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 ПАРАЛЛЕЛЬНОГО ВЫПОЛНЕНИЯ ОПЕРАЦИЙ НАД МАТРИЦАМИ НА GPU (NVIDIA GEFORCE RTX 3060 LAPTOP) С CUDA И CPU (INTEL CORE I5-11400H)**

БГУИР КП 1-40 04 01 01 007

Студент Т. П. Власенко

Руководитель А. Н. Марков

Нормоконтролер А. А. Калиновская

Минск 2023

**СОДЕРЖАНИЕ**

[Введение 4](#_Toc152756559)

[1 Архитектура вычислительной системы 5](#_Toc152756560)

[2 Платформа программного обеспечения 11](#_Toc152756561)

[3 Теоретическое обоснование разработки программного продукта 17](#_Toc152756562)

[3.1 Обоснование необходимости разработки 17](#_Toc152756563)

[3.2 Технологии программирования, используемые для решения поставленных задач 18](#_Toc152756564)

[3.3 Связь архитектуры вычислительной системы с разрабатываемым программным обеспечением 23](#_Toc152756565)

[4 Проектирование функциональных возможностей программы 24](#_Toc152756566)

[4.1 Настройки запуска 24](#_Toc152756567)

[4.2 Входные данные 24](#_Toc152756568)

[4.3 Алгоритм умножения матриц 24](#_Toc152756569)

[4.4 Оценка производительности 26](#_Toc152756570)

[4.5 Общая структура программ 27](#_Toc152756571)

[4.6 Описание функциональной схемы алгоритма программы 27](#_Toc152756572)

[4.7 Описание блок-схемы алгоритма программы 28](#_Toc152756573)

[5 Архитектура разрабатываемой программы 29](#_Toc152756574)

[5.1 Запуск программ 29](#_Toc152756575)

[5.2 Запуск на CPU на одном ядре с использованием HTT и без HTT 29](#_Toc152756576)

[5.3 Запуск на CPU на всех ядрах с использованием HTT и без HTT 30](#_Toc152756577)

[5.4 Запуск на GPU на одном CUDA ядре 32](#_Toc152756578)

[5.5 Запуск на GPU на многих CUDA ядрах 32](#_Toc152756579)

[5.6 Анализ полученных результатов 33](#_Toc152756580)

[Заключение 39](#_Toc152756585)

[Список использованных источников 40](#_Toc152756586)

[Приложение А (обязательное) Ведомость 42](#_Toc152756587)

[Приложение Б (обязательное) Исходный код 43](#_Toc152756589)

[Приложение В (обязательное) Функциональная схема алгоритма 53](#_Toc152756590)

[Приложение Г (обязательное) Блок-схема алгоритма 54](#_Toc152756591)

[Приложение Д (обязательное) Графический интерфейс пользователя 55](#_Toc152756592)

# ВВЕДЕНИЕ

В современном мире задачи обработки и анализа больших объемов данных становятся все более распространенными. Матричные операции являются важной составной частью многих вычислительных задач, таких как машинное обучение, обработка сигналов, компьютерное зрение и другие. Параллельное выполнение операций над матрицами на графическом процессоре (*GPU*) с использованием *CUDA* и на центральном процессоре (*CPU*) представляет собой активно развивающуюся область исследований.

Центральные процессоры и *GPU* обладают различными архитектурами и специализированными возможностями, поэтому для повышения эффективности вычислений следует учитывать эти особенности.

Целью данного курсового проекта является сравнение производительности параллельного выполнения операций над матрицами на *GPU* (*NVIDIA* *GeForce* *RTX* 3060 *laptop*) с использованием технологии *CUDA* и на *CPU* (*Intel* *Core* i5-11400*H*) в целях определения эффективности каждого подхода.

В ходе работы будут рассмотрены способы реализации многопоточных алгоритмов работы с матрицами (умножение) на *GPU* и *CPU*.

В результате данного проекта будут реализованы несколько программ, выполняющих операции над матрицами на *GPU* и на *CPU*. Критерием сравнения будет выступать время, затраченное на выполнение программы. Полученные результаты необходимо проанализировать и сделать вывод, в каких случаях параллельное выполнение на *GPU* и *CPU* наиболее эффективно.

Курсовой проект оформлен с помощью СТП 01-2017 (стандарт предприятия). [1]

# 1 АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНОЙ СИСТЕМЫ

В данном разделе описаны основные компоненты вычислительной системы, их взаимодействие и особенности.

Центральный процессор (ЦП) является основным исполнительным устройством компьютера. Он обрабатывает и выполняет инструкции программы, управляет памятью и взаимодействует с другими компонентами системы. При параллельном выполнении матричных операций следует учитывать следующие аспекты:

**Многопоточность**: современные ЦП поддерживают многопоточность, что позволяет выполнять несколько потоков инструкций одновременно. Правильное использование многопоточности может значительно повысить производительность алгоритма умножения матриц. Необходимо учитывать количество ядер и потоков для эффективного распараллеливания.

**Кэш-память**: ЦП обычно имеет несколько уровней кэш-памяти различной емкости. Кэш-память используется для временного хранения данных, которые часто используются процессором. Эффективное использование кэш-памяти может существенно сократить временные задержки и улучшить производительность.

Процессор, который был использован в данном курсовом проекте – *Intel* *Core* *i*5-11400*H*:

1. Архитектура: *x*86;
2. Количество ядер: 6;
3. Количество потоков: 12;
4. Базовая тактовая частота: 2.7 ГГц;
5. Максимальная тактовая частота: 4.5 ГГц;
6. Кэш-память *L*1: 480 КБ;
7. Кэш-память *L*2: 7680 КБ;
8. Кэш-память *L*3: 12 МБ;
9. Техпроцесс: 10 нм;
10. Интегрированная графика: *Intel* *UHD* *Graphics* 16*EU* *Mobile;*
11. Поддержка памяти: *DDR*4 (до 3200 МГц);
12. *TDP*: 45 Вт. [2]

Intel Core i5-11400H является мобильным процессором из серии *Tiger* *Lake*-*H* 11-го поколения от компании *Intel*. Он представляет собой мощный процессор, разработанный специально для использования в ноутбуках. *Intel* *Core* *i*5-11400*H* оснащен шестью физическими ядрами и двенадцатью логическими ядрами, что позволяет обрабатывать до двенадцати потоков инструкций. Это обеспечивает высокую производительность и позволяет справляться с множеством задач.

Архитектура *Tiger Lake-H* обеспечивает улучшенные характеристики производительности и энергоэффективности. Она использует передовые технологии, такие как улучшенный кэш, для оптимизации работы процессора и ускорения выполнения задач. C ее появлением производительность в однопоточном режиме выросла на 32 процента, а в многопоточном – на 65 процентов. [3]

Для графических вычислений *Core* *i*5-11400*H* использует встроенную графику *Intel UHD Graphics*, которая предоставляет базовые графические возможности. Однако, процессор также поддерживает дискретные графические карты, что делает его подходящим для игр и других графически интенсивных приложений.

Также данный процессор оснащен технологией *Hyper-Threading* (*HTT*), которая позволяет виртуально удваивать количество ядер в процессоре. Когда *HT* выключен, то каждое физическое ядро ведет себя как одно логическое (рисунок 1.1). Когда *HT* включен, каждое физическое ядро процессора может обрабатывать несколько потоков одновременно, работая как два логических ядра (рисунок 1.2). [4]

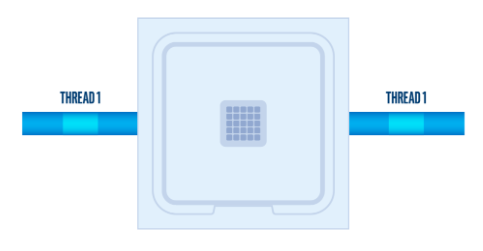


Рисунок 1.1 – Физическое ядро работает как одно логическое ядро без применения *HTT*

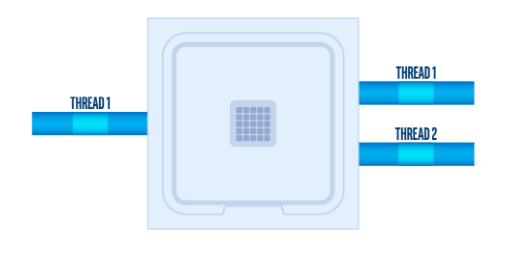


Рисунок 1.2 – Физическое ядро работает как два логических ядра при применении *HTT*

В основе *Hyper-Threading* лежит идея о том, что во время выполнения задачи процессор может использовать не все свои ресурсы. Некоторые стадии выполнения задачи могут быть блокированы ожиданием данных или доступа к памяти. В таких случаях, вместо простоя процессора, *HT* позволяет ему переключаться на другую задачу, которая может быть обработана параллельно. Таким образом, одно физическое ядро может эффективно обрабатывать два потока инструкций.

В целом, *Intel* *Core* *i*5-11400*H* – это мощный мобильный процессор, который обеспечивает высокую производительность, энергоэффективность и поддерживает широкий спектр задач.

Графический процессор (ГП) является мощным устройством для параллельной обработки данных. Он специально разработан для выполнения графических и высокопроизводительных вычислений. При параллельном выполнении матричных операций следует учитывать следующие аспекты:

**Параллелизм**: *GPU* состоит из множества вычислительных ядер, которые могут выполнять одну и ту же операцию над различными данными параллельно. Параллельное программирование и разделение задач на потоки являются ключевыми аспектами оптимизации вычислений на GPU. Учитывая особенности технологии *CUDA* (*Compute* *Unified* Device *Architecture*), необходимо правильно подобрать размер сетки блоков и размер самих блоков.

**Память**: *GPU* имеет свою собственную память, называемую глобальной памятью. Она имеет большую пропускную способность, но также имеет высокую задержку доступа. Эффективное использование локальной, разделяемой и константной памяти может существенно повысить производительность вычислений на *GPU*.

Графический процессор, который был использован в рамках данного курсового проекта – *NVIDIA RTX GeForce* 3060 *laptop*:

1. Архитектура: *NVIDIA Ampere;*
2. Количество *CUDA*-ядер: 3840;
3. Базовая частота равна 900 МГц;
4. *Boost*-частота 1425 МГц;
5. Техпроцесс: 8 нм;
6. Видеопамять: 6 ГБ *GDDR*6;
7. Ширина шины памяти: 192 бит;
8. Пропускная способность памяти: 336 Гбит/с;
9. Количество *SM*: 30;
10. Тензорных ядер: 120;
11. *RT* ядер: 30;
12. *L*1 кэш: 128 KB (на один SM);
13. *L*2 кэш: 3 MB ;
14. Поддержка технологий: *CUDA*, *OpenCL* 3.0, *DirectX*, *Vulkan*, *PhysX*, *OpenGL* 4.6;
15. *TDP*: 80 Вт. [5]

*NVIDIA GeForce RTX* 3060 *laptop* – это мощный графический процессор, специально разработанный для использования в ноутбуках. Он предлагает высокую производительность и передовые графические возможности, что делает его подходящим выбором для игр и выполнения графически интенсивных задач.

Архитектура *NVIDIA Ampere* представляет собой поколение графических архитектур, разработанное компанией *Nvidia*. Она была представлена в 2020 году и стала основой для графических процессоров серии *GeForce* *RTX* 30 (строение *SM* блока представлено на рисунке 1.3).

Предшественником архитектуры *Nvidia* *Ampere* была архитектура *Nvidia* *Turing*, которая была выпущена в 2018 году. Архитектура *Turing* представила ряд инноваций в графических процессорах, включая улучшенную архитектуру ядра, расширенные возможности машинного обучения с помощью тензорных ядер (*Tensor* *Cores*) и встроенную аппаратную поддержку трассировки лучей с помощью *RT* *Cores*.



Рисунок 1.3 – *SM* блок в *Nvidia* *Ampere* архитектуре

Некоторые преимущества архитектуры *Ampere* перед архитектурой Turing (рисунок 1.4):

1. Увеличенная производительность: архитектура *Ampere* может похвастаться значительным повышением производительности по сравнению с архитектурой *Turing*. Она предлагает более широкие и эффективные ядра *CUDA*, улучшенные блоки исполнения и более высокую ширину памяти, что позволяет достигать лучшей общей производительности в широком спектре графических задач.

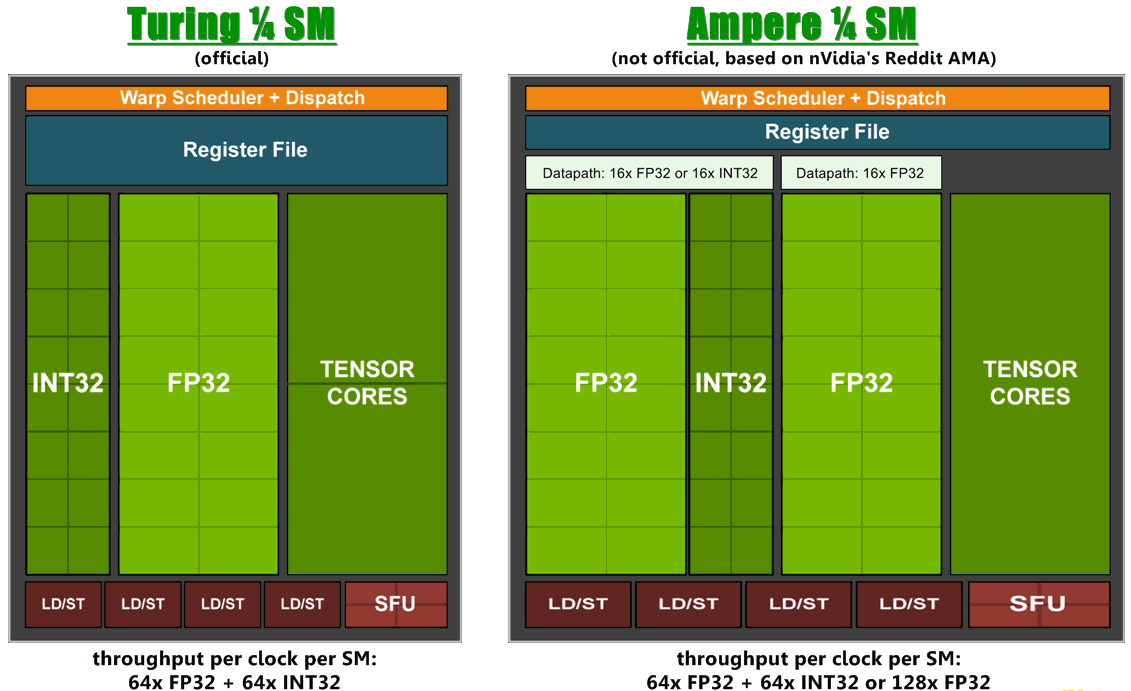


Рисунок 1.4 – Сравнение *SM* в *Turing* и *Ampere*

1. Улучшенная энергоэффективность: *Ampere* включает в себя улучшенные техники управления энергопотреблением и более эффективные блоки исполнения, что позволяет достигать высокой производительности при более низком энергопотреблении.
2. Улучшенная трассировка лучей: *Ampere* внедряет новое поколение ядер *RT* (*Ray Tracing*). Они обладают улучшенной производительностью и эффективностью по сравнению с *RT* ядрами в архитектуре *Turing*, что позволяет достигать лучшей визуальной реалистичности в играх и приложениях, использующих трассировку лучей. Ядра *RT* второго поколения архитектуры *Ampere* позволяют значительно ускорить такие процессы, как фотореалистичный рендеринг киноматериалов, оценка архитектурных проектов, виртуальное прототипирование продуктов, рендеринг сцен движения и другие. [6]

В целом, *NVIDIA GeForce RTX* 3060 *Laptop* представляет собой мощное графическое решение для ноутбуков, обеспечивающее высокую производительность и передовые графические возможности.

В данном разделе были рассмотрены ключевые характеристики центрального и графического процессора, которые были использованы в данном курсовом проекте, их архитектура и особенности, которые следует учитывать при разработке приложений для вычислений над матрицами.

# 2 ПЛАТФОРМА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Платформа разработки основана на операционной системе *Windows*.

*Windows* – операционная система, разработанная и распространяемая компанией *Microsoft*.

*Windows* обладает обширным функциональным набором. Она предоставляет графический интерфейс пользователя, который облегчает взаимодействие с компьютером. Рабочий стол в *Windows* позволяет пользователю размещать ярлыки, файлы и папки для быстрого доступа. Оконное управление позволяет открывать и управлять несколькими приложениями и окнами одновременно.

Операционная система *Windows* обладает встроенными сетевыми возможностями, что позволяет подключаться к Интернету, обмениваться данными в локальных сетях и управлять сетевыми ресурсами. Безопасность также является важной составляющей *Windows*. Система предлагает механизмы защиты, включая антивирусную защиту, брандмауэры и систему обновлений для обеспечения безопасности и стабильности работы.

Ядро *Windows*, также известное как *NT*-ядро (*Windows NT Kernel*), является центральной составляющей операционной системы Windows. Оно предоставляет основные функции управления ресурсами компьютера и обеспечивает взаимодействие между аппаратным и программным обеспечением.

В *NT*-ядре *Windows* реализованы такие ключевые компоненты, как управление памятью, ввод-вывод, обработка прерываний, драйверы устройств и системные вызовы. Он предоставляет абстракции и интерфейсы для работы с аппаратными ресурсами, такими как процессоры, память, диски, сетевые карты и другие устройства.

Ядро *Windows* обеспечивает многозадачность, что позволяет одновременно выполнять несколько процессов. Оно содержит планировщик задач, который определяет, каким образом и когда каждое приложение или процесс получает доступ к ресурсам системы, чтобы обеспечить рациональное распределение ресурсов и общую отзывчивость системы.

Предложенная компанией *Nvidia* технология *CUDA* (*Compute* *Unified* *Device* *Architecture*) заметно облегчает написание *GPGPU*-приложений. Она не использует графических *API* и свободна от ограничений, свойственных этим *API*. Данная технология предназначена для разработки приложений для массивно-параллельных вычислительных устройств. На сегодняшний момент поддерживаемыми устройствами являются все *GPU* компании *Nvidia*, начиная с серии *GeForce*8, а также специализированные для решения расчетных задач *GPU* семейства *Tesla*.

Основными преимуществами технологии *CUDA* являются ее простота – все программы пишутся на расширении языке *С*, наличие хорошей документации, набор готовых инструментов, включающих профайлер, набор готовых библиотек, кроссплатформенность (поддерживаются *Microsoft* *Windows*, *Linux* и *Mac* *OS* *X*).

*CUDA* строится на концепции, что *GPU* (называемый устройством, *device*) выступает в роли массивно-параллельного сопроцессора к центральному процессору (называемому *host*). Программа на *CUDA* задействует как *CPU*, так и *GPU*. При этом обычный (последовательный, то есть непараллельный) код выполняется на *CPU*, а для массивно-параллельных вычислений соответствующий код выполняется на *GPU* как набор одновременно выполняющихся нитей (потоков, *threads*). [7]

*NVIDIA CUDA Toolkit* – это пакет разработчика, предоставляемый *NVIDIA*, который включает в себя необходимые инструменты и библиотеки для разработки приложений, использующих *CUDA*. *CUDA Toolkit* включает в себя компилятор *CUDA*, библиотеки *CUDA* *Runtime*, инструменты для профилирования и отладки, а также другие компоненты.

Перед появлением *CUDA*, использование *GPU* ограничивалось преимущественно графическими вычислениями для отображения графики и обработки видео. Однако, с развитием *GPU* и его параллельных вычислительных возможностей, стало ясно, что *GPU* может быть использован для выполнения других задач, требующих большой вычислительной мощности.

*CUDA* была разработана для упрощения программирования на *GPU*, чтобы разработчики могли использовать его параллельные вычислительные возможности для общего назначения. Согласно модели *CUDA*, программист разбивает задачу на блоки, а блоки на потоки:

1. Каждый блок будет полностью выполнен на выделенном ему *SM*.
2. Распределением блоков по *SM* занимается *GPU*, не программист.
3. Все потоки блока *X* будут разбиты на группы, называемые *warps* (варпы), и выполнены на *SM*. Размер этих групп зависит от модели *GPU*. Все потоки из одного варпа выполняются одновременно, занимая определенную часть ресурсов *SM*. Причем они либо выполняют одну и ту же инструкцию (но на разных данных), либо простаивают. [8]

Первая версия *CUDA* была выпущена в 2007 году. С течением времени *CUDA* продолжала развиваться и улучшаться. *NVIDIA* выпускала новые версии *CUDA* *Toolkit*, добавляя новые функции, оптимизации и поддержку новых архитектур *GPU*. *CUDA* стала популярной в научных исследованиях, высокопроизводительных вычислениях, машинном обучении и других областях, где требуется параллельная обработка больших объемов данных.

*CUDA* также предоставляет доступ к различным библиотекам, таким как *cuBLAS* (библиотека линейной алгебры), *cuFFT* (быстрое преобразование Фурье), *cuDNN* (библиотека глубокого обучения) и другим, которые упрощают разработку и оптимизацию приложений.

Сегодня *CUDA* остается одной из ведущих платформ для параллельных вычислений на *GPU* и продолжает развиваться, предоставляя разработчикам средства для эффективного использования современных графических процессоров в различных областях применения.

Для работы с *CUDA* необходимо установить драйвер *NVIDIA* для имеющейся видеокарты. Драйвер обеспечивает взаимодействия между приложением и аппаратным обеспечением *GPU*.

В качестве среды разработки была выбрана *Visual Studio* 2022. *Visual* *Studio* – это мощное средство разработчика, которое можно использовать для выполнения всего цикла разработки в одном месте. Это комплексная интегрированная среда разработки (*IDE*), которую можно использовать для написания, редактирования, отладки и сборки кода, а затем развертывания приложения. Помимо редактирования и отладки кода *Visual* *Studio* включает компиляторы, расширения и многое другое, чтобы улучшить каждый этап процесса разработки программного обеспечения. [9]

Среда разработки *Visual* *Studio* обладает следующими особенностями:

**Обширный набор инструментов**: *Visual Studio* предоставляет разработчикам широкий набор инструментов для разработки приложений различных типов, включая языки программирования *C*++, *C*#, *C* и другие. Она поддерживает разработку приложений для различных платформ, включая Windows, веб и мобильные платформы.

**Отладка и профилирование**: *Visual Studio* предоставляет мощные инструменты для отладки кода, включая установку точек останова, пошаговое выполнение, просмотр значений переменных и другие функции. Она также поддерживает профилирование приложений, позволяя разработчикам исследовать производительность и оптимизировать свой код.

**Интегрируется** с другими инструментами и платформами *Microsoft*, такими как *Azure*, *SQL* *Server* и другими. Это позволяет разработчикам удобно работать с различными сервисами и инфраструктурой *Microsoft*.

В дополнение к *Visual Studio* использовались *NVIDIA Nsight Compute* и *NVIDIA Nsight Systems*.

*NVIDIA Nsight Compute* – это инструмент разработки и профилирования, предоставляемый компанией *NVIDIA*, который помогает разработчикам анализировать и оптимизировать вычислительные приложения, работающие на графических процессорах (*GPU*) от *NVIDIA*. [10]

*Nsight Compute* предоставляет подробную информацию о выполнении ядер *GPU*, позволяя разработчикам получить глубокое понимание производительности и эффективности своего кода. С помощью этого инструмента можно профилировать и анализировать различные аспекты работы приложения на *GPU*, включая использование регистров, памяти, арифметических инструкций и других ключевых ресурсов.

Одной из основных возможностей *Nsight* *Compute* является генерация графического представления временной линии выполнения, которая позволяет визуализировать и анализировать, как различные ядра и потоки выполняются на *GPU*. Также инструмент предоставляет детальные метрики производительности, такие как количество инструкций, загрузка и промахи кэша.

*Nsight* *Compute* также предоставляет возможности для оптимизации кода на уровне инструкций и памяти. Разработчики могут исследовать производительность своего кода, выявлять узкие места и искать способы оптимизации, чтобы достичь максимальной эффективности выполнения на *GPU*.

Кроме того, *Nsight* *Compute* интегрируется с другими инструментами *NVIDIA*, такими как *Nsight* *Systems* и *Nsight Visual Studio Edition*, что обеспечивает полный набор инструментов для разработки, профилирования и отладки приложений на *GPU*.

*NVIDIA* *Nsight* *Systems* – это инструмент профилирования и анализа производительности, предоставляемый компанией *NVIDIA*. Он предназначен для разработчиков, которые хотят получить полное представление о производительности своих приложений на графических процессорах (*GPU*) и центральных процессорах (*CPU*). [11]

*Nsight Systems* позволяет разработчикам собирать и анализировать данные производительности приложений в реальном времени. Инструмент предоставляет детальную информацию о времени выполнения, загрузке и использовании ресурсов, таких как процессорное время, память, дисковое пространство и другие.

Одной из ключевых особенностей *Nsight Systems* является возможность визуализации данных производительности в виде графической временной линии. Это позволяет разработчикам наглядно представить, как различные компоненты приложения взаимодействуют и какие узкие места могут замедлять его работу.

*Nsight* *Systems* также обеспечивает интеграцию с другими инструментами *NVIDIA*, такими как *Nsight Compute* и *Nsight Visual Studio Edition*, что позволяет разработчикам получить более полное представление о производительности приложений на GPU и взаимодействии между GPU и CPU.

Для измерения времени вычислений и запуска программ использовалась операционная система *Linux* (*Ubuntu* 22.04)*.* Это открытая операционная система, основанная на ядре *Linux*.

Ядро – это программа, расположенная в памяти компьютера и отдающая распоряжения центральному процессору. Ядро управляет аппаратными средствами и выступает главным образом в качестве интерфейса между аппаратными средствами и любой запущенной программой. [12]

*Ubuntu* 22.04 *LTS* – это один из вариантов операционной системы *Linux*, основанный на ядре *Linux*. *Ubuntu* является одним из самых популярных и простых в использовании дистрибутивов *Linux*. Он ориентирован на обеспечение простоты установки, использования и обновления, а также на поддержку широкого спектра аппаратного обеспечения.

Таким образом, платформа программного обеспечения (для разработки) базируется на операционной системе *Windows*, так как на ней доступно большое количество удобных инструментов (*Visual* *Studio*, *Nsight* *Compute* и другие), и на платформе *CUDA*, предназначенной для параллельных вычислений на *GPU*. Тестирование, в свою очередь, проводилось на операционной системе *Linux* по причине ее легковесности и отсутствия излишней нагрузки на центральный процессор.

# 3 ТЕОРЕТИЧЕСКОЕ ОБОСНОВАНИЕ РАЗРАБОТКИ ПРОГРАММНОГО ПРОДУКТА

## 3.1 Обоснование необходимости разработки

В современном информационном обществе вычислительные системы играют важную роль в обработке и анализе больших объемов данных. Одной из ключевых задач является эффективное выполнение операций над матрицами, которые широко применяются в различных областях, таких как компьютерное зрение, машинное обучение, научные и инженерные расчеты и другие.

Операции над матрицами требуют значительных вычислительных ресурсов, особенно при работе с большими матрицами. В связи с этим возникает потребность в использовании специализированных вычислительных устройств, способных обеспечить высокую производительность при выполнении таких операций. Одним из таких устройств является графический процессор (*GPU*), который обладает параллельными вычислительными возможностями.

Разработка системы для работы с матрицами на *GPU* с *CUDA* необходима из-за следующих преимуществ:

1. Увеличение производительности вычислений: благодаря большому количеству ядер и особенностям архитектуры, использование *GPU* позволяет значительно увеличить производительность для данной задачи.
2. Перспективы применения вычислений на *GPU*: разработка оптимального для графических вычислительных устройств параллельного алгоритма требует много ресурсов, однако полученный результат может сэкономить большое количество времени, что нивелирует время, затраченное на разработку и реализацию данного алгоритма.
3. Уменьшение времени выполнения умножения: использование *GPU* позволяет серьезно уменьшить время выполнения умножения, что приводит к уменьшению времени работы программы, работающей с матрицами.

Однако, особенности работы *GPU* требуют отличного от *CPU* подхода к разработке приложений, поэтому необходимо оптимально разработать алгоритм на *GPU* и оптимизировать работу с памятью, иначе *CPU* может показать лучшие результаты.

Таким образом, стоит выбор между *CPU* и *GPU*. Необходимо проанализировать их вычислительные возможности на различных наборах данных и выбрать наиболее эффективное устройство для выполнения матричных операций.

## 3.2 Технологии программирования, используемые для решения поставленных задач

Для написания программы, вычисляющей произведение матриц на *GPU*, использовались следующие технологии программирования:

1. *CUDA API*: расширение языка *С* (также можно использовать *C*++), позволяющее разработчикам не тратить много времени на изучение данной технологии.
2. Драйвер устройства: обеспечивает взаимодействие между хостом и графическим процессором.
3. Программный интерфейс (*API*): предоставляет доступ к функциям и ресурсам графического процессора.
4. Библиотека *Thrust*, позволяющая работать с *CUDA API* в парадигме *RAII*, которая является основной в языке *C++*.

Программа разработана на языке программирования *C*++. Язык *C*++ является высокоуровневым языком программирования, который был разработан на основе языка *C*. Он объединяет в себе возможности низкоуровневого программирования, характерные для C, дополнительные функции и абстракции, такие как объектно-ориентированное программирование (ООП) и обобщенное программирование (шаблоны). [13]

Основные особенности *C*++:

* 1. Объектно-ориентированное программирование (ООП): *C*++ позволяет создавать классы и объекты, инкапсулировать данные и методы, наследоваться от других классов, реализовывать полиморфизм и другие концепции ООП. Это позволяет разрабатывать модульные, гибкие и повторно используемые программы.
  2. Обобщенное программирование (шаблоны): *C*++ поддерживает шаблоны, которые позволяют создавать обобщенные алгоритмы и контейнеры данных. Шаблоны позволяют писать код, который может работать с различными типами данных, без необходимости создания дублирующегося кода.
  3. Эффективность: *C*++ обеспечивает близкое к машинному коду управление памятью и низкоуровневый доступ к аппаратным ресурсам компьютера. Это позволяет разработчикам оптимизировать производительность программы и управлять ресурсами напрямую.
  4. Библиотеки: *C*++ имеет обширную стандартную библиотеку, которая включает контейнеры данных, алгоритмы, ввод/вывод, обработку строк, работы с файлами и другие функциональные возможности. Кроме того, существует множество сторонних библиотек, расширяющих функциональность языка.
  5. Поддержка многопоточности: *C*++ предоставляет средства для создания многопоточных программ, включая потоки выполнения, синхронизацию и обмен данными между потоками.
  6. Переносимость: код, написанный на *C*++, может быть скомпилирован и запущен на различных платформах, включая *Windows*, *macOS*, *Linux*.

В качестве компилятора для *C*++ был выбран *g*++. Он является частью комплекта программ *GNU* *Compiler* *Collection* (*GCC*), который предоставляет набор компиляторов для различных языков программирования. Данный компилятор поддерживает множество стандартов и библиотек языка программирования *C*++. [14]

Компилятор g++ разрабатывается и поддерживается проектом *GNU*, и его исходный код является открытым и свободно распространяемым. Он доступен для различных операционных систем, включая *Linux*, macOS и *Windows*.

Основная функция *g*++ – это компиляция исходного кода на *C*++ в исполняемый файл, который может быть запущен на целевой платформе. Компилятор обрабатывает исходый код и выполняет следующие шаги:

* 1. Лексический анализ: исходный код разбивается на лексемы, такие как ключевые слова, операторы и идентификаторы.
  2. Синтаксический анализ: лексемы проверяются на соответствие синтаксису языка *C*++. Если обнаружена ошибка, компилятор выдаст сообщение об ошибке.
  3. Семантический анализ: компилятор проверяет семантические аспекты кода, такие как типы данных, правильность использования переменных и функций, наследование классов и другие аспекты языка *C*++.
  4. Генерация промежуточного кода: Компилятор создает промежуточный код, который представляет собой низкоуровневое представление программы.
  5. Оптимизация: промежуточный код проходит через этап оптимизации, где компилятор применяет различные техники для улучшения производительности и эффективности программы.
  6. Генерация машинного кода: наконец, компилятор преобразует промежуточный код в машинный код, специфичный для целевой аппаратной платформы.

Помимо компиляции, *g*++ также предоставляет различные опции и флаги для управления процессом компиляции, включая определение символов препроцессора, выбор оптимизаций, указание путей к заголовочным файлам и библиотекам, включение отладочной информации и другие.

Компилятор *g*++ является одним из наиболее распространенных компиляторов *C*++ и широко используется в сообществе разработчиков для создания *C*++ приложений и библиотек.

*NVCC* (*NVIDIA CUDA Compiler*) – это компилятор, разработанный компанией *Nvidia*, который используется для компиляции и сборки программ, написанных на языке программирования *CUDA*, для выполнения на графических процессорах *NVIDIA*.

Основные особенности и возможности *NVCC*:

1. Поддержка языка программирования *CUDA*: *NVCC* понимает расширения языка *C/C*++ для *CUDA*, которые позволяют разработчикам явно указывать и управлять выполнением кода на графическом процессоре. Эти расширения включают спецификаторы типов, квалификаторы памяти, директивы и встроенные функции, которые позволяют использовать возможности параллельных вычислений на графическом процессоре.
2. Компиляция смешанного кода: *NVCC* позволяет создавать смешанный код, который состоит из обычного (*host*) кода, выполняемого на центральном процессоре (*CPU*), и кода, выполняемого на графическом процессоре (*device* *code*). Это позволяет разработчикам использовать гибридные вычисления, где часть задач выполняется на *CPU*, а часть – на *GPU*.
3. Оптимизации для графических процессоров *NVIDIA*: *N­VCC* включает оптимизации, которые специально нацелены на графические процессоры *Nvidia*. Компилятор оптимизирует код для максимальной производительности на конкретной архитектуре *GPU*.
4. Интеграция с *CUDA Runtime API*: *NVCC* взаимодействует с *CUDA* *Runtime API*, предоставляя функции для управления выполнением программы на графическом процессоре. Разработчики могут использовать функции для управления памятью, запуска ядер выполнения на *GPU*, синхронизации выполнения и других операций, связанных с выполнением программы на графическом процессоре. [15]

*NVCC* является важным инструментом для разработки программ, использующих возможности параллельных вычислений на графических процессорах *Nvidia*. Он обеспечивает удобную интеграцию с языками программирования *C/C*++ и *CUDA*, позволяет разработчикам использовать гибридные вычисления и оптимизировать код для максимальной производительности на графических процессорах *Nvidia*.

Одним из наиболее популярных средств программирования для компьютеров с общей памятью, базирующихся на традиционных языках программирования и использовании специальных комментариев, в настоящее время является технология *OpenMP*. За основу берётся последовательная программа, а для создания её параллельной версии пользователю предоставляется набор директив, функций и переменных окружения. [16]

Одной из ключевых особенностей *OpenMP* является простота использования. Благодаря директивам компилятора, включенным в библиотеку, разработчики могут легко указать участки кода, которые должны быть выполнены параллельно. Например, с использованием директивы *pragma omp parallel*, разработчик может создать параллельный регион, в котором несколько потоков будут выполнять инструкции независимо друг от друга. Это позволяет распараллелить выполнение циклов, операций на матрицах, обработку массивов данных и другие участки кода, которые могут быть разделены на независимые задачи.

Библиотека *OpenMP* также обладает возможностями для управления потоками и синхронизации. Разработчики могут получить информацию о текущем потоке с помощью функции *omp\_get\_thread\_num*, а также узнать общее количество потоков с помощью функции *omp\_get\_num\_threads*. Это позволяет создавать логику программы, которая учитывает текущий поток и делает выборки задач в соответствии с доступными потоками. Библиотека также предоставляет средства синхронизации, такие как барьеры (*pragma* *omp* *barrier*), критические секции (*pragma omp critical*), атомарные операции и другие, чтобы обеспечить правильное выполнение параллельного кода.

Другим важным аспектом *OpenMP* является поддержка различных архитектур и операционных систем. Библиотека доступна для множества платформ, включая *Linux*, *macOS* и *Windows*, что позволяет разработчикам создавать параллельные программы, независимо от выбранной операционной системы. Более того, *OpenMP* может использоваться с различными языками программирования, включая *C*, *C*++ и *Fortran*, что делает ее универсальным инструментом для разработки параллельных программ на разных языках.

В целом, библиотека *OpenMP* представляет собой мощный и удобный инструмент для разработки параллельных программ.

Для измерения времени выполнения программы использовалась библиотека *chrono*. Библиотека *chrono* – это библиотека *C*++, введенная в стандарте *C*++11, которая предоставляет возможности работы со временем и измерения временных интервалов с высокой точностью. Она предоставляет типы, функции и классы для измерения времени выполнения операций с временными точками, длительностями и интервалами.

Одним из классов, предоставляемых библиотекой *chrono*, является *high\_resolution\_clock*. Этот класс представляет собой часы высокого разрешения, которые предоставляют максимально доступное разрешение для измерения времени на данной системе. [17]

Класс *high\_resolution\_clock* обеспечивает доступ к текущему времени в форме временной точки (*time*\_*point*) и позволяет измерять временные интервалы с высокой точностью с использованием длительности (*duration*). Он может быть использован для выполнения точных измерений времени и профилирования кода.

Для чтения и записи тестовых данных в файл использовалась библиотека *fstream.* Она предоставляет удобные классы для работы с файлами. Она позволяет открывать файлы для чтения (*ifstream*), записи (*ofstream*) или как комбинированный режим чтения и записи (*fstream*). Библиотека предоставляет набор флагов, которые можно указывать для определения режима работы с файлом. Флаги открытия файла в библиотеке fstream позволяют указать дополнительные параметры при открытии файла. Некоторые из наиболее часто используемых флагов включают:

1. Флаг ios::binary (бинарный режим) указывает, что файл должен быть открыт в бинарном режиме. В бинарном режиме данные записываются и читаются в их исходном двоичном формате, без преобразования.
2. Флаг ios::out (режим записи) указывает, что файл должен быть открыт для записи. Если файл уже существует, его содержимое будет удалено и заменено новыми данными. Если файл не существует, он будет создан.
3. Флаг ios::trunc (очистка файла) указывает, что файл должен быть очищен перед открытием. Если файл уже существует, его содержимое будет удалено при открытии. Этот флаг обычно используется вместе с ios::out для создания нового файла или перезаписи существующего.

Также для работы с файловой системой в дополнение к библиотеке *fstream* использовалась библиотека *filesystem*. Она предоставляет простой и удобный функционал для анализа содержимого директории, получение абсолютного пути к файлу, получения расширения файла и так далее.

## 3.3 Связь архитектуры вычислительной системы с разрабатываемым программным обеспечением

Связь архитектуры вычислительной системы с разрабатываемым ПО заключается в том, что архитектура определяет возможности и ограничения системы, а также влияет на выбор и разработку программного обеспечения. Архитектура определяет, какие типы вычислительных операций могут быть выполнены на данной системе, а также какие ресурсы и алгоритмы могут быть использованы для решения конкретных задач.

Например, если система имеет многоядерный процессор, то можно использовать параллельные алгоритмы, которые будут распределять вычисления между ядрами, что приведет к увеличению скорости решения задачи. А при разработке программного обеспечения для мобильных устройств, таких как смартфоны или планшеты, важно учитывать ограниченные ресурсы процессора, памяти и энергии. В таких случаях разработчики могут использовать оптимизированные языки программирования или фреймворки, которые позволяют создавать эффективное и энергосберегающее ПО.

В нашем случае мы будем иметь особенности работы *CPU* и *GPU* с памятью, а также то, что *CPU* – более универсальное вычислительное устройство, которое может иметь возможность исполнения более сложных инструкций, в то время как *GPU* имеет большое количество простых исполнительных устройств.

Сравнение архитектур *GPU* и *CPU* в контексте выполнения операций над матрицами позволит оценить преимущества и ограничения каждого устройства. *GPU* обычно показывает высокую эффективность при параллельных вычислениях, в то время как *CPU* может быть более эффективным при выполнении последовательных операций.

Таким образом, в данном разделе была обоснована необходимость разработки приложения для выполнения умножения матриц (в особенности с использованием графического процессора), так как в современном мире матрицам находят все больше и больше применения. Были описаны средства и технологии, использованные в данном курсовом проекте (*OpenMP*, *CUDA, C++* и другие). Была выявлена связь архитектуры вычислительной системы с разрабатываемым ПО, учтены особенности строения *GPU* и *CPU*.

# 4 ПРОЕКТИРОВАНИЕ ФУНКЦИОНАЛЬНЫХ ВОЗМОЖНОСТЕЙ ПРОГРАММЫ

Процесс проектирования программного обеспечения для сравнения эффективности выполнения матричных операций на центральном процессоре *Intel Core i5*-11400*H* и графическом процессоре *NVIDIA GeForce RTX* 3060 *laptop* требует всестороннего подхода. Ключевое внимание уделяется разработке функций, выполняющих вычисления. Этот раздел описывает проектирование основных функциональных возможностей программы.

## 4.1 Варианты запуска

Как упоминалось в разделе 1, рассматриваемый центральный процессор оснащен технологией *Hyper-Threading* (*HTT*).

Для более детального исследования и сравнительного анализа производительности необходимо осуществить запуск программного обеспечения с включенной и выключенной функцией *Hyper-Threading*, чтобы проанализировать насколько данная технология влияет на производительность и конкурентоспособность с *GPU.*

Так как у *GPU* во время выполнения алгоритма есть несколько вариантов работы с памятью (использование глобальной или разделяемой памяти), то необходимо проанализировать, какой вариант покажет лучший результат.

Таким образом, имеется несколько вариантов запуска программ: на одном ядре на *CPU* с и без *HTT*, на всех ядрах на *CPU* с и без *HTT*, на одном и на всех *CUDA*-ядрах на *GPU* используя только глобальную память и использую глобальную вместе с разделяемой памятью.

## 4.2 Тестовые данные

В качестве тестовых данных выступают файлы, содержащие бинарные данные, которые программа считывает и заполняет ими матрицу. Для полноты исследования использовались входные данные разных объемов (разный размер матриц): от 1000 до 9500 элементов в строке квадратной матрицы. В данном курсовом проекте тип данных элементов матрицы ­­­­­­­– числа с плавающей точкой, занимающие в памяти 4 байта.

## 4.3 Алгоритм умножения матриц

Алгоритм умножения матриц – это вычислительно интенсивная операция, которая может сильно нагружать процессор. Умножение матриц требует выполнения множества умножений и сложений между элементами матриц, что приводит к большому количеству операций, которые должны быть выполнены последовательно или параллельно. Это делает алгоритм умножения матриц одним из наиболее ресурсоемких заданий в области вычислительных операций.

Для обеспечения качественного анализа производительности необходимо разработать эффективный алгоритм. В данной курсовой работе использовался итеративный алгоритм умножения матриц. По определению умножения матриц для матриц *A* и *B* размера *N* *N* произведением является матрица *N* *N*, состоящая из элементов . Отсюда можно построить простой алгоритм путём организации вложенных циклов по индексу *i* от 1 до N и *j* от 1 до *N*, и осуществляя вычисления по вышеприведённой формуле. [18, 19]

Однако самая простая реализация данного алгоритма не является оптимальной. Три цикла при итеративном умножении матриц можно произвольным образом менять друг с другом местами без влияния на правильность алгоритма или асимптотическое время работы. Порядок циклов может влиять на практические характеристики доступа памяти и на использование кэша. Для анализа использования кэша во время выполнения программы использовался инструмент *cachegrind*. [20]

В целях выявления наиболее эффективного расположения циклов было произведено несколько тестовых запусков программы с различным порядков циклов.

При умножении матриц 1000 1000 и размещении циклов в порядке *i*, *j*, *k* частота кэш-промахов при обращении к кэшу данных первого уровня *D*1 составила 35,3% (рисунок 4.1).

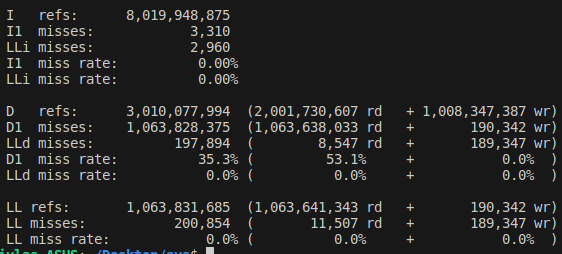


Рисунок 4.1 ­– Результаты обращения к кэшу при порядке циклов *i*, *j*, *k*

При умножении матриц 1000 1000 и размещении циклов в порядке *i*, *k*, *j* частота кэш-промахов при обращении к кэшу данных первого уровня *D*1 составила 8,2% (рисунок 4.2).

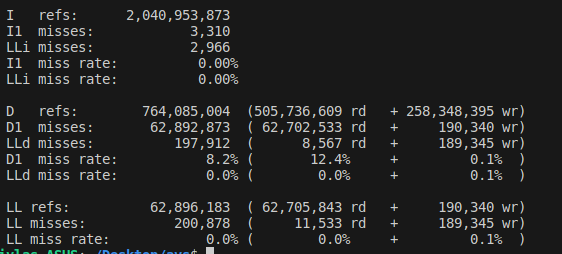


Рисунок 4.2 – Результаты обращения к кэшу при порядке циклов *i*, *k*, *j*

Таким образом, в разработанном алгоритме циклы по *j* и *k* были переставлены местами, что привело к четырехкратному уменьшению количества кэш-промахов и большей производительности.

## 4.4 Оценка производительности

В качестве критерия оценки производительности процессора в данном курсовом проекте использовалось время выполнения алгоритма.

Принцип измерения времени заключается в фиксации начального и конечно моментов времени и последующего вычисления времени выполнения алгоритма. В случае параллельной реализации алгоритма для корректного измерения необходимо дождаться окончания работы параллельно запущенной задачи.

## 4.5 Общая структура программы

Все разработанные программы имеют одинаковую общую структуру, но также есть небольшие различия в реализации: разные используемые библиотеки в зависимости от типа алгоритма (параллельный или нет) и процессора, на котором код выполняется (*CPU*, *GPU*).

Начало работы программы начинается с передачи ей аргументов командной строки: путь к файлу, в который будет записан результат измерений времени, и путь к директории, в которой расположены файлы с тестовыми данными. Программа считывает из директории пути ко всем тестовым файлам и записывает их в массив.

Далее обрабатывается каждый файл из массива: производятся вычисления и измерение времени для конкретного набора входных данных. Полученное значение времени записывается в файл, выполнение программы продолжается до тех пор, пока не будут обработаны все файлы в директории.

## 4.6 Описание функциональной схемы алгоритма программы

На функциональной схеме (приложение Б) представлено описание основных компонентов и их взаимодействия в разрабатываемой программе, предназначенной для сравнительного анализа производительности центрального процессора *Intel* *Core* *i*5-11400*H* и графического процессора Nvidia *GeForce* *RTX* 3060 *laptop* при вычислении произведения матриц.

**Генерация тестовых данных**: одна из написанных программ принимает на вход размер матрицы, тип элементов матрицы и путь к файлу, в который нужно сохранить сгенерированную матрицу. Для ускорения генерации больших объемов данных использовался графический процессор.

**Сохранение тестовых данных в файл**: сгенерированная графическим процессором матрица копируется в оперативную память и затем сохраняется в файл для дальнейшего использования.

**Чтение тестовых данных из каждого файла в директории**: программы, выполняющие умножение, в качестве одного из параметров принимают на вход директорию, в которой расположены файлы с данными. Затем начинает выполнение цикл: из каждого файла по очереди считываются данные и записываются в матрицу. Происходит вычисление и измерение времени. Цикл выполняется пока все файлы (все наборы тестовых данных) не будут обработаны.

**Начало отсчета времени выполнения алгоритма**: перед началом вычислений программа запоминает текущий момент времени.

**Выполнение умножения матриц**: две матрицы, содержащие считанные из файла данные, перемножаются и результат сохраняется в третьей матрице.

**Конец отсчета времени выполнения алгоритма**: после выполнения вычислений программа запоминает текущий момент времени и находит разницу между ним и ранее сохраненным моментом времени перед началом вычислений, получая время выполнения алгоритма.

**Запись времени выполнения в файл**: для каждого набора тестовых данных программа записывает время выполнения в файл.

## 4.7 Описание блок-схемы алгоритма программы

Алгоритм программы описан на блок-схеме (приложение В).

Для умножения матриц был использован итеративный метод.

Как видно из блок-схемы, программа начинает свое выполнение с получение от пользователя следующих данных:

1. Путь к файлу (*outFileName*), в который будут записываться результаты измерения времени для каждого набора тестовых данных;
2. Путь к директории (*dataDir*), в которой располагаются файлы, содержащие тестовые данные.

Затем программа просматривает директорию *dataDir* и создает массив *fileNames*, содержащий пути к файлам из директории.

Далее начинает выполнение цикл: пока не будут обработаны все файлы из массива *fileNames*, программа считывает размер матрицы и ее элементы из текущего файла, заполняет этими данными матрицу *A*, копирует в матрицу *B* и создает матрицу *C* такого же размера, заполненную нулями (необходимо для корректной работы алгоритма умножения).

После формирования матриц программа запоминает текущий момент времени *start* и начинает выполнение умножения, используя итеративный алгоритм. После завершения вычислений, программа запоминает текущий момент времени *end*, находит разность *end* и *start* и получает время выполнения алгоритма.

Полученное время записывается в *outFileName* для дальнейшего анализа.

Реализации итеративного алгоритма могут отличаться в зависимости от используемого аппаратного обеспечения (*CPU* или *GPU*), программного обеспечения (используемый язык программирования, компилятор и так далее). В данном курсовом проекте было реализовано несколько вариантов итеративного алгоритма умножения матриц:

1. Параллельный алгоритм на *CPU*;
2. Последовательный алгоритм на *CPU*;
3. Параллельный алгоритм на *GPU* с использованием глобальной памяти;
4. Параллельный алгоритм на *GPU* с использованием глобальной и разделяемой памяти;
5. Последовательный алгоритм на *GPU*.

Таким образом, в данной главе были описаны функциональные возможности программы: принцип работы с тестовыми данными, варианты алгоритма умножения матриц, общая структура программы, способ сравнения производительности *GPU* и *CPU* при выполнении последовательного и параллельного умножения матриц. Особое внимание уделялось описанию тонкостей реализации алгоритма умножения (оптимизация работы с кэшем на *CPU*, с памятью на *GPU*), описанию функциональной схемы алгоритма и блок-схемы алгоритма.

# 5 Архитектура разрабатываемой программы

## 5.1 Запуск программ

Все запуски на CPU производились на операционной системе *Linux* с использованием *Live* *USB*. Запуски на GPU производились на установленной ОС *Linux*. В качестве размера входных данных были выбраны следующие значения: 1000, 1500, 2000, 2500, 3000, 3500, 4000, 5500, 6000, 6500, 7000, 7500, 8000, 8500, 9000 и 9500 элементов в одной строке квадратной матрицы. Также для каждого значения размера матрицы производилось по 5 запусков, чтобы минимизировать влияние случайных отклонений во время вычислений.

## 5.2 Запуск на CPU на одном ядре с использованием HTT и без HTT

После нескольких запусков были получение следующие результаты умножения на *CPU* с использованием *HTT* (таблицу 5.1). Программа запускалась с помощью команды *taskset*, чтобы указать номер ядра, на котором выполнять код.

Таблица 5.1 – Результаты запуска на CPU на одном ядре с HTT

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Номер запуска | Время выполнения при заданном размере матрицы, с | | | | | | | | |
| 1000 | 1500 | 2000 | 2500 | 3000 | 3500 | 4000 | 4500 | 5000 |
| 1 | 0,11 | 0,52 | 1,68 | 3,36 | 5,54 | 8,98 | 13,40 | 19,30 | 26,72 |
| 2 | 0,11 | 0,53 | 1,68 | 3,37 | 5,42 | 8,85 | 13,40 | 19,19 | 26,52 |
| 3 | 0,11 | 0,54 | 1,67 | 3,36 | 5,44 | 8,87 | 13,40 | 19,31 | 26,68 |
| 4 | 0,11 | 0,55 | 1,67 | 3,35 | 5,46 | 8,86 | 13,39 | 19,31 | 26,72 |
| 5 | 0,11 | 0,55 | 1,66 | 3,37 | 5,74 | 8,85 | 13,39 | 19,32 | 26,71 |
| Среднее | 0,11 | 0,54 | 1,67 | 3,36 | 5,52 | 8,88 | 13,40 | 19,23 | 26,67 |

Продолжение таблицы 5.1

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Номер запуска | Время выполнения при заданном размере матрицы, с | | | | | | | | |
| 5500 | 6000 | 6500 | 7000 | 7500 | 8000 | 8500 | 9000 | 9500 |
| 1 | 35,73 | 46,62 | 59,89 | 75,31 | 93,29 | 113,07 | 136,65 | 162,64 | 191,72 |
| 2 | 35,51 | 46,28 | 59,52 | 75,00 | 92,69 | 112,99 | 135,55 | 161,24 | 190,68 |
| 3 | 35,76 | 46,60 | 59,80 | 75,35 | 93,14 | 113,20 | 136,00 | 162,16 | 191,01 |
| 4 | 35,75 | 46,65 | 59,93 | 75,47 | 93,40 | 113,71 | 136,57 | 162,10 | 191,56 |
| 5 | 35,73 | 46,62 | 59,90 | 75,31 | 93,29 | 113,07 | 136,65 | 162,64 | 191,72 |
| Среднее | 35,70 | 46,55 | 59,80 | 75,30 | 93,16 | 113,20 | 136,28 | 162,16 | 191,34 |

После нескольких запусков были получение следующие результаты умножения на *CPU* без использования *HTT* (таблица 5.2). Программа запускалась с помощью команды *taskset*, чтобы указать номер ядра, на котором выполнять код.

Таблица 5.2 – Результаты запуска на CPU на одном ядре без HTT

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Номер запуска | Время выполнения при заданном размере матрицы, с | | | | | | | | |
| 1000 | 1500 | 2000 | 2500 | 3000 | 3500 | 4000 | 4500 | 5000 |
| 1 | 0,11 | 0,54 | 1,65 | 3,34 | 5,59 | 8,84 | 13,42 | 19,19 | 26,70 |
| 2 | 0,11 | 0,53 | 1,66 | 3,37 | 5,48 | 8,79 | 13,40 | 19,29 | 26,52 |
| 3 | 0,11 | 0,53 | 1,68 | 3,36 | 5,45 | 8,85 | 13,42 | 19,30 | 26,61 |
| 4 | 0,11 | 0,54 | 1,68 | 3,33 | 5,47 | 8,76 | 13,44 | 19,22 | 26,72 |
| 5 | 0,11 | 0,55 | 1,66 | 3,37 | 5,74 | 8,85 | 13,39 | 19,32 | 26,75 |
| Среднее | 0,11 | 0,54 | 1,66 | 3,35 | 5,55 | 8,82 | 13,41 | 19,26 | 26,66 |

Продолжение таблицы 5.2

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Номер запуска | Время выполнения при заданном размере матрицы, с | | | | | | | | |
| 5500 | 6000 | 6500 | 7000 | 7500 | 8000 | 8500 | 9000 | 9500 |
| 1 | 35,60 | 46,50 | 59,82 | 75,51 | 93,30 | 113,71 | 137,08 | 162,93 | 191,75 |
| 2 | 35,41 | 46,30 | 59,54 | 75,02 | 92,69 | 112,94 | 135,55 | 161,24 | 190,68 |
| 3 | 35,75 | 46,61 | 59,83 | 75,35 | 93,24 | 113,23 | 136,11 | 162,13 | 191,21 |
| 4 | 35,76 | 46,65 | 59,91 | 75,48 | 93,42 | 113,72 | 136,55 | 162,20 | 191,46 |
| 5 | 35,73 | 46,61 | 59,90 | 75,35 | 93,29 | 113,07 | 136,65 | 162,84 | 191,57 |
| Среднее | 35,65 | 46,53 | 59,80 | 75,34 | 93,19 | 113,33 | 136,39 | 162,27 | 191,33 |

## 5.3 Запуск на CPU на всех ядрах с использованием HTT и без HTT

После нескольких запусков были получение следующие результаты умножения на *CPU* с использованием *HTT* (таблица 5.3). Программа запускалась с помощью команды *taskset*, чтобы указать номера ядер, на которых выполнять код.

Таблица 5.3 – Результаты запуска на CPU на всех ядрах с HTT

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Номер запуска | Время выполнения при заданном размере матрицы, с | | | | | | | | |
| 1000 | 1500 | 2000 | 2500 | 3000 | 3500 | 4000 | 4500 | 5000 |
| 1 | 0,02 | 0,10 | 0,28 | 0,56 | 0.98 | 1,62 | 2,37 | 3,32 | 4,73 |
| 2 | 0,02 | 0,09 | 0,27 | 0,55 | 0,95 | 1,52 | 2,27 | 3,26 | 4,56 |
| 3 | 0,02 | 0,09 | 0,27 | 0,55 | 0,96 | 1,52 | 2,28 | 3,44 | 4,50 |
| 4 | 0,02 | 0,09 | 0,27 | 0,56 | 0,99 | 1,53 | 2,29 | 3,30 | 4,52 |
| 5 | 0,02 | 0,09 | 0,27 | 0,56 | 0,95 | 1,53 | 2,30 | 3,41 | 4,57 |
| Среднее | 0,02 | 0,09 | 0,27 | 0,56 | 0,96 | 1,54 | 2,30 | 3,35 | 4,58 |

Продолжение таблицы 5.3

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Номер запуска | Время выполнения при заданном размере матрицы, с | | | | | | | | |
| 5500 | 6000 | 6500 | 7000 | 7500 | 8000 | 8500 | 9000 | 9500 |
| 1 | 6,07 | 7,87 | 10,42 | 12,46 | 17,21 | 19,18 | 22,86 | 28,86 | 33,66 |
| 2 | 6,27 | 8,20 | 9,99 | 13,10 | 15,19 | 21,05 | 23,36 | 27,25 | 34,37 |
| 3 | 5,99 | 7,86 | 9,87 | 12,80 | 15,10 | 19,68 | 21,89 | 26,35 | 37,12 |
| 4 | 6,03 | 8,30 | 9,93 | 12,39 | 15,80 | 19,09 | 22,07 | 28,64 | 36,41 |
| 5 | 6,04 | 7,83 | 10,61 | 12,53 | 16,38 | 19,10 | 23,01 | 27,87 | 34,12 |
| Среднее | 6,08 | 8,01 | 10,16 | 12,67 | 15,94 | 19,62 | 22,64 | 27,79 | 35,14 |

После нескольких запусков были получение следующие результаты умножения на *CPU* без использования *HTT* (таблица 5.4). Программа запускалась с помощью команды *taskset*, чтобы указать номера ядер, на которых выполнять код.

Таблица 5.4 – Результаты запуска на CPU на всех ядрах без HTT

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Номер запуска | Время выполнения при заданном размере матрицы, с | | | | | | | | |
| 1000 | 1500 | 2000 | 2500 | 3000 | 3500 | 4000 | 4500 | 5000 |
| 1 | 0,02 | 0,13 | 0,43 | 0,90 | 1,58 | 2,50 | 3,86 | 5,33 | 7,30 |
| 2 | 0,02 | 0,14 | 0,44 | 0,90 | 1,57 | 2,49 | 4,20 | 5,30 | 7,29 |
| 3 | 0,02 | 0,13 | 0,45 | 0,90 | 1,57 | 2,67 | 3,72 | 5,30 | 7,28 |
| 4 | 0,02 | 0,13 | 0,44 | 0,90 | 1,57 | 2,65 | 4,03 | 5,36 | 7,29 |
| 5 | 0,02 | 0,13 | 0,44 | 0,90 | 1,57 | 2,50 | 3,72 | 5,33 | 7,45 |
| Среднее | 0,02 | 0,13 | 0,44 | 0,90 | 1,57 | 2,56 | 3,91 | 5,32 | 7,32 |

Продолжение таблицы 5.4

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Номер запуска | Время выполнения при заданном размере матрицы, с | | | | | | | | |
| 5500 | 6000 | 6500 | 7000 | 7500 | 8000 | 8500 | 9000 | 9500 |
| 1 | 9,73 | 13,22 | 16,10 | 20,73 | 26,43 | 30,30 | 41,34 | 46,56 | 54,58 |
| 2 | 9,71 | 13,53 | 16,06 | 21,02 | 27,37 | 32,26 | 38,04 | 48,34 | 54,64 |
| 3 | 9,69 | 12,80 | 16,05 | 21,31 | 24,93 | 31,91 | 36,28 | 45,00 | 56,30 |
| 4 | 9,71 | 12,87 | 16,09 | 21,02 | 26,62 | 32,15 | 37,90 | 51,95 | 58,70 |
| 5 | 9,70 | 13,54 | 16,09 | 21,84 | 27,73 | 30,77 | 38,36 | 48,62 | 56,58 |
| Среднее | 9,71 | 13,19 | 16,08 | 21,18 | 26,62 | 31,49 | 38,38 | 48,09 | 56,16 |

## 5.4 Запуск на GPU на одном CUDA ядре

После нескольких запусков были получение следующие результаты умножения на *GPU* на одном *CUDA* ядре (таблица 5.5). Так как время выполнения кода на *GPU* на одному *CUDA* ядре гораздо больше, чем на *CPU* на одном ядре (таблица 5.1), то было принято решение сократить набор входных данных до 1000, 1500 и 2000 элементов в строке матрицы и провести три запуска вместо пяти.

Таблица 5.5 – Результаты запуска на GPU на одном CUDA ядре

|  |  |  |  |
| --- | --- | --- | --- |
| Номер запуска | Время выполнения при заданном размере матрицы, с | | |
| 1000 | 1500 | 2000 |
| 1 | 82,90 | 318,19 | 744,93 |
| 2 | 82,12 | 314,40 | 742,32 |
| 3 | 83,21 | 317,89 | 745,16 |
| Среднее | 82,74 | 316,83 | 744,14 |

## 5.5 Запуск на GPU на всех CUDA ядрах

После нескольких запусков были получение следующие результаты умножения на *GPU* на всех *CUDA*-ядрах без использования разделяемой памяти (таблица 5.6).

Таблица 5.6 – Результаты запуска на GPU на всех CUDA-ядрах без использования разделяемой памяти

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Номер запуска | Время выполнения при заданном размере матрицы, с | | | | | | | | |
| 1000 | 1500 | 2000 | 2500 | 3000 | 3500 | 4000 | 4500 | 5000 |
| 1 | 0,01 | 0,03 | 0,06 | 0,16 | 0,20 | 0,37 | 0,47 | 0,81 | 0,94 |
| 2 | 0,01 | 0,03 | 0,06 | 0,16 | 0,20 | 0,38 | 0,47 | 0,82 | 0,95 |
| 3 | 0,01 | 0,03 | 0,06 | 0,16 | 0,20 | 0,39 | 0,48 | 0,83 | 0,96 |
| 4 | 0,01 | 0,03 | 0,06 | 0,16 | 0,20 | 0,39 | 0,48 | 0,83 | 0,96 |
| 5 | 0,01 | 0,03 | 0,06 | 0,16 | 0,20 | 0,39 | 0,48 | 0,83 | 0,96 |
| Среднее | 0,01 | 0,03 | 0,06 | 0,16 | 0,20 | 0,38 | 0,48 | 0,82 | 0,95 |

Продолжение таблицы 5.6

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Номер запуска | Время выполнения при заданном размере матрицы, с | | | | | | | | |
| 5500 | 6000 | 6500 | 7000 | 7500 | 8000 | 8500 | 9000 | 9500 |
| 1 | 1,50 | 1,59 | 2,49 | 2,59 | 3,86 | 3,83 | 5,67 | 5,60 | 8,06 |
| 2 | 1,51 | 1,60 | 2,53 | 2,62 | 3,93 | 3,87 | 5,76 | 5,65 | 8,13 |
| 3 | 1,53 | 1,62 | 2,56 | 2,64 | 3,96 | 3,90 | 5,80 | 5,69 | 8,19 |
| 4 | 1,53 | 1,62 | 2,56 | 2,65 | 3,97 | 3,91 | 5,82 | 5,70 | 8,19 |
| 5 | 1,54 | 1,62 | 2,56 | 2,65 | 3,97 | 3,91 | 5,82 | 5,70 | 8,19 |
| Среднее | 1,52 | 1,61 | 2,54 | 2,63 | 3,94 | 3,88 | 5,77 | 5,67 | 8,15 |

После нескольких запусков были получение следующие результаты умножения на *GPU* на всех *CUDA*-ядрах с использованием разделяемой памяти (таблица 5.7).

Таблица 5.7 – Результаты запуска на GPU на всех CUDA-ядрах c использованием разделяемой памяти

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Номер запуска | Время выполнения при заданном размере матрицы, с | | | | | | | | |
| 1000 | 1500 | 2000 | 2500 | 3000 | 3500 | 4000 | 4500 | 5000 |
| 1 | 0,002 | 0,008 | 0,018 | 0,038 | 0,064 | 0,097 | 0,128 | 0,188 | 0,259 |
| 2 | 0,002 | 0,008 | 0,018 | 0,038 | 0,063 | 0,097 | 0,128 | 0,190 | 0,263 |
| 3 | 0,002 | 0,008 | 0,018 | 0,038 | 0,063 | 0,097 | 0,129 | 0,190 | 0,262 |
| 4 | 0,002 | 0,008 | 0,018 | 0,035 | 0,061 | 0,087 | 0,130 | 0,193 | 0,264 |
| 5 | 0,002 | 0,008 | 0,018 | 0,036 | 0,061 | 0,096 | 0,130 | 0,195 | 0,268 |
| Среднее | 0,002 | 0,008 | 0,018 | 0,037 | 0,062 | 0,095 | 0,129 | 0,191 | 0,263 |

Продолжение таблицы 5.7

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Номер запуска | Время выполнения при заданном размере матрицы, с | | | | | | | | |
| 5500 | 6000 | 6500 | 7000 | 7500 | 8000 | 8500 | 9000 | 9500 |
| 1 | 0,350 | 0,450 | 0,591 | 0,736 | 0,946 | 1,122 | 1,376 | 1,622 | 1,932 |
| 2 | 0,352 | 0,457 | 0,596 | 0,739 | 0,958 | 1,135 | 1,396 | 1,637 | 1,952 |
| 3 | 0,357 | 0,459 | 0,595 | 0,739 | 0,959 | 1,138 | 1,394 | 1,646 | 1,966 |
| 4 | 0,359 | 0,464 | 0,605 | 0,751 | 0,973 | 1,151 | 1,410 | 1,664 | 1,967 |
| 5 | 0,362 | 0,464 | 0,608 | 0,760 | 0,993 | 1,170 | 1,426 | 1,675 | 2,004 |
| Среднее | 0,356 | 0,459 | 0,599 | 0,745 | 0,966 | 1,143 | 1,400 | 1,649 | 1,964 |

## 

## 5.6 Анализ полученных результатов

### 5.6.1 Анализ результатов запуска на одном ядре CPU с использованием HTT и без HTT

Результаты приведены в таблице 5.1 и таблице 5.2.

Как видно из графика (рисунок 5.1), время выполнения с использованием HTT практически полностью совпадает с временем выполнения без использования HTT.

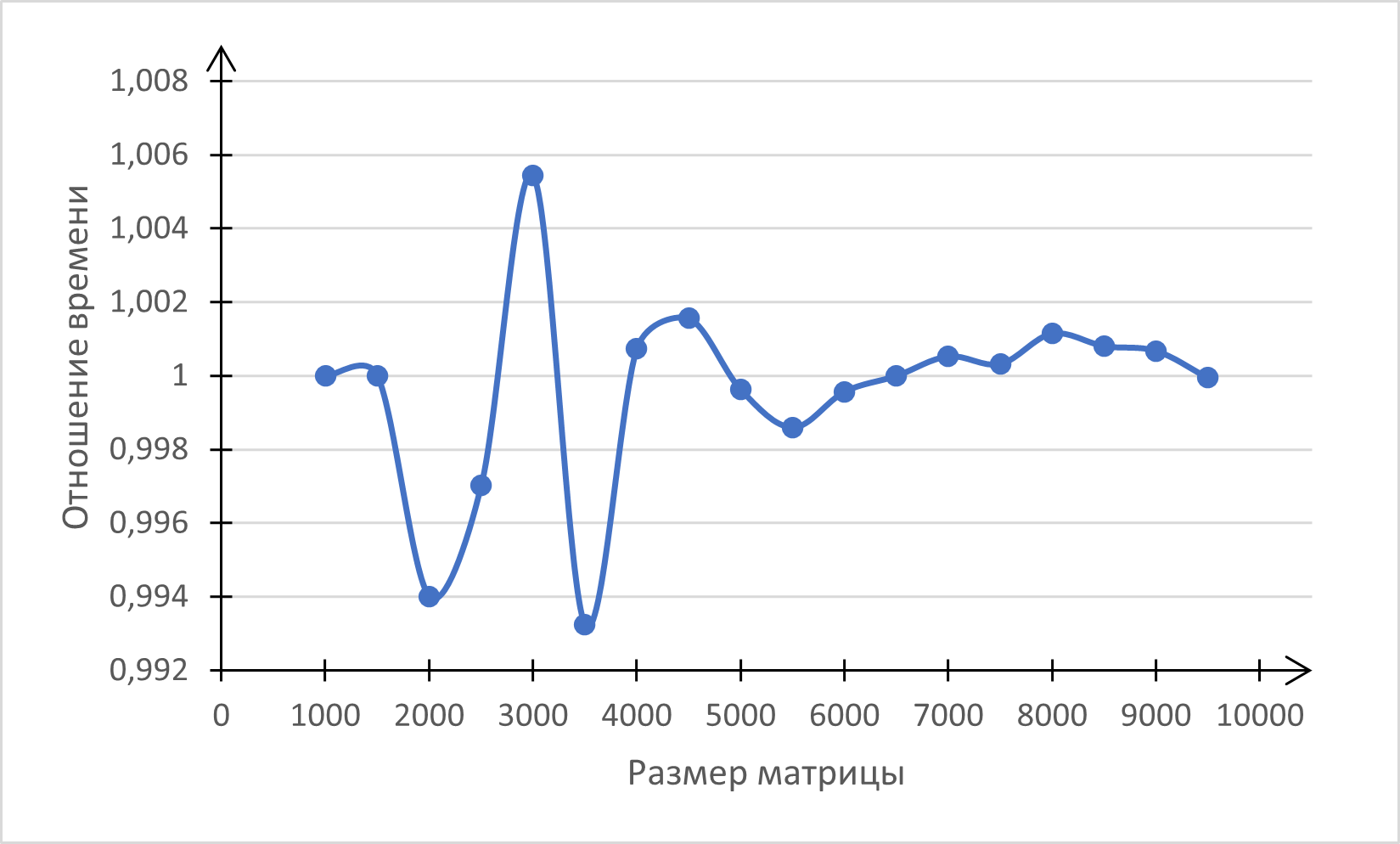


Рисунок 5.1 – График отношения времени выполнения умножения на одном ядре CPU без HTT ко времени выполнения на одном ядре с HTT

Это объясняется тем, что процессору на вход поступает всего один поток инструкций (на одном ядре запущена всего одна программа), а значит нет необходимости во втором логическом ядре и все необходимые вычислительные устройства находятся в распоряжении одного потока.

### 5.6.2 Анализ результатов запуска на всех ядрах CPU с использованием HTT и без HTT

Результаты приведены в таблице 5.3 и 5.4.

Как видно из графика (рисунок 5.2), производительность процессора без использования HTT заметно уступает производительности процессора с использованием HTT.

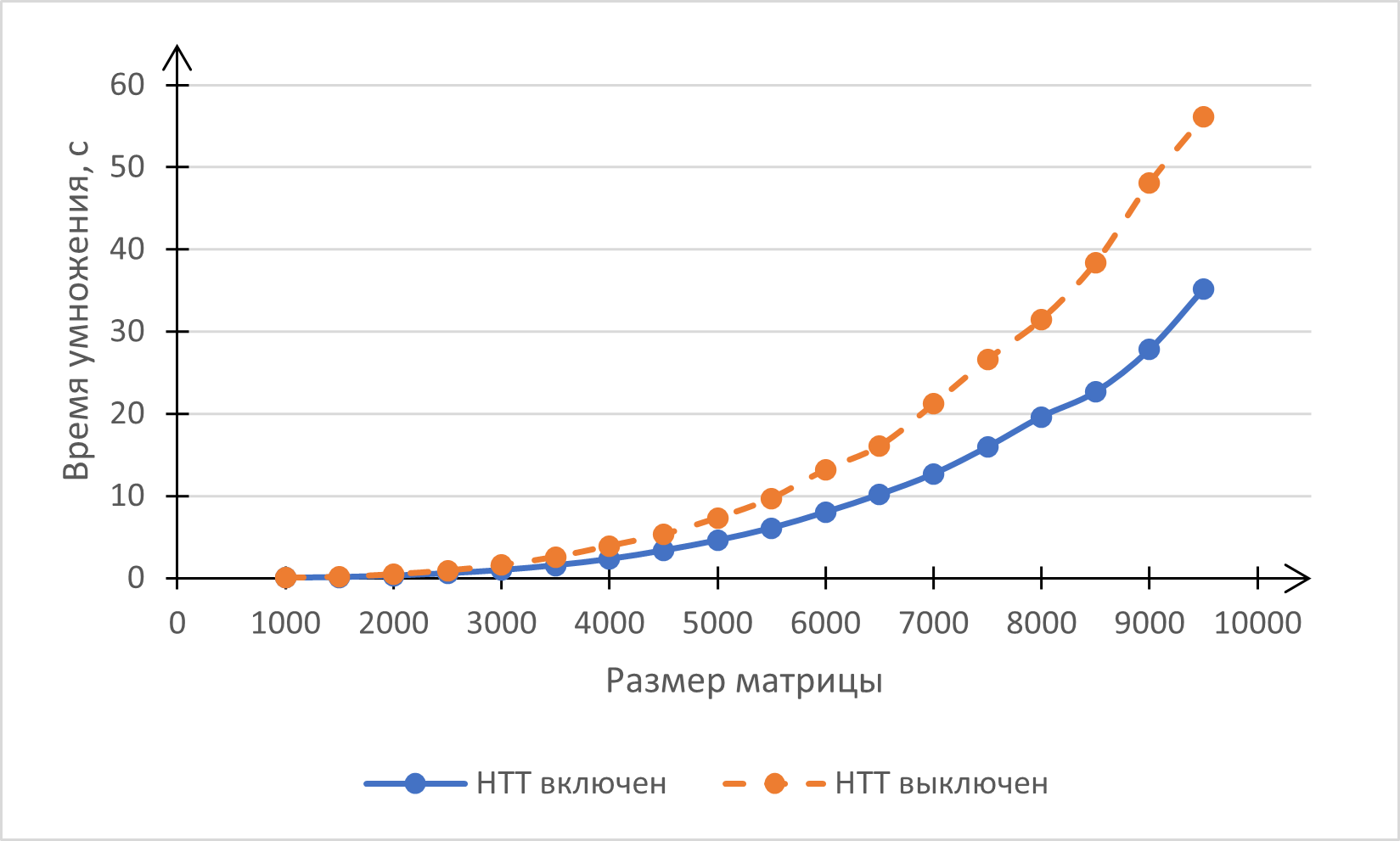


Рисунок 5.2 – График сравнения времени выполнения умножения на всех ядрах CPU с HTT и без HTT

Среднее значение отношения времени выполнения умножения без использования *HTT* ко времени умножения с использованием *HTT* (рисунок 5.3) составило 1,59, а медианное – 1,61.

Причиной может служить то, что при выключенном *HT* процессор не задействует все доступные вычислительный устройства (*FPU* в данном случае) и, как следствие, показывает меньшую производительность. Однако при включении *HT* операционная система подает на вход второй поток инструкций (два логических ядра вместо одного), используя при этом оставшиеся доступные вычислительные устройства. В результате производительность повышается.

Далее сравнение производительности процессора в однопоточном и многопоточном режиме работы (запуск программы на одном и на всех ядрах).

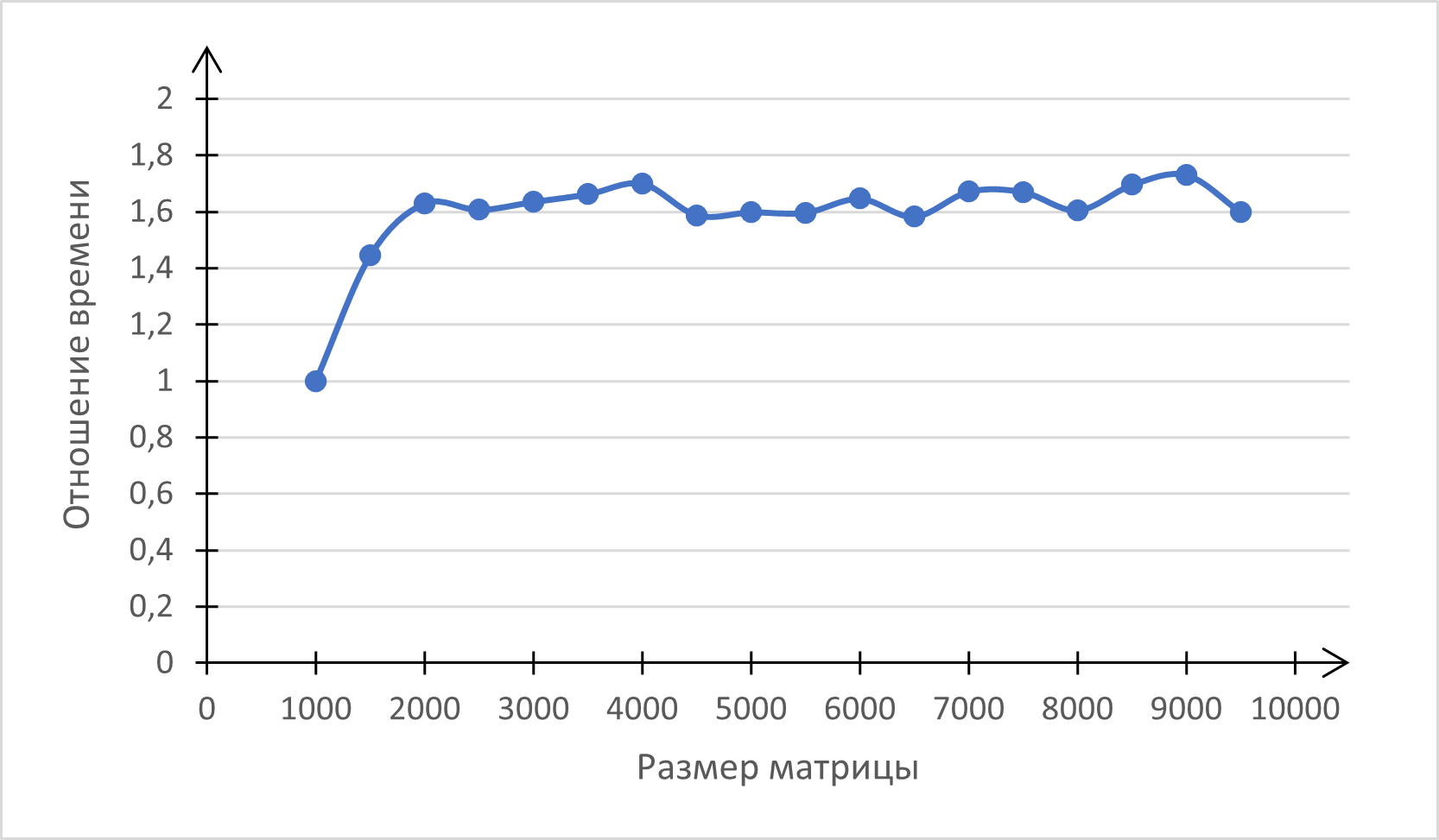


Рисунок 5.3 – График отношения времени выполнения умножения на всех ядрах CPU без HTT ко времени выполнения на всех ядрах CPU с HTT

Как видно из графика (рисунок 5.4), максимальный прирост производительности составил 6,19 раз, а минимальный – 5.44 раза.

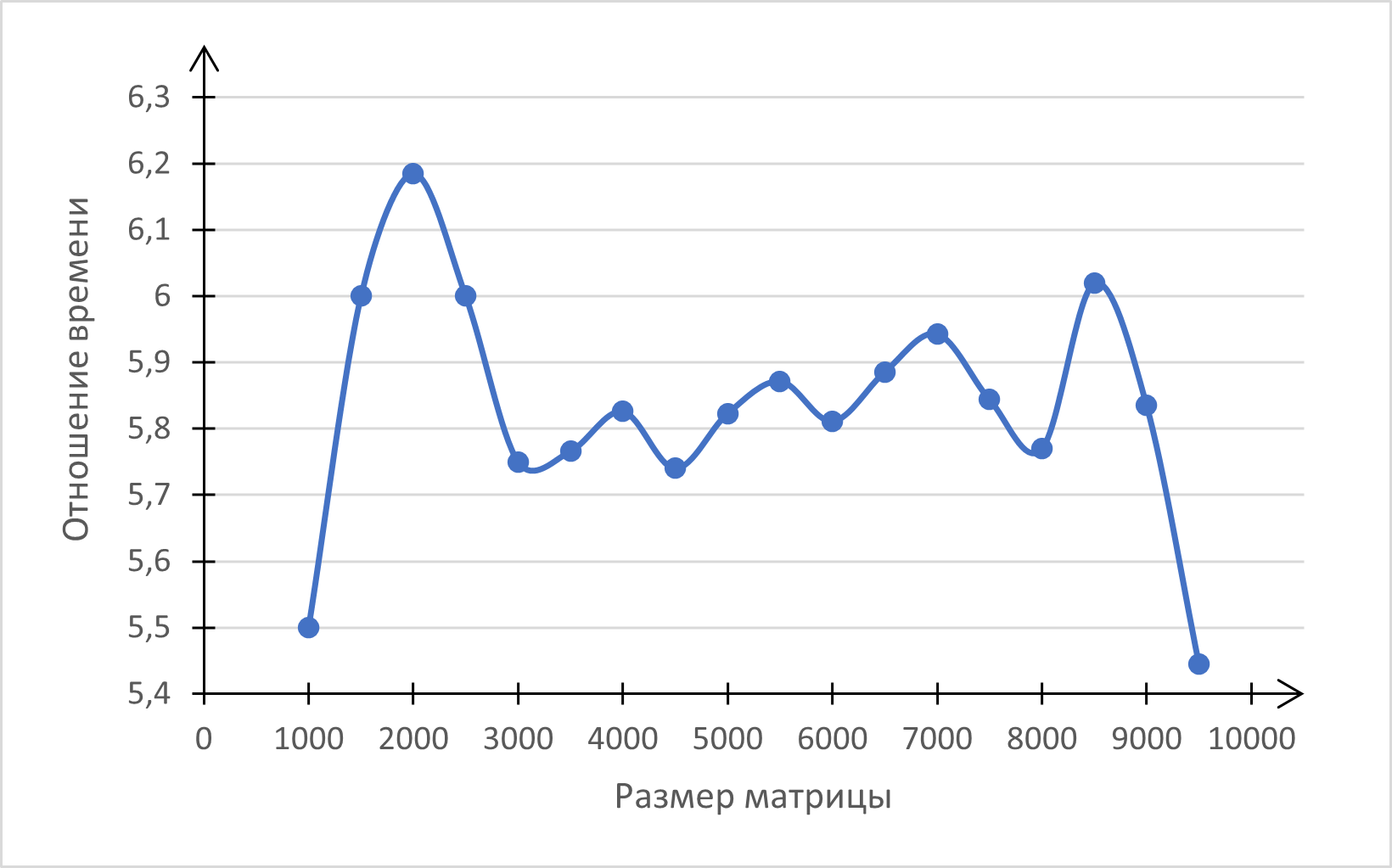


Рисунок 5.4 – График отношения времени выполнения умножения на одном ядре CPU с HTT и ко времени умножения на всех ядрах CPU с HTT

Повышение производительности, очевидно, объясняется тем, что в процессе вычислений используется в 6 раз больше физических ядер.

### 5.6.3 Анализ запусков на CPU на одном ядре с HTT и на GPU на одном CUDA ядре

Результаты запусков приведены в таблице 5.1 и 5.5.

Как видно из графика (рисунок 1 приложения Г), CPU значительно опережает GPU в производительности при одноядерном режиме работы. Время работы на CPU в сотни раз меньше времени работы на GPU (рисунок 2 приложения Г). Это объясняется строением ядра центрального процессора и *CUDA*-ядра на *GPU*. Во-первых, ядро на центральном процессоре содержит больше вычислительных устройств, чем CUDA-ядро (рисунок 5.5).

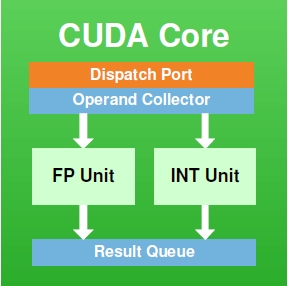


Рисунок 5.5 – Строение *CUDA*-ядра

Во-вторых, тактовая частота ядер на центральном процессоре в несколько раз больше, чем на *GPU*. В-третьих, *CPU* поддерживает принцип *SIMD* (*single* *instruction*, *multiple* *data*) и может выполнять более сложные наборы инструкций (*AVX*, *SSE*, *MMX* и т.д.), чем *GPU*.

### 5.6.4 Анализ запусков на CPU на всех ядрах с HTT и на GPU на всех CUDA-ядрах

Результаты запусков приведены в таблице 5.3 и таблице 5.6.

Сравнение времени выполнения умножения (в данном случае на GPU не использовалась разделяемая память) представлено на рисунке 3 в приложении Г. Из графика видно, что GPU требуется намного меньше времени для завершения операции умножения, чем CPU. Среднее значение отношения времени выполнения на CPU ко времени выполнения на GPU составило 4,20 (рисунок 4 приложение Г). Как описано выше, процессор, выполняя задачу на одном ядре, показывает гораздо большую производительность, чем GPU на одном CUDA-ядре. Однако в отличии от процессора (в данном курсовом проекте рассматривался Intel Core i5-11400H, который имеет 6 физических ядер), GPU имеет тысячи CUDA-ядер (Nvidia GeForce RTX 3060 mobile имеет 3840 CUDA-ядер). Благодаря значительному перевесу по количеству исполняемых устройств GPU выдает большую производительность для массивно-параллельных задач.

Сравним результаты выполнения умножения на GPU с использованием разделяемой памяти и без нее (рисунок 5 приложение Г). Из графика, что использование разделяемой памяти повышает производительность. В среднем повышается в 3,88 раза. Это объясняется тем

При использовании разделяемой памяти производительность относительно многопоточного режима работы CPU с HTT увеличилась в среднем в 16 раз (рисунок 6 приложение Г).

Также, результат сравнения работы *GPU* на одном и тысячах *CUDA*-ядрах показывает, что в многопоточном режиме работы GPU в тысячи раз производительнее, чем в однопоточном режиме работы (рисунок 5.5).

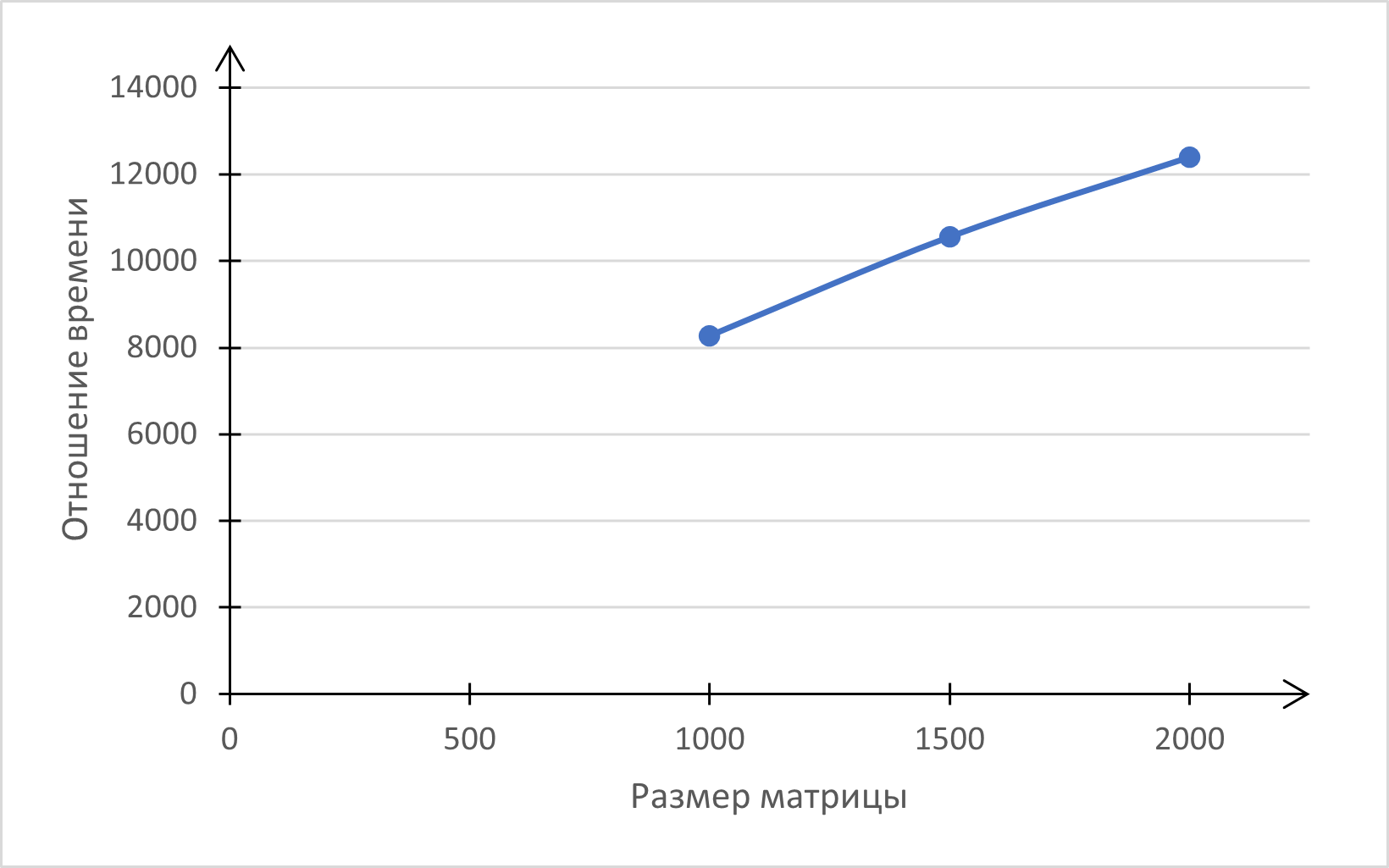


Рисунок 5.5 – График отношения времени выполнения на одном CUDA-ядре на GPU ко времени выполнения на тысячах CUDA-ядер

Очевидной причиной является тысячекратное увеличение количества задействованных вычислительных устройств.

# ЗАКЛЮЧЕНИЕ

В результате выполнения данного курсового проекта было проведено сравнение параллельного выполнения операции умножения матриц на графическом процессоре (GPU) Nvidia GeForce RTX 3060 mobile с использованием технологии CUDA и на центральном процессоре (CPU) Intel Core i5-11400H.

В процессе работы были рассмотрены способы реализации многопоточных алгоритмов умножения матриц. Были разработаны программы, выполняющие умножение как на GPU (в однопоточном и многопоточном режиме), так и на CPU (в однопоточном и многопоточном режиме).

Для сравнения эффективности использования GPU и CPU использовался такой критерий, как время выполнения вычислений (умножения матриц).

Анализ полученных результатов показал, что параллельное выполнение умножения матриц на GPU с использованием технологии CUDA может значительно ускорить вычисления по сравнению с выполнением на CPU. GPU обладает большим количеством ядер и параллельных вычислительных ресурсов, что позволяет эффективно обрабатывать большие объемы данных.

Однако эффективность использования GPU зависит от характеристик конкретной задачи и размера матриц. В случае малых матриц или задач, требующих интенсивного взаимодействия с памятью, CPU может быть более эффективным выбором.

Таким образом, для оптимального выбора между GPU и CPU необходимо учитывать особенности конкретной задачи, размеры матриц и доступные ресурсы.

# СПИСОК ЛИТЕРАТУРНЫХ ИСТОЧНИКОВ

* + - 1. Доманов, А. Т. Стандарт предприятия / А. Т. Доманов, Н. И. Сорока. – Минск: Учреждение образования "Белорусский государственный университет информатики и радиоэлектроники", 2017. – 168 с.
      2. Intel® Core™ i5-11400H Processor [Электронный ресурс]. – Режим доступа: https://ark.intel.com/content/www/us/en/ark/products/213805/intel-core-i5-11400h-processor-12m-cache-up-to-4-50-ghz.html?wapkw=intel%20core%20i5%2011400H. – Дата доступа: 19.09.2023.
      3. Tiger Lake H [Электронный ресурс]. – Режим доступа: https://www.intel.com/content/www/us/en/products/platforms/details/tiger-lake-h.html. – Дата доступа: 19.09.2023.
      4. What Is Hyper-Threading? [Электронный ресурс]. – Режим доступа: https://www.intel.com/content/www/us/en/gaming/resources/hyper-threading.html. – Дата доступа: 19.09.2023.
      5. NVIDIA GeForce RTX 3060 Mobile [Электронный ресурс]. – Режим доступа: https://www.techpowerup.com/gpu-specs/geforce-rtx-3060-mobile.c3757. – Дата доступа: 21.09.2023.
      6. Архитектура NVIDIA Ampere [Электронный ресурс]. – Режим доступа: https://www.nvidia.com/ru-ru/data-center/ampere-architecture/. – Дата доступа: 21.09.2023.
      7. Боресков, А. В. Основы работы с технологией CUDA / А. В. Боресков, А. А. Харламов. – М: ДМК Пресс, 2010. – 232 с.
      8. Параллельное программирование с CUDA. Часть 2: Аппаратное обеспечение GPU и шаблоны параллельной коммуникации [Электронный ресурс]. – Режим доступа: https://habr.com/ru/companies/epam\_systems/articles/245523/. – Дата доступа: 24.09.2023.
      9. Что такое Visual Studio? [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/visualstudio/get-started/visual-studio-ide?view=vs-2022. – Дата доступа: 24.09.2023.
      10. NVIDIA Nsight Compute [Электронный ресурс]. – Режим доступа: https://developer.nvidia.com/nsight-compute. – Дата доступа: 27.09.2023.
      11. NVIDIA Nsight Systems [Электронный ресурс]. – Режим доступа: https://developer.nvidia.com/nsight-systems. – Дата доступа: 27.09.2023.
      12. Уорд, Б. Внутреннее устройство Linux / Б. Уорд. – СПб: Питер, 2016. – 384 с.
      13. Страуструп, Б. Язык программирования C++ / Б. Страуструп. – Бостон: Addison-Wesley, 2013. – 1054 с.
      14. Stallman, R. Using the GNU Compiler Collection / R. Stallman. – Boston: Free Software Foundation, 2023. – 1093 с.
      15. NVIDIA CUDA Compiler Driver NVCC [Электронный ресурс]. – Режим доступа: https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html. – Дата доступа: 11.10.2023.
      16. Антонов, А. С. Параллельное программирование с использованием технологии OpenMP / А. С. Антонов. – М.: Изд-во МГУ, 2009. – 77 с.
      17. chrono [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/cpp/standard-library/chrono?view=msvc-170. – Дата доступа: 15.10.2023.
      18. Корн, Г. Справочник по математике / Г. Корн, Т. Корн. – М.: Наука, 1973. – 832 с.
      19. Кормен, Т. И др. Алгоритмы: построение и анализ / Т. Кормен. – М.: ООО "И. Д. Вильямс", 2013. – 1328 с.
      20. Cachegrind: a cache-miss profiler [Электронный ресурс]. – Режим доступа: https://wwwcdf.pd.infn.it/valgrind/cg\_main.html. – Дата доступа: 18.11.2023.

# Приложение А

**(обязательное)**

**Исходный код**

#include <atomic>

#include "cuda\_runtime.h"

#include <thrust/random.h>

#include <thrust/transform.h>

#include <thrust/device\_vector.h>

#include <thrust/host\_vector.h>

#include <fstream>

#include <type\_traits>

template <typename T>

void saveToBinary(const thrust::host\_vector<T>& v, const std::string& filename) {

std::ofstream file(filename, std::ios::binary | std::ios::trunc | std::ios::out);

file.seekp(0);

size\_t size = v.size();

file.write(reinterpret\_cast<const char\*>(&size), sizeof(size\_t));

file.write(reinterpret\_cast<const char\*>(v.data()), size \* sizeof(T));

file.close();

}

template <typename T>

using DistributionType = typename std::conditional<std::is\_integral\_v<T>, thrust::uniform\_int\_distribution<T>, thrust::uniform\_real\_distribution<T>>::type;

template <typename T>

struct GenRand {

thrust::default\_random\_engine rand\_eng;

DistributionType<T> uni\_dist;

\_\_device\_\_

GenRand(): rand\_eng(thrust::default\_random\_engine{static\_cast<unsigned int>(clock())}) {

}

\_\_device\_\_

T operator () (int idx) {

rand\_eng.discard(idx);

return uni\_dist(rand\_eng);

}

};

template <typename T>

void generate(const std::string& fileName, size\_t size) {

thrust::device\_vector<T> a(size);

thrust::transform(thrust::make\_counting\_iterator(0ULL), thrust::make\_counting\_iterator(static\_cast<unsigned long long>(size)), a.begin(), GenRand<T>());

thrust::host\_vector<T> a\_copy = a;

saveToBinary<T>(a\_copy, fileName);

}

int main(int argc, char\* argv[]) {

size\_t N = 3;

if(argc != 4) {

std::cout << "Requires 3 args\n";

return 0;

}

else {

N = std::atoi(argv[2]);

if(N < 2 || N > 30000) {

std::cout << "Invalid size parameter (2 <= size <= 30000).\n";

return 0;

}

std::string dataType = argv[3];

if(dataType != "f" && dataType != "i") {

std::cout << "Invalid data type (f or i).\n";

return 0;

}

}

std::string fileName = argv[1];

std::string dataType = argv[3];

const size\_t size = N \* N;

if(dataType == "f") {

generate<float>(fileName, size);

}

else if (dataType == "i") {

generate<int>(fileName, size);

}

else {

std::cout << "Nothing was generated! Invalid data type.";

}

return 0;

}

#include <omp.h>

#include <atomic>

#include <chrono>

#include <cmath>

#include <exception>

#include <filesystem>

#include <fstream>

#include <functional>

#include <iostream>

#include <thread>

#include <vector>

template <typename T>

void readFromBinary(std::vector<T>& v, const std::filesystem::path& fileName) {

std::ifstream file(fileName, std::ios::binary);

if (!file.good())

throw std::invalid\_argument("Invalid file! " + fileName.string());

size\_t size = 0;

file.read(reinterpret\_cast<char\*>(&size), sizeof(size\_t));

v.resize(size);

file.read(reinterpret\_cast<char\*>(v.data()), size \* sizeof(T));

file.close();

}

template <typename T>

void matrixMul(const std::vector<T>& a, const std::vector<T>& b,

std::vector<T>& result) {

size\_t size = sqrt(a.size());

int j, k;

#pragma omp parallel for

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

for (k = 0; k < size; k++) {

for (j = 0; j < size; j++) {

result[i \* size + j] += a[i \* size + k] \* b[k \* size + j];

}

}

}

}

template <typename T>

void matrixMulNotParallel(const std::vector<T>& a, const std::vector<T>& b,

std::vector<T>& result) {

size\_t size = sqrt(a.size());

int j, k;

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

for (k = 0; k < size; k++) {

for (j = 0; j < size; j++) {

result[i \* size + j] += a[i \* size + k] \* b[k \* size + j];

}

}

}

}

void printProgressIndicator(std::atomic<bool>& isCalculating) {

const std::string indicators = "-\\|/";

int index = 0;

while (isCalculating) {

std::cout << "\rCalculating... " << indicators[index++];

index %= indicators.size();

std::cout.flush();

std::this\_thread::sleep\_for(std::chrono::milliseconds(200));

}

std::cout << '\n';

}

template <typename T>

auto measureTime(const std::vector<T>& a, const std::vector<T>& b,

std::vector<T>& result, const std::string& operationType) {

auto start = std::chrono::high\_resolution\_clock::now();

if (operationType == "p") {

matrixMul(a, b, result);

} else if (operationType == "np") {

matrixMulNotParallel(a, b, result);

} else {

throw std::invalid\_argument("Invalid operationType argument!");

}

auto end = std::chrono::high\_resolution\_clock::now();

auto elapsed\_time =

std::chrono::duration\_cast<std::chrono::nanoseconds>(end - start);

return elapsed\_time.count();

}

template <typename T>

void calculate(const std::vector<std::filesystem::path>& filePaths,

const std::filesystem::path& outFileName,

const std::string& operationType) {

std::fstream outfile(outFileName, std::ios::trunc | std::ios::out);

if (!outfile.is\_open()) {

throw std::invalid\_argument("Invalid outFileName! " +

outFileName.string());

}

std::atomic<bool> isCalculating;

isCalculating.store(true);

std::thread progressThread(printProgressIndicator, std::ref(isCalculating));

std::this\_thread::sleep\_for(std::chrono::milliseconds(500));

for (const auto& filePath : filePaths) {

std::vector<T> a;

readFromBinary(a, filePath);

std::vector<T> b = a;

std::vector<T> result(a.size(), 0);

auto time = measureTime<T>(a, b, result, operationType);

outfile << std::to\_string(time) + " " +

filePath.filename().string() + "\n";

}

outfile.close();

isCalculating.store(false);

progressThread.join();

}

int main(int argc, char\* argv[]) {

omp\_set\_num\_threads(omp\_get\_max\_threads());

std::filesystem::path outFileName = std::filesystem::absolute(argv[1]);

std::string dataDir = argv[2];

std::string operationType = argv[3];

if (operationType != "p" && operationType != "np") {

std::cout << "3rd argument nust be equal to p (parallel) or np (not "

"parallel)!";

return 0;

}

std::filesystem::directory\_iterator iterator(dataDir);

std::vector<std::filesystem::path> fileNames;

for (const auto& entry : iterator) {

if (entry.is\_regular\_file()) {

std::filesystem::path filePath =

std::filesystem::absolute(entry.path()).lexically\_normal();

fileNames.push\_back(filePath);

}

}

calculate<float>(fileNames, outFileName, operationType);

}

#include <thrust/device\_vector.h>

#include <thrust/host\_vector.h>

#include <atomic>

#include <chrono>

#include <filesystem>

#include <fstream>

#include <thread>

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

template <typename T>

\_\_global\_\_ void matrixMulKernel(T\* c, const T\* a, const T\* b, size\_t N) {

size\_t j = blockIdx.x \* blockDim.x + threadIdx.x;

size\_t i = blockIdx.y \* blockDim.y + threadIdx.y;

if (i < N && j < N) {

auto idx = i \* N + j;

for (size\_t k = 0; k < N; k++) {

c[idx] += a[i \* N + k] \* b[k \* N + j];

}

}

}

template <typename T>

\_\_global\_\_ void matrixMulKernelNotParallel(T\* c, const T\* a, const T\* b,

size\_t N) {

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

for (int k = 0; k < N; k++) {

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

c[i \* N + j] += a[i \* N + k] \* b[k \* N + j];

}

}

}

}

#define BLOCK\_DIM 16

template <typename T>

\_\_global\_\_ void matrixMulKernelSharedMemory(T\* c, const T\* a, const T\* b,

const size\_t N) {

int blockX = blockIdx.x;

int blockY = blockIdx.y;

int threadX = threadIdx.x;

int threadY = threadIdx.y;

int i = blockY \* blockDim.y + threadY;

int j = blockX \* blockDim.x + threadX;

\_\_shared\_\_ T aShared[BLOCK\_DIM][BLOCK\_DIM];

\_\_shared\_\_ T bShared[BLOCK\_DIM][BLOCK\_DIM];

T sum = 0;

for (int part = 0; part < ceil(static\_cast<T>(N) / BLOCK\_DIM); ++part) {

int iRow = i;

int iCol = part \* BLOCK\_DIM + threadX;

int jCol = j;

int jRow = part \* BLOCK\_DIM + threadY;

aShared[threadY][threadX] =

(iRow < N && iCol < N) ? a[iRow \* N + iCol] : 0;

bShared[threadY][threadX] =

(jRow < N && jCol < N) ? b[jRow \* N + jCol] : 0;

\_\_syncthreads();

for (int idx = 0; idx < BLOCK\_DIM; ++idx) {

sum += aShared[threadY][idx] \* bShared[idx][threadX];

}

\_\_syncthreads();

}

if (i < N && j < N) {

c[i \* N + j] = sum;

}

}

template <typename T>

void readFromBinary(thrust::host\_vector<T>& v,

const std::filesystem::path& fileName) {

std::ifstream file(fileName, std::ios::binary);

if (!file.good())

throw std::invalid\_argument("Invalid file! " + fileName.string());

size\_t size = 0;

file.read(reinterpret\_cast<char\*>(&size), sizeof(size\_t));

v.resize(size);

file.read(reinterpret\_cast<char\*>(v.data()), size \* sizeof(T));

file.close();

}

void printProgressIndicator(std::atomic<bool>& isCalculating) {

const std::string indicators = "-\\|/";

int index = 0;

while (isCalculating) {

std::cout << "\rCalculating... " << indicators[index++];

index %= indicators.size();

std::cout.flush();

std::this\_thread::sleep\_for(std::chrono::milliseconds(200));

}

std::cout << '\n';

}

template <typename T>

long long measureTime(const thrust::device\_vector<T>& a,

thrust::device\_vector<T>& b,

thrust::device\_vector<T>& result, size\_t N,

const std::string& kernelType) {

dim3 blockSize;

dim3 numBlocks;

if (kernelType == "ps" || kernelType == "p") {

blockSize = dim3(BLOCK\_DIM, BLOCK\_DIM);

numBlocks = dim3((N + blockSize.x - 1) / blockSize.x,

(N + blockSize.y - 1) / blockSize.y);

} else {

blockSize = dim3(1, 1);

numBlocks = dim3(1, 1);

}

auto start = std::chrono::high\_resolution\_clock::now();

auto rawResult = thrust::raw\_pointer\_cast(result.data());

auto rawA = thrust::raw\_pointer\_cast(a.data());

auto rawB = thrust::raw\_pointer\_cast(b.data());

if (kernelType == "p") {

matrixMulKernel<<<numBlocks, blockSize>>>(rawResult, rawA, rawB, N);

} else if (kernelType == "np") {

matrixMulKernelNotParallel<<<numBlocks, blockSize>>>(rawResult, rawA,

rawB, N);

} else if (kernelType == "ps") {

matrixMulKernelSharedMemory<<<numBlocks, blockSize>>>(rawResult, rawA,

rawB, N);

} else {

throw std::invalid\_argument("Invalid kernelType argument!");

}

cudaDeviceSynchronize();

auto end = std::chrono::high\_resolution\_clock::now();

auto elapsed\_time =

std::chrono::duration\_cast<std::chrono::nanoseconds>(end - start);

return elapsed\_time.count();

}

template <typename T>

void calculate(const std::vector<std::filesystem::path>& filePaths,

const std::filesystem::path& outFileName,

const std::string& kernelType) {

std::fstream outfile(outFileName, std::ios::trunc | std::ios::out);

if (!outfile.is\_open()) {

throw std::invalid\_argument("Invalid outFileName! " +

outFileName.string());

}

std::atomic<bool> isCalculating;

isCalculating.store(true);

std::thread progressThread(printProgressIndicator, std::ref(isCalculating));

std::this\_thread::sleep\_for(std::chrono::milliseconds(500));

for (const auto& filePath : filePaths) {

thrust::host\_vector<T> h\_a;

readFromBinary<T>(h\_a, filePath);

thrust::device\_vector<T> d\_a = h\_a;

thrust::device\_vector<T> d\_b = h\_a;

thrust::device\_vector<T> d\_c(h\_a.size());

const size\_t N = sqrt(h\_a.size());

auto time = measureTime<T>(d\_a, d\_b, d\_c, N, kernelType);

outfile << std::to\_string(time) + " " +

filePath.filename().string() + "\n";

}

outfile.close();

isCalculating.store(false);

progressThread.join();

}

int main(int argc, char\* argv[]) {

std::filesystem::path outFileName = std::filesystem::absolute(argv[1]);

std::string dataDir = argv[2];

std::string kernelType = argv[3];

if (kernelType != "p" && kernelType != "ps" && kernelType != "np") {

std::cout << "3rd argument must be equal to p (parallel) or ps "

"(parallel with shared memory) or np (not parallel)!";

return 0;

}

std::filesystem::directory\_iterator iterator(dataDir);

std::vector<std::filesystem::path> fileNames;

for (const auto& entry : iterator) {

if (entry.is\_regular\_file()) {

std::filesystem::path filePath =

std::filesystem::absolute(entry.path()).lexically\_normal();

fileNames.push\_back(filePath);

}

}

calculate<float>(fileNames, outFileName, kernelType);

return 0;

}

# Приложение Б

**(обязательное)**

**Функциональная схема алгоритма**

# Приложение В

**(обязательное)**

**Блок схема алгоритма**

# Приложение Г

**(обязательное)**

**Графический интерфейс пользователя**

# Приложение Д

**(обязательное)**

**Ведомость документов**