrix A(5, 5);

A.Increase(1, 1, -2);

std::cout << std::string(A) << "\n\n";

A.Increase(2, 2, 2);

int i = 1;

Function p\_t = [](float x)->float { return 0; };

Function q\_t = [](float x) { return 1; };

Function f\_t = [](float x)->float { return x \* x + 3 \* x - 7; };

Function lambda = [](float x)->float { return x \* x + 3 \* x - 9; };

Piece piece\_t = { 0, 1 };

Cond cond\_t = { -9, -5 };

const int numbOfMembers = 1000;

const int numbOfPoints = 1000;

dotest(p\_t, q\_t, f\_t, lambda, piece\_t, cond\_t, i, numbOfMembers, numbOfPoints);

Function e3 = [](float x)->float {return expf(3 \* x); };

p\_t = [](float x)->float {return x \* x + 6; };

q\_t = [](float x)->float {return pow(2, -x); };

f\_t = [e3](float x)->float {return 9 \* e3(x) + 3 \* (x \* x + 6) \* e3(x) + pow(2, -x) \* e3(x); };

lambda = [](float x)->float {return expf(3 \* x); };

piece\_t = { 0, 1 };

cond\_t = { 1, expf(3) };

i = 2;

dotest(p\_t, q\_t, f\_t, lambda, piece\_t, cond\_t, i, numbOfMembers, numbOfPoints);

p\_t = [](float x)->float {return 0; };

q\_t = [](float x)->float {return 1; };

f\_t = [](float x)->float {return 2 \* x - PI; };

lambda = [](float x)->float {return 2 \* x - PI + PI \* cos(x); };

cond\_t = { 0, lambda(1) };

piece\_t = { 0, 1 };

i = 3;

dotest(p\_t, q\_t, f\_t, lambda, piece\_t, cond\_t, i, numbOfMembers, numbOfPoints);

p\_t = [](float x)->float {return 2; };

q\_t = [](float x)->float {return 1; };

f\_t = [](float x)->float {return 0; };

lambda = [](float x)->float {return expf(-x) + x \* expf(-x); };

piece\_t = { 0, 1 };

cond\_t = { lambda(0), lambda(1) };

i = 4;

dotest(p\_t, q\_t, f\_t, lambda, piece\_t, cond\_t, i, numbOfMembers, numbOfPoints);

}

*GpuUnopotimized:Matrix.cu:*

#include "Matrix.h"

#include <cmath>

#define SIZE \_rows \* \_columns \* sizeof(float)

#define ERRORCHECKF(op) auto err = op; if(err != cudaSuccess) std::cout << cudaGetErrorString(err) << "\n";

#define ERRORCHECK(op) err = op; if(err != cudaSuccess) std::cout << cudaGetErrorString(err) << "\n";

int Matrix::count = 0;

float prec1 = 1000000;

\_\_global\_\_ void setZeros(float\* arr, int \_rows, int \_columns) {

for (int i = 0; i < \_rows; ++i)

for (int j = 0; j < \_columns; ++j)

arr[i \* \_columns + j] = 0;

}

Matrix::Matrix(int rows, int columns) : \_rows(rows), \_columns(columns) {

ERRORCHECKF(cudaMalloc(&arr, \_rows \* \_columns \* sizeof(float)));

setZeros << <1, 1 >> > (arr, \_rows, \_columns);

cudaDeviceSynchronize();

}

Matrix::Matrix(const Matrix& other) :Matrix(other.\_rows, other.\_columns) {

//cudaMalloc(&arr, SIZE);

ERRORCHECKF(cudaMemcpy(arr, other.arr, SIZE, cudaMemcpyDeviceToDevice));

}

Matrix::Matrix(Matrix&& other) {

arr = other.arr;

\_columns = other.\_columns;

\_rows = other.\_rows;

other.arr = nullptr;

}

Matrix::Matrix() :\_rows(0), \_columns(0) {

}

Matrix& Matrix::operator=(const Matrix& other) {

ERRORCHECKF(cudaFree(arr));

ERRORCHECK(cudaMalloc(&arr, SIZE));

ERRORCHECK(cudaMemcpy(arr, other.arr, SIZE, cudaMemcpyDeviceToDevice));

return \*this;

}

Matrix::~Matrix() {

cudaFree(arr);

}

\_\_global\_\_ void cudaGet(int i, int j, float\* arr, float\* ref, int \_rows, int \_columns) {

//cudaMemcpy(ref,&(arr[i \* \_columns + j]), sizeof(float), cudaMemcpyDeviceToHost);

}

float Matrix::get(int i, int j) const {

float ref = 0.1f;

cudaMemcpy(&ref, &(arr[i \* \_columns + j]), sizeof(float), cudaMemcpyDeviceToHost);

//cudaGet << <1, 1 >> > (i, j, arr, &ref, \_rows, \_columns);

//ERRORCHECKF(cudaDeviceSynchronize());

return ref;

}

\_\_global\_\_ void cudaSet(int i, int j, float\* arr, float value, int \_rows, int \_columns) {

arr[i \* \_columns + j] = value;

}

void Matrix::set(int i, int j, float value) {

cudaSet << <1, 1 >> > (i, j, arr, value, \_rows, \_columns);

cudaDeviceSynchronize();

}

\_\_global\_\_ void addMatrix(float\* dst, float\* src, int \_rows, int \_columns) {

for (int i = 0; i < \_rows \* \_columns; ++i)

dst[i] += src[i];

}

Matrix& Matrix::operator+=(const Matrix& other) {

addMatrix << <1, 1 >> > (arr, other.arr, \_rows, \_columns);

cudaDeviceSynchronize();

return \*this;

}

Matrix Matrix::operator+(const Matrix& other) {

Matrix m(\*this);

return m += other;

}

\_\_global\_\_ void multOnFloat(float\* src, float l, int \_rows, int \_columns) {

for (int i = 0; i < \_rows \* \_columns; ++i)

src[i] \*= l;

}

Matrix& Matrix::operator\*=(float l) {

multOnFloat << <1, 1 >> > (arr, l, \_rows, \_columns);

cudaDeviceSynchronize();

return\*this;

}

Matrix Matrix::operator\*(float l) {

Matrix m(\*this);

return m \*= l;

}

\_\_global\_\_ void divOnFloat(float\* src, float l, int \_rows, int \_columns) {

for (int i = 0; i < \_rows \* \_columns; ++i)

src[i] /= l;

}

Matrix& Matrix::operator/=(float l) {

divOnFloat << <1, 1 >> > (arr, l, \_rows, \_columns);

cudaDeviceSynchronize();

return\*this;

}

Matrix Matrix::operator/(float l) {

Matrix m(\*this);

return m /= l;

}

Matrix& Matrix::operator=(Matrix&& other) {

cudaFree(arr);

arr = other.arr;

\_columns = other.\_columns;

\_rows = other.\_rows;

other.arr = nullptr;

return \*this;

}

\_\_global\_\_ void multRow(float\* src, int row, float l, int rows, int columns) {

for (int i = columns \* (row); i < (row + 1) \* columns; ++i)

src[i] \*= l;

}

void Matrix::MultiplyRow(int row, float l) {

//std::cout << "columns " << \_columns << " blocks " << \_blocksForRow << " threads " << \_threadsForRow << "\n";

multRow << <1, 1 >> > (arr, row, l, \_rows, \_columns);

cudaDeviceSynchronize();

}

\_\_global\_\_ void plusRows(float\* src, int rowDst, int rowSrc, int rows, int columns) {

for (int i = (rowDst - 1) \* columns, i1 = (rowSrc - 1) \* columns; i < rowDst \* columns; ++i, ++i1)

src[i] += src[i1];

}

void Matrix::PlusRows(int row1, int row2) {

plusRows << <1, 1 >> > (arr, row1, row2, \_rows, \_columns);

cudaDeviceSynchronize();

}

\_\_global\_\_ void minusRows(float\* src, int rowDst, int rowSrc, int rows, int columns) {

for (int i = (rowDst) \* columns, i1 = rowSrc \* columns; i < (rowDst + 1) \* columns; ++i, ++i1)

src[i] -= src[i1];

}

void Matrix::MinusRows(int row1, int row2) {

minusRows << <1, 1 >> > (arr, row1, row2, \_rows, \_columns);

cudaDeviceSynchronize();

}

\_\_global\_\_ void swapRows(float\* src, int row1, int row2, int rows, int columns) {

for (int i = (row1 - 1) \* columns, i1 = (row2 - 1) \* columns; i < row1 \* columns; ++i, ++i1) {

float tmp = src[i];

src[i] = src[i1];

src[i1] = tmp;

}

}

void Matrix::swapLines(int line1, int line2) {

swapRows << <1, 1 >> > (arr, line1, line2, \_rows, \_columns);

cudaDeviceSynchronize();

}

\_\_global\_\_ void currentColumnToZero(int current, float\* arr, int \_rows, int \_columns) {

for (int i = current + 1; i < \_rows; ++i) {

float c = arr[i \* \_columns + current];

for (int j = current; j < \_columns; ++j) {

float f = arr[current \* \_columns + j];

f \*= c;

arr[i \* \_columns + j] -= f;

}

}

}

void Matrix::ToUpTriangle() {

for (int i = 0; i < \_rows; ++i) {

float ii = get(i, i);

if (ii == 0) continue;

MultiplyRow(i, 1 / get(i, i));

if (i != \_rows - 1) currentColumnToZero << <1, 1 >> > (i, arr, \_rows, \_columns);

cudaDeviceSynchronize();

ERRORCHECKF(cudaGetLastError());

}

}

\_\_global\_\_ void backMoveFunc(float\* x, float\* arr, int current, int \_rows, int \_columns) {

for (int i = 0; i < current; ++i) {

x[i] += x[current] \* arr[i \* \_columns + current];

}

}

float\* Matrix::backMove() {

float\* x\_d;

cudaMalloc(&x\_d, sizeof(float) \* \_rows);

setZeros << <1, 1 >> > (x\_d, 1, \_rows);

cudaDeviceSynchronize();

for (int i = \_rows - 1; i >= 0; --i) {

if (i == \_rows - 1) {

cudaSet << <1, 1 >> > (0, i, x\_d, get(i, \_columns - 1), \_rows, \_columns);

}

else {

float sum = 0.1f;

cudaMemcpy(&sum, &(x\_d[i]), sizeof(float), cudaMemcpyDeviceToHost);

cudaSet << <1, 1 >> > (0, i, x\_d, get(i, \_columns - 1) - sum, \_rows, \_columns);

cudaDeviceSynchronize();

}

if (i != 0) {

backMoveFunc << <1, 1 >> > (x\_d, arr, i, \_rows, \_columns);

cudaDeviceSynchronize();

}

}

float\* x = new float[\_rows];

cudaMemcpy(x, x\_d, sizeof(float) \* \_rows, cudaMemcpyDeviceToHost);

return x;

}

Matrix::operator std::string() const {

std::string ans = "";

for (int i = 0; i < \_rows; i++) {

for (int j = 0; j < \_columns; ++j)

ans += std::to\_string(get(i, j)) + " ";

ans += "\n";

}

return ans;

}

\_\_global\_\_ void forMult(float\* dst, float\* arr1, float\* arr2, int \_rows, int \_columns) {

for (int i = 0; i < \_rows; ++i) {

for (int j = 0; j < \_columns; ++j) {

dst[i \* \_columns + j] = arr1[i \* \_columns + i] \* arr2[j \* \_columns + j];

}

}

}

Matrix Matrix::Multiply(const Matrix& other) {

Matrix dst(\_rows, \_columns);

forMult << <1, 1 >> > (dst.arr, this->arr, other.arr, \_rows, \_columns);

cudaDeviceSynchronize();

return dst;

}

\_\_global\_\_ void forDiffSquare(float\* arr, int current, int \_rows, int \_columns) {

for (int i = 0; i < \_rows; ++i) {

if (i == current) continue;

for(int j = 0; j < \_columns; ++j) {

if (j == current) continue;

arr[i \* \_columns + j] = 0;

}

}

}

Matrix Matrix::DiffSquare(int variable) {

Matrix ret(\*this);

forDiffSquare << <1, 1 >> > (ret.arr, variable, \_rows, \_columns);

cudaDeviceSynchronize();

return ret;

}

\_\_global\_\_ void increm(float\* arr, int i, int j, float inc, int \_rows, int \_columns) {

arr[i \* \_columns + j] += inc;

}

void Matrix::Increase(int i, int j, float inc) {

increm << <1, 1 >> > (arr, i, j, inc, \_rows, \_columns);

cudaDeviceSynchronize();

}

\_\_global\_\_ void copyDiffToMatrixFunc(float\* arr, float\* diffArr, int variable, int \_rows, int \_columns, int tfr) {

for (int j = 1; j < \_columns ; ++j) {

arr[(variable - 1) \* \_columns + j - 1] =

diffArr[(variable)\*\_columns + j] + diffArr[(j)\*\_columns + variable];

}

}

void Matrix::CopyDiffToMatrix(Matrix& diff, int variable) {

copyDiffToMatrixFunc << <1, 1 >> > (arr, diff.arr, variable, \_rows, \_columns, \_threadsForRow);

cudaDeviceSynchronize();

set(variable - 1, \_columns - 1, -(diff.get(variable, 0) + diff.get(0, variable)));

}

\_\_global\_\_ void writeToDiagFunc(float\* dst, float\* src, int \_rows, int \_columns, int tfr) {

for (int i = 0; i < \_rows; ++i) {

dst[i \* \_columns + i] = src[i];

}

}

void Matrix::WriteToDiag(float\* diagArr) {

float\* src;

cudaMalloc(&src, \_columns \* sizeof(float));

cudaMemcpy(src, diagArr, \_columns \* sizeof(float), cudaMemcpyHostToDevice);

writeToDiagFunc << <1, 1 >> > (arr, src, \_rows, \_columns, \_threadsForRow);

cudaDeviceSynchronize();;

cudaFree(src);

}

*GpuUnoptimized:kernel.cu:*

#include <iostream>

#include "FuncRow.h"

#include "Matrix.h"

#include <functional>

const float PI = 3.1415926f;

typedef std::function<Matrix(float x)> CoeffMatrixCounter;

typedef std::function<float(float)> Function;

struct Piece {

float a;

float b;

};

struct Cond {

float y1;

float y2;

};

float prec2 = 10000000;

Matrix\* GetY(CoeffMatrixCounter f, Piece piece, int numbOfPieces) {

float h = (piece.b - piece.a) / numbOfPieces;

float x = piece.a;

Matrix\* Y = new Matrix[numbOfPieces + 1];

for (int i = 0; i < numbOfPieces + 1; ++i) {

Y[i] = f(x);

x += h;

}

return Y;

}

Generator Anis(Cond cond, Piece piece) {

return [cond, piece](int i)->Polynom {

if (i == 0) {

float k = (cond.y2 - cond.y1) / (piece.b - piece.a);

float c = cond.y1 - k \* piece.a;

float arr[] = { c,k,0 };

return Polynom(arr);

}

float arr2[3] = { piece.a \* piece.b, -(piece.a + piece.b), 1 };

Polynom polly(arr2);

return polly.IncreasePow(i - 1);

};

}

Matrix IntSimpson(CoeffMatrixCounter f, int numbOfPieces, Piece piece) {

float h = (piece.b - piece.a) / numbOfPieces;

float x = piece.a;

Matrix s = f(x) + f(piece.b);

x += h;

//std::cout << std::string(s) << "\n\n";

for (int i = 1; i < numbOfPieces; ++i) {

if (i % 2 == 0) s += f(x) \* 2;

else s += f(x) \* 4;

x += h;

}

return s \* (h / 3);

}

Matrix Mult(Matrix& m1, Matrix& m2) {

return m1.Multiply(m2);

}

Matrix DiffSquare(Matrix& m, int variable) {

return m.DiffSquare(variable);

}

FuncRow MnkInt(Function p, Function q, Function f, int numbOfMembers, Generator memb, int numbOfPieces,

Piece piece) {

FuncRow frow(numbOfMembers, memb);

auto lambda =

[numbOfMembers, frow, p, q, f](float x)->Matrix {

static int counter = 0;

Matrix A(numbOfMembers, numbOfMembers);

float\* incArr = new float[numbOfMembers];

for (int i = 0; i < numbOfMembers; ++i) {

incArr[i] = frow[i].Diff().Diff().Count(x);

incArr[i] += frow[i].Diff().Count(x) \* p(x);

incArr[i] += frow[i].Count(x) \* q(x);

}

incArr[0] -= f(x);

A.WriteToDiag(incArr);

delete[] incArr;

auto M = Mult(A, A);

++counter;

return M;

};

Matrix intgr = IntSimpson(lambda, numbOfPieces, piece);

Matrix matr(numbOfMembers - 1, numbOfMembers);

for (int i = 1; i < numbOfMembers; ++i) {

Matrix df = DiffSquare(intgr, i);

matr.CopyDiffToMatrix(df, i);

}

matr.ToUpTriangle();

//std::cout << "После треуголирования\n" << std::string(matr) << "\n\n";

float\* coeffs = matr.backMove();

for (int i = 1; i < numbOfMembers; ++i) frow[i] \*= coeffs[i - 1];

delete[] coeffs;

return frow;

}

void ShowDataTest(FuncRow frow, Function ans, Piece piece, int numbOfPoints) {

float h = (piece.b - piece.a) / numbOfPoints;

float x = piece.a;

float\* delt = new float[numbOfPoints + 1];

for (int i = 0; i <= numbOfPoints; ++i, x += h) {

delt[i] = abs(frow.Count(x) - ans(x));

}

float Norm = 0;

for (int i = 0; i < numbOfPoints + 1; ++i) if (Norm < delt[i]) Norm = delt[i];

delete[] delt;

std::cout << "Норма невязки: " << Norm << "\n";

}

void dotest(Function p\_t, Function q\_t, Function f\_t, Function lambda, Piece piece,

Cond cond, int i, int numbOfMembers, int numbOfPoints) {

std::cout << "Интегральный МНК Тестовый пример " << i << "\n";

auto beg = clock();

FuncRow frow = MnkInt(p\_t, q\_t, f\_t, numbOfMembers, Anis(cond, piece), 30, piece);

auto end = clock();

std::cout << "Время выполнения " << end - beg << " миллисекунд" << "\n";

ShowDataTest(frow, lambda, piece, numbOfPoints);

}

int main() {

setlocale(0, "");

Matrix A(5, 5);

A.Increase(1, 1, -2);

std::cout << std::string(A) << "\n\n";

A.Increase(2, 2, 2);

int i = 1;

Function p\_t = [](float x)->float { return 0; };

Function q\_t = [](float x) { return 1; };

Function f\_t = [](float x)->float { return x \* x + 3 \* x - 7; };

Function lambda = [](float x)->float { return x \* x + 3 \* x - 9; };

Piece piece\_t = { 0, 1 };

Cond cond\_t = { -9, -5 };

const int numbOfMembers = 1000;

const int numbOfPoints = 20;

dotest(p\_t, q\_t, f\_t, lambda, piece\_t, cond\_t, i, numbOfMembers, numbOfPoints);

Function e3 = [](float x)->float {return expf(3 \* x); };

p\_t = [](float x)->float {return x \* x + 6; };

q\_t = [](float x)->float {return pow(2, -x); };

f\_t = [e3](float x)->float {return 9 \* e3(x) + 3 \* (x \* x + 6) \* e3(x) + pow(2, -x) \* e3(x); };

lambda = [](float x)->float {return expf(3 \* x); };

piece\_t = { 0, 1 };

cond\_t = { 1, expf(3) };

i = 2;

dotest(p\_t, q\_t, f\_t, lambda, piece\_t, cond\_t, i, numbOfMembers, numbOfPoints);

p\_t = [](float x)->float {return 0; };

q\_t = [](float x)->float {return 1; };

f\_t = [](float x)->float {return 2 \* x - PI; };

lambda = [](float x)->float {return 2 \* x - PI + PI \* cos(x); };

cond\_t = { 0, lambda(1) };

piece\_t = { 0, 1 };

i = 3;

dotest(p\_t, q\_t, f\_t, lambda, piece\_t, cond\_t, i, numbOfMembers, numbOfPoints);

p\_t = [](float x)->float {return 2; };

q\_t = [](float x)->float {return 1; };

f\_t = [](float x)->float {return 0; };

lambda = [](float x)->float {return expf(-x) + x \* expf(-x); };

piece\_t = { 0, 1 };

cond\_t = { lambda(0), lambda(1) };

i = 4;

dotest(p\_t, q\_t, f\_t, lambda, piece\_t, cond\_t, i, numbOfMembers, numbOfPoints);

}

**ПРИЛОЖЕНИЕ Б**

**(обязательное)**

**Функциональная схема алгоритма, реализующего программное средство**

**ПРИЛОЖЕНИЕ В**

**(обязательное)**

**Блок схема алгоритма, реализующего программное средство**

**ПРИЛОЖЕНИЕ Г**

**(обязательное)**

**Графики сравнения производительности**

**ПРИЛОЖЕНИЕ Д**

**(обязательное)**

**Ведомость документов**