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 AMD RYZEN 5 3500U И AMD RYZEN 7 7700 НА ПРИМЕРЕ АЛГОРИТМОВ ПОИСКА В ШИРИНУ (BFS) И В ГЛУБИНУ (DFS)

БГУИР КП 6-05 0612 02 020 ПЗ

Студент М. М. Петроченко

Руководитель А. А. Калиновская

Нормоконтролёр А. А. Калиновская

Минск 2025

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В условиях стремительного развития вычислительных технологий эффективность аппаратного обеспечения остается одним из ключевых факторов, определяющих возможности современных программных решений. Особую актуальность приобретает сравнительный анализ производительности процессоров, поскольку это позволяет оптимально подобрать аппаратную платформу под конкретные задачи. В рамках данного курсового проекта рассматривается сравнительная оценка производительности двух моделей процессоров компании *AMD – Ryzen 5 3500U* и *Ryzen 7 7700 –* на основе реализации классических алгоритмов поиска в ширину (*BFS*) и поиска в глубину (*DFS*).

Цель данной работы – провести сравнительный анализ производительности процессоров *AMD Ryzen 5 3500U* и *AMD Ryzen 7 7700* при выполнении алгоритмов поиска в ширину (*BFS*) и поиска в глубину (*DFS*). Это позволит выявить влияние технических характеристик процессоров на эффективность выполнения классических алгоритмических задач.

Для достижения цели планируется решить следующие задачи:

1 Разработать условия проведения измерений, позволяющие максимально объективно провести сравнительную оценку производительности.

2 Подготовить реализации алгоритмов BFS и DFS для тестирования.

3 Провести измерения времени их выполнения на обоих процессорах в разработанных условиях и выполнить сравнительный анализ результатов с учетом особенностей архитектуры и технических параметров исследуемых моделей.

Пояснительная записка оформлена в соответствии с СТП 01-2024 [1].

Данный курсовой проект выполнен мной лично, проверен на заимствования, процент оригинальности составляет 87% (отчёт о проверке на заимствования прилагается).

1 АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНОЙ СИСТЕМЫ

1.1 Выбор вычислительной системы

Для сравнения производительности был проведён анализ ряда доступных процессоров. В результате анализа были выбраны модели *AMD Ryzen 5 3500U* и *AMD Ryzen 7 7700*. Данные модели показаны на рисунке 1.1.

а

б

а – процессор *AMD Ryzen 5 3500U*;
б – процессор *AMD Ryzen 7 7700*
Рисунок 1.1 – Изображения выбранных процессоров

Основные характеристики процессоров представлены в таблице 1.1 [2].

Таблица 1.1 – Основные технические характеристики выбранных процессоров

	AMD Ryzen 5 3500U	*AMD Ryzen 7 7700*
Год выхода	2019	2023
Категория	Мобильный	Десктопный
Кодовое имя архитектуры	*Picasso-U* (*Zen+*)	*Raphael* (*Zen 4*)
Количество физических ядер	4	8
Количество логических ядер	8	16
Базовая частота, ГГц	2,1	3,8
Максимальная частота, ГГц	3,7	5,3
Кэш L1, Кб	96	64
Кэш L2, Кб	512	1024
Кэш L3, Мб	4	32
Технологический процесс, нм	12	5
TDP, Вт	15	65
Встроенная графика	*Radeon Vega 8*	*Radeon Graphics*

Примечание – Данные о размере кэшей L1 и L2 указаны на одно ядро, данные о размере кэша L3 – на все ядра.

Выбор данных моделей был обусловлен следующими факторами:

1 Наличие оборудования. Обладая обеими моделями процессоров, есть возможность удобно провести сравнительный анализ и оценить производительность каждого из устройств в выбранном классе задач.

2 Наличие встроенной графики. За счёт наличия встроенной графики в обеих моделях процессоров появляется возможность оценить и сравнить производительность не только вычислительных ядер, но и графических.

1.2 История, версии и достоинства выбранной архитектуры

В качестве процессоров для оценки производительности в ходе данного курсового проекта были использованы процессоры *Ryzen 5 3500U* и *Ryzen 7 7700* производства компании *AMD*. Оба процессора относятся к архитектуре *x86-64*, иначе известной как *AMD-64*. Данная архитектура, представленная в 2000 году компанией *AMD*, является версией архитектуры *x86*, хотя изначально разрабатывалась как расширение. Впервые данная архитектура была реализована в микропроцессоре *AMD Opteron*, выпущенном в апреле 2003 года.

В процессоре *AMD Ryzen 5 3500U* архитектура *x86-64* реализована в микроархитектуре *Zen+*. Данная микроархитектура является улучшением микроархитектуры *Zen*. На рисунке 1.2 представлена упрощённая схема ядра процессора на основе микроархитектуры *Zen*.

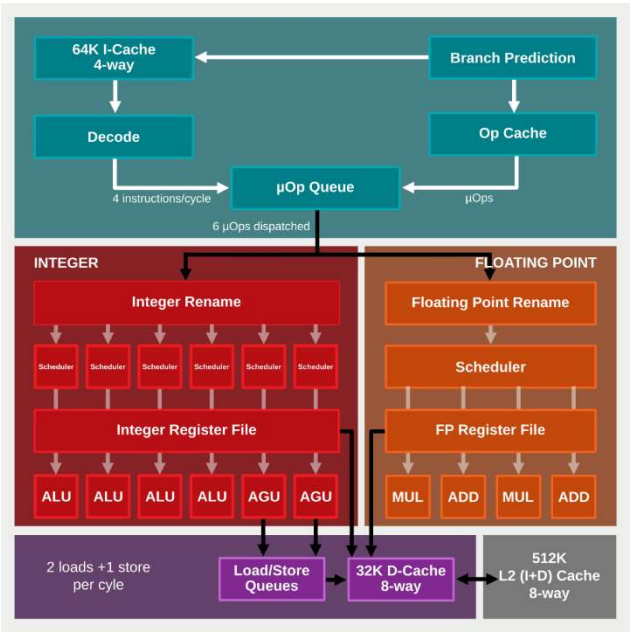


Рисунок 1.2 – Упрощённая иллюстрация ядра на основе микроархитектуры *Zen*

В процессоре *AMD Ryzen 7 7700* архитектура *x86-64* реализована в микроархитектуре *Zen 4*. Для более наглядного сравнения перечисленных

микроархитектур в таблице 1.2 представлены ключевые особенности и отличия микроархитектур от *Zen* до *Zen 5*. Стоит отметить, что для микроархитектуры *Zen* будут представлены отличия от семейства предыдущих микроархитектур компании *AMD – Bulldozer*.

Таблица 1.2 – Ключевые особенности и отличия микроархитектур семейства *Zen*

Микроархитектура	Ключевые особенности и отличия
Zen	Два потока на ядро (*SMT*), добавлен кэш декодированных микроопераций [3], увеличен размер кэша L1, увеличена пропускная способность кэш-памяти, оптимизация задержек доступа к кэш-памяти [4], переход на 14-нм технологический процесс.
Zen+	Улучшение регулировки тактовой частоты в зависимости от нагрузки [5], улучшения латентности кэша L2 и памяти [6], переход на 12-нм технологический процесс.
Zen 2	Расширения набора инструкций: *WBNOINVD, CLWB, RDPID, RDPRU, MCOMMIT* (каждой из инструкций присвоен свой *CPUID* бит) [7], аппаратная защита от уязвимости *Spectre V4* [8], переход на 7-нм технологический процесс.
Zen 3	Изменения в расположении компонентов на чипе (рисунок 1.3), переход от двух четырёхядерных комплексов на чиплете к одному восьмиядерному, увеличение количества исполняемых за цикл инструкций на 19% [9].
Zen 4	Увеличение кэша микроопераций на 69%, удвоение размера кэша *L2*, увеличение максимальной тактовой частоты до 5,7 ГГц, расширены векторные (вещественные) регистры для работы с *AVX-512*, улучшение предсказания для прямых и косвенных ветвлений [10].
Zen 5	Добавление двух дополнительных АЛУ, улучшение блока предсказания ветвлений (предсказание двух ветвлений за один тактовый цикл), увеличение размера кэша L1, увеличение ассоциативности кэша *L2* с 8 до 16.

Также для справки приведена хронология выпуска микроархитектур, описанных выше:

– *Zen* – 2017;

– *Zen+* – 2018;

– *Zen 2* – 2019;
– *Zen 3* – 2020;
– *Zen 4* – 2022;
– *Zen 5* – 2024.

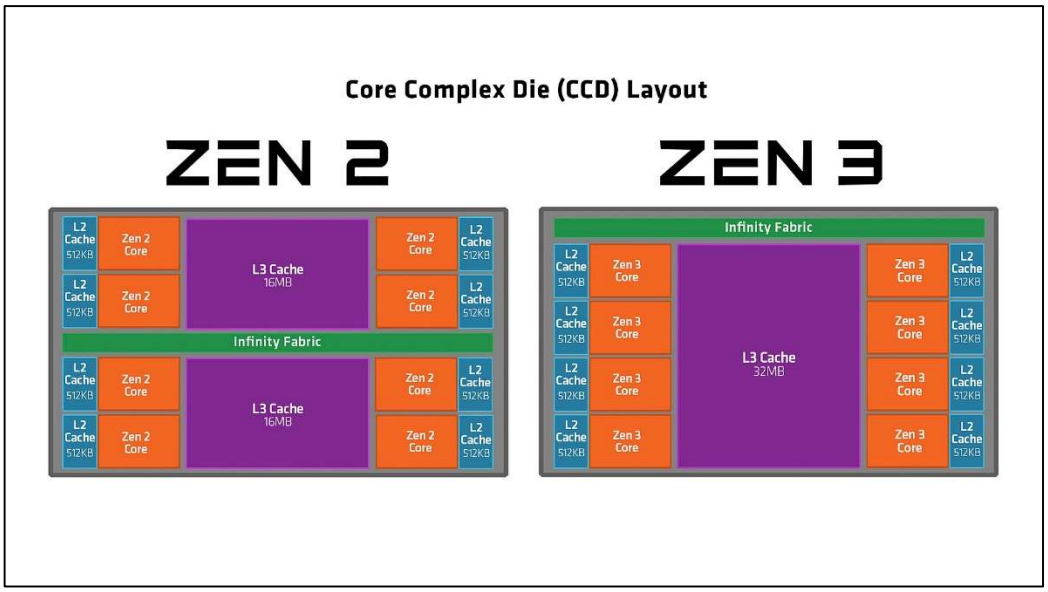


Рисунок 1.3 Сравнение расположения компонентов в микроархитектурах *Zen 2* и *Zen 3*

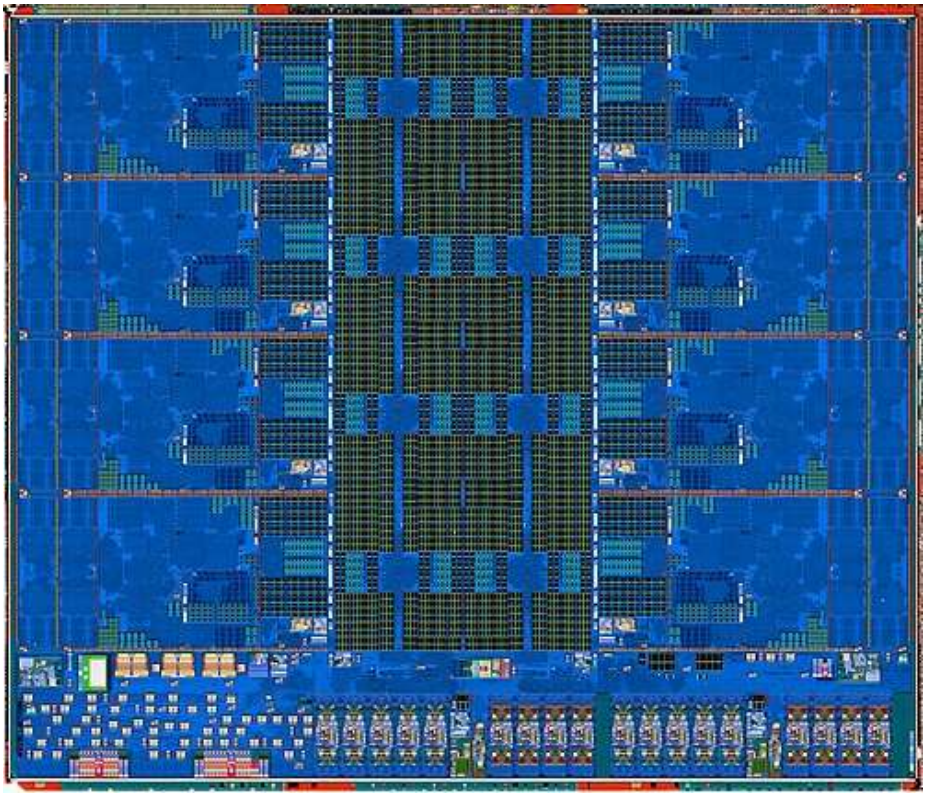


Рисунок 1.4 – *Die-shot* (снимок кристалла) процессора *AMD Ryzen 5 9600X* с микроархитектурой *Zen 5*

Сама же архитектура *x86* берёт своё начало в 1976 году, когда была начата разработка процессора *Intel 8086*, выпущенного в 1979 году. Архитектура набора команд, реализованная в этом процессоре, стала основой архитектуры *x86*. По этой причине все современные процессоры этой архитектуры могут выполнять все команды этого набора. Название архитектуры пошло от 2 последних цифр названий ранних моделей процессоров *Intel – 8086*, *80186*, *80286* (*i286*), *80386* (*i386*), *80486* (*i486*).

Архитектура развивалась одновременно с развитием технологий разработки и производства микропроцессоров, в таблице 1.3 представлены избранные вехи в развитии архитектуры и процессоров, разработанных в соответствии с этой архитектурой.

Таблица 1.3 – Избранные вехи в развитии архитектуры *x86*

Событие	Важность этапа для развития архитектуры
Выпуск *Intel 8086* (1978)	Первый коммерческий процессор с реализованной в нём архитектурой *x86.*
Выпуск *Intel 80286* (1982)	Появление такого понятия как защищённый режим и виртуальная память.
Выпуск *Intel 80386* (1985)	Первый 32-х разрядный процессор, появление режима виртуального *8086*, аппаратной отладки и страничного преобразования.
Выпуск *Intel Pentium* (*i586*) (1993)	Первый суперскалярный и суперковейерный процессор. Номерные названия ушли по причине невозможности запатентовать число.
Выпуск *Intel Pentium Pro* (*i686*) (1995)	Появление блоков предсказания ветвлений, переименования регистров, *RISC*-ядра, интеграция *L2* кэша в один корпус с ядром.
Выпуск *Intel Pentium MMX* (1997)	Появление поддержки технологии *MMX.*
Выпуск *Intel LV-Xeon DP* (2002)	Появление технологии *Hyper-Threading*.
Выпуск *AMD Opteron* (2003)	Первый процессор с архитектурой *x86-64*

К началу 2000-х годов стало очевидно, что 32-битное адресное пространство архитектуры *x86* ограничивает производительность приложений, работающих с большими объёмами данных. 32-разрядное адресное пространство позволяет процессору осуществлять непосредственную адресацию лишь 4 ГБ данных. Этого может оказаться недостаточным для некоторых приложений, связанных, например, с обработкой видео или обслуживанием баз данных.

Для решения этой проблемы *Intel* разработала новую архитектуру *IA-64*

– основу семейства процессоров *Itanium*. Для обеспечения обратной совместимости со старыми приложениями, использующими 32-разрядный код, в *IA-64* был предусмотрен режим эмуляции. Однако на практике данный режим работы оказался чрезвычайно медленным. Архитектура процессоров семейства Itanium представлена на рисунке 1.5.

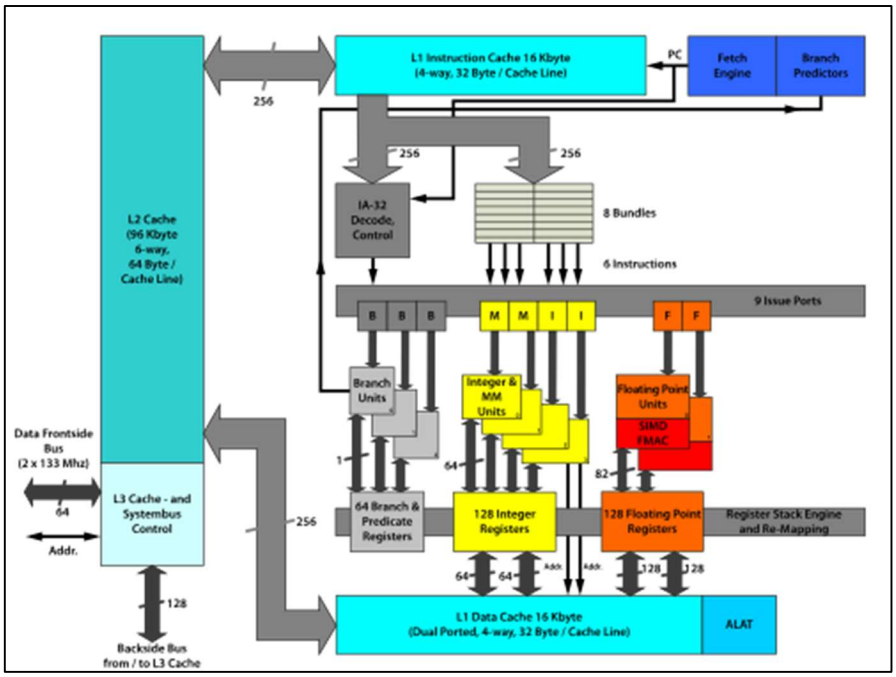


Рисунок 1.5 – Схема архитектуры процессоров семейства *Itanium*

Компания *AMD* предложила альтернативное решение проблемы увеличения разрядности процессора. Вместо того, чтобы изобретать совершенно новую систему команд, было предложено ввести 64-разрядное расширение к уже существующей 32-разрядной архитектуре *x86*. Первоначально новая архитектура называлась *x86-64*, позже она была переименована в *AMD64*. Первоначально новый набор инструкций поддерживался процессорами семейств *Opteron*, *Athlon 64* и *Turion 64* компании *AMD*. Успех процессоров, использующих технологию *AMD64*, наряду с вялым интересом к архитектуре *IA-64*, побудили *Intel* приобрести лицензию на набор инструкций *AMD64*. При этом был добавлен ряд специфических инструкций, не присутствовавших в изначальном наборе *AMD64*. Новая версия архитектуры получила название *EM64T*.

В литературе и названиях версий своих программных продуктов компании *Microsoft* и *Sun* используют объединённое именование *AMD64/EM64T*, когда речь заходит о 64-разрядных версиях их операционных систем *Windows* и *Solaris* соответственно. В то же время, поставщики программ для операционных систем семейства *Linux*, *BSD* используют метки «*x86-64*» или «*amd64*», а в *Mac OS X* используется метка «*x86_64*», если необходимо подчеркнуть, что данное ПО использует 64-разрядные

инструкции.

Процессоры данной архитектуры имеют два режима работы: *Long mode* и *Legacy mode* (режим совместимости с 32-битным процессорами на основе *x86*). На рисунке 1.6 представлена диаграмма, содержащая данные о режимах работы и процессорах, для которых эти режимы работы были введены и которые их поддерживают.

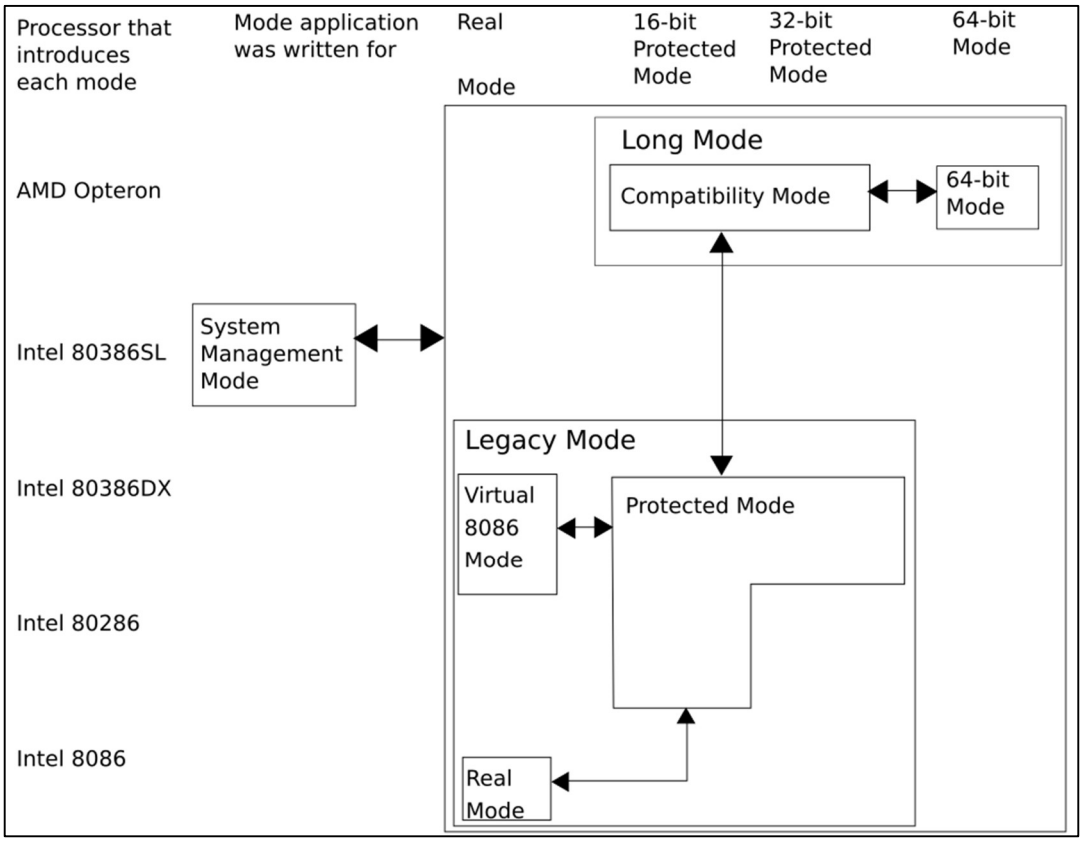


Рисунок 1.6 – Диаграмма режимов работы микропроцессоров

«Длинный» режим является основным для процессоров *AMD64*. Этот режим даёт возможность воспользоваться всеми преимуществами архитектуры *x86-64*. Для использования этого режима необходима любая 64-битная операционная система.

Этот режим позволяет выполнять 64-битные программы. Также, для обратной совместимости, предусмотрена поддержка выполнения 32-битного кода, включая 32-битные приложения. Однако при запуске 32-битных программ в 64-битной системе, они не смогут использовать 64-битные системные библиотеки, и наоборот. Для решения этой задачи большинство 64-разрядных операционных систем предоставляет два набора системных *API*: один для нативных 64-битных приложений и другой для 32-битных программ. Этот подход аналогичен методике, использовавшейся в ранних 32-битных системах, таких как *Windows 95* и *Windows NT*, для выполнения 16-битных программ.

В режиме «*Long Mode*» были ликвидированы некоторые особенности архитектуры *x86-32*, такие как режим виртуального *8086* и сегментная модель памяти. Однако, осталась возможность использования сегментов *FS* и *GS*, что полезно для быстрого нахождения важных данных потока при переключении задач. Также аппаратная многозадачность и некоторые команды, связанные с устаревшими возможностями и работой с *BCD*-числами, которые редко используются в новых программах, были исключены. «Длинный» режим активируется установкой флага *CR0.PG*, который используется для включения страничного *MMU*. Таким образом, исполнение 64-битного кода с запрещённым страничным преобразованием невозможно. Это может вызвать определённые сложности в программировании, поскольку при переключении между «*Long Mode*» и «*Legacy Mode*» (например, для вызова функций *BIOS* или *DOS*, монитором виртуальной машины, и т. д.) требуется двойной сброс *MMU*, для чего код переключения должен располагаться в одинаково отображенной странице.

«Унаследованный» режим позволяет процессору *x86-64* выполнять команды, предназначенные для процессоров *x86*, обеспечивая таким образом полную совместимость с 32-битным кодом и операционными системами для *x86*. В этом режиме процессор ведёт себя точно так же, как x86-процессор (например, как *Athlon* или *Pentium III*). Функции и возможности, предоставляемые архитектурой *x86-64* (например, 64-битные регистры), в этом режиме, естественно, недоступны. В этом режиме 64-битные программы и операционные системы работать не будут.

Разрабатывая архитектуру *AMD64*, инженеры корпорации *AMD* решили навсегда покончить с главным «рудиментом» архитектуры *x86* – сегментной моделью памяти, которая поддерживалась ещё со времён *8086*. Однако из-за этого при разработке первой *x86-64*-версии своего продукта для виртуализации программисты компании *VMware* столкнулись с непреодолимыми трудностями при реализации виртуальной машины для 64-битных гостевых систем [11]: поскольку для отделения кода монитора от кода «гостя» программой использовался механизм сегментации, эта задача стала практически неразрешимой.

2 ПЛАТФОРМА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

2.1 Структура и архитектура платформы

Для выполнения данного курсового проекта была выбрана операционная система семейства *Linux* (*GNU/Linux*). Ниже рассмотрены основные компоненты данной платформы, её аспекты и характеристики, а также роль платформы в разработке и выполнении прикладных программ.

Платформа *Linux* имеет модульную структуру, что позволяет ей быть конкурентноспособной во множестве различных сценариев применения и обеспечивать высокую надёжность и производительностъ. ОС *Linux* состоит из следующих компонентов:

1 Ядро *Linux* (*Linux Kernel*) – центральная часть операционной системы, обеспечивающая приложениям координированный доступ к ресурсам компьютера, таким как процессорное время, память, внешнее аппаратное обеспечение, внешнее устройство ввода и устройства вывода. Также обычно ядро предоставляет сервисы файловой системы и сетевых протоколов. Как основополагающий элемент ядро представляет собой наиболее низкий уровень абстракции для доступа приложений к ресурсам системы, необходимым для своей работы. Как правило, ядро предоставляет такой доступ исполняемым процессам соответствующих приложений за счёт использования механизмов межпроцессного взаимодействия и обращения приложений к системным вызовам ОС.

2 Стандартная библиотека языка Си (*libc*) – библиотека Си, которая обеспечивает системные вызовы и основные функции, такие как open, malloc и т. д. Стандартная библиотека Си используется для всех динамически скомпонованных программ. По сути стандартная библиотека является обёрткой над системными вызовами ядра Linux выполненной на языке Си, как показано на рисунке 2.1. Стоит также отметить, что *libc* является лишь частью стандарта *ANSI C*, то есть описанием программного интерфейса, а не настоящей библиотекой, используемой при компиляции. Существуют различные реализации стандартной библиотеки. В большинстве дистрибутивов Linux *glibc* (*GNU C Library*) выступает в качестве стандартной библиотеки. Данная реализация стремится предоставить пользователям наиболее оптимизированный относительно скорости выполнения вариант. Предназначена преимущественно для динамической компановки, поэтому оптимизация размера библиотеки на диске и в оперативной памяти не была наиболее приоритетной задачей при её разработке. Наиболее популярный конкуррент *glibc* – *musl*. *musl* – реализация стандартной библиотеки языка Си, нацеленная на эффективность в том числе и при статической компановке. Данная реализация стандартной библиотеки была разработана с нуля чтобы обеспечить надежность в реальном времени, избегая состояний гонки, внутренних сбоев при исчерпании ресурсов и различных других плохих

случаев поведения, присутствующих в существующих реализациях [12].

3 Системные компоненты – программы, обеспечивающие работу системы или определённых подсистем. К системным компонентам относят: подсистемы инициализации (*init daemon*), системные демоны, оконные системы, библиотеки для работы с графикой и т.д. Примером подсистемы инициализации может послужить используемая в большинстве дистрибутивов *systemd* или *OpenRC*, используемая в *Alpine Linux* и других дистрибутивах.

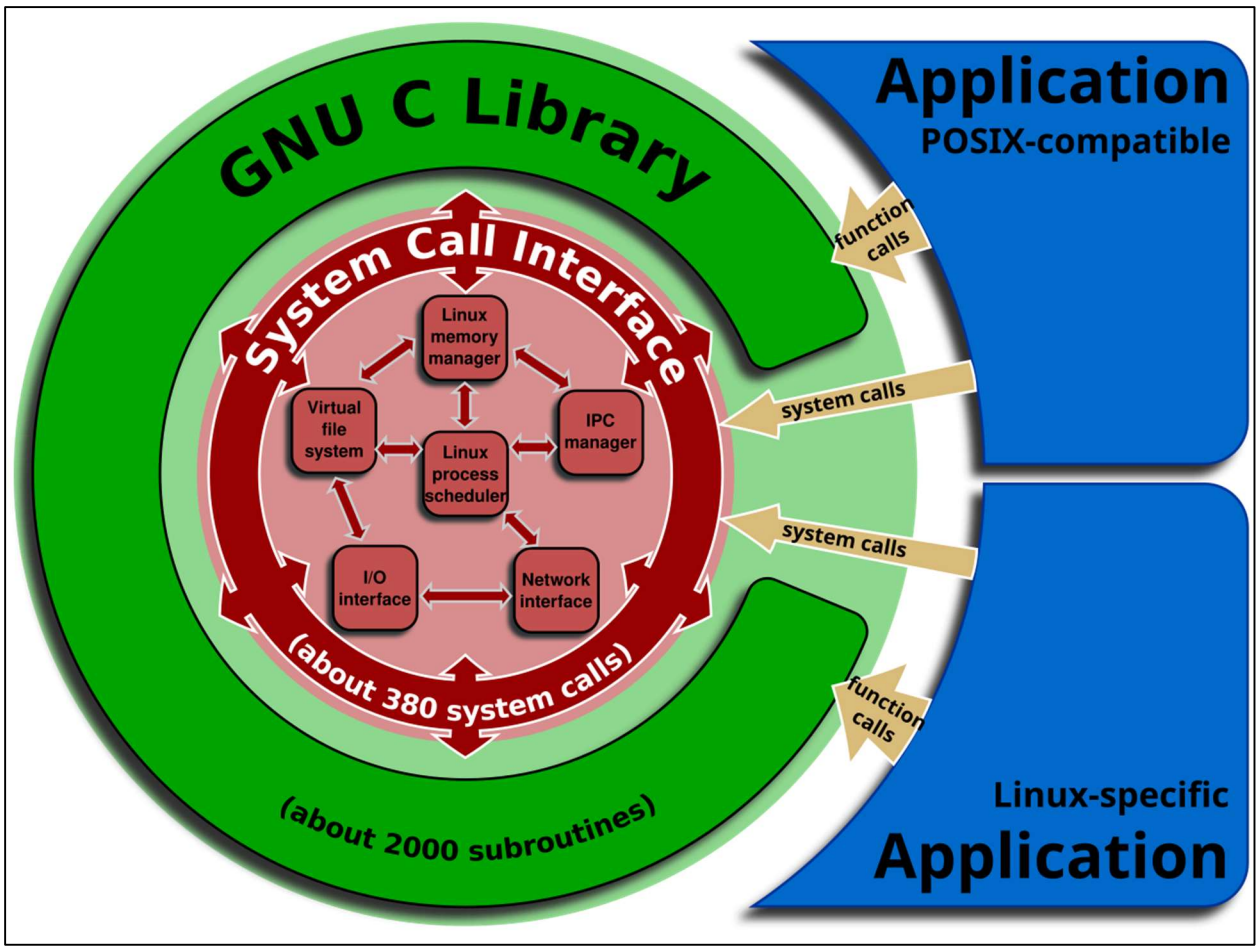


Рисунок 2.1 – Схематичное изображение отношения между стандартной библиотекой, приложениями и ядром *Linux*.

Ядро *Linux* относится к классу монолитных ядер операционных систем. Основными особенностями этого архитектурного класса являются работа всех частей ядра в одном адресном пространстве и богатый набор абстракций оборудования. Альтернативой данного класса ядер являются микроядра, которые реализуют минимальную функциональность (управление физической и виртуальной памятью компьютера, управление процессорным временем, управление доступом к устройствам ввода-вывода,коммуникация и синхронизация процессов) в пространстве ядра, а все другие функции (драйверы устройств, реализации файловых систем и др.) являются процессами в пользовательском пространстве и осуществляют

взаимодействие с ядром с помощью системных вызовов, а взаимодействие между собой с помощью средств межпроцессного взаимодействия. Также существуют гибридные архитектуры ядер, к которым например относится ядро *Windows NT*. Микроядро *NT* слишком велико (более 1 Мбайт, кроме того, в ядре системы находится, например, ещё и модуль графического интерфейса), чтобы носить приставку «микро». Компоненты ядра *Windows NT* располагаются в вытесняемой памяти и взаимодействуют друг с другом путём передачи сообщений, как и положено в микроядерных операционных системах. В то же время все компоненты ядра работают в одном адресном пространстве и активно используют общие структуры данных, что свойственно операционным системам с монолитным ядром.

Старые монолитные ядра требовали перекомпиляции при любом изменении состава оборудования. Большинство современных ядер, такие как *Linux*, позволяет во время работы динамически подгружать и выгружать модули, выполняющие часть функций ядра. Модульность ядра осуществляется на уровне бинарного образа, а не на архитектурном уровне ядра, так как динамически подгружаемые модули загружаются в адресное пространство ядра и в дальнейшем работают как непосредственная часть ядра. Модульные монолитные ядра не следует путать с архитектурным уровнем модульности, присущим микроядрам и гибридным ядрам. Практически, динамическая загрузка модулей — это просто более гибкий способ изменения образа ядра во время выполнения — в отличие от перезагрузки с другим ядром. Модули позволяют легко расширить возможности ядра по мере необходимости. Динамическая подгрузка модулей помогает сократить размер кода, работающего в пространстве ядра, до минимума, например, свести к минимуму размер ядра для встраиваемых устройств с ограниченными аппаратными ресурсами.

2.2 История, версии и достоинства

Семейство операционных систем *GNU/Linux* является одной из наиболее значимых ступеней в развитии операционных систем и в компьютерной индустрии в целом. История *Linux* началась в 1991 году и уже с 2000-х годов системы *Linux* являются основными для серверов и суперкомпьютеров, распространение среди пользователей настольных компьютеров также растёт, хотя и значительно меньшими темпами, и по статистике на сентябрь 2025 года достигло 3,17% [13].

Предшественником и идейным вдохновителем *GNU/Linux* послужила операционная система *Unix*, являющаяся предком или вдохновителем большинства используемых операционных систем, что показано на рисунке 2.2.

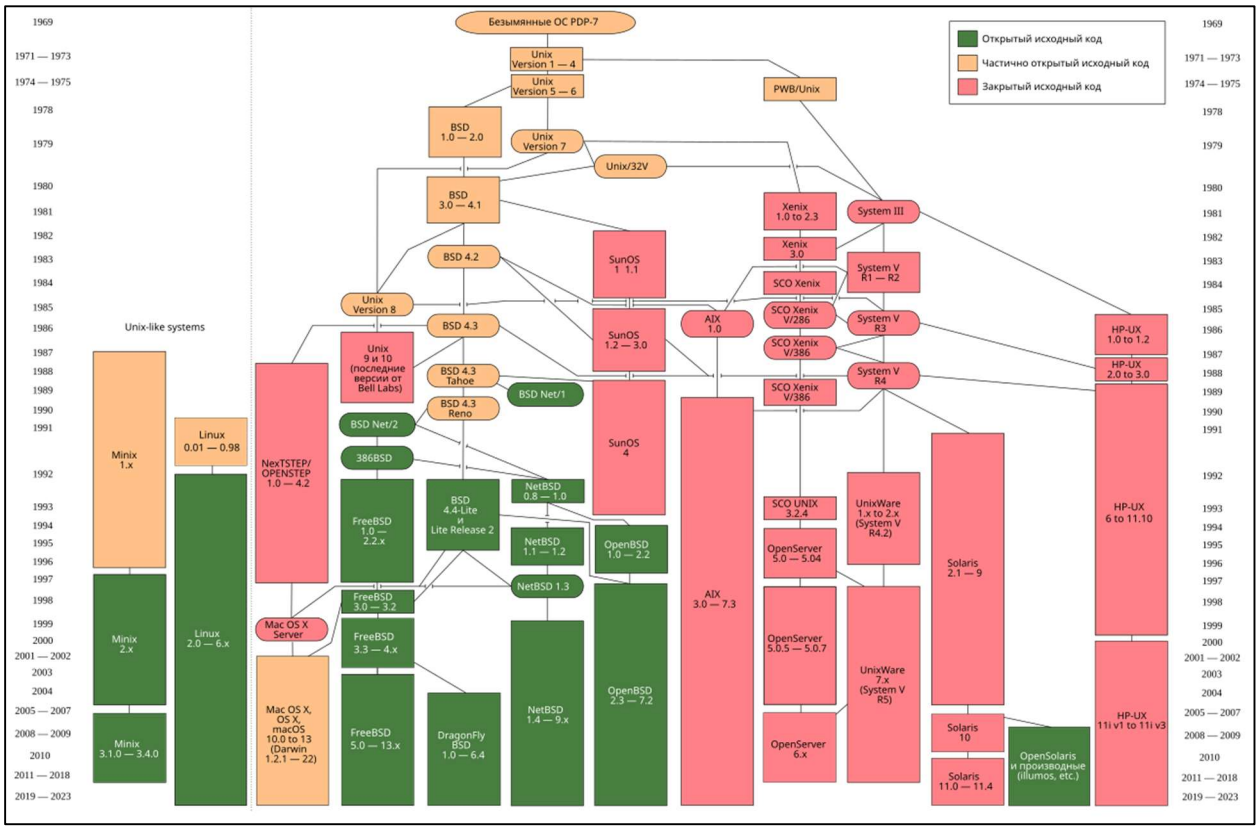


Рисунок 2.2 – Гениалогическое древо *Unix*-систем

В 1984 году разработавшая *Unix* компания *Bell Labs*, принадлежащая *AT&T*, начала продавать ОС как проприетарный продукт, модификация которого была юридически запрещена. В связи с закрытостью *Unix* в 1983 Ричард Столлман основал проект *GNU*, основной целью которого было создание свободной Unix-подобной операционной системы с открытым исходным кодом. Аббревиатура *GNU* это рекурсивный акроним от английского *GNU's Not Unix* – «*GNU* – не *Unix*». К началу 1990-х годов был готов ряд программ, необходимых для операционной системы, таких как библиотеки, компиляторы, текстовые редакторы и оболочка коммандной строки, однако разработка низкоуровневых элементов таких как драйверы, демоны и ядро, вместе называемых *GNU Hurd*, была затянута и данные элементы были далеки от готовности.

В 1990 году студент Хельсинкского университета Линус Торвальдс записался на курс по *Unix*-системам. На данном курсе в качестве одного из обязательных материалов для изучения фигурировала книга «Операционные системы: разработка и реализация» за авторством Эндрю Таненбаума. В комплекте с книгой распространялся также исходный код операционной системы *Minix*, разработанной самим Таненбаумом в целях обучения студентов конструированию компьютеров и созданию операционных систем. Данная операционная система была лицензирована только для учебного использования.

Из-за проблем с лицензированием *Minix* в 1991 году Торвальдс начал работу над своим ядром операционной системы, которое впоследствии стало ядром Linux.

В феврале 1992 Линус Торвальдс перевёл свой проект на лицензию *GPLv2* (*GNU General Public License version 2*) [14], которая, в отличие от его собственной лицензии, позволяла свободно коммерчески распространять ядро. Это позволило с одной стороны использовать *Linux* в качестве ядра для операционной системы *GNU*, а с другой стороны заменить *Unix* и *Minix* приложения на свободные приложения *GNU*. По этой причине технически семейство операционных систем называемое Linux правильнее называть *GNU/Linux*, потому как оба компонента обеспечивают существование операционной системы как общего целого.

В мае 1992 была выпущена версия ядра 0.96, позволявшая запускать оконную систему *X11*.

Версия 1.0, первая версия *Linux*, подходящая для работы в окружении конечного пользователя, была выпущена в марте 1994.

В июне 1996 была выпущена версия 2.0, основными нововведениями которой были симметричная многопроцессорная обработка (*SMP*), поддержка большего количества типов процессоров и поддержка выбора конкретного аппаратного обеспечения для обеспечения специфических особенностей архитектуры и оптимизации, а также загружаемые модули ядра.

В версии 2.2, выпущенной в январе 1999 года была добавлена поддержка 64-разрядных архитектур, а также множества файловых систем, включая поддержку *NTFS* компании *Microsoft* в режиме только для чтения.

В версии 2.4.0 была добавлена поддержка микроархитектур *Pentium* 4 и *Itanium* а также поддержка *Bluetooth, RAID* и *LVM.*

В версии 2.6.0 в ядро была интегрирована реализация стандартной библиотеки языка Си *uClinux*, предназначенная для использования во встраиваемых системах, максимальное количество пользователей было увеличено с 2^{16} до 2^{32}, а максимальное количество процессов – до 2^{29}, но только для 64-разрядных систем и ещё множество других нововведений.

Изменение нумерации с 2.6.39 на 3.0 и с 3.19 на 4.0 не имело существенных технических отличий; номер основной версии был просто увеличен, чтобы избежать большого числа во второстепенном номере [15].

В апреле 2015 было выпущено ядро версии 4.0, а в марте 2019 – ядро версии 5.0.

Актуальная на момент написания проекта стабильная версия ядра *Linux* – 6.17.5, именно она и используется в качестве ядра программной платформы.

По поводу наименования данного семейства операционных систем по сей день ведутся споры. Сторонники *FSF* (*Free Software Foundation*), авторы программ *GNU*, и Ричард Столлман в частности считают, что операционная система должна быть названа GNU/Linux в соответствии с тем, кто разработал какие части системы. Столлман писал: «Проект *GNU* не был и не является проектом по разработке определенных пакетов программ. […] Многие

сделали серьезный вклад в свободные программы этой системы, и все они заслуживают признательности за свои программы. Но причина, по которой это является целостной системой – а не только набором полезных программ – то, что проект *GNU* постановил сделать это такой системой. Мы составили список программ, необходимых, чтобы свободная система была полной, и систематически отыскивали, писали или отыскивали людей для написания всего, что стояло в списке».

Сам Торвальдс относится к данным спорам с долей иронии. В документальном фильме «Революционная ОС» он говорит: «Ну, я думаю, что это справедливо, однако справедливо в том случае, если вы сделаете *GNU* дистрибутив *Linux* … так же как я считаю, что "*Red Hat Linux*" это нормально, или "*SUSE Linux*", или "*Debian Linux*", потому что если вы реально сделаете свой дистрибутив *Linux* вы можете дать ему название, но называть *Linux* в общем "*GNU/Linux*" я считаю попросту смешным». Однако общее его вовлечение в этот спор невелико.

Весь долгий путь пройденный *Linux* свидетельствует о постоянной эволюции и стремлении сообщества к улучшению производительности, безопасности и удобства использования для миллионов пользователей по всему миру. *Linux* остается одной из наиболее популярных и важных операционных систем на сегодняшний день.

2.3 Обоснование выбора платформы

В ходе курсового проекта выбор программной платформы играет ключевую роль. В качестве кандидатов на роль программной платформы при выполнении данного курсового проекта были выделены *Arch Linux, Windows 10* и *Windows 11*. В данном случае, было принято решение использовать дистрибутив из семейства операционных систем *Linux Arch Linux* в качестве платформы для проведения сравнения выбранных моделей процессоров. Существует несколько важных аргументов, которые обосновывают выбор *Arch Linux:*

1 Легковесность. Дистрибутив *Arch Linux* является одним из наиболее оптимизированных дистрибутивов в мире. Это означает, что множество инструментов и программ не установлены по умолчанию. Такой подход позволяет наиболее точно оценить производительность процессора в отсутствие задач заднего плана и даже графической оболочки, в чём и заключается основное преимущество над основным конкурентом в данном выборе – операционной системой *Windows 10*.

2 Поддержка многих процессоров. *Linux* обладает высокой совместимостью с различными аппаратными конфигурациями, включая многоядерные процессоры. Это предоставляет отличную возможность для исследования и оптимизации вычислений на различных системах.

3 Современность. *Arch Linux* выпускается в соответствии с моделью *Rolling Release*, то есть позволяет пользователю всегда иметь последние версии устанавливаемых программ, избавляя его от необходимости периодической переустановки системы. Данная модель также относится и к версии ядра *Linux*, поэтому при использовании данного дистрибутива у пользователя всегда есть доступ к самым новым нововведениям и особенностям ПО.

4 Доступность библиотек и фреймворков. Платформа *Linux* на данный момент де-факто является стандартом индустрии, поэтому большинство библиотек и фреймворков доступно и имеет наилучшую поддержку именно для *Linux*.

Таким образом, выбор операционной системы *Arch Linux* в качестве платформы для проведения сравнения производительности процессоров обоснован легковесностью, поддержкой множества процессоров и архитектур, наличием разнообразных инструментов разработки, доступом к наиболее современным версиям программ, богатстве библиотек и удобстве использования. Эта платформа позволяет провести обширное исследование в области оптимизации производительности вычислений.

3 ТЕОРЕТИЧЕСКОЕ ОБОСНОВАНИЕ РАЗРАБОТКИ ПРОГРАММНОГО ПРОДУКТА

3.1 Обоснование необходимости разработки

В современном информационном обществе, где высокая производительность вычислений играет ключевую роль, разработка программного продукта, направленного на измерение производительности вычислений на различных процессорах с использованием классических алгоритмических задач, является важным шагом в обеспечении эффективного использования вычислительных ресурсов.

Современный рынок процессоров предлагает широкий выбор многоядерных моделей, каждая из которых обладает уникальными характеристиками по частоте, числу ядер и энергоэффективности. Среди популярных представителей – *AMD Ryzen 5 3500U* и *AMD Ryzen 7 7700*, которые демонстрируют различные уровни производительности и архитектурные решения. Исследование и сравнение их вычислительных возможностей на примере типовых алгоритмов поиска в графах — поиска в ширину (*BFS*) и поиска в глубину (*DFS*) — позволяет оценить влияние аппаратных особенностей процессоров на эффективность алгоритмических задач.

Основные преимущества проведения сравнения производительности данных процессоров с использованием алгоритмов *BFS* и *DFS* заключаются в следующем:

1 Практическая оценка вычислительной мощности. Выполнение алгоритмов обхода графа на обеих моделях процессоров позволяет объективно измерить время выполнения, загрузку ядер и влияние различных параметров архитектуры на скорость обработки данных.

2 Анализ масштабируемости и параллелизма. Алгоритмы поиска в ширину и глубину имеют разные структуры обхода и требования к памяти, что дает возможность изучить, как каждый процессор справляется с параллельной обработкой и управлением ресурсами при типичных задачах.

3 Информативность для выбора аппаратной платформы. Результаты сравнения помогают разработчикам и исследователям сделать обоснованный выбор процессора для задач, связанных с обработкой графов и алгоритмической логикой, что актуально в приложениях от анализа социальных сетей до оптимизации маршрутов и систем искусственного интеллекта.

Таким образом, выполнение сравнительного анализа производительности *AMD Ryzen 5 3500U* и *AMD Ryzen 7 7700* на базе алгоритмов *BFS* и *DFS* оправдано с точки зрения глубокого понимания архитектурных преимуществ, оптимального распределения вычислительных ресурсов и повышения эффективности программных решений, зависящих от

производительности процессора. Такой подход способствует расширению практических знаний и разработке более адаптированных и производительных программных продуктов.

3.2 Технологии программирования, используемые для решения поставленных задач

Выбор языка программирования является одним из ключевых решений, существенно влияющих на успешность и качество разработки программного продукта, особенно если речь идет об оптимизации производительности вычислений. В данном проекте был сделан осознанный выбор в пользу языка *C++*, что обусловлено рядом весомых преимуществ, выгодно выделяющих его среди других языков программирования для решения подобных задач.

Прежде всего, *C++* отличается выдающейся производительностью, близкой к языку ассемблера, что позволяет создавать максимально эффективный и быстрый код. Высокая скорость выполнения программ на C++ критически важна для задач, где требуется глубокая оптимизация вычислительных процессов и минимизация времени отклика.

Несмотря на то, что *C++* обеспечивает мощный контроль над низкоуровневыми аспектами программирования, он при этом сохраняет статус высокоуровнего языка. Это означает, что разработчику доступны современные средства абстракции, объектно-ориентированные и шаблонные механизмы, что значительно упрощает написание, поддержку и расширение сложных программных систем. Благодаря этому разработка на C++ становится более продуктивной и надежной, снижая вероятность ошибок и обеспечивая читаемость кода.

Важным преимуществом *C++* является его богатая и зрелая экосистема библиотек и инструментов. Стандартная библиотека шаблонов (*STL*) предоставляет мощные контейнеры, алгоритмы и средства управления памятью, а многочисленные сторонние фреймворки и библиотеки ускоряют процесс создания программ и расширяют функциональные возможности. Такой широкий выбор готовых решений позволяет сосредоточиться на специфике алгоритмов и оптимизации, не тратя время на разработку базовых компонентов.

Еще одним критически важным фактором является возможность тонкого управления памятью и ресурсами системы. В *C++* разработчик самостоятельно контролирует выделение и освобождение памяти, что особенно ценно при реализации высокопроизводительных вычислительных алгоритмов, где экономия ресурсов напрямую влияет на общую производительность и стабильность работы программы.

Кроме того, *C++* сохраняет обратную совместимость с языком C, что обеспечивает легкую интеграцию с уже существующими проектами и библиотеками, написанными на *C*. Это расширяет возможности использования кода и позволяет постепенно модернизировать системы без

необходимости полного переписывания.

Таким образом, выбор *C++* является оптимальным компромиссом между высокой производительностью, гибкостью разработки и удобством использования современных средств программирования. Он обеспечивает разработчикам все необходимые инструменты для создания эффективных, быстрых и масштабируемых программных продуктов, что особенно важно при решении задач оптимизации вычислительных процессов. Благодаря этим свойствам, *C++* становится идеальным языком для разработки программных решений, где требуются как максимальная скорость работы, так и возможность гибкой поддержки и развития кода.

Помимо языка *C++* были рассмотрены также следующие варианты языков программирования:

1 *C#*. *C#* – объектно-ориентированный язык программирования общего назначения, разработанный в 1998 – 2001 годах группой инженеров компании *Microsoft* как язык разработки приложений для платформы *Microsoft .NET Framework* и *.NET Core*. Вопреки названию платформа *Microsoft .NET* кроссплатформенна и поддерживает выполнение под управлением *Linux*. *C#* относится к семье языков с *C*-подобным синтаксисом, из них его синтаксис наиболее близок к *C++* и *Java*. Язык имеет статическую типизацию, поддерживает полиморфизм, перегрузку операторов (в том числе операторов явного и неявного приведения типа), делегаты, атрибуты, события, переменные, свойства, обобщённые типы и методы, итераторы, анонимные функции с поддержкой замыканий, *LINQ*, исключения и другие абстракции, позволяющие программисту писать высокоуровневый но высокопроизводительный код. *C#* не был выбран в качестве языка программирования для данного курсового проекта потому что не является строго компилируемым языком, а выполняется с помощью *CLR* (*Common Language Runtime*), которой делегируются низкоуровневые задачи, такие как работа с памятью. Стоит отметить, что существует возможность скомпилировать код, написанный на *C#*. Данная модель развёртывания называется *Native AOT* от выражения *Ahead Of Time* компиляция, которая противопоставлена *JIT – Just In Time* компиляции, являющейся основной моделью развёртывания приложений на *C#*. Однако такой способ развёртывания накладывает некоторые ограничения, такие как невозможность подключения динамических библиотек, требует дополнительных зависимостей и менее удобен на практике.

2 *Rust*. *Rust* – мультипарадигменный компилируемый язык программирования общего назначения, сочетающий парадигмы функционального и процедурного программирования. Управление памятью осуществляется через механизм «владения» с использованием аффинных типов, что позволяет обходиться без системы сборки мусора во время исполнения программы. Rust гарантирует безопасную работу с памятью благодаря встроенной в компилятор системе статической проверки ссылок (*borrow checker*). Имеются средства, позволяющие использовать приёмы

объектно-ориентированного программирования. Ключевые приоритеты языка: безопасность, скорость и параллелизм. Rust пригоден для системного программирования, в частности, он рассматривается как перспективный язык для разработки ядер операционных систем. Rust сопоставим по скорости и возможностям с *C/C++*, однако даёт большую безопасность при работе с памятью, что обеспечивается встроенными в язык механизмами контроля ссылок. Производительности программ на Rust способствует использование «абстракций с нулевой стоимостью». Данный язык не был выбран в качестве языка программирования для данного курсового проекта в связи с излишней сложностью, которая в контексте данного проекта не приводит к видимым преимуществам. Например, мощная система типов языка *Rust* не находит применения в задачах, стоящих при разработке программного продукта для данной курсовой работы.

3 *Zig*. *Zig* – императивный, статически типизированный, компилируемый язык программирования общего назначения. Язык был спроектирован для «создания надёжного, оптимального и переиспользуемого ПО». Поддерживает обобщённое программирование и рефлексию во время компиляции, кросс-компиляцию и ручное управление памятью. Главная цель языка – быть более удобным, чем *C* [16] в задачах системного программирования, и в то же время быть более простым, чем *C++* и *Rust*. Разработка *Zig* поддерживается некоммерческой организацией Zig Software Foundation, основанной в 2020 году автором языка Эндрю Келли. Язык имеет много средств для низкоуровневого программирования, среди таковых: упакованные структуры (структуры с нулевым выравниванием между полями), целочисленные типы произвольной длины (вплоть до 65535 бит), несколько типов указателей. Данный язык не был выбран из-за своей незрелости, хотя стоит отметить, что относительно своих конкурентов он лучше всего подходит для задач данного курсового проекта. Также данный язык полностью совместим с *C*, что является огромным преимуществом на рынке системных языков программирования.

Таким образом можно сделать вывод, что рынок языков системного программирования постоянно растёт и развивается, и не теряет востребованности. Идеи и абстракции, привносимые новыми языками, такими как *Rust* и *Zig,* уже успели зарекомендовать себя как полезные нововведения, а не проходящая мода. Однако вместе с этим необходимо отметить развитие уже существующих языков программирования, таких как *C/C++* и *C#*, которые пополняются нововведениями, библиотеками и абстркациями.

Для построения графиков и анализа данных выбор был сделан в пользу языка программирования *Python*. *Python* – мультипарадигменный высокоуровневый язык программирования общего назначения с динамической строгой типизацией и автоматическим управлением памятью, ориентированный на повышение производительности разработчика, читаемости кода и его качества, а также на обеспечение переносимости написанных на нём программ. Язык является полностью объектно-

ориентированным в том плане, что всё является объектами. Необычной особенностью языка является выделение блоков кода отступами. Синтаксис ядра языка минималистичен, за счёт чего на практике редко возникает необходимость обращаться к документации. *Python* – интерпретируемый язык, использующийся в том числе для написания скриптов. Недостатками языка являются зачастую более низкая скорость работы и более высокое потребление памяти написанными на нём программами по сравнению с аналогичным кодом, написанным на компилируемых языках, таких как *C* или *C++*. Большим преимуществом *Python* над другими языками программирования является богатая экосистема библиотек под любые нужды и требования. На момент написания на *PyPi* (*Python Package Index*) зарегистрировано 692723 проекта [17].

Для выполнения анализа данных в программном продукте применяется библиотека *pandas* – программная библиотека на языке Python для обработки и анализа данных. Работа *pandas* с данными строится поверх библиотеки *NumPy*, являющейся инструментом более низкого уровня. Предоставляет специальные структуры данных и операции для манипулирования числовыми таблицами и временными рядами. Основные возможности *pandas*:

– объект DataFrame для манипулирования индексированными массивами двумерных данных;

– инструменты для обмена данными между структурами в памяти и файлами различных форматов;

– встроенные средства совмещения данных и способы обработки отсутствующей информации;

– переформатирование наборов данных, в том числе создание сводных таблиц;

– срез данных по значениям индекса, расширенные возможности индексирования, выборка из больших наборов данных;

– вставка и удаление столбцов данных;

– возможности группировки позволяют выполнять трёхэтапные операции типа «разделение, изменение, объединение;

– слияние и объединение наборов данных;

– иерархическое индексирование позволяет работать с данными высокой размерности в структурах меньшей размерности;

– работа с временными рядами: формирование временных периодов и изменение интервалов.

Библиотека оптимизирована для высокой производительности, наиболее важные части кода написаны на *Cython* и *C*.

Для визуализации данных применяется *Matplotlib* – библиотека на языке программирования *Python* для визуализации данных двумерной и трёхмерной графикой.

Matplotlib является гибким, легко конфигурируемым пакетом. В настоящее время пакет работает с несколькими графическими библиотеками, включая *wxWindows* и *PyGTK*. Пакет поддерживает многие виды графиков и

диаграмм:

– графики;
– диаграммы рассеяния;
– столбчатые диаграммы и гистограммы;
– круговые диаграммы;
– диаграммы стебель-листья;
– контурные графики;
– поля градиентов;
– спектральные диаграммы.

Таким образом, выбор *Python* вместе с библиотеками *pandas* и *Matplotlib* обеспечивает эффективное и гибкое решение для выполнения анализа и визуализации данных. Высокоуровневые абстракции *pandas* в сочетании с оптимизированным низкоуровневым кодом позволяют работать с большими и сложными наборами данных, а *Matplotlib* предоставляет расширенные возможности для создания разнообразных графических представлений. Такая технология используется в программных продуктах для обработки данных, где требуется сочетание удобства разработки, производительности и качественной визуализации.

Для выполнения алгоритма поиска в ширину на графическом процессоре была использована технология *HIP*. *HIP* (*Heterogeneous-Compute Interface for Portability* – интерфейс для портируемых вычислений на гетерогенных системах) – тонкий уровень абстракции над *Nvidia CUDA* и *AMD ROCm*, предоставляющий возможность писать код для выполнения на *GPU* на диалекте языка *C++*.Вызовы к функциям *HIP* будут использовать библиотеки *ROCm* при выполнении в среде с графическими процессорами производства компании *AMD*, при выполнении в среде с графическими процессорами производства компании *Nvidia* вызовы будут перенаправлены к библиотекам *CUDA* [18].

При выполнении данного курсового проекта был проведён анализ рынка технологий и были выбраны наиболее подходящие из доступных по критериям аппаратного и программного обеспечения.

4 ПРОЕКТИРОВАНИЕ ФУНКЦИОНАЛЬНЫХ ВОЗМОЖНОСТЕЙ ПРОГРАММЫ

4.1 Анализ алгоритмов

В настоящее время, в условиях постоянного развития технологий, одним из важнейших аспектов в области вычислений является повышение производительности. Для сравнения производительности компьютерных систем и процессоров, в частности, необходимо тщательно выбирать соответствующие алгоритмы. Этот раздел посвящен анализу алгоритмов с целью определения их пригодности для оценки и сравнения производительности процессоров.

Перед тем как приступить к выбору конкретного алгоритма, необходимо понять основные параметры, которые должны соответствовать алгоритму для успешной оценки производительности. Важными факторами являются:

1 Распространённость и применимость – алгоритм должен быть известен и применим для решения широкого ряда задач для большей объективности проведённых измерений.

2 Простота реализации – излишняя сложность алгоритма может привести к ошибкам реализации и необходимости обращения к различным библиотекам, что снизит объективность измеренной оценки. Однако излишняя простота также может снизить объективность измеренной оценки из-за оптимизаций процессоров для выполнения простых задач.

3 Нелинейность доступа к памяти – если доступ к памяти алгоритма зависит от входных данных и не является последовательным, это позволяет лучше протестировать производительность процессора и его подсистему памяти.

С учетом вышеуказанных параметров, особое внимание уделяется алгоритмам поиска в глубину (*DFS*) и в ширину (*BFS*), широко применяемым в алгоритмах анализа графов, таких как определение циклов, связности, построение остовных деревьев и проверка двудольности графа.

4.2 Поиск в ширину

Алгоритмы поиска в графах можно разделить на несколько классов в зависимости от подхода к обходу и требованиям к результату. Одним из фундаментальных методов является алгоритм поиска в ширину (*BFS*), который используется для обхода или поиска в графах и деревьях.

Метод основан на пошаговом исследовании вершин графа, начиная с заданной стартовой точки, и переходе к вершинам, находящимся на следующем уровне расстояния, с помощью очереди. В отличие от других

методов, которые могут использовать рекурсию или жадные подходы, BFS гарантирует, что каждое посещение вершины происходит в порядке увеличения минимального количества ребер до неё.

Данный алгоритм широко применяется в задачах поиска кратчайшего пути в невзвешенных графах, проверки связности, топологической сортировки и других областях теории графов. В данном разделе будет представлен разбор математической модели и детальное описание работы алгоритма поиска в ширину.

Пусть задан граф $G = (V, E)$ и выделена исходная вершина s. Алгоритм поиска в ширину систематически обходит все ребра графа G для «открытия» всех вершин, достижимых из s, вычисляя при этом расстояние (минимальное количество рёбер) от s до каждой достижимой из s вершины. Порядок обхода вершин произвольного графа представлен на рисунке 4.1.

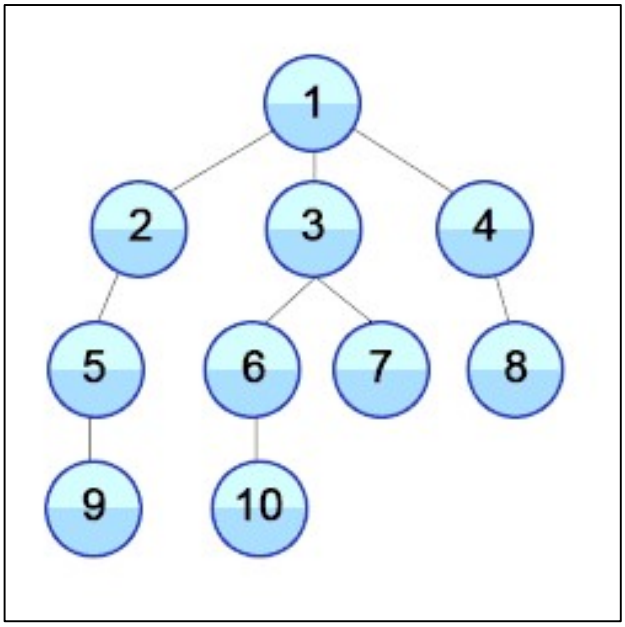


Рисунок 4.1. – Порядок обхода вершин графа в ширину начиная от корня дерева

Алгоритм работает как для ориентированных, так и для неориентированных графов [19].

Поиск в ширину имеет такое название потому, что в процессе обхода мы идём вширь, то есть перед тем, как приступить к поиску вершин на расстоянии $k + 1$, выполняется обход вершин на расстоянии k.

Поиск в ширину является одним из неинформированных алгоритмов поиска.

Обозначим количество вершин и рёбер в графе как $|V|$ и $|E|$ соответственно. Так как в памяти хранятся все развёрнутые вершины, пространственная сложность алгоритма составляет $O(|V| + |E|)$. Так как в худшем случае алгоритм посещает все вершины, при хранении графа в виде

списков смежности временная сложность алгоритма составляет $O(|V| + |E|)$.

Если у каждого узла имеется конечное число преемников, алгоритм является полным: если решение существует, алгоритм поиска в ширину его находит, независимо от того, является ли граф конечным. Однако если решения не существует, на бесконечном графе поиск не завершается.

Поиск в ширину применяется для решения задач, связанных с теорией графов:

– волновой алгоритм поиска пути в лабиринте;

– волновая трассировка печатных плат;

– поиск компонент связности в графе;

– поиск кратчайшего пути между двумя узлами невзвешенного графа;

– поиск в пространстве состояний: нахождение решения задачи с наименьшим числом ходов, если каждое состояние системы можно представить вершиной графа, а переходы из одного состояния в другое – рёбрами графа;

– нахождение кратчайшего цикла в ориентированном невзвешенном графе;

– нахождение всех вершин и рёбер, лежащих на каком-либо кратчайшем пути между двумя вершинами;

– поиск увеличивающего пути в алгоритме Форда-Фалкерсона (алгоритм Эдмондса-Карпа).

Стоит также отметить, что поиск в ширину легко параллелизуется.

4.3 Поиск в глубину

Поиск в глубину (*DFS, Depth-First Search*) является одним из базовых алгоритмов обхода графов и деревьев, основанным на углублении в построение пути. В отличие от поиска в ширину, DFS исследует как можно дальше по каждому из возможных путей, прежде чем перейти к следующему. Алгоритм начинается с заданной стартовой вершины и рекурсивно или с помощью стека углубляется в смежные с текущей вершиной непосещённые вершины, возвращаясь назад только при достижении вершины без новых непосещённых соседей.

DFS полезен для задач обхода, проверки связности, топологической сортировки ориентированных ацикличных графов, поиска компонент связности, а также для решения задач, где важен полный обход всех путей. Он работает как с ориентированными, так и с неориентированными графами.

Метод функционирует путём погружения вдоль смежных вершин, используя стек (явный или рекурсивный) для запоминания пути, что отличает его от *BFS*, который использует очередь. *DFS* относится к классу неинформированных алгоритмов поиска и позволяет получить порядок обхода, при котором вершины посещаются в глубину, обеспечивая возможность построить глубинное дерево обхода графа. Таким образом, поиск

в глубину систематически посещает вершины графа, двигаясь от стартовой к наиболее глубоким соседям, пока не исчерпает путь, после чего возвращается и прорабатывает остальные варианты.

Пусть задан граф $G = (V, E)$. Обозначим количество вершин и рёбер в графе как $|V|$ и $|E|$ соответственно. Предположим, что в начальный момент времени все вершины графа окрашены в белый цвет. Алгоритм выполняет следующие действия:

1 Осуществляется прохождение по всем вершинам $v \in V$. Если вершина v белая, для неё выполняется процедура $DFS(v)$.

Процедура $DFS(u)$ (параметр – вершина $u \in V$) выполняется следующим образом:

2 Вершина u перекрашивается в серый цвет.

3 Для всякой вершины w, смежной с вершиной u и окрашенной в белый цвет рекурсивно выполняется процедура $DFS(w)$.

4 Вершина u перекрашивается в чёрный цвет.

Часто используют двухцветные метки – без серого, на первом шаге красят сразу в чёрный цвет. Пример порядка обхода произвольного графа представлен на рисунке 4.2.

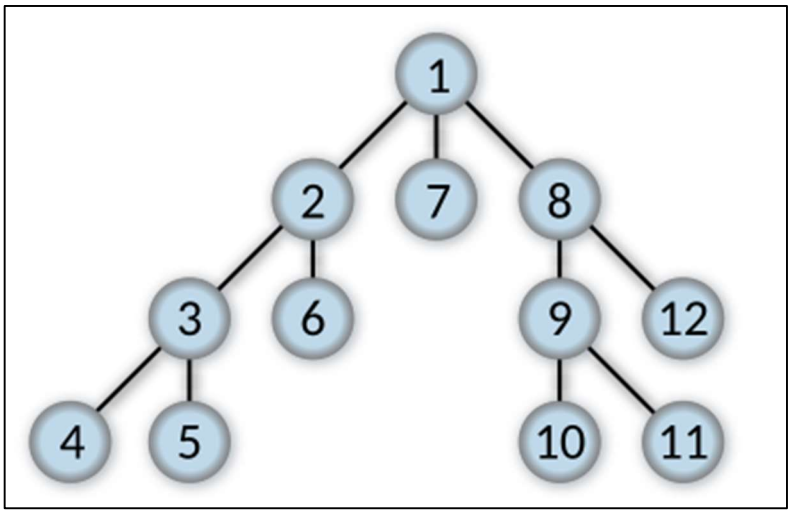


Рисунок 4.2. – Порядок обхода графа в глубину

Процедура DFS вызывается от каждой вершины не более одного раза, а внутри процедуры рассматриваются все такие ребра $\{e \mid begin(e) = u\}$. Всего таких ребер для всех вершин в графе $O(|E|)$, следовательно, время работы алгоритма оценивается как $O(|V| + |E|)$.

Поиск в глубину ограниченно применяется как собственно поиск, чаще всего на древовидных структурах: когда расстояние между точками мало, поиск в глубину может «плутать» где-то далеко. Зато поиск в глубину – хороший инструмент для исследования топологических свойств графов. Например:

– в качестве подпрограммы в алгоритмах поиска одно- и двусвязных компонент;

– в топологической сортировке;

– для поиска точек сочленения, мостов;

– для преобразования синтаксического дерева в строку (любую: префиксную, инфиксную, обратную польскую);

– в различных расчётах на графах (как часть алгоритма Диница поиска максимального потока или других).

Алгоритм поиска в глубину плохо поддаётся распараллеливанию, некоторые исследователи называют его «кошмар для параллельной обработки» [20].

Таким образом алгоритм поиска в глубину является одним из важнейших и наиболее широко применимых классических алгоритмов на графах.

4.4 Описание функциональной схемы алгоритма

Функциональная схема программы представлена в приложении В. В рамках разработки функциональной схемы были выделены 5 ключевых блоков:

1 Считывание графа из файла.

2 Запуск алгоритма на графах.

3 Получение данных о времени работы алгоритма.

4 Запись промежуточных данных о времени работы алгоритма.

5 Запись полноценных данных о времени работы алгоритма в файл.

Данная схема описывает функциональные возможности программного продукта.

4.5 Описание блок-схемы алгоритма

Блок схема алгоритма программы представлена в приложении Г. Блок-схема состоит из 5 ключевых функций:

1 Точка входа в программу – основная функция, осуществляющая управление остальными.

2 Функция замера времени работы алгоритма.

3 Функция поиска кратчайших расстояний от начальной вершины до всех вершин графа с помощью поиска в ширину.

4 Поиск цикла в графе с помощью поиска в глубину.

5 Рекурсивный поиск цикла в графе от конкретной вершины с помощью поиска в глубину.

Данная схема детально описывает реализацию алгоритмов, реализующих программное средство

4.6 Реализация программы

В данном подразделе представлена детальная реализация алгоритмов поиска в ширину и в глубину использованием языка программирования *C++*. Код включает реализацию алгоритмов, а также инструменты для замера времени, вывода подробной информации в консоль и сохранения результатов в формате *CSV*.

Функция замера времени работы функции (листинг А.2) предназначена для измерения времени выполнения переданной в качестве аргумента функции в наносекундах.

Шаги функции:

1 Сохранение текущего момента времени с помощью средств библиотеки <*crono*>.

2 Вызов переданной в качестве аргумента функции.

3 Сохранение текущего момента времени с помощью средств библиотеки <*chrono*>.

4 Вычисление времени, затраченного на выполнение функции-аргумента в наносекундах путём вычисления разницы между моментом времени после запуска функции-аргумента и моментом до запуска функции-аргумента.

5 Возврат вычисленных результатов.

Функция поиска наикратчайшего пути до любой вершины из начальной вершины (листинг А.3) предназначена для поиска наикратчайшего пути с помощью поиска в ширину.

Шаги функции:

1 Инициализация массивов для хранения уже посещённых вершин и для хранения расстояний до каждой вершины.

2 Инициализация очереди для обхода графа.

3 Добавление начальной вершины в очередь.

4 Инициализация начальных значений для работы алгоритма.

5 Вычисление наикратчайшего пути с помощью нескольких циклов. Во внешнем цикле происходит выборка вершины из начала очереди пока очередь не опустеет, во внутреннем происходит добавление каждой вершины смежной выбранной вершине в очередь, если выбранная вершина ещё не была рассмотрена.

6 Возврат массива расстояний от начальной вершины.

Функция поиска наикратчайшего пути до любой вершины в параллельной реализации (листинг А.4) выполняет те же шаги, но с небольшими изменениями:

1 Инициализация массивов выполняется аналогично, но массив посещённых вершин реализован через атомарные флаги для безопасного доступа из нескольких потоков.

2 Внешний цикл while обрабатывает текущий фронт обхода, пока он не опустеет.

3 Вместо последовательного обхода вершин фронта, множество вершин разбивается на блоки, которые обрабатываются параллельно в нескольких потоках:

– количество потоков определяется на основе аппаратной поддержки;

– каждый поток обрабатывает свой диапазон вершин в текущем фронте;

– для каждой вершины v в своём диапазоне поток проверяет всех соседей u.

4 При обработке соседей используется атомарная операция для безопасной проверки и установки флага посещения вершины. Если удалось атомарно установить флаг:

– Расстояние до вершины u устанавливается равным $distances[v] + 1$;

– Вершина добавляется в локальный буфер.

5 После обработки своего диапазона каждый поток защищённо объединяет свой локальный буфер в общий следующий фронт обхода с использованием мьютекса.

6 После завершения всех потоков текущий фронт обхода заменяется на следующий для последующей итерации.

Функция поиска наикратчайшего пути до любой вершины в реализации для вычисления на графическом процессоре (листинг А.5) имеет схожую структуру с параллельной реализацией для вычисления на центральном процессоре.

Функция поиска цикла в графе (листинг А.6) предназначена для определения является ли граф ациклическим с помощью поиска в глубину.

Шаги функции:

1 Пометить текущую вершину v как посещённую.

2 Для каждой смежной вершины u из списка соседей графа:

– если u ещё не посещена, рекурсивно вызвать функцию для u, передав текущую вершину v в качестве родителя;

– если рекурсивный вызов вернул false, немедленно вернуть false;

– если u уже посещена и при этом $u \neq q$, возвращаем false, так как найдена обратная дуга, указывающая на цикл в неориентированном графе.

3 Если обход всех соседей не выявил проблем, вернуть true.

Таким образом, представленный раздел охватывает детальную реализацию алгоритмов поиска в глубину и поиска в ширину для решения задач, связанных с графами на языке программирования *C++*. Включены также инструменты для замера времени выполнения программы.

5 СРАВНЕНИЕ ПРОИЗВОДИТЕЛЬНОСТИ ПРОЦЕССОРОВ

5.1 Условия проведения сравнения

Для наиболее объективного сравнения двух процессоров были соблюдены следующие условия:

1 Операционная система *Arch Linux* была установлена на *USB-флеш-накопитель*. Это позволяет сравнивать производительность вычислений, не заботясь об остановке различных фоновых процессов и различиях в скорости *SSD-накопителей*.

2 Лимит оперативной памяти для операционной системы был установлен равным 8 гигабайт. Это позволяет сравнивать производительность вычислений, не заботясь о разнице в объеме оперативной памяти между двумя компьютерами.

3 Каждый замер проведён 10 раз, в качестве значения времени выполнения было взято среднее арифметическое всех значений времени выполнения для данного размера графа.

Для подведения итогов методологии сравнения процессоров важно отметить, что применённый подход обеспечивает высокий уровень объективности за счёт минимизации внешних факторов, влияющих на измерения. Такая методика позволяет получить сбалансированную и репрезентативную оценку производительности процессоров в контролируемых условиях, что соответствует ключевым принципам корректного сравнительного анализа вычислительных систем.

5.2 Основные сведения о тестировании

Далее будут рассмотрены результаты экспериментов по сравнению производительности алгоритмов на графах на процессорах *AMD Ryzen 5 3500U* и *AMD Ryzen 7 7700*.

При проведении экспериментов замер времени выполнения алгоритма осуществлялся с использованием библиотеки *<chrono>*. Результаты экспериментов были сохранены в файл в формате *CSV* для последующего анализа. Для сравнения были выбраны графы размером 1000, 2000, 5000, 10000, 20000, 50000 и 100000 вершин. Такой выбор размеров графов позволяет исследовать влияние размера графа на эффективность выполнения алгоритма процессором.

Исследования начались с проведения тестов на обоих компьютерах с включенной гиперпоточностью. После этого функциональность гиперпоточности была отключена, и тесты были повторно запущены для получения дополнительных данных.

Пример полученных данных представлен в таблице 5.1.

Таблица 5.1 – Результаты эксперимента по измерению производительности для алгоритма поиска в ширину на *iGPU*

Количество вершин в графе	Время выполнения алгоритма, мкс	
	AMD Ryzen 5 3500U	*AMD Ryzen 7 7700*
1000	750.6472	383.6374
2000	759.5520	407.3502
5000	702.0163	336.1356
10000	894.7784	428.9151
20000	1562.4891	1055.7595
50000	11823.9889	4658.8191
100000	37765.5262	19427.4481

Стоит отметить, что контринтуитивные на первый взгляд результаты измерений, такие как 702.0163 мкс для 5000 вершин и 759.5520 мкс для 2000 вершин, объясняются сутью алгоритма: граф с большим количеством вершин может быть более плотно связан, что в свою очередь ведёт к более быстрому выполнению алгоритма, так как ему нужно проходить в среднем более короткий путь. Корректность алгоритма подтверждает соответствующие результаты для обоих процессоров. Среднее отношение времени выполнения для данного алгоритма ≈ 1.994.

5.3 Анализ результатов

Графики сравнения производительности обоих процессоров представлены в приложении Г. На рисунках Г.1–Г.2 находятся графики, относящиеся к результатам тестов для алгоритма поиска в глубину, на рисунках Г.3–Г.7 – для алгоритма поиска в ширину в однопоточной реализации, многопоточной реализации и в реализации для *iGPU*. Для алгоритма поиска в глубину представлены графики тестирования с включённой гиперпоточностью и с без неё. Для алгоритма поиска в ширину представлены графики тестирования в однопоточном, многопоточном и *GPU* режимах, для всех режимов помимо режима выполнения на графическом процессоре представлены варианты графиков с включённой гиперпоточностью и без неё.

На оси абсцисс отображены размеры тестируемых графов (*Number of Nodes)*. Синий график представляет собой время выполнения на *AMD Ryzen 3500U*. Оранжевый график представляет собой время выполнения на *AMD Ryzen 7 7700*.

Влияние размера графа на эффективность вычислений на различных процессора является важным аспектом анализа. На небольших графах (например, $100, 200, 500$ вершин) время выполнения алгоритма сравнительно

близко. Таким образом, при малых размерах графа аппаратные преимущества *Ryzen 7 7700*, такие как большее количество ядер, более высокая тактовая частота и более современная архитектура не имеют выраженного преимущества над *Ryzen 5 3500U*, и их время выполнения становится близким. Сравнительный выигрыш начинает проявляться при увеличении объёмов данных, когда вычислительные ресурсы *Ryzen 7 7700* начинают полноценно использоваться, а накладные расходы на управление уменьшаются относительно общего времени вычислений.

В таблице 5.2 также представлено среднее отношение времени выполнения для каждого из алгоритмов.

Таблица 5.2 – Среднее отношение времени выполнения для протестированных алгоритмов

Алгоритм	Среднее отношение времени выполнения
DFS с гиперпоточностью	6.342
DFS без гиперпоточности	6.594
Однопоточная реализация *BFS* с гиперпоточностью	4.998
Однопоточная реализация *BFS* без гиперпоточности	4.821
Многопоточная реализация *BFS* с гиперпоточностью	3.878
Многопоточная реализация *BFS* без гиперпоточности	4.310
Реализация *BFS* для *iGPU*	1.994

Процессор *AMD Ryzen 5 3500U* оснащён 4 физическими ядрами и 8 логическими потоками, что характерно для мобильных решений среднего уровня. В то время как *AMD Ryzen 7 7700* предлагает 8 физических и 16 логических ядер, ориентируясь на производительные настольные системы.

Результаты тестов показывают, что при одинаковой нагрузке *Ryzen 7 7700* стабильно демонстрирует значительно более быстрое выполнение алгоритмов *BFS* и *DFS*. Это связано не только с большим количеством ядер, но и с более высокой тактовой частотой, улучшенной микроархитектурой и усовершенствованной системой кэширования. *Ryzen 7 7700* способен обрабатывать большие графы с меньшим временем отклика и более высокой пропускной способностью.

В случае *Ryzen 5 3500U* наблюдаются более высокие времена исполнения, особенно на масштабных графах, что объясняется ограничениями мобильной архитектуры и меньшим числом вычислительных ресурсов. Кроме того, возможности памяти и внутренняя шина у *Ryzen 5 3500U* уступают *Ryzen 7 7700*, что также сказывается на быстродействии при больших объемах

данных.

Особо стоит отметить, что для обоих процессоров характерно увеличение времени исполнения алгоритмов пропорционально росту размера графов. Однако *Ryzen 7 7700* показывает более плавный рост времени, демонстрируя лучшую масштабируемость и оптимизацию работы с данными.

Таким образом, сравнительный анализ подчёркивает существенные преимущества *Ryzen 7 7700* в задачах классических алгоритмов на графах, что делает его более подходящим выбором для ресурсоёмких вычислительных приложений. В то же время *Ryzen 5 3500U* остаётся хорошим вариантом для сценариев с умеренными требованиями к производительности и ограничениями по энергопотреблению.

ЗАКЛЮЧЕНИЕ

В рамках данного курсового проекта было проведено исследование и сравнительный анализ производительности процессоров *AMD Ryzen 5 3500U* и *AMD Ryzen 7 7700* при выполнении алгоритмов поиска в ширину (*BFS*) и поиска в глубину (*DFS*). В ходе работы изучались особенности архитектуры и вычислительных возможностей обоих процессоров, а также их влияние на эффективность выполнения алгоритмов обхода графа.

Основным объектом исследования стали реализации алгоритмов *BFS* и *DFS* с акцентом на измерение времени выполнения и использование ресурсов процессоров в различных условиях. Особое внимание уделялось оценке масштабируемости и поведению алгоритмов при увеличении объёма обрабатываемых данных.

Полученные результаты показали, что более производительный AMD Ryzen 7 7700 обеспечивает значительный прирост скорости выполнения по сравнению с AMD Ryzen 5 3500U при обработке больших графов. При этом на малых объёмах данных разница в производительности менее заметна, что связано с накладными расходами и особенностями доступа к памяти.

Таким образом, исследование демонстрирует влияние архитектурных особенностей и количества ядер процессоров на эффективность алгоритмов *BFS* и *DFS*. Полученные выводы представляют практическую ценность для выбора аппаратных решений при реализации графовых алгоритмов на современных вычислительных платформах и могут служить основой для дальнейших оптимизаций выполнения классических алгоритмов на графах.

СПИСОК ЛИТЕРАТУРНЫХ ИСТОЧНИКОВ

[1] Доманов, А. Т. Стандарт предприятия / А. Т. Доманов, Н. И. Сорока. – Минск: БГУИР, 2024. – 178 с.

[2] Technical city [Электронный ресурс]. – Режим доступа: https://technical.city/ru/cpu/Ryzen-5-3500U-protiv-Ryzen-7-7700. – Дата доступа: 20.09.2025.

[3] AnandTech [Электронный ресурс]. – Режим доступа: https://www.anandtech.com/print/10578/amd-zen-microarchitecture-dual-schedulers-micro-op-cache-memory-hierarchy-revealed. – Дата доступа: 17.12.2019.

[4] AnandTech [Электронный ресурс]. – Режим доступа: https://www.anandtech.com/show/10578/amd-zen-microarchitecture-dual-schedulers-micro-op-cache-memory-hierarchy-revealed/2. – Дата доступа: 17.12.2019.

[5] Forbes [Электронный ресурс]. – Режим доступа: https://www.forbes.com/sites/antonyleather/2018/01/07/amd-confirms-new-zen-ryzen-cpus-for-april-2018-x470-chipset-threadripper-and-apus-inbound-too/. – Дата доступа: 20.09.2025.

[6] PC World [Электронный ресурс]. – Режим доступа: https://www.pcworld.com/article/3246211/computers/amd-reveals-ryzen-2-threadripper-2-7nm-navi-and-more-in-ces-blockbuster.html. – Дата доступа: 13.01.2018.

[7] Phoronix [Электронный ресурс]. – Режим доступа: https://www.phoronix.com/news/AMD-Zen-2-New-Instructions. – Дата доступа: 20.09.2025.

[8] TechPowerUp [Электронный ресурс]. – Режим доступа: https://www.techpowerup.com/256478/amd-zen-2-has-hardware-mitigation-for-spectre-v4. – Дата доступа: 20.09.2025.

[9] AnandTech [Электронный ресурс]. – Режим доступа: https://www.anandtech.com/show/16214/amd-zen-3-ryzen-deep-dive-review-5950x-5900x-5800x-and-5700x-tested. – Дата доступа: 12.01.2020.

[10] AnandTech [Электронный ресурс]. – Режим доступа: https://web.archive.org/web/20220926130934/https://www.anandtech.com/show/17585/amd-zen-4-ryzen-9-7950x-and-ryzen-5-7600x-review-retaking-the-high-end. – Дата доступа: 26.09.2022.

[11] Pagetable [Электронный ресурс]. – Режим доступа: http://www.pagetable.com/?p=25. – Дата доступа: 18.07.2011.

[12] musl – Introduction [Электронный ресурс]. – Режим доступа: https://www.musl-libc.org/intro.html. – Дата доступа: 26.10.2025.

[13] StatCounter – Desktop Operating Systems Market Share Worldwide [Электронный ресурс]. – Режим доступа: https://gs.statcounter.com/os-market-share/desktop. – Дата доступа: 26.10.2025.

[14] kernel.org [Электронный ресурс]. – Режим доступа: https://www.kernel.org/pub/linux/kernel/Historic/old-versions/RELNOTES-0.12. – Дата доступа: 27.10.2025.

[15] Linux-Kernel Archive [Электронный ресурс]. – Режим доступа: https://lkml.indiana.edu/hypermail/linux/kernel/1107.2/01843.html. – Дата доступа: 27.10.2025.

[16] ziglang.org [Электронный ресурс]. – Режим доступа: https://ziglang.org/#Zig-competes-with-C-instead-of-depending-on-it. – Дата доступа: 27.10.2025.

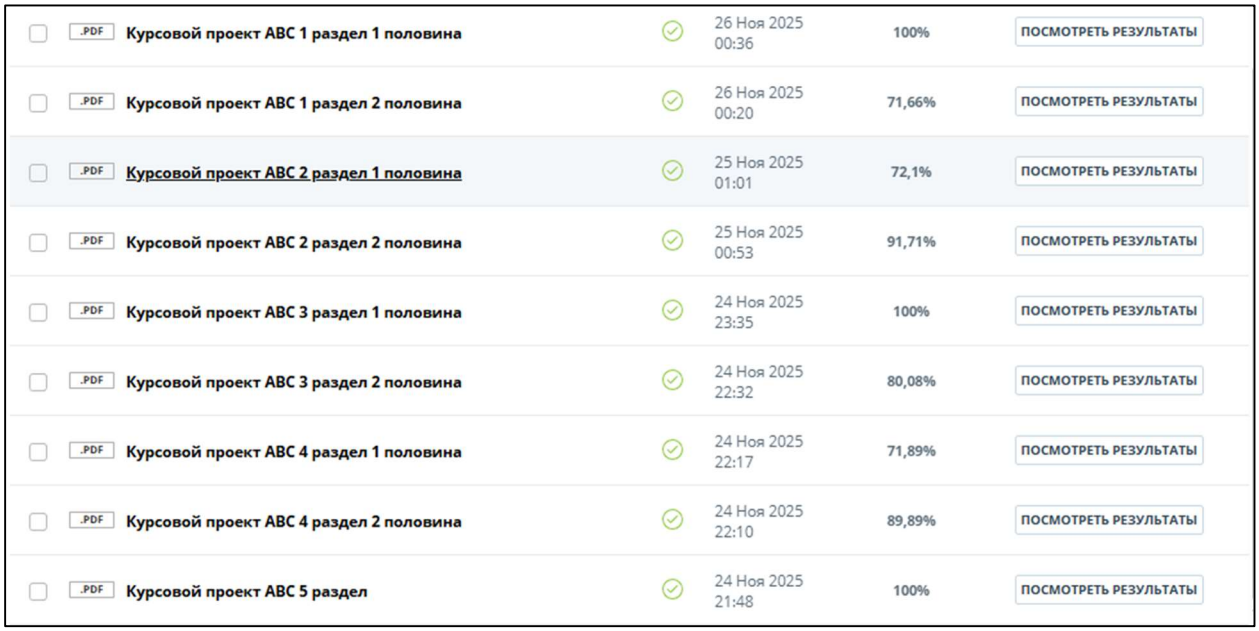
[17] PyPI · The Python Package Index [Электронный ресурс]. – Режим доступа: https://pypi.org/. – Дата доступа: 27.10.2025.

[18] HIP 7.2.0 Documentation [Электронный ресурс]. – Режим доступа: https://rocm.docs.amd.com/projects/HIP/en/develop/what_is_hip.html. – Дата доступа: 29.10.2025.

[19] Алгоритмы: построение и анализ / Кормен Т. [и др.]. – М. : Издательский дом «Вильямс», 2013. – 1328 с.

[20] Algorithms and Data Structures: The Basic Toolbox. – Mehlhorn, Kurt; Sanders, Peter, 2008.

ПРИЛОЖЕНИЕ А
(обязательное)
Справка о проверке на заимствования



.PDF	Курсовой проект АВС 1 раздел 1 половина	26 Ноя 2025 00:36	100%	ПОСМОТРЕТЬ РЕЗУЛЬТАТЫ
.PDF	Курсовой проект АВС 1 раздел 2 половина	26 Ноя 2025 00:20	71,66%	ПОСМОТРЕТЬ РЕЗУЛЬТАТЫ
.PDF	Курсовой проект АВС 2 раздел 1 половина	25 Ноя 2025 01:01	72,1%	ПОСМОТРЕТЬ РЕЗУЛЬТАТЫ
.PDF	Курсовой проект АВС 2 раздел 2 половина	25 Ноя 2025 00:53	91,71%	ПОСМОТРЕТЬ РЕЗУЛЬТАТЫ
.PDF	Курсовой проект АВС 3 раздел 1 половина	24 Ноя 2025 23:35	100%	ПОСМОТРЕТЬ РЕЗУЛЬТАТЫ
.PDF	Курсовой проект АВС 3 раздел 2 половина	24 Ноя 2025 22:32	80,08%	ПОСМОТРЕТЬ РЕЗУЛЬТАТЫ
.PDF	Курсовой проект АВС 4 раздел 1 половина	24 Ноя 2025 22:17	71,89%	ПОСМОТРЕТЬ РЕЗУЛЬТАТЫ
.PDF	Курсовой проект АВС 4 раздел 2 половина	24 Ноя 2025 22:10	89,89%	ПОСМОТРЕТЬ РЕЗУЛЬТАТЫ
.PDF	Курсовой проект АВС 5 раздел	24 Ноя 2025 21:48	100%	ПОСМОТРЕТЬ РЕЗУЛЬТАТЫ

ПРИЛОЖЕНИЕ Б
(обязательное)
Листинг программного кода

Листинг Б.1 – Зависимости программы

```
#include <hip/hip_runtime.h>
#include <chrono>
#include <cstdint>
#include <ratio>
#include <format>
#include <fstream>
```

Листинг Б.2 – Функция замера времени работы функции

```
uint64_t benchmark(void (*func)()) {
    auto start = std::chrono::high_resolution_clock::now();
    func();
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<uint64_t, std::nano> duration = end - start;
    return duration.count();
}
```

Листинг Б.3 – Функция поиска наикратчайшего пути до любой вершины из начальной вершины

```
vector<int64_t> bfs(graph_t& graph) {
    auto used = vector<bool>(graph.size(), false);
    auto distances = vector<int64_t>(graph.size(), -1);
    queue<int64_t> q;
    q.push(0);
    used[0] = true;
    distances[0] = 0;

    while (!q.empty()) {
        int64_t cur = q.front();
        q.pop();

        for (auto neighbor : graph[cur]) {
            if (!used[neighbor]) {
                q.push(neighbor);
                used[neighbor] = true;
                distances[neighbor] = distances[cur] + 1;
            }
        }
    }

    return distances;
}
```

Листинг Б.4 – Функция поиска наикратчайшего пути до любой вершины в параллельной реализации

```
vector<int64_t> parallel_bfs(graph_t& graph) {
    int n = graph.size();
    vector<std::atomic<bool>> used(n);
    vector<int64_t> distances(n, -1);

    for (auto& u : used) {
        u.store(false);
    }
```

```
    vector<int64_t> frontier = {0};
    used[0].store(true);
    distances[0] = 0;

    std::mutex mtx;

    while (!frontier.empty()) {
        vector<int64_t> next_frontier;

        auto worker = [&](int start, int end) {
            vector<int64_t> local_next;
            for (int i = start; i < end; ++i) {
                int64_t v = frontier[i];
                for (int64_t u : graph[v]) {
                    bool expected = false;
                    if (used[u].compare_exchange_strong(expected, true)) {
                        distances[u] = distances[v] + 1;
                        local_next.push_back(u);
                    }
                }
            }

            if (!local_next.empty()) {
                std::lock_guard<std::mutex> lock(mtx);
                next_frontier.insert(next_frontier.end(), local_next.begin(),
                                     local_next.end());
            }
        };

        int num_threads = std::thread::hardware_concurrency();
        if (num_threads == 0) {
            num_threads = 4;
        }

        vector<std::thread> threads;
        int block_size = (int)frontier.size() / num_threads + 1;
        for (int t = 0; t < num_threads; ++t) {
            int start = t * block_size;
            int end = std::min((t + 1) * block_size, (int)frontier.size());
            if (start >= end) {
                break;
            }
            threads.emplace_back(worker, start, end);
        }

        for (auto& th : threads) {
            th.join();
        }

        frontier = std::move(next_frontier);
    }

    return distances;
}
```

Листинг Б.5 – Функция поиска наикратчайшего пути до любой вершины в реализации для вычисления на графическом процессоре

```
__global__ void bfs_kernel(int n, const int* row_offsets,
                           const int* col_indices, int* distances,
```

```
                            int* frontier, int frontier_size, int*
next_frontier,
                            int* next_frontier_size) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < frontier_size) {
        int u = frontier[tid];
        int dist_u = distances[u];
        int start = row_offsets[u];
        int end = row_offsets[u + 1];
        for (int i = start; i < end; i++) {
            int v = col_indices[i];
            if (atomicCAS(&distances[v], INF, dist_u + 1) == INF) {
                int pos = atomicAdd(next_frontier_size, 1);
                next_frontier[pos] = v;
            }
        }
    }
}

std::vector<int> bfs_hip(int n, const graph_t& adj_list, int root) {
    std::vector<int> row_offsets(n + 1, 0);
    int edge_count = 0;
    for (int i = 0; i < n; i++) {
        row_offsets[i + 1] = row_offsets[i] + adj_list[i].size();
        edge_count += adj_list[i].size();
    }
    std::vector<int> col_indices(edge_count);
    int idx = 0;
    for (int i = 0; i < n; i++) {
        for (int v : adj_list[i]) {
            col_indices[idx++] = v;
        }
    }

    int *d_row_offsets, *d_col_indices, *d_distances;
    hipMalloc(&d_row_offsets, (n + 1) * sizeof(int));
    hipMalloc(&d_col_indices, edge_count * sizeof(int));
    hipMalloc(&d_distances, n * sizeof(int));

    hipMemcpy(d_row_offsets, row_offsets.data(), (n + 1) * sizeof(int),
              hipMemcpyHostToDevice);
    hipMemcpy(d_col_indices, col_indices.data(), edge_count * sizeof(int),
              hipMemcpyHostToDevice);

    std::vector<int> distances(n, INF);
    distances[root] = 0;

    int *d_frontier, *d_next_frontier, *d_next_frontier_size;
    hipMalloc(&d_frontier, n * sizeof(int));
    hipMalloc(&d_next_frontier, n * sizeof(int));
    hipMalloc(&d_next_frontier_size, sizeof(int));

    hipMemcpy(d_frontier, &root, sizeof(int), hipMemcpyHostToDevice);
    int frontier_size = 1;
    hipMemcpy(d_distances, distances.data(), n * sizeof(int),
              hipMemcpyHostToDevice);

    while (frontier_size > 0) {
        hipMemset(d_next_frontier_size, 0, sizeof(int));

        int block_size = 256;
        int grid_size = (frontier_size + block_size - 1) / block_size;
```

```
        hipLaunchKernelGGL(bfs_kernel, dim3(grid_size), dim3(block_size), 0,
0,
                           n, d_row_offsets, d_col_indices, d_distances,
                           d_frontier, frontier_size, d_next_frontier,
                           d_next_frontier_size);

        hipMemcpy(&frontier_size, d_next_frontier_size, sizeof(int),
                  hipMemcpyDeviceToHost);

        std::swap(d_frontier, d_next_frontier);
    }

    hipMemcpy(distances.data(), d_distances, n * sizeof(int),
              hipMemcpyDeviceToHost);

    hipFree(d_row_offsets);
    hipFree(d_col_indices);
    hipFree(d_distances);
    hipFree(d_frontier);
    hipFree(d_next_frontier);
    hipFree(d_next_frontier_size);

    return distances;
}
```

Листинг Б.6 – Функция поиска цикла в графе

```
bool dfs(graph_t& graph, int64_t v, int64_t p = -1) {
    used[v] = true;

    for (int64_t u : graph[v]) {
        if (!used[u]) {
            if (!dfs(graph, u, v)) {
                return false;
            }
        } else if (u != p) {
            return false;
        }
    }

    return true;
}

bool has_cycles(graph_t& graph) {
    for (auto i = 0; i < graph.size(); i++) {
        if (!used[i]) {
            if (!dfs(graph, i)) {
                return true;
            }
        }
    }
    return false;
}
```

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

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

ПРИЛОЖЕНИЕ Д
(обязательное)
Графики сравнения производительности

ПРИЛОЖЕНИЕ Е
(обязательное)
Диаграмма развёртывания

ПРИЛОЖЕНИЕ Ж
(обязательное)
Ведомость курсового проекта