Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина Архитектура Вычислительных Систем

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к курсовому проекту
на тему

«СРАВНЕНИЕ ПРОИЗВОДИТЕЛЬНОСТИ ПРОЦЕССОРОВ INTEL CORE I5-12450H И AMD RYZEN 7 5800H НА ОСНОВЕ ВЫПОЛНЕНИЯ ПРЕОБРАЗОВАНИЙ ФУРЬЕ»

БГУИР КП 6-05-0612-02 023 ПЗ

Студент Д. Ю. Себелев

Руководитель А. А. Калиновская

Нормоконтроллёр А. А. Калиновская

Минск 2025

СОДЕРЖАНИЕ

Введение 5
1 Архитектура вычислительной системы 6
1.1 Структура и архитектура вычислительной системы 6
1.2 История, версии и достоинства 8
1.3 Обоснование выбора вычислительной системы 9
1.4 Анализ выбранной вычислительной системы для написания программы 9
2 Платформа программного обеспечения 11
2.1 Структура и архитектура платформы 11
2.2 История, версии и достоинства 12
2.3 Обоснование выбора платформы 14
2.4 Анализ операционной системы для написания программы 14
3 Теоретическое обоснование разработки программного продукта 16
3.1 Обоснование необходимости разработки 16
3.2 Технологии программирования, используемые для решения поставленных задач 17
3.3 Связь архитектуры вычислительной системы с разрабатываемым программным обеспечением 18
4 Проектирование функциональных возможностей программы 20
4.1 Общая цель программы 20
4.2 Теоретические сведения о алгоритме быстрого преобразования Фурье 20
4.3 Описание функциональных возможностей 21
4.4 Описание функциональной схемы алгоритма 22
4.5 Описание блок схемы алгоритма 23
5 Сравнение производительности процессоров 25
5.1 Общая структура программы 25
5.2 Подготовка к тестированию 25
5.3 Результаты тестирования с включенным SMT/Hyper-Threading 26
5.4 Результаты тестирования с выключенным SMT/Hyper-Threading 30
5.5 Результаты тестирования на iGPU 33
Заключение 35
Список литературных источников 36
Приложение А (обязательное) Справка о проверке на заимствования 37
Приложение Б (обязательное) Листинг программного кода 38
Приложение В (обязательное) Функциональная схема алгоритма, реализующая программное средство 46
Приложение Г (обязательное) Блок схема алгоритма, реализующего программное средство 47
Приложение Д (обязательное) Графики сравнения производительности процессоров 48
Приложение Е (обязательное) Графическое представления нагрузки на ядра процессоров 50
Приложение Ж (обязательное) Ведомость курсового проекта 51

ВВЕДЕНИЕ

В эпоху цифровой трансформации производительность центральных процессоров становится ключевым фактором эффективности вычислительных систем. Развитие технологий приводит к постоянному появлению новых архитектурных решений, направленных на повышение производительности и энергоэффективности.

Вследствие этого возникает многообразие архитектур, использующих различные подходы к построению многоядерных систем. Понимание того, как эти подходы влияют на производительность при решении конкретных вычислительных задач, представляет значительный научный и практический интерес.

Актуальность темы данной курсового проекта обусловлена несколькими ключевыми факторами. Выбранные для сравнения процессоры являются популярными представителями своих линеек в сегменте производительных ноутбуков и построены на принципиально разных архитектурных подходах. Их прямое сравнение позволяет на конкретном примере оценить преимущества и недостатки современных тенденций в проектировании центральных процессоров.

В качестве тестовой нагрузки выбрано быстрое преобразование Фурье (БПФ). Это вычислительно интенсивная задача, эффективность выполнения которой напрямую зависит от архитектурных особенностей процессора. Поэтому анализ производительности БПФ является репрезентативным тестом для широкого класса научных и инженерных приложений.

Целью данной курсового проекта является проведение сравнительного анализа производительности процессоров *Intel Core i5-12450H* и *AMD Ryzen 7 5800H* при выполнении алгоритмов преобразования Фурье.

Для достижения поставленной цели необходимо решить следующие задачи:

1 Изучение технических характеристик и архитектурных особенностей рассматриваемых процессоров.

2 Разработка программы для выполнения преобразования Фурье.

3 Проведение вычислительных экспериментов с различными размерами данных, числом ядер и потоков.

4 Анализ полученных данных для оценки производительности процессоров.

В результате исследования будут построены графики, демонстрирующие зависимость времени выполнения преобразования Фурье от размера данных, что позволит провести детальный анализ производительности рассматриваемых процессоров.

Данный курсовой проект выполнен мной лично, проверен на заимствования, процент оригинальности составляет XX% (отчет о проверке на заимствования прилагается).

1 АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНОЙ СИСТЕМЫ

1.1 Структура и архитектура вычислительной системы

Процессоры *AMD Ryzen 7 5800H* и *Intel Core i5-12450H* основаны на принципиально разных архитектурных подходах к построению вычислительных систем и ориентированы мобильные вычислительные системы. Их характеристики приведены в таблице 1.1.

Таблица 1.1 – Сравнение характеристик

Характеристики	*AMD Ryzen 7 5800H*	*Intel Core i5-12450H*
Кодовое имя архитектуры	*Cezanne-H (Zen 3)*	*Alder Lake-H*
Физические ядра	8	8
Количество потоков	16	12
L1 кэш на ядро	64 КБ	80 КБ + 96 КБ
L2 кэш на ядро	512 КБ	1.25 МБ + 2 МБ[1)]
L3 кэш	16 МБ	12 МБ
Тактовая частота	3.2 - 4.4 Ггц	2 - 4.4 Ггц
Техпроцесс	7 нм	10 нм
Встроенная графика	*AMD Radeon RX Vega 8*	*Intel UHD Graphics Xe G4*
Поддерживаемая память	*DDR4*	*DDR4, DDR5*
TDP	45 Вт	45 Вт
Hyper-Threading/SMT	+	+

Процессор *AMD Ryzen 7 5800H*, использует монолитную архитектуру *Zen 3*, произведенную по техпроцессу 7 нм [1]. Данная архитектура предполагает использование 8 ядер, которые с помощью технологии *SMT* (*Simultaneous Multithreading*) обеспечивают обработку 16 потоков. Максимальная тактовая частота составляет 4.4 ГГц. Процессор оснащен 16 МБ кэш-памяти третьего уровня (*L3*), а его тепловой пакет (*TDP*) составляет 45 Вт. За обработку графики отвечает встроенный видеоадаптер *AMD Radeon RX Vega 8*. Он построен на проверенной временем архитектуре *Vega*, имеет в своем составе 8 вычислительных блоков и способен работать на частоте до 2.0 ГГц.

Процессор *Intel Core i5-12450H* базируется на гибридной архитектуре *Alder Lake*, изготовленной по техпроцессу 10 нм [2]. В его состав входят 4 производительных *P*-ядра и 4 энергоэффективных *E*-ядра, что обеспечивает поддержку 12 потоков. Распределение нагрузки между ядрами осуществляется аппаратным планировщиком *Intel Thread Director*. Пиковая частота производительных ядер достигает 4.4 ГГц, объем кэш-памяти *L3*

[1)] E-ядра используют общий 2 МБ кластер

составляет 12 МБ, а тепловой пакет (*TDP*) — 45 Вт. В него интегрировано графическое ядро *Intel UHD Graphics Xe G4*, принадлежащее к более современной архитектуре *Xe-LP*. Оно включает 48 исполнительных блоков, но работает на более низкой пиковой частоте в 1.2 ГГц.

Диаграммы топологии, полученные с помощью утилиты *lstopo*, наглядно демонстрируют различие в архитектуре исследуемых процессоров. На рисунке 1.1 видна однородная структура *AMD Ryzen 7 5800H* с восемью идентичными ядрами.

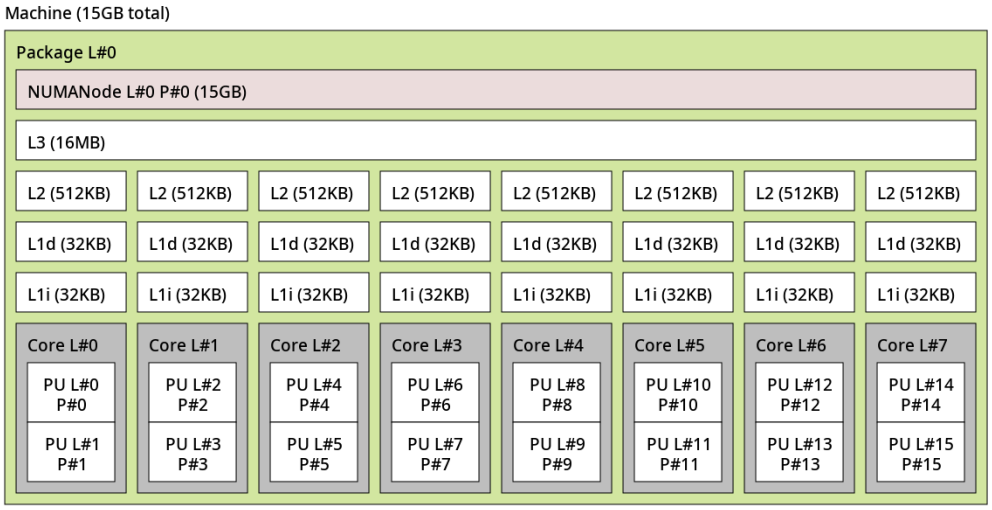

Рисунок 1.1 – Топология *AMD Ryzen 7 5800H*

На рисунке 1.2 представлена гибридная архитектура *Intel Core i5-12450H* с разделением на производительные и энергоэффективные ядра. Четыре производительных *P*-ядра обладают индивидуальными кэшами *L2* размером 1.25 МБ каждое, в то время как четыре энергоэффективных *E*-ядра используют общий кластер кэш-памяти *L2* размером 2 МБ.

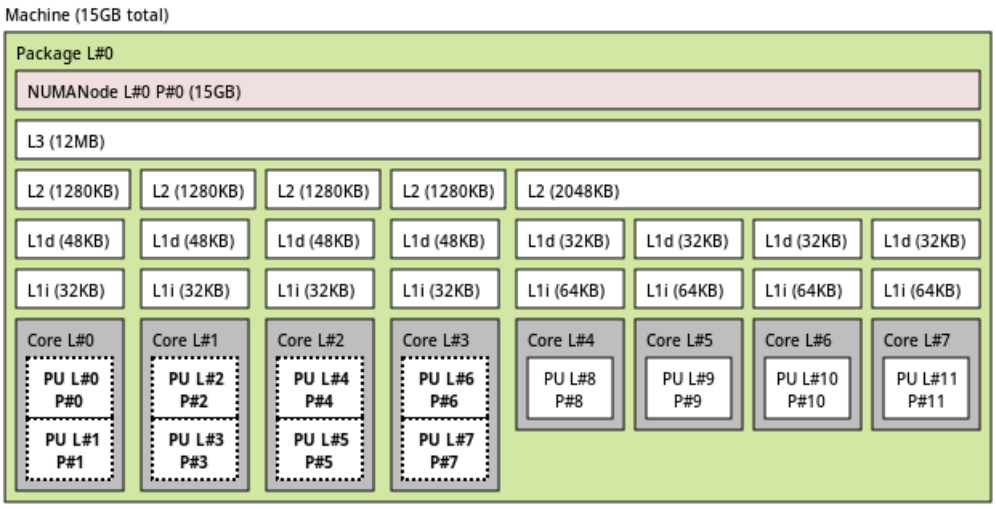

Рисунок 1.2 – Топология *Intel Core i5-12450H*

Таким образом, ключевое различие между процессорами заключается в подходе к многоядерности. Архитектура *Zen 3*, позволяющая процессору оперировать 16 потоками, потенциально лучше подходит для тяжелых параллельных вычислений, где важна максимальная производительность каждого потока, а увеличенный объем кэш-памяти *L3* способствует ускорению работы с большими наборами данных. В свою очередь, архитектура *Alder Lake* использует более современную модель с разнородными ядрами, которая обеспечивает гибкость и энергоэффективность за счет распределения задач между производительными и эффективными ядрами с помощью аппаратного планировщика *Intel Thread Director*. Однако эффективность такого подхода сильно зависит от оптимизации операционной системы и программного обеспечения под гибридную архитектуру. Эти фундаментальные различия в дизайне и станут основой для последующего сравнительного анализа их производительности в реальных задачах.

1.2 История, версии и достоинства

Архитектура *Zen 3*, на которой базируется *Ryzen 7 5800H*, является кульминацией многолетних усовершенствований. Её история начинается с революционной архитектуры *Zen* (2017), которая ознаменовала возвращение *AMD* в сегмент высокопроизводительных процессоров. Последующая *Zen 2* привнесла 7-нм техпроцесс и чиплетный дизайн, значительно увеличив количество ядер. *Zen 3* стала её логическим развитием, сфокусированным на повышении показателя *IPC* (числа инструкций за такт) и снижении задержек. Ключевым нововведением стало объединение ядер и кэш-памяти *L3* в единый комплекс, что улучшило межъядерное взаимодействие. Основные достоинства этой архитектуры заключаются в её монолитной структуре с восемью одинаково производительными ядрами, что обеспечивает предсказуемую производительность. Благодаря технологии *SMT* процессор обрабатывает 16 потоков, что делает его отлично подходящим для параллельных вычислений, а большой объединенный *L3* кэш ускоряет работу с крупными наборами данных.

В свою очередь, архитектура *Alder Lake*, лежащая в основе *Core i5-12450H*, знаменует собой самый радикальный сдвиг для *Intel* за последнее десятилетие. Ей предшествовал долгий период доминирования архитектуры *Skylake* и её многочисленных итераций (от *Kaby Lake* до *Comet Lake*), производившихся по 14-нм техпроцессу. Столкнувшись с растущей конкуренцией, *Intel* разработала *Alder Lake* как комплексный ответ. Эта архитектура не только перешла на новый техпроцесс *Intel 7* (10 нм), но и впервые в настольном сегменте представила гибридную модель. Она сочетает в себе производительные ядра (*P-cores*) для выполнения сложных задач и энергоэффективные ядра (*E-cores*) для фоновых процессов. Управление распределением задач между ними осуществляет аппаратный планировщик *Intel Thread Director*. Преимущества такого подхода заключаются в гибкости и энергоэффективности, поскольку система адаптируется к нагрузке, оптимизируя энергопотребление. *P*-ядра обеспечивают высокую

однопоточную производительность, а поддержка современных технологий, таких как память *DDR5*, гарантирует высокую пропускную способность.

1.3 Обоснование выбора вычислительной системы

Выбор процессоров *AMD Ryzen 7 5800H* и *Intel Core i5-12450H* для сравнительного анализа обусловлен тем, что они, принадлежа к одному классу мобильных решений с одинаковым тепловым пакетом, основываются на принципиально разных архитектурных философиях. Их сопоставление позволяет наглядно оценить сильные и слабые стороны двух доминирующих подходов в современном процессоростроении: «классического» монолитного с однородными ядрами и нового гибридного с разнородными. Ключевым аспектом для сравнения является реализация технологий виртуальной многопоточности: *SMT* (*Simultaneous Multithreading*) у *AMD* и *Hyper-Threading* у *Intel*. В процессоре *Ryzen 7 5800H* технология *SMT* применяется ко всем восьми однородным ядрам, удваивая количество потоков до 16. Это создает симметричную и предсказуемую среду для параллельных вычислений. В свою очередь, в *Intel Core i5-12450H* технология *Hyper-Threading* активна только на четырех производительных *P*-ядрах, в то время как четыре энергоэффективных *E*-ядра не поддерживают ее. В результате общее число потоков составляет 12 (8 от *P*-ядер и 4 от *E*-ядер), что создает асимметричную структуру. Именно анализ того, как операционная система и приложение для тестов будут распределять нагрузку в этих двух принципиально разных моделях многопоточности, и является одной из центральных задач исследования. Наконец, принадлежность процессоров к одному сегменту и схожий уровень энергопотребления создают равные условия для анализа именно архитектурных различий, что делает сравнение объективным.

Выбор данных моделей также обусловлен их широкой распространенностью на рынке, что делает исследование актуальным для многих пользователей. В качестве тестовой нагрузки было выбрано быстрое преобразование Фурье (БПФ) — фундаментальный алгоритм, чувствительный к производительности ядер и подсистемы памяти. Его параллельная природа позволяет эффективно оценить, как каждая архитектура справляется с масштабированием вычислительной нагрузки.

1.4 Анализ выбранной вычислительной системы для написания программы

Анализ производительности выбранных процессоров целесообразно начать с рассмотрения их типичных результатов в популярных синтетических бенчмарках, таких как *Cinebench R24*, *Geekbench 5* и *Geekbench 6*. Эти тесты являются индустриальным стандартом для оценки производительности *CPU* и позволяют сделать предварительные выводы о поведении процессоров в различных сценариях.

Cinebench R24 — это бенчмарк, который оценивает производительность процессора путем рендеринга сложной трехмерной сцены [3]. Тест доступен

в двух режимах: однопоточном, который измеряет производительность одного ядра, и многопоточном, который задействует все доступные процессорные потоки. Этот бенчмарк хорошо показывает пиковую производительность *CPU* в задачах, которые эффективно распараллеливаются.

Geekbench 6 — это кросс-платформенный бенчмарк, который измеряет производительность системы в задачах, имитирующих реальные сценарии использования, от просмотра веб-страниц до обработки изображений и машинного обучения [4]. Он также предоставляет результаты для однопоточного и многопоточного режимов, что позволяет составить комплексное представление о возможностях процессора.

Сравнивая результаты тестирования, можно отметить, что в многопоточном тесте *Cinebench R24* процессор *AMD Ryzen 7 5800H* опережает *Intel Core i5-12450H* примерно на 20%, что показывает его эффективность при работе в многопотоке, в то время как в однопоточном режиме разница составляет 10% в пользу *Intel Core i5-12450H*, что подчеркивает его высокую производительность при выполнении задач с использованием одного ядра.

Тест *Geekbench 6* показывает довольно схожие результаты. В многопоточном тесте *Ryzen 7 5800H* вырывается вперед на 16%, оставляя позади *Core i5-12450H*. Но в однопоточном режиме по прежнему лидирует *Core i5-12450H* с отрывом в 12.5%, что доказывает его преимущество в задачах, не требующих использования нескольких ядер.

Экстраполируя эти результаты на задачу вычисления БПФ, следует ожидать аналогичной картины. В однопоточном режиме *Intel Core i5-12450H*, вероятнее всего, покажет лучшее время. В многопоточном же режиме, где алгоритм может быть эффективно распараллелен, преимущество будет за *AMD Ryzen 7 5800H* благодаря его превосходству в многопоточных вычислениях.

2 ПЛАТФОРМА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

2.1 Структура и архитектура платформы

В качестве операционной системы для разработки и тестирования программного обеспечения была выбрана ОС *Linux*. *Linux* — это семейство *Unix*-подобных операционных систем с открытым исходным кодом, основанных на одноименном ядре. Впервые выпущенное Линусом Торвальдсом в 1991 году, ядро *Linux* стало основой для множества операционных систем, известных как дистрибутивы.

Ключевыми особенностями *Linux*, определившими его выбор, являются высокая стабильность, безопасность, гибкость настройки и модульность. Эти качества сделали его доминирующей системой в серверном сегменте, суперкомпьютерах и встраиваемых системах, а также популярной платформой для разработчиков программного обеспечения.

Архитектура *Linux* строится на нескольких фундаментальных принципах, унаследованных от *Unix*, но получивших собственное развитие:

1 Монолитное ядро с модульной структурой. Ядро *Linux* является монолитным, то есть все основные сервисы работают в едином адресном пространстве (пространстве ядра), имея прямой доступ к аппаратному обеспечению. Однако оно поддерживает динамическую загрузку модулей (драйверов), что позволяет сохранять компактность, обеспечивая широкую поддержку оборудования.

2 Системные библиотеки. Они формируют ключевой промежуточный слой между пользовательскими приложениями и ядром системы. Программы, как правило, не используют системные вызовы напрямую. Вместо этого они обращаются к функциям из системных библиотек, самой известной из которых является *GNU C Library (glibc)*. Такой подход упрощает разработку и повышает переносимость программного обеспечения.

3 Многозадачность и многопользовательский режим. Система изначально проектировалась для одновременной работы нескольких пользователей и параллельного выполнения множества процессов в пространстве пользователя. Ядро использует механизм вытесняющей многозадачности для справедливого распределения процессорного времени.

4 Стандарт иерархии файловой системы (*FHS*). Структура каталогов в *Linux* строго стандартизирована. *FHS* определяет назначение основных директорий, таких как /*bin*, /*etc*, /*home* и /*var*, что обеспечивает предсказуемость и совместимость программного обеспечения.

5 Разделение пространства. Система четко разделена на пространство ядра (*kernel space*) и пространство пользователя (*user space*), где выполняются все прикладные программы. Взаимодействие между ними происходит через строго определенный интерфейс системных вызовов, что повышает стабильность и безопасность системы.

6 Широкая поддержка оборудования. Огромное количество драйверов для самого разного оборудования включено непосредственно в исходный

код ядра *Linux*. Это, в сочетании с модульной структурой, позволяющей подгружать только необходимые драйверы, обеспечивает широкую поддержку устройств «из коробки». Активное участие сообщества и производителей оборудования постоянно расширяет этот список.

На рисунке 2.1 изображена архитектура операционной системы *Linux*.

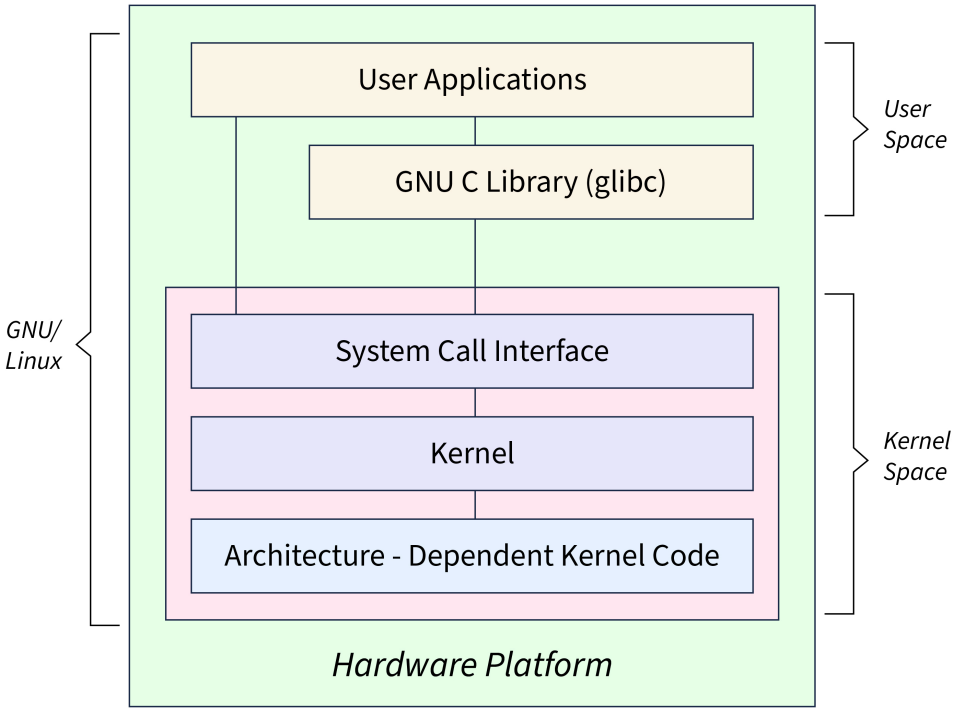

Рисунок 2.1 – Архитектура операционной системы Linux

2.2 История, версии и достоинства

Возникновение операционной системы *Linux* датируется 1991 годом и связано с инициативой финского исследователя Линуса Торвальдса [5]. Изначальной целью разработки было создание *UNIX*-подобного ядра, которое было бы свободно от лицензионных ограничений, присущих существовавшим на тот момент системам, таким как *MINIX*. Первая официальная версия ядра, 0.01, была представлена в сентябре 1991 года.

Фундаментальным этапом в развитии *Linux* стала его интеграция с набором системных утилит и библиотек проекта *GNU*, инициированного Ричардом Столлманом в 1983 году. Проект *GNU* ставил своей целью создание полностью свободной *UNIX*-совместимой операционной системы, и к началу 1990-х годов располагал практически всеми необходимыми компонентами (т.н. *userland*), за исключением низкоуровневого ядра. Объединение ядра *Linux* и пользовательского окружения *GNU* привела к формированию полноценной операционной системы, которую корректно именовать *GNU/Linux*. Перевод проекта под юрисдикцию лицензии *GNU General Public License (GPL) v2* в 1992 году стал решающим фактором, обеспечившим правовую основу для свободного распространения, модификации и коллективной разработки.

Разработка *Linux* в значительной степени была вдохновлена его предшественниками, в первую очередь *UNIX* и *MINIX*. Система *UNIX*, созданная в лабораториях *Bell Labs* в конце 1960-х — начале 1970-х годов, заложила фундаментальные принципы проектирования, ставшие стандартом для современных операционных систем: иерархическая файловая система, концепция процессов и функциональный интерфейс командной строки. Её портируемость и многозадачность сделали её стандартом в академической и коммерческой среде.

Однако со временем лицензирование *UNIX* становилось всё более ограничительным. Это побудило Эндрю Таненбаума в конце 1980-х годов создать *MINIX* — *UNIX*-подобную операционную систему на основе микроядра, предназначенную для образовательных целей. Её исходный код был доступен, но лицензия всё ещё накладывала ограничения на свободное изменение и распространение. Именно опыт работы Линуса Торвальдса с *MINIX* и его желание создать по-настоящему свободную *UNIX*-подобную операционную систему с открытым исходным кодом, которую он мог бы адаптировать для собственного оборудования, и послужили прямой причиной для создания ядра *Linux*.

Ядро *Linux* само по себе не является готовой к использованию операционной системой. Оно служит основой для дистрибутивов — комплексных программных сборок, которые включают также системные утилиты, библиотеки и прикладное программное обеспечение. Дистрибутивы, такие как *Debian*, *Fedora* и *Ubuntu*, различаются системами управления пакетами, набором ПО и целевым назначением, предоставляя пользователям готовые к работе системы для различных задач [6].

Ключевые достоинства *Linux* как платформы для разработки и научных вычислений включают следующие аспекты:

1 Доступность исходного кода: Лицензия *GPL* гарантирует право на изучение, модификацию и распространение исходного кода. Это способствует проведению аудита безопасности, адаптации системы под специфические задачи и ускоряет цикл внедрения новых технологий.

2 Высокая стабильность и надежность: Архитектурные принципы, унаследованные от *UNIX*, обеспечивают высокую отказоустойчивость и способность к длительной непрерывной работе, что является критически важным для серверных систем и длительных вычислительных экспериментов.

3 Модель безопасности: Система разграничения прав доступа для пользователей, групп и прочих субъектов, а также механизм изоляции процессов, значительно снижают риски, связанные с вредоносным программным обеспечением.

4 Гибкость и модульность: Возможность детальной конфигурации всех компонентов системы, от параметров ядра до выбора графической среды, позволяет оптимизировать платформу для конкретных аппаратных ресурсов и прикладных задач.

5 Эффективное управление ресурсами: Планировщик задач ядра *Linux* и подсистема управления памятью обеспечивают эффективное использование

вычислительных ресурсов, что часто выражается в более высокой производительности по сравнению с альтернативными ОС на идентичном оборудовании.

2.3 Обоснование выбора платформы

Для проведения исследования в качестве дистрибутива операционной системы *GNU/Linux* был выбран *Arch Linux* [7]. Этот выбор обусловлен рядом ключевых преимуществ, которые делают его оптимальной платформой для проведения точного и воспроизводимого сравнительного анализа производительности.

Центральным принципом *Arch Linux* является минимализм. Система поставляется в виде базового набора компонентов, предоставляя пользователю полный контроль над устанавливаемым программным обеспечением. Такой подход позволяет создать «чистую» среду для тестирования, свободную от фоновых процессов и служб, которые могли бы вносить помехи в результаты измерений. Это гарантирует, что полученные данные о времени выполнения алгоритма БПФ будут максимально точно отражать производительность самих процессоров, а не влияние стороннего ПО.

Другим важным достоинством является модель непрерывных обновлений (*rolling release*). *Arch Linux* предоставляет доступ к самым последним стабильным версиям программного обеспечения, включая ядро *Linux*, компиляторы (*GCC*, *Clang*) и системные библиотеки. Использование новейшего ядра особенно актуально для данного исследования, поскольку оно включает последние оптимизации планировщика задач, что критически важно для корректной работы с гибридной архитектурой процессора *Intel Core i5-12450H* и его технологией *Thread Director*. Современные компиляторы, в свою очередь, могут предложить улучшенные возможности для оптимизации кода, что напрямую влияет на итоговую производительность.

Ключевым преимуществом *Arch Linux* является его философия, предполагающая полный контроль над конфигурацией системы. Прозрачность и простота структуры дистрибутива позволяют точно документировать и, что более важно, воспроизводить тестовое окружение с высокой степенью достоверности. Это, в сочетании с минимализмом и доступом к новейшему программному обеспечению, обеспечивает создание изолированной и предсказуемой среды, необходимой для получения объективных и сопоставимых результатов производительности.

2.4 Анализ операционной системы для написания программы

Анализ операционной системы в контексте поставленной задачи заключается в определении конкретных инструментов и методологий, которые будут использованы для разработки, компиляции и выполнения программы для тестирования. Выбранная платформа, *Arch Linux*, предоставляет все необходимые средства для проведения глубокого и точного исследования.

Компиляция программы будет производиться с помощью компилятора *GCC* последней версии, доступной в дистрибутиве. Для достижения максимальной производительности компиляция будет выполняться с флагами, активирующими высокий уровень оптимизации, такие как *-O3* и *-march=native*. Флаг *-march=native* заставляет компилятор генерировать код, оптимизированный под специфический набор инструкций конкретного процессора, на котором производится сборка. Это позволит в полной мере задействовать архитектурные преимущества каждого из тестируемых *CPU*.

Ключевым аспектом анализа станет управление выполнением потоков. Операционная система *Linux* предоставляет мощные механизмы для привязки процессов и потоков к конкретным физическим ядрам процессора (*CPU affinity*). С помощью системных вызовов, таких как *sched_setaffinity*, можно будет изолированно измерять производительность:

1 Производительных (*P-cores*) и энергоэффективных (*E-cores*) ядер процессора *Intel Core i5-12450H*.

2 Различного числа ядер (от 1 до 8) процессора *AMD Ryzen 7 5800H*.

Такой подход позволит детально изучить вклад каждого типа ядер в общую производительность и оценить эффективность планировщика задач.

Для получения более глубоких данных о производительности, помимо простого измерения времени выполнения, планируется использование утилиты *perf*. Этот инструмент позволяет собирать данные с аппаратных счетчиков производительности процессора, такие как количество выполненных инструкций, промахи кэша и неверно предсказанные переходы. Анализ этих метрик даст возможность сделать более обоснованные выводы о причинах наблюдаемых различий в производительности.

Таким образом, *Arch Linux* предоставляет полный набор инструментов для разработки, низкоуровневой оптимизации и детального анализа производительности, что делает его идеальной средой для решения задач данного исследования.

3 ТЕОРЕТИЧЕСКОЕ ОБОСНОВАНИЕ РАЗРАБОТКИ ПРОГРАММНОГО ПРОДУКТА

3.1 Обоснование необходимости разработки

Для объективной оценки производительности современных центральных процессоров недостаточно простого сопоставления их формальных технических характеристик. Различия в архитектуре, объеме кэш-памяти и реализации технологий многопоточности требуют проведения практических испытаний в реалистичных сценариях. В связи с этим для детального и объективного сравнительного анализа производительности двух популярных мобильных процессоров, *Intel Core i5-12450H* и *AMD Ryzen 7 5800H*, необходимо использовать специализированное программное обеспечение. Стандартные бенчмарки, давая общее представление, не позволяют углубленно изучить поведение процессоров в специфических, вычислительно интенсивных задачах.

Для этих целей в качестве тестовой нагрузки был выбран алгоритм быстрого преобразования Фурье (БПФ). БПФ является фундаментальной операцией, находящей широкое применение в цифровой обработке сигналов, анализе изображений, решении дифференциальных уравнений и многих других научных и инженерных областях. Эффективность его выполнения напрямую зависит от ключевых архитектурных особенностей процессора: скорости выполнения операций с плавающей запятой, пропускной способности подсистемы памяти и эффективности работы кэша. Это делает БПФ идеальным инструментом для выявления сильных и слабых сторон процессорных архитектур.

Разработка собственного приложения позволяет создать полностью контролируемую и воспроизводимую среду для тестирования, свободную от влияния стороннего программного обеспечения. Программа предоставит возможность точно контролировать количество вычислительных потоков и привязывать их к конкретным физическим ядрам (*CPU affinity*). Такой гранулярный контроль имеет ключевое значение для данного исследования, поскольку позволит детально изучить масштабируемость производительности, сравнить эффективность технологии *SMT* от *AMD* и гибридного подхода *Intel*, а также изолированно измерить производительность производительных (*P-cores*) и энергоэффективных (*E-cores*) ядер.

Таким образом, создание специализированного программного продукта является не просто технической задачей, а необходимым методологическим инструментом. Он позволит получить глубокие и детализированные данные, необходимые для того, чтобы выйти за рамки поверхностного сравнения и сделать обоснованные выводы об архитектурных преимуществах и недостатках исследуемых процессоров в контексте параллельных научных вычислений.

3.2 Технологии программирования, используемые для решения поставленных задач

Для реализации программного продукта, направленного на исследование производительности центральных процессоров в вычислительно интенсивных задачах, выбран язык программирования *C++* с использованием стандарта *C++20* [8].

Данный выбор обоснован ключевыми факторами, связанными с эффективностью и возможностями, необходимыми для реализации алгоритма быстрого преобразования Фурье (*БПФ*). *C++* обеспечивает высочайшую производительность благодаря генерации кода, близкого к машинным инструкциям, и отсутствию фоновых процессов, таких как сборка мусора, что позволяет проводить «чистые» замеры производительности процессора. Гибкость управления памятью в *C++* также играет ключевую роль. Язык позволяет организовывать данные в памяти таким образом, чтобы обеспечить их локальность и последовательный доступ. Для алгоритмов, подобных *БПФ*, где производятся многочисленные «баттерфляй»-преобразования, такая оптимизация работы с кэш-памятью процессора является определяющим фактором для достижения высокой производительности.

Важной частью технологического стека является инструментарий для сборки и компиляции. В качестве системы сборки используется кросс-платформенный инструмент *CMake* [9], который автоматизирует процесс компиляции исходного кода в исполняемый файл. *CMake* обеспечивает переносимость проекта между различными операционными системами и средами разработки, а также управляет зависимостями, такими как тестовый фреймворк *Google Test*, что гарантирует воспроизводимость результатов. Исходный код преобразуется в машинный с помощью одного из современных оптимизирующих компиляторов, например, *GCC* [10]. Этот компилятор производит глубокий анализ кода и применяет множество техник оптимизации для генерации максимально эффективного исполняемого файла, что напрямую влияет на точность и достоверность полученных в ходе тестирования данных.

Для реализации ключевых аспектов программы были задействованы специализированные стандартные библиотеки *C++*.

Библиотека *<thread>* [11] предоставляет набор инструментов для создания и управления потоками выполнения в рамках одного процесса. В данном проекте она является основой для реализации многопоточного режима тестирования. С ее помощью основная вычислительная задача распределяется между несколькими потоками, которые операционная система может исполнять параллельно на разных физических или логических ядрах процессора. Это позволяет в полной мере загрузить многоядерные архитектуры и собрать данные об эффективности параллельной обработки и масштабируемости производительности исследуемых процессоров.

Библиотека *<chrono>* [12] используется для точного измерения временных интервалов. Она предоставляет доступ к различным системным часам, включая *std::chrono::high_resolution_clock*, которые обеспечивают максимально возможную точность измерений. В контексте тестирования

производительности, где время выполнения операций может составлять миллисекунды или даже микросекунды, использование часов высокого разрешения является обязательным условием для получения достоверных и воспроизводимых результатов. Данная библиотека позволяет с высокой точностью фиксировать время начала и окончания вычислений, что является основой для всех последующих оценок производительности.

Таким образом, выбор языка *C++* в сочетании с современной системой сборки *CMake* и мощными стандартными библиотеками, такими как <*thread*> и <*chrono*>, формирует надежную технологическую основу. Этот стек технологий позволяет разработать специализированное программное обеспечение, способное с высокой точностью и эффективностью проводить измерения производительности, что является необходимым условием для объективного сравнительного анализа процессорных архитектур.

3.3 Связь архитектуры вычислительной системы с разрабатываемым программным обеспечением

Архитектурные особенности выбранных процессоров оказывают прямое и определяющее влияние на проектирование и реализацию программного обеспечения. Понимание этих взаимосвязей является ключевым для создания эффективного тестового приложения, способного в полной мере раскрыть потенциал каждой архитектуры и выявить их характерные особенности.

Разработанное приложение предоставляет возможность изолированного тестирования как производительных (*P-cores*), так и энергоэффективных (*E-cores*) ядер на *Intel Core i5-12450H*, позволяя оценить их вклад в общую производительность и раскрыть эффективность аппаратного планировщика *Intel Thread Director*. Для *AMD Ryzen 7 5800H* с его симметричной структурой из восьми одинаковых ядер программа позволяет тестирование с произвольным количеством рабочих потоков, что обеспечивает оценку линейности масштабирования производительности.

Реализация алгоритма *БПФ* в контексте данных архитектур требует особого внимания к организации работы с данными. Объём кэш-памяти различных уровней напрямую влияет на эффективность алгоритма: *AMD Ryzen 7 5800H* обладает 16 МБ объединённого кэша *L3*, в то время как *Intel Core i5-12450H* — 12 МБ. Для алгоритмов, оперирующих большими объёмами данных и характеризующихся сложными паттернами доступа к памяти, эта разница может быть значимой. Правильная организация циклов, выравнивание данных в памяти и минимизация конфликтов кэша позволяют снизить количество промахов кэша и повысить пропускную способность доступа к памяти на обеих платформах.

Флаги компиляции *-O3* и *-march=native* являются решающими для достижения максимальной производительности. Флаг *-march=native* инструктирует компилятор *GCC* использовать все доступные инструкции целевого процессора, включая расширения *SIMD*. Для *AMD Ryzen 7 5800H* это означает использование инструкций *AVX2*, а для *Intel Core i5-12450H* —

как *AVX2*, так и потенциально расширенные наборы инструкций, доступные на платформе *Alder Lake*. Автоматическая векторизация циклов компилятором позволяет обрабатывать несколько элементов данных в рамках одной инструкции, что существенно ускоряет выполнение барочка-трансформаций в *БПФ*.

Микроархитектурные различия учитываются при проектировании приложения. *Zen 3* с её фокусом на повышение *IPC* (инструкций за такт) лучше справляется с задачами, требующими выполнения большого количества независимых инструкций в единицу времени. Барочка-трансформация в *БПФ* содержит множество независимых арифметических операций, что делает эту архитектуру особенно подходящей для данного алгоритма. Архитектура *Alder Lake* ориентирована на оптимизацию однопоточной производительности производительных ядер, что требует учёта при распределении нагрузки между различными типами ядер.

Разработанное приложение включает возможность запуска алгоритма с переменным количеством потоков, привязку потоков к конкретным физическим ядрам через механизм *CPU affinity*, использование современных инструкций *SIMD* для векторизации операций над комплексными числами и минимизацию системных издержек за счёт минимальной конфигурации *Arch Linux*. Такой подход позволяет создать инструмент, способный максимально объективно и детально оценить производительность исследуемых архитектур в контексте вычислительно интенсивной задачи, не искажённую влиянием системных процессов или неправильной оптимизации компилятора.

4 ПРОЕКТИРОВАНИЕ ФУНКЦИОНАЛЬНЫХ ВОЗМОЖНОСТЕЙ ПРОГРАММЫ

4.1 Общая цель программы

Целью разработки программного обеспечения является создание приложения для сравнительного анализа производительности двух современных мобильных процессоров — *AMD Ryzen 7 5800H* и *Intel Core i5-12450H* — при выполнении параллельных вычислений. Программа позволяет исследовать влияние архитектурных различий, в частности различных моделей многопоточности и организации кэш-памяти, на производительность многоядерных систем. Разработанное приложение предоставляет объективные данные о производительности каждого процессора в различных сценариях распараллеливания, что позволяет сделать обоснованные выводы об эффективности каждой архитектуры.

4.2 Теоретические сведения о алгоритме быстрого преобразования Фурье

Быстрое преобразование Фурье (БПФ) — это алгоритм, который вычисляет дискретное преобразование Фурье (ДПФ) последовательности за время $O(n \log n)$ вместо наивного подхода $O(n^2)$ [13]. Дискретное преобразование Фурье преобразует последовательность комплексных чисел $x_0, x_1, \ldots, x_{\{n-1\}}$ в последовательность $X_0, X_1, \ldots, X_{\{n-1\}}$ согласно формуле:

$$X_k = \sum_{j=0}^{n-1} x_j e^{\frac{-2\pi ijk}{n}}, \tag{4.1}$$

где X_k – спектральный коэффициент преобразования Фурье для частоты k;
x_j – исходный элемент последовательности с индексом j;
n – длина последовательности;
k – индекс спектрального компонента, где $k = 0, 1, \ldots, n - 1$;
i – мнимая единица.

Существует два основных подхода к реализации алгоритма *БПФ*: рекурсивный и итеративный. Рекурсивный подход, известный как алгоритм Кули-Тьюки, основан на разложении исходной последовательности на две подпоследовательности (чётные и нечётные индексы), что приводит к следующему соотношению:

$$X_k = \sum_{j=0}^{\frac{n}{2}-1} x_{2j} e^{\frac{-4\pi ijk}{n}} + e^{\frac{-2\pi ik}{n}} \sum_{j=0}^{\frac{n}{2}-1} x_{2j+1} e^{\frac{-4\pi ijk}{n}}, \tag{4.2}$$

где X_k – спектральный коэффициент преобразования Фурье;
x_{2j} – элементы последовательности с чётными индексами;

x_{2j+1} – элементы последовательности с нечётными индексами;
$n/2$ – размер подпоследовательностей;
$e^{-2\pi ik/n}$ – поворачивающий множитель (*twiddle factor*).

Однако в данной работе использован итеративный подход, который реализует тот же принцип разложения Кули-Тьюки, но без рекурсивных вызовов. Итеративный алгоритм начинает с перестановки входных данных согласно битовой инверсии индексов, а затем выполняет логарифмическое количество проходов ($\log_2(n)$ проходов). На каждом проходе применяются бабочковые операции (*butterfly operation*) с различными поворачивающими множителями W. Такой подход избегает рекурсивных вызовов, снижает издержки на управление стеком вызовов и повышает локальность доступа к данным, что способствует лучшему использованию кэш-памяти процессора.

Ключевой операцией в итеративном алгоритме является бабочковая операция, в которой два входных элемента a и b комбинируются с поворачивающим множителем W по следующей схеме: первый выходной элемент равен $a + Wb$, а второй равен $a - Wb$. Эта операция выполняется миллионы раз во время вычисления и является основной нагрузкой на процессор. Такая структура операций позволяет алгоритму содержать на каждом проходе операции с независимыми данными, что особенно хорошо подходит для параллельного выполнения на многоядерных архитектурах.

Благодаря своему широкому применению в обработке сигналов, анализе изображений и решении дифференциальных уравнений, эффективность реализации БПФ напрямую влияет на производительность многих научных и инженерных приложений. Именно поэтому БПФ выбран в качестве тестовой нагрузки: алгоритм чувствителен к архитектурным особенностям процессора, таким как размер кэш-памяти, ширина пропускной способности памяти и эффективность параллелизма, что позволяет провести глубокий анализ различий между архитектурами *AMD Ryzen 7 5800H* и *Intel Core i5-12450H*.

4.3 Описание функциональных возможностей

Разработанное приложение предоставляет комплексный набор инструментов для проведения детального анализа производительности многоядерных процессоров при выполнении алгоритма быстрого преобразования Фурье. Программа позволяет исследовать влияние архитектурных различий на эффективность параллельных вычислений и выявить закономерности масштабируемости производительности в зависимости от количества активных рабочих потоков.

Приложение реализует следующий функциональный набор:

1 Генерация входных данных. Создается синтетический набор комплексных чисел размером 2^n, где n варьируется в диапазоне от 5 до 24, что позволяет исследовать производительность на различных масштабах данных от нескольких килобайт до нескольких мегабайт.

2 Реализация итеративного алгоритма БПФ. Вычисление быстрого преобразования Фурье выполняется с использованием итеративного подхода

Кули-Тьюки, который избегает рекурсивных вызовов и обеспечивает лучшее использование кэш-памяти процессора.

3 Многопоточное выполнение. Программа поддерживает произвольное количество рабочих потоков от 1 до максимально доступного на платформе, с возможностью динамического масштабирования параллелизма и анализа зависимости производительности от степени параллелизации.

4 Привязка потоков к ядрам. Осуществляется управление сродством потоков к физическим ядрам процессора через системный вызов *sched_setaffinity*, что исключает влияние планировщика операционной системы и обеспечивает воспроизводимость и детерминированность результатов.

5 Изолированное тестирование ядер. На архитектуре *Intel Core i5-12450H* реализована возможность раздельного анализа производительных (P-cores) и энергоэффективных (E-cores) ядер без их взаимного влияния, что позволяет оценить архитектурные различия.

6 Точное измерение времени. Для регистрации времени выполнения используются высокоточные системные часы с разрешением в микросекунды через библиотеку *std::chrono*, что обеспечивает достоверность и точность полученных метрик.

7 Сохранение результатов. Результаты измерений сохраняются в формате *CSV*, содержащем информацию о количестве потоков, размере входных данных, времени выполнения и вычисленной пропускной способности, что упрощает последующий анализ и визуализацию данных.

8 Верификация корректности. Проводится тестирование правильности вычисления преобразования Фурье через сравнение результатов итеративного алгоритма с эталонной реализацией, что обеспечивает надёжность измеряемых результатов.

Таким образом, разработанное приложение представляет собой полнофункциональный инструмент для объективного сравнения архитектур *AMD Ryzen 7 5800H* и *Intel Core i5-12450H* в контексте вычислительно интенсивных задач, обеспечивающий надёжность и воспроизводимость результатов.

4.4 Описание функциональной схемы алгоритма

Функциональная схема алгоритма БПФ представлена в приложении В и отражает последовательность основных этапов работы программы. Алгоритм состоит из пяти функциональных блоков, каждый из которых выполняет определённую задачу в процессе тестирования производительности.

Работа программы начинается с блока инициализации параметров алгоритма. На этом этапе выполняется разбор аргументов командной строки, определяется размер входных данных (степень двойки n), количество рабочих потоков и режим привязки к ядрам процессора. Также инициализируются структуры данных для хранения результатов измерений и настраиваются параметры многопоточной среды выполнения.

Следующим этапом является выбор режима выполнения. Программа определяет, какой сценарий тестирования будет использован: последовательное выполнение на одном ядре либо параллельное выполнение с заданным количеством потоков. На основе выбранного режима настраивается политика распределения потоков по физическим ядрам процессора с учётом архитектурных особенностей целевой платформы.

Блок генерации тестовых значений отвечает за создание входного массива комплексных чисел. Формируется синтетический набор данных размером $2n$ элементов, где каждый элемент представляет собой комплексное число с действительной и мнимой частями. Генерация выполняется с использованием детерминированного алгоритма, что обеспечивает воспроизводимость результатов при повторных запусках программы.

Центральным этапом является выполнение алгоритмов быстрого преобразования Фурье. В этом блоке реализуется итеративный алгоритм Кули-Тьюки с распараллеливанием бабочковых операций между рабочими потоками. Каждый поток обрабатывает независимый диапазон индексов, а синхронизация между проходами алгоритма осуществляется с помощью барьеров. Одновременно с вычислениями производится измерение времени выполнения с использованием высокоточных системных часов.

Завершающий блок выполняет сохранение результатов и освобождение ресурсов. Результаты измерений, включающие время выполнения, количество использованных потоков и размер обработанных данных, записываются в файл формата CSV. После этого освобождается выделенная память, завершаются рабочие потоки и программа корректно завершает своё выполнение.

4.5 Описание блок схемы алгоритма

Блок-схема алгоритма программного средства представлена в приложении Г и состоит из трёх взаимосвязанных частей: главного алгоритма, алгоритма однопоточного БПФ и алгоритма многопоточного БПФ.

Главный алгоритм определяет общую логику работы программы. После запуска выполняется получение набора входных данных, представляющего собой массив комплексных чисел заданного размера. Далее осуществляется проверка выбранного режима вычислений. Если выбран однопоточный режим, управление передаётся соответствующей подпрограмме. В случае многопоточного режима предварительно определяется количество потоков для вычислений, после чего вызывается подпрограмма многопоточного БПФ. После завершения вычислений результаты тестирования сохраняются в файл формата *CSV*.

Подпрограмма однопоточного алгоритма БПФ реализует итеративный подход Кули-Тьюки. Внешний цикл обеспечивает проход по заданным размерам входных данных для проведения серии измерений. Для каждого размера сначала выполняется бит-реверсивная перестановка элементов массива, которая подготавливает данные для последующих вычислений. Затем начинается основной вычислительный процесс, организованный в виде трёх вложенных циклов. Внешний цикл проходит по стадиям вычислений,

количество которых равно $log_2(n)$. На каждой стадии определяется размер групп, после чего выполняется проход по всем группам в массиве. Внутренний цикл обрабатывает пары элементов внутри каждой группы, выполняя для них вычисление поворотного множителя и операцию «бабочка», которая заключается в вычислении суммы и разности двух элементов с учётом поворотного множителя.

Подпрограмма многопоточного алгоритма БПФ расширяет однопоточную версию механизмами параллельного выполнения. После входа в цикл по размерам данных создаётся и запускается заданное количество рабочих потоков. Бит-реверсивная перестановка выполняется параллельно, после чего происходит синхронизация потоков для обеспечения корректности последующих вычислений. На каждой стадии алгоритма применяется адаптивная стратегия распараллеливания. Если количество групп на текущей стадии больше или равно количеству потоков, используется крупнозернистый параллелизм — каждый поток обрабатывает свою часть групп независимо. В противном случае применяется мелкозернистый параллелизм, при котором потоки совместно обрабатывают пары элементов внутри одной группы. Такой подход обеспечивает эффективную загрузку всех потоков на любой стадии вычислений. После завершения каждой стадии выполняется синхронизация потоков с помощью барьера, что гарантирует завершение всех операций перед переходом к следующей стадии. По окончании всех стадий рабочие потоки завершают свою работу.

5 СРАВНЕНИЕ ПРОИЗВОДИТЕЛЬНОСТИ ПРОЦЕССОРОВ

5.1 Общая структура программы

Программа представляет собой систему для сравнительного анализа производительности итеративного алгоритма быстрого преобразования Фурье (БПФ) по основанию 2, реализованную для выполнения вычислений как на центральном (*CPU*), так и на графическом (*GPU*) процессоре. Архитектура проекта является модульной, что обеспечивается системой сборки *CMake*, и разделена на несколько независимых исполняемых файлов: основной бенчмарк для *CPU* и бенчмарк для *GPU* на базе *OpenCL*.

Центральным компонентом является класс *benchmark*, инкапсулирующий логику тестирования на *CPU*. Он содержит реализации как однопоточной (*fft_iterative*), так и многопоточной (*fft_iterative_multithreaded*) версий алгоритма. Многопоточная реализация построена с использованием нативных средств языка: для создания параллельных потоков вычислений применяются *std::thread*, а для их синхронизации между стадиями БПФ — *std::barrier*.

Тестирование производительности выполняется на прогрессивном наборе из 21 размера входных данных, от 32 до 33,554,432 элементов. Для каждого запуска генерируется массив комплексных чисел (*std::vector<std::complex<double>>*) с использованием генератора псевдослучайных чисел *std::mt19937*. Для обеспечения полной воспроизводимости тестов генератор инициализируется фиксированным значением. Точность измерений времени выполнения гарантируется использованием высокоточных часов *std::chrono::high_resolution_clock*.

Модуль *GPU*-вычислений представляет собой отдельную программу, использующую технологию *OpenCL* для выполнения БПФ на графическом процессоре. Основная логика вычислений вынесена в ядро *fft_kernel.cl*, которое выполняет одну стадию преобразования. Управляющий код на *C++* отвечает за инициализацию *OpenCL*, передачу данных на устройство, последовательный вызов ядра для каждой стадии алгоритма и сбор результатов.

Управление режимами работы программы осуществляется через аргументы командной строки. Флаги *--mode* и *--threads* позволяют запускать однопоточную или многопоточную версию с заданным количеством потоков. Результаты каждого теста, содержащие размер входных данных и время выполнения в миллисекундах, сохраняются в структурированный *CSV*-файл для последующего анализа и построения графиков.

5.2 Подготовка к тестированию

Для проведения сравнительного анализа производительности процессоров *AMD Ryzen 7 5800H* и *Intel Core i5-12450H* была сформирована

стандартизированная тестовая среда. В качестве входных данных для каждого запуска генерировался массив комплексных чисел заданного размера. Для обеспечения воспроизводимости результатов генерация случайных чисел выполнялась с фиксированным начальным значением.

Ключевым аспектом подготовки стало управление средой исполнения. Тесты проводились в нескольких конфигурациях: для одного ядра, а также для 4 и 8 физических ядер. Каждый из этих сценариев был выполнен дважды: с включенной и принудительно отключенной технологией одновременной многопоточности (*SMT* для *AMD*, *Hyper-Threading* для *Intel*).

Чтобы гарантировать выполнение потоков на строго определенных физических ядрах и исключить влияние системного планировщика, использовалась утилита *taskset*. Стратегия привязки потоков (*CPU affinity*) зависела от режима *SMT/HT*. При включенной технологии многопоточности потоки привязывались к последовательному набору логических процессоров (0, 1, 2, …). При отключенной технологии *SMT/HT* применялась специальная стратегия для работы с физическими ядрами: для процессора *AMD* потоки привязывались к ядрам с четными номерами (0, 2, 4, …), а для гибридной архитектуры *Intel* потоки сначала занимали все доступные производительные *P*-ядра (0, 2, 4, 6), и только после их исчерпания назначались на энергоэффективные *E*-ядра (8, 9, 10, 11).

Для сбора данных о производительности использовался набор стандартных утилит. Основные аппаратные метрики собирались с помощью инструмента *perf stat* по следующим событиям: *cycles*, *instructions*, *cache-references*, *cache-misses*, *branch-instructions* и *branch-misses*. Эти данные позволили в дальнейшем рассчитать такие показатели, как *IPC* (инструкций за такт) и процент промахов. Одновременно с этим, утилита *mpstat* использовалась для фонового мониторинга и сбора данных о нагрузке на каждое логическое ядро процессора, что позволяло верифицировать корректность распределения нагрузки в ходе теста.

5.3 Результаты тестирования с включенным SMT/Hyper-Threading

Анализ результатов тестирования процессора *AMD Ryzen 7 5800H* позволяет выявить характерные закономерности масштабирования производительности при параллельном выполнении алгоритма быстрого преобразования Фурье. На малых размерах входных данных однопоточное выполнение демонстрирует значительное преимущество перед многопоточными конфигурациями. Это объясняется природой накладных расходов на организацию параллельных вычислений: создание потоков, распределение работы и синхронизация на барьерах требуют фиксированного времени, которое не зависит от объёма полезных вычислений.

По мере увеличения размера входного массива соотношение между полезными вычислениями и накладными расходами изменяется в пользу первых. Использование четырёх ядер начинает приносить выигрыш раньше, чем использование восьми, поскольку меньшее количество потоков требует меньших затрат на синхронизацию. На очень больших размерах данных,

превышающих объём кэш-памяти третьего уровня, разница между четырьмя и восемью ядрами становится минимальной, что свидетельствует о достижении предела пропускной способности подсистемы памяти. Результаты измерений времени выполнения представлены в таблице 5.1.

Таблица 5.1 – Замеры времени выполнения на *AMD Ryzen 7 5800H*

Размер	Одно ядро	4 ядра	8 ядер
32	0.009858	0.482569	1.0882
64	0.001393	0.378853	0.564479
128	0.002305	0.317166	0.517923
256	0.004418	0.307974	0.542613
512	0.009889	0.347999	0.557287
1024	0.034955	0.378583	0.650579
2048	0.040055	0.330224	0.570899
4096	0.104135	0.321323	0.591723
8192	0.189985	0.374917	0.713431
16384	0.412492	0.434351	0.710907
32768	0.976186	0.571204	0.721745
65536	2.18888	0.89526	0.960985
131072	4.65683	1.38313	2.19718
262144	9.88293	2.52728	2.03694
524288	20.8727	4.38266	3.32503
1048576	47.7608	10.7941	7.11852
2097152	155.939	77.1275	70.2616
4194304	365.071	188.538	182.289
8388608	698.916	437.435	437.477
16777216	1427.81	918.334	895.01
33554432	3054.66	1973.62	1925.17

Метрики производительности раскрывают внутренние причины наблюдаемого поведения. Показатель *IPC* существенно снижается при переходе к многопоточному режиму, что связано с возросшей нагрузкой на подсистему памяти и работой протокола когерентности кэшей. Процент промахов кэша демонстрирует резкий рост при увеличении количества активных ядер: все ядра разделяют общий кэш *L3*, что приводит к взаимному вытеснению данных. Низкий процент промахов ветвлений свидетельствует о предсказуемой структуре алгоритма БПФ. Ключевые метрики производительности, полученные с помощью *perf*, сведены в таблицу 5.2.

Таблица 5.2 – Замеры метрик на *AMD Ryzen 7 5800H*

Метрика	Одно ядро	4 ядра	8 ядер
IPC	1.62	0.37	0.21
Промахи кэша (%)	5.98	36.49	39.39
Промахи ветвлений (%)	0.13	0.14	0.15

Результаты тестирования процессора *Intel Core i5-12450H* демонстрируют существенно отличающуюся картину, обусловленную гибридной архитектурой *Alder Lake* На малых размерах данных Intel показывает схожее с *AMD* поведение, однако на больших объёмах демонстрирует значительно лучшую масштабируемость — при максимальном размере входного массива восьмиядерная конфигурация *Intel* опережает аналогичную конфигурацию *AMD* более чем в два раза.

На некоторых промежуточных размерах данных конфигурация с четырьмя ядрами показывает лучшие результаты, чем с восемью. Это объясняется тем, что при использовании четырёх потоков все они размещаются на производительных *P* ядрах, тогда как при восьми потоках часть попадает на менее производительные *E* ядра, замедляя общее выполнение из-за ожидания на барьерах синхронизации. Детальные замеры времени выполнения для процессора *Intel* приведены в таблице 5.3.

Таблица 5.3 – Замеры времени выполнения на *Intel Core i5-12450H*

Размер	Одно ядро	4 ядра	8 ядер
32	0.010597	0.491682	1.25722
64	0.002319	0.256993	1.03671
128	0.003945	0.225747	0.833765
256	0.007881	0.255611	0.715128
512	0.016516	0.315792	0.599665
1024	0.035551	0.342733	0.524364
2048	0.081465	0.39702	0.700515
4096	0.184372	0.409595	0.604209
8192	0.380837	0.462682	0.603044
16384	0.687353	0.551071	0.728383
32768	1.08981	0.652562	0.953188
65536	3.06556	0.940648	1.66175
131072	4.48471	2.14488	4.0922
262144	9.49335	3.1405	5.66165

Продолжение таблицы 5.3

Размер	Одно ядро	4 ядра	8 ядер
524288	21.2841	6.81683	11.6393
1048576	60.7808	15.9238	16.1062
2097152	140.678	43.0497	40.349
4194304	292.681	101.034	85.3587
8388608	618.562	208.171	175.443
16777216	1274.66	445.21	377.736
33554432	2630.61	909.939	783.953

Метрики *Intel* представлены с разделением на *P* ядра и *E* ядра. Производительные ядра демонстрируют исключительно высокий *IPC*, значительно превосходящий результаты *AMD*, что свидетельствует о более широких возможностях суперскалярного выполнения в микроархитектуре *Golden Cove*. Энергоэффективные ядра показывают существенно более скромные результаты, соответствующие их проектному назначению — оптимизации для фоновых задач с минимальным энергопотреблением.

Высокий процент промахов кэша на *Intel* по сравнению с *AMD* может объясняться различиями в методиках подсчёта событий производительности между архитектурами, иной организацией кэш-иерархии, а также агрессивным механизмом предвыборки данных. Аномальный скачок промахов ветвлений на *E* ядрах указывает на особенности предсказателя ветвлений в микроархитектуре *Gracemont*. Сводные данные по метрикам производительности для *P* и *E* ядер представлены в таблице 5.4.

Таблица 5.4 – Замеры метрик на *Intel Core i5-12450H*

Метрика	Одно ядро	4 ядра	8 ядер
IPC	8.01 / 2.45	3.43 / 0.32	4.48 / 0.81
Промахи кэша (%)	75.65 / 68.59	44.40 / 61.93	49.53 / 62.25
Промахи ветвлений (%)	0.10 / 1.09	0.14 / 8.31	0.13 / 0.16

Сравнение результатов двух процессоров при включенных технологиях многопоточности выявляет ряд закономерностей. В однопоточном режиме *AMD* демонстрирует преимущество на малых и средних размерах данных благодаря меньшим накладным расходам, однако на больших массивах *Intel* показывает сопоставимые или лучшие результаты. При использовании четырёх ядер процессоры демонстрируют схожую производительность на малых данных, но *Intel* значительно опережает *AMD* на больших размерах. Наиболее существенное различие наблюдается в восьмиядерной

конфигурации: *Intel* превосходит *AMD* более чем вдвое на максимальных размерах массивов, что объясняется эффективным использованием *Hyper-Threading* на производительных ядрах. При этом *AMD* демонстрирует значительно лучшую эффективность работы с кэш-памятью — процент промахов кэша на *AMD* в несколько раз ниже, чем на *Intel*, что свидетельствует о более оптимальной организации подсистемы памяти архитектуры *Zen 3*.

5.4 Результаты тестирования с выключенным SMT/Hyper-Threading

Результаты AMD Ryzen 7 5800H без SMT демонстрируют улучшение однопоточной производительности на малых размерах данных — при отключенном SMT исключается интерференция с фоновыми задачами, которые планировщик мог размещать на том же физическом ядре. Многопоточная производительность изменяется неоднозначно: на средних размерах наблюдается улучшение, тогда как на очень больших результаты остаются сопоставимыми, что подтверждает гипотезу об ограничении пропускной способностью памяти. Результаты измерений времени выполнения с отключенной технологией *SMT* представлены в таблице 5.5.

Таблица 5.5 – Замеры времени выполнения на *AMD Ryzen 7 5800H*

Размер	Одно ядро	4 ядра	8 ядер
32	0.007954	0.454898	0.622964
64	0.001022	0.157984	0.546454
128	0.001613	0.195502	0.31052
256	0.002925	0.212372	0.288239
512	0.00614	0.249379	0.299651
1024	0.013242	0.187417	0.821468
2048	0.036031	0.186556	0.501861
4096	0.062546	0.213815	0.368627
8192	0.13561	0.249489	0.37026
16384	0.293567	0.317211	0.451349
32768	0.67833	0.511159	0.554569
65536	1.58606	0.96691	0.804016
131072	3.45899	1.59716	1.54787
262144	6.94271	4.83196	3.51917
524288	15.8176	5.63401	5.2375
1048576	40.1054	17.033	9.12598
2097152	135.52	95.2771	73.1935

Продолжение таблицы 5.5

Размер	Одно ядро	4 ядра	8 ядер
4194304	319.344	238.415	204.893
8388608	695.518	443.021	437.013
16777216	1426.43	983.963	1039.93
33554432	3061.06	2069.51	2053.05

Метрики *AMD* без *SMT* показывают существенное улучшение *IPC* в многопоточном режиме и снижение процента промахов кэша. Отключение *SMT* устраняет конкуренцию за исполнительные устройства, декодеры инструкций и кэш первого уровня, позволяя каждому потоку использовать все ресурсы ядра монопольно. Соответствующие изменения в метриках производительности показаны в таблице 5.6.

Таблица 5.6 – Замеры метрик на *AMD Ryzen 7 5800H*

Метрика	Одно ядро	4 ядра	8 ядер
IPC	1.59	0.69	0.41
Промахи кэша (%)	6.02	28.29	37.74
Промахи ветвлений (%)	0.13	0.14	0.14

Результаты *Intel Core i5-12450H* без *Hyper-Threading* демонстрируют иную динамику. Важно отметить, что *Hyper-Threading* доступен только на *P* ядрах, тогда как *E* ядра изначально не поддерживают эту технологию. При отключенном *Hyper-Threading* наблюдается ухудшение производительности на больших данных: восьмипоточная конфигурация вынуждена использовать все *P* и *E* ядра, что приводит к гетерогенному выполнению. Производительные ядра завершают работу быстрее и простаивают на барьерах синхронизации, ожидая более медленные *E* ядра. Данные по времени выполнения для этого сценария приведены в таблице 5.7.

Таблица 5.7 – Замеры времени выполнения на *Intel Core i5-12450H*

Размер	Одно ядро	4 ядра	8 ядер
32	0.010997	0.813273	0.870665
64	0.00199	0.352655	0.625475
128	0.003755	0.290983	0.49094
256	0.007266	0.317632	0.622206
512	0.014694	0.394683	0.415895

Продолжение таблицы 5.7

Размер	Одно ядро	4 ядра	8 ядер
1024	0.03285	0.368615	1.27804
2048	0.076268	0.330625	0.664094
4096	0.147757	0.39088	0.524475
8192	0.324351	0.527037	0.584163
16384	0.727194	0.923695	0.996862
32768	1.71755	1.32323	1.05719
65536	2.48678	2.34384	2.90096
131072	4.21892	2.34555	5.11337
262144	8.88698	4.13068	6.239
524288	20.0271	7.94803	12.6238
1048576	61.6581	24.4118	18.4065
2097152	141.617	56.57	48.4533
4194304	307.985	126.102	100.53
8388608	642.509	267.714	208.784
16777216	1346.7	544.755	444
33554432	2784.31	1168.18	932.592

Метрики *Intel* без *Hyper-Threading* показывают существенный рост *IPC* для *P* ядрер, подтверждая эффективность монопольного использования ресурсов ядра. *E* ядра демонстрируют *IPC* около единицы, что отражает архитектурные ограничения микроархитектуры *Gracemont*. Высокий процент промахов кэша на обоих типах ядер указывает на подсистему памяти как ключевой ограничивающий фактор. Сводка по метрикам производительности для режима с отключенным *Hyper-Threading* находится в таблице 5.8.

Таблица 5.8 – Замеры метрик на *Intel Core i5-12450H*

Метрика	Одно ядро	4 ядра	8 ядер
IPC	1.82 / -	5.41 / -	7.29 / 1.00
Промахи кэша (%)	69.54 / -	51.18 / -	66.53 / 75.47
Промахи ветвлений (%)	0.10 / -	0.12 / -	0.11 / 0.17

Сравнение результатов двух процессоров при отключенных технологиях многопоточности демонстрирует изменение баланса сил между платформами. В однопоточном режиме *AMD* сохраняет преимущество на малых размерах данных, однако разрыв сокращается по сравнению с режимом с включенным

SMT. При использовании четырёх ядер *Intel* показывает худшие результаты на малых данных из-за больших накладных расходов, но на средних и больших размерах демонстрирует преимущество благодаря более высокой производительности *P* ядер.

Наиболее интересная картина наблюдается в восьмиядерной конфигурации. *AMD* показывает стабильные результаты, сопоставимые с режимом с включенным *SMT*, тогда как *Intel* демонстрирует заметное падение производительности по сравнению с режимом с *Hyper-Threading*. Это объясняется тем, что при отключенном *Hyper-Threading Intel* вынужден задействовать менее производительные *E* ядра, которые становятся узким местом системы. Тем не менее, на больших размерах данных *Intel* по-прежнему опережает *AMD*, хотя разрыв существенно сокращается — примерно до двукратного преимущества вместо более чем двукратного при включенном *Hyper-Threading*.

По метрикам производительности отключение технологий многопоточности положительно сказывается на обеих платформах: *IPC* возрастает, а процент промахов кэша снижается. Однако *AMD* получает больший относительный выигрыш от отключения *SMT*, тогда как *Intel* теряет в абсолютной производительности из-за необходимости использования гетерогенных ядер.

5.5 Результаты тестирования на iGPU

Сравнение производительности встроенных графических ускорителей (*iGPU*) процессоров *AMD* и *Intel* на задаче БПФ выявляет сложную и нелинейную зависимость производительности от размера входных данных. Сравнительные результаты тестирования представлены в таблице 5.9.

Таблица 5.9 – Замеры времени выполнения на *iGPU*

Размер	*AMD Radeon RX Vega 8*	*Intel Iris Xe Graphics*
32	0.0111	0.032654
64	0.009496	0.037497
128	0.01026	0.034422
256	0.045364	0.039265
512	0.051054	0.054421
1024	0.018754	0.049995
2048	0.02192	0.056352
4096	0.024006	0.065255
8192	0.033823	0.082076
16384	0.057185	0.098327
32768	0.540957	0.137391

Продолжение таблицы 5.9

Размер	*AMD Radeon RX Vega 8*	*Intel Iris Xe Graphics*
65536	0.641145	0.279056
131072	0.883801	0.42114
262144	2.28955	0.743377
524288	4.03693	1.72015
1048576	8.58995	4.59515
2097152	17.7182	22.2319
4194304	36.9868	42.414
8388608	78.3595	86.3744
16777216	162.237	118.307

Анализ результатов показывает, что ни одно из интегрированных решений не имеет абсолютного преимущества. На малых размерах данных (до 16К) *iGPU* от *AMD* демонстрирует более низкое время выполнения. Это может быть связано с меньшими накладными расходами на запуск *OpenCL* ядра и передачу данных.

Однако при переходе к средним размерам (от 32К до 1М) наблюдается инверсия производительности: *Intel Iris Xe* показывает значительно лучшие результаты, опережая конкурента в полтора раза. Вероятно, на этих размерах данных архитектура *Intel* с большим количеством исполнительных блоков (*Execution Units*) и более эффективной работой с локальной памятью обеспечивает существенный прирост производительности.

При дальнейшем увеличении массива данных (от 2М до 8М) *AMD* вновь выходит вперед. Это может указывать на то, что при работе с объёмами, значительно превышающими кэши *GPU*, узким местом становится пропускная способность системной памяти, и в этом аспекте архитектура *AMD* оказывается более сбалансированной. Наконец, на максимальном размере в 16М *Intel* снова показывает лучшее время, что может быть следствием совокупности факторов, включая чистую вычислительную мощность, которая на предельных нагрузках перевешивает недостатки в работе с памятью.

ЗАКЛЮЧЕНИЕ

На основе проведенной работы можно сделать следующие выводы. Оба процессора демонстрируют значительное ускорение при использовании параллельных версий алгоритма *БПФ*. Однако архитектурные особенности процессоров по-разному влияют на производительность. *AMD Ryzen 7 5800H* показывает более стабильную и предсказуемую производительность благодаря однородной архитектуре ядер. *Intel Core i5-12450H* демонстрирует высокую производительность в многопоточных задачах благодаря гибридной архитектуре, но эффективность зависит от оптимизации распределения нагрузки между производительными (*P*-ядрами) и энергоэффективными (*E*-ядрами).

При этом *Intel Core i5-12450H* показывает лучшее абсолютное время выполнения для *CPU*-реализаций на больших объемах данных, в то время как *AMD iGPU* демонстрирует высокую эффективность при работе с большими объемами данных, опережая *Intel iGPU* на некоторых диапазонах размеров.

Технология *Hyper-Threading/SMT* оказывает различное влияние на производительность. На платформе *Intel* ее отключение может приводить к снижению производительности в многопоточных сценариях из-за необходимости задействовать *E*-ядра, в то время как на *AMD* изменение производительности чаще незначительное или даже приводит к небольшому улучшению *IPC*.

Анализ аппаратных метрик показал, что *AMD Ryzen 7 5800H* демонстрирует более эффективную работу с кэш-памятью (меньший процент промахов) и предсказуемую структуру ветвлений для алгоритма *БПФ*. *Intel Core i5-12450H* показывает исключительно высокий *IPC* на *P*-ядрах, но при этом имеет более высокий процент промахов кэша и аномальные скачки промахов ветвлений на *E*-ядрах.

Интегрированная графика (*iGPU*) демонстрирует конкурентоспособную производительность только при обработке больших объемов данных, где преимущества массового параллелизма компенсируют накладные расходы на передачу данных.

В ходе проведенного исследования установлено, что в большинстве тестовых сценариев процессор *Intel Core i5-12450H* демонстрирует превосходство над *AMD Ryzen 7 5800H* на больших размерах данных. Это преимущество особенно заметно в задачах, где эффективно задействуются производительные *P*-ядра гибридной архитектуры.

СПИСОК ЛИТЕРАТУРНЫХ ИСТОЧНИКОВ

[1] AMD Ryzen 7 5800H Specs [Электронный ресурс]. – Режим доступа: https://www.notebookcheck.net/AMD-Ryzen-7-5800H-Processor-Benchmarks-and-Specs.512759.0.html. – Дата доступа: 23.09.2025.

[2] Intel Core i5-12450H Specs [Электронный ресурс]. – Режим доступа: https://technical.city/en/cpu/Core-i5-12450H. – Дата доступа: 23.09.2025.

[3] Cinebench benchmark [Электронный ресурс]. – Режим доступа: https://cinebench.net/. – Дата доступа: 28.09.2025.

[4] Geekbench 6 [Электронный ресурс]. – Режим доступа: https://www.geekbench.com/. – Дата доступа: 28.09.2025.

[5] History of Linux [Электронный ресурс]. – Режим доступа: https://linuxsimply.com/linux-basics/introduction/history-of-linux/. – Дата доступа: 02.11.2025.

[6] 8 Most Popular Linux Distributions (2025) [Электронный ресурс]. – Режим доступа: https://www.geeksforgeeks.org/linux-unix/8-most-popular-linux-distributions/. – Дата доступа: 02.11.2025.

[7] Плюсы и минусы Arch Linux [Электронный ресурс]. – Режим доступа: https://mivocloud.com/ru/blog/plusy-i-minusy-arch-linux. – Дата доступа: 02.11.2025.

[8] Working Draft, Standard for Programming Language C++ [Электронный ресурс]. – Режим доступа: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/n4861.pdf. – Дата доступа: 02.11.2025.

[9] CMake Documentation [Электронный ресурс]. – Режим доступа: https://cmake.org/documentation/. – Дата доступа: 02.11.2025.

[10] GCC, the GNU Compiler Collection [Электронный ресурс]. – Режим доступа: https://gcc.gnu.org/. – Дата доступа: 02.11.2025.

[11] Thread support library (C++20) [Электронный ресурс]. – Режим доступа: https://en.cppreference.com/w/cpp/thread. – Дата доступа: 02.11.2025.

[12] Date and time utilities (C++20) [Электронный ресурс]. – Режим доступа: https://en.cppreference.com/w/cpp/chrono. – Дата доступа: 02.11.2025.

[13] The Fourier Analysis: The Fast Fourier Transform (FFT) Method [Электронный ресурс]. – Режим доступа: https://www.electronics-lab.com/article/the-fourier-analysis-the-fast-fourier-transform-fft-method/. – Дата доступа: 02.11.2025.

ПРИЛОЖЕНИЕ А
(обязательное)
Справка о проверке на заимствования

Рисунок А.1 – Справка о проверке на заимствования

ПРИЛОЖЕНИЕ Б
(обязательное)
Листинг программного кода

```
#include "benchmark.h"

#include <barrier>
#include <chrono>
#include <complex>
#include <fstream>
#include <iostream>
#include <random>
#include <thread>
#include <vector>


// Define the input sizes to be benchmarked
const std::vector<int> INPUT_SIZES = {
    32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384,
    32768, 65536, 131072, 262144, 524288, 1048576, 2097152, 4194304,
8388608, 16777216,
    33554432
};

benchmark::benchmark() = default;

benchmark::~benchmark() = default;

void benchmark::run_single_threaded_benchmark(const string&
output_file_path)
{
    ofstream results_file_stream(output_file_path);

    if (!results_file_stream.is_open())
    {
        cerr << "Failed to create single results file: " <<
output_file_path << endl;
        return;
    }

    results_file_stream << "Input_Size,Time_ms" << endl;
    cout << "Running single-threaded benchmark..." << endl;

    for (int size : INPUT_SIZES)
    {
        // Generate random data
        vector<complex<double>> data;
        data.reserve(size);
        std::mt19937 gen(1234); // Fixed seed for reproducibility
        std::uniform_real_distribution<> dis(-1000.0, 1000.0);
        for (int i = 0; i < size; ++i)
        {
            data.emplace_back(dis(gen), 0.0);
        }

        // Run and measure
        auto start = chrono::high_resolution_clock::now();
        fft_iterative(data);
        auto end = chrono::high_resolution_clock::now();
        chrono::duration<double, milli> duration = end - start;

        results_file_stream << size << "," << duration.count() << endl;
        cout << "  Input size " << size << ": " << duration.count() << "
ms"
            << endl;
    }

    results_file_stream.close();
```

```
    cout << "Single-threaded benchmark finished. Results saved to " <<
output_file_path << endl;
}

void benchmark::run_multithreaded_benchmark(const string&
output_file_path, unsigned int num_threads)
{
    ofstream results_file_stream(output_file_path);

    if (!results_file_stream.is_open())
    {
        cerr << "Failed to create multi results file: " <<
output_file_path << endl;
        return;
    }

    results_file_stream << "Input_Size,Time_ms" << endl;
    cout << "Running multi-threaded benchmark with " << num_threads << "
threads..." << endl;

    for (int size : INPUT_SIZES)
    {
        // Generate random data
        vector<complex<double>> data;
        data.reserve(size);
        std::mt19937 gen(1234); // Fixed seed for reproducibility
        std::uniform_real_distribution<> dis(-1000.0, 1000.0);
        for (int i = 0; i < size; ++i)
        {
            data.emplace_back(dis(gen), 0.0);
        }

        // Run and measure
        auto start = chrono::high_resolution_clock::now();
        fft_iterative_multithreaded(data, num_threads);
        auto end = chrono::high_resolution_clock::now();
        chrono::duration<double, milli> duration = end - start;

        results_file_stream << size << "," << duration.count() << endl;
        cout << "  Input size " << size << ": " << duration.count() << "
ms" << endl;
    }

    results_file_stream.close();
    cout << "Multi-threaded benchmark finished. Results saved to " <<
output_file_path << endl;
}



unsigned int benchmark::reverse_bits(unsigned int n, unsigned int bits)
{
    unsigned int reversed = 0;
    for (unsigned int i = 0; i < bits; ++i)
    {
        reversed <<= 1;
        reversed |= (n & 1);
        n >>= 1;
    }
    return reversed;
}

void benchmark::fft_iterative(vector<complex<double>>& data)
{
    const size_t N = data.size();
    int logN = 0;
    while ((1 << logN) < N) ++logN;

    for (int i = 0; i < N; ++i)
    {
        unsigned int reversed_i = reverse_bits(i, logN);
```

```
            if (i < reversed_i)
            {
                swap(data[i], data[reversed_i]);
            }
        }

        for (int s = 1; s <= logN; ++s)
        {
            int m = (1 << s);
            complex<double> wm = polar(1.0, -2 * M_PI / m);
            for (int k = 0; k < N; k += m)
            {
                complex<double> w(1.0, 0);
                for (int j = 0; j < m / 2; ++j)
                {
                    complex<double> t = w * data[k + j + m / 2];
                    complex<double> u = data[k + j];
                    data[k + j] = u + t;
                    data[k + j + m / 2] = u - t;
                    w *= wm;
                }
            }
        }
    }

    void benchmark::fft_iterative_multithreaded(vector<complex<double>>& data,
    unsigned int num_threads)
    {
        const size_t N = data.size();
        if (N < 2) return;
        int logN = 0;
        while ((1 << logN) < N) ++logN;

        auto worker = [&](unsigned int thread_id, std::barrier<>& sync_point)
        {
            // 1. Parallel Bit-Reversal
            // Each thread handles a chunk of the array.
            const size_t chunk_size = (N + num_threads - 1) / num_threads;
            const size_t start_index = thread_id * chunk_size;
            const size_t end_index = std::min(start_index + chunk_size, N);

            for (size_t i = start_index; i < end_index; ++i)
            {
                unsigned int reversed_i = reverse_bits(i, logN);
                if (i < reversed_i)
                {
                    swap(data[i], data[reversed_i]);
                }
            }

            sync_point.arrive_and_wait();

            // 2. Parallel FFT Stages
            for (int s = 1; s <= logN; ++s)
            {
                const int m = (1 << s);
                const complex<double> wm = polar(1.0, -2 * M_PI / m);
                const int num_groups = N / m;

                if (num_groups >= num_threads)
                {
                    // Strategy 1: Coarse-grained parallelism for early
    stages
                    for (int group_idx = thread_id; group_idx < num_groups;
    group_idx += num_threads)
                    {
                        const int k = group_idx * m;
                        complex<double> w(1.0, 0);
                        for (int j = 0; j < m / 2; ++j)
                        {
                            complex<double> t = w * data[k + j + m / 2];
```

```
                            complex<double> u = data[k + j];
                            data[k + j] = u + t;
                            data[k + j + m / 2] = u - t;
                            w *= wm;
                        }
                    }
                }
                else
                {
                    // Strategy 2: Fine-grained parallelism for later stages
                    const int threads_per_group = num_threads / num_groups;
                    const int my_group = thread_id / threads_per_group;
                    const int my_local_thread_id = thread_id %
threads_per_group;

                    if (my_group < num_groups)
                    {
                        const int k = my_group * m;
                        const int butterflies_in_group = m / 2;
                        const int work_per_local_thread =
                            (butterflies_in_group + threads_per_group -
1) / threads_per_group;
                        const int start_j = my_local_thread_id *
work_per_local_thread;
                        const int end_j = std::min(start_j +
work_per_local_thread, butterflies_in_group);

                        complex<double> w = pow(wm, start_j);
                        for (int j = start_j; j < end_j; ++j)
                        {
                            complex<double> t = w * data[k + j + m / 2];
                            complex<double> u = data[k + j];
                            data[k + j] = u + t;
                            data[k + j + m / 2] = u - t;
                            w *= wm;
                        }
                    }
                }

                sync_point.arrive_and_wait();
            }
        };

        // Run threads
        vector<thread> threads;
        barrier sync_point(num_threads);
        for (unsigned int i = 0; i < num_threads; ++i)
        {
            threads.emplace_back(worker, i, std::ref(sync_point));
        }

        // Wait for all threads
        for (auto& thread : threads)
        {
            thread.join();
        }
}


#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <chrono>
#include <complex>
#include <random>
#include <algorithm> // For std::reverse

// OpenCL headers
#include <CL/cl.h>
```

```
#ifndef KERNEL_FILE_PATH
#define KERNEL_FILE_PATH "fft_kernel.cl" // Fallback for non-CMake builds
#endif

// Helper function to check OpenCL errors
void checkError(cl_int err, const char* name) {
    if (err != CL_SUCCESS) {
        std::cerr << "ERROR: " << name << " (" << err << ")" <<
std::endl;
        exit(EXIT_FAILURE);
    }
}

// Function to load OpenCL kernel source code from file
std::string loadKernelSource(const char* filename) {
    std::ifstream file(filename);
    if (!file.is_open()) {
        std::cerr << "ERROR: Could not open kernel file " << filename <<
std::endl;
        exit(EXIT_FAILURE);
    }
    std::string source((std::istreambuf_iterator<char>(file)),
std::istreambuf_iterator<char>());
    return source;
}

// Function to reverse bits (for bit-reversal permutation)
unsigned int reverse_bits(unsigned int n, unsigned int bits) {
    unsigned int reversed = 0;
    for (unsigned int i = 0; i < bits; ++i) {
        reversed <<= 1;
        reversed |= (n & 1);
        n >>= 1;
    }
    return reversed;
}

int main(int argc, char* argv[]) {
    std::string output_file_path;
    for (int i = 1; i < argc; ++i) {
        std::string arg = argv[i];
        if (arg == "--output-file" && i + 1 < argc) {
            output_file_path = argv[++i];
        }
    }

    if (output_file_path.empty()) {
        std::cerr << "Error: Please provide an output file path with --
output-file" << std::endl;
        return 1;
    }

    std::ofstream results_file_stream(output_file_path);
    if (!results_file_stream.is_open()) {
        std::cerr << "Failed to create results file: " <<
output_file_path << std::endl;
        return 1;
    }

    results_file_stream << "Input_Size,Time_ms" << std::endl;
    // Define input sizes to be benchmarked (same as CPU for comparison)
    const std::vector<int> INPUT_SIZES = {
        32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384,
        32768, 65536, 131072, 262144, 524288, 1048576, 2097152, 4194304,
8388608, 16777216
    };

    cl_int err;
    cl_platform_id platform;
    cl_device_id device;
    cl_context context;
```

```
    cl_command_queue queue;
    cl_program program;
    cl_kernel kernel;

    err = clGetPlatformIDs(1, &platform, NULL);
    checkError(err, "clGetPlatformIDs");

    err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);
    if (err == CL_DEVICE_NOT_FOUND) {
        std::cout << "No GPU found, trying CPU..." << std::endl;
        err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_CPU, 1, &device,
NULL);
        checkError(err, "clGetDeviceIDs (CPU)");
    } else {
        checkError(err, "clGetDeviceIDs (GPU)");
    }

    char deviceName[128];
    clGetDeviceInfo(device, CL_DEVICE_NAME, sizeof(deviceName),
deviceName, NULL);
    std::cout << "Using device: " << deviceName << std::endl;

    context = clCreateContext(NULL, 1, &device, NULL, NULL, &err);
    checkError(err, "clCreateContext");

    queue = clCreateCommandQueue(context, device,
CL_QUEUE_PROFILING_ENABLE, &err);
    checkError(err, "clCreateCommandQueue");

    std::string kernelSource = loadKernelSource(KERNEL_FILE_PATH);
    const char* kernelSourcePtr = kernelSource.c_str();
    program = clCreateProgramWithSource(context, 1, &kernelSourcePtr,
NULL, &err);
    checkError(err, "clCreateProgramWithSource");

    err = clBuildProgram(program, 1, &device, NULL, NULL, NULL);
    if (err != CL_SUCCESS) {
        size_t logSize;
        clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG, 0,
NULL, &logSize);
        std::vector<char> buildLog(logSize);
        clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG,
logSize, buildLog.data(), NULL);
        std::cerr << "ERROR: OpenCL Build Log:\n" << buildLog.data() <<
std::endl;
        checkError(err, "clBuildProgram"); // This will exit
    }

    kernel = clCreateKernel(program, "fft_kernel", &err);
    checkError(err, "clCreateKernel");

    std::cout << "Running GPU benchmark..." << std::endl;

    for (int N : INPUT_SIZES) {
        // Generate random data (real part only for now, complex part 0)
        std::vector<cl_float2> h_data(N); // Host data
        std::mt19937 gen(1234); // Fixed seed for reproducibility
        std::uniform_real_distribution<> dis(-1000.0, 1000.0);
        for (int i = 0; i < N; ++i) {
            h_data[i] = { (cl_float)dis(gen), 0.0f };
        }

        // Perform bit-reversal permutation on host (or could be a
separate kernel)
        int logN = 0;
        while ((1 << logN) < N) ++logN; // N must be a power of 2 for
this FFT

        std::vector<cl_float2> h_data_permuted = h_data; // Copy for
permutation
        for (int i = 0; i < N; ++i) {
```

```
            unsigned int reversed_i = reverse_bits(i, logN);
            if (i < reversed_i) {
                std::swap(h_data_permuted[i],
h_data_permuted[reversed_i]);
            }
        }

        cl_mem d_data = clCreateBuffer(context, CL_MEM_READ_WRITE |
CL_MEM_COPY_HOST_PTR,
                                       sizeof(cl_float2) * N,
h_data_permuted.data(), &err);
        checkError(err, "clCreateBuffer");

        err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_data);
        checkError(err, "clSetKernelArg 0");
        err = clSetKernelArg(kernel, 1, sizeof(cl_int), &N);
        checkError(err, "clSetKernelArg 1");

        size_t globalWorkSize[1] = { (size_t)N / 2 }; // Each work-item
processes one butterfly
        size_t localWorkSize[1] = { 64 }; // Smaller local work size for
broader compatibility

        if (globalWorkSize[0] < localWorkSize[0]) {
            localWorkSize[0] = globalWorkSize[0];
        }

        double total_kernel_duration_ms = 0;
        auto start_host = std::chrono::high_resolution_clock::now();

        for (int stage = 1; stage <= logN; ++stage) {
            err = clSetKernelArg(kernel, 2, sizeof(cl_int), &stage);
            checkError(err, "clSetKernelArg 2 (stage)");

            cl_event event;
            err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL,
globalWorkSize, localWorkSize, 0, NULL, &event);
            checkError(err, "clEnqueueNDRangeKernel");

            err = clWaitForEvents(1, &event);
            checkError(err, "clWaitForEvents");

            // Get kernel execution time from event profiling
            cl_ulong time_start, time_end;
            clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START,
sizeof(cl_ulong), &time_start, NULL);
            clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END,
sizeof(cl_ulong), &time_end, NULL);
            total_kernel_duration_ms += (time_end - time_start) /
1000000.0;
            clReleaseEvent(event);
        }
        auto end_host = std::chrono::high_resolution_clock::now();

        results_file_stream << N << "," << total_kernel_duration_ms <<
std::endl;
        std::cout << "  Input size " << N << ": Kernel " <<
total_kernel_duration_ms << " ms" << std::endl;

        // std::vector<cl_float2> d_results(N);
        // err = clEnqueueReadBuffer(queue, d_data, CL_TRUE, 0,
sizeof(cl_float2) * N, d_results.data(), 0, NULL, NULL);
        // checkError(err, "clEnqueueReadBuffer");

        clReleaseMemObject(d_data);
    }

    clReleaseKernel(kernel);
    clReleaseProgram(program);
    clReleaseCommandQueue(queue);
    clReleaseContext(context);
```

```
    results_file_stream.close();

    std::cout << "GPU benchmark finished. Results saved to " <<
output_file_path << std::endl;

    return 0;
}


// fft_kernel.cl

// Function to multiply two complex numbers (a + bi) * (c + di) = (ac -
bd) + (ad + bc)i
float2 complex_mul(float2 a, float2 b) {
    return (float2)(a.x * b.x - a.y * b.y, a.x * b.y + a.y * b.x);
}

// Function to compute the twiddle factor (e^(-2*PI*k/N))
float2 twiddle_factor(int k, int N) {
    float angle = -2.0f * M_PI_F * (float)k / (float)N;
    return (float2)(cos(angle), sin(angle));
}

__kernel void fft_kernel(
    __global float2* data, // Input/output array of complex numbers
    int N,                  // Total size of the FFT
    int stage               // Current stage (s from 1 to logN)
) {
    int gid = get_global_id(0);

    // Boundary check to ensure we don't go out of bounds
    if (gid >= N / 2) {
        return;
    }

    // m = 2^stage
    int m = 1 << stage;
    int m_half = m / 2;

    // Derive 'j' (index within the butterfly group) and 'k' (start of
the current m-sized block)
    // from the global ID.
    // There are N/2 total butterflies in a stage.
    // gid ranges from 0 to N/2 - 1.
    int j = gid % m_half;
    int k_group_start = (gid / m_half) * m;

    // Calculate the twiddle factor for this specific butterfly
    // w = e^(-2*PI*j/m)
    float2 w = twiddle_factor(j, m);

    // Perform the butterfly operation
    float2 u = data[k_group_start + j];
    float2 t = complex_mul(w, data[k_group_start + j + m_half]);

    data[k_group_start + j] = u + t;
    data[k_group_start + j + m_half] = u - t;
}
```

ПРИЛОЖЕНИЕ В
(обязательное)
Функциональная схема алгоритма, реализующая программное средство

ПРИЛОЖЕНИЕ Г
(обязательное)
Блок схема алгоритма,
реализующего программное средство

ПРИЛОЖЕНИЕ Д
(обязательное)
Графики сравнения производительности процессоров

ПРИЛОЖЕНИЕ Е
(обязательное)
Графическое представления нагрузки на ядра процессоров

ПРИЛОЖЕНИЕ Ж
(обязательное)
Ведомость курсового проекта