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 I7 12700H И AMD RYZEN 7 5800H НА ОСНОВЕ АЛГОРИТМА БИНАРНОГО И ЛИНЕЙНОГО ПОИСКА С ПАРАЛЛЕЛИЗАЦИЕЙ ВЫЧИСЛЕНИЙ

БГУИР КП 6-05-0612-02 013 ПЗ

Студент А. И. Кохан

Руководитель А. Н. Марков

Нормоконтролер А. А. Калиновская

Минск 2025

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В современной вычислительной технике производительность процессоров является критически важным фактором, определяющим эффективность решения широкого круга задач. Сравнительный анализ процессоров с различной архитектурой, таких как *Intel Core i7-12700H и AMD Ryzen 7 5800H*, представляет значительный практический интерес, особенно в контексте оптимизации алгоритмов. Алгоритмы поиска, в частности бинарный и линейный, относятся к фундаментальным и широко применяются в информатике для обработки и анализа данных.

Актуальность данного исследования обусловлена растущими требованиями к скорости выполнения операций с большими объемами информации. Использование параллельных вычислений позволяет значительно ускорить работу алгоритмов, что делает изучение эффективности распараллеливания на многопоточных процессорах чрезвычайно важным.

Цель работы заключается в проведении сравнительного анализа производительности процессоров *Intel Core i7-12700H* и *AMD Ryzen 7 5800H* при реализации алгоритмов бинарного и линейного поиска с использованием параллелизации вычислений. Для достижения поставленной цели необходимо решить следующие задачи:

1 Изучение архитектурных особенностей и технических характеристик процессоров *Intel Core i7-12700H и AMD Ryzen 7 5800H*.

2 Разработка программной реализации алгоритмов бинарного и линейного поиска с поддержкой параллельных вычислений на языке *C++*.

3 Проведение серии вычислительных экспериментов с варьированием размеров входных данных и количества используемых потоков.

4 Анализ полученных результатов производительности, включая время выполнения и эффективность распараллеливания.

В результате исследования будут получены данные о времени выполнения алгоритмов на каждом из процессоров, что позволит провести объективное сравнение их производительности и сделать выводы об эффективности использования их многопоточных возможностей для задач поиска.

Данный курсовой проект выполнен мной лично, проверен на заимствования, процент оригинальности составляет 77% (отчет о проверке на заимствования прилагается).

1 АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНОЙ СИСТЕМЫ

1.1 Структура и архитектура вычислительной системы

Современные мобильные процессоры *AMD Ryzen* и *Intel Core* представляют собой высокотехнологичные продукты, ориентированные на обеспечение высокой производительности в условиях ограниченного энергопотребления. Сравнительный анализ их эффективности является актуальной задачей, особенно применительно к вычислительно сложным алгоритмам.

Объектами данного исследования выступают два современных 8-ядерных процессора для игровых ноутбуков и рабочих станций: *AMD Ryzen 7 5800H* на микроархитектуре *Zen 3* и *Intel Core i7-12700H* на гибридной архитектуре *Alder Lake-H*. Их ключевые характеристики представлены в таблице 1.1 [1].

Таблица 1.1 – Основные характеристики выбранных процессоров

Характеристики	*AMD Ryzen 7 5800H*	*Intel Core i7-12700H*
Кодовое имя архитектуры	*Cezanne-H (Zen 3)*	*Alder Lake-H*
Дата выпуска	*Январь 2021г*	*Январь 2022г*
Физические ядра	*8*	*14*
Количество потоков	*16*	*20*
L1 кэш	*64K (на ядро)*	*80K (на ядро)*
L2 кэш	*512K (на ядро)*	*1.25 Мб (на ядро)*
L3 кэш	*16 Мб (всего)*	*24 Мб (всего)*
Базовая частота	*3.2 ГГц*	*2.3 ГГЦ*
Максимальная частота	*4.4 ГГц*	*4.7 ГГц*
Техпроцесс	*7 нм*	*10 нм*
Размер кристалла	*156 мм^2*	*217 мм^2*
Энергоэффективность	*9.23*	*13.63*
Встроенная графика	*AMD Radeon RX Vega 8*	*Intel Iris Xe 96*
Тип памяти	*DDR4*	*DDR4*
Архитектура	*x86-64*	*x86-64*
Hyper-Threading/SMT	+	+
Теплопакет (*TDP)*	*54 Вт*	*45 Вт*

AMD Ryzen 7 5800H – мобильный процессор с 8 ядрами, выпущенный в январе 2021 года [2]. Он является частью линейки *Ryzen 7*, использующей архитектуру *Zen 3 (Cezanne)* с *Socket FP6*. Благодаря технологии *AMD Simultaneous Multithreading (SMT)* количество ядер фактически удваивается и составляет 16 потоков. *Ryzen 7 5800H* имеет 16 МБ кэш-памяти L3 и работает на частоте 3,2 ГГц по умолчанию, но может увеличиваться до 4,4 ГГц, в зависимости от рабочей нагрузки. AMD производит *Ryzen 7 5800H* на 7-нм производственном узле с использованием 10 700 миллионов транзисторов.

Кремниевый кристалл чипа изготавливается не на *AMD*, а на литейном заводе *TSMC*. На *Ryzen 7 5800H* множитель заблокирован, что ограничивает его разгонный потенциал. При *TDP* 45 Вт *Ryzen 7 5800H* потребляет типичные для современного ПК уровни мощности. Процессор *AMD* поддерживает память *DDR4* с двухканальным интерфейсом. Самая высокая официально поддерживаемая скорость памяти составляет 4266 МТ/с. Для связи с другими компонентами в системе *Ryzen 7 5800H* использует соединение *PCI-Express Gen 3*. Этот процессор оснащен интегрированным графическим решением *Radeon Vega 8*.

Аппаратная виртуализация доступна на *Ryzen 7 5800H*, что значительно повышает производительность виртуальной машины. Программы, использующие *Advanced Vector Extensions (AVX)*, могут выполняться на этом процессоре, повышая производительность приложений, требующих больших вычислительных ресурсов. Помимо *AVX, AMD* также включает более новый стандарт *AVX2*, но не *AVX-512*.

Intel Core i7-12700H – мобильный процессор с 14 ядрами, выпущенный в январе 2022 года [3]. Он является частью линейки *Core i7*, использующей архитектуру *Alder Lake-H* с *BGA 1744*. Доступна технология Intel Hyper-Threading, которая значительно удваивает количество ядер *P*-ядер до 20 потоков. *Core i7-12700H* имеет 24 МБ кэш-памяти *L3* и работает на частоте 2,3 ГГц по умолчанию, но может разгоняться до 4,7 ГГц, в зависимости от рабочей нагрузки. *Intel* строит *Core i7-12700H* по 10-нм техпроцессу, количество транзисторов неизвестно. На *Core i7-12700H* множитель заблокирован, что ограничивает его возможности по разгону.

Обладая *TDP* 45 Вт, *Core i7-12700H* потребляет типичный для современного ПК уровень мощности. Процессор *Intel* поддерживает память *DDR4* и *DDR5* с двухканальным интерфейсом. Для связи с другими компонентами машины *Core i7-12700H* использует соединение *PCI-Express Gen 4*. В этом процессоре используется интегрированное графическое решение *Iris Xe 96EU*.

Аппаратная виртуализация доступна на *Core i7-12700H*, что значительно повышает производительность виртуальной машины. Кроме того, поддерживается виртуализация *IOMMU* (сквозная передача *PCI*), что позволяет гостевым виртуальным машинам напрямую использовать оборудование узла. Программы, использующие *Advanced Vector Extensions (AVX),* будут выполняться на этом процессоре, повышая производительность приложений, требующих больших вычислительных ресурсов. Помимо *AVX*, *Intel* также включает более новый стандарт *AVX2*, но не *AVX-512*.

1.2 История, версии и достоинства

AMD Ryzen 7 5800H, выпущенный в начале 2021 года, является частью серии Ryzen 5000, построенной на микроархитектуре *Zen 3* (кодовое имя *Cezanne*) [2]. Это поколение стало важным этапом для *AMD*, ознаменовавшим переход на новую, более эффективную архитектуру ядра. По сравнению с

предыдущим поколением *Zen 2*, архитектура *Zen 3* принесла существенное увеличение производительности на тактовую частоту за счет унифицированного кэша *L3* и переработанного исполнительного модуля. *Ryzen 7 5800H* унаследовал ключевые достоинства *Zen 3*: высокую энергоэффективность 7-нм техпроцесса, мощную многопоточную производительность благодаря 8 ядрам и 16 потокам, а также поддержку современных стандартов памяти. Его основными достоинствами являются выдающаяся многопоточная производительность в рамках заданного *TDP* и зрелость платформы на базе чипсета *Socket FP6*.

Intel Core i7-12700H, анонсированный в начале 2022 года, представляет собой процессор 12-го поколения (*Alder Lake-H*) и знаменует собой революционный переход *Intel* на гибридную гетерогенную архитектуру [3]. Это первое поколение, сочетающее высокопроизводительные ядра (*P-cores*) и энергоэффективные ядра (*E-cores*) в рамках одного кристалла. Данная архитектура требует тесной интеграции с операционной системой для эффективного планирования задач (технология Thread Director). Ключевыми достоинствами *i7-12700H* являются его высочайшая многопоточная производительность, обеспечиваемая 14 ядрами (6P+8E) и 20 потоками, графическое решение *Iris Xe* по сравнению с предыдущими поколениями. Процессор демонстрирует значительный прирост производительности, особенно в многопоточных задачах.

1.3 Обоснование выбора вычислительной системы

Выбор процессоров *AMD Ryzen 7 5800H* и *Intel Core i7-12700H* для сравнительного анализа производительности при реализации алгоритмов бинарного и линейного поиска с параллелизацией вычислений обосновывается следующими факторами:

1 Разнородная многопоточная архитектура – процессор *Intel Core i7-12700H* использует гибридную архитектуру *Performance-cores* (*P-cores*) и *Efficient-cores* (*E-cores*), что позволяет исследовать эффективность различных типов ядер при параллельном выполнении алгоритмов поиска. *AMD Ryzen 7 5800H*, в свою очередь, обладает однородной 8-ядерной архитектурой, что предоставляет возможность сравнить два разных подхода к организации многопоточности.

2 Поддержка современных технологий параллелизма – оба процессора поддерживают расширенные наборы инструкций для многопоточных вычислений и обладают развитыми системами кэширования. Это особенно важно для алгоритмов поиска, где эффективное использование кэша разных уровней может значительно повлиять на производительность при работе с большими объемами данных.

3 Актуальность и распространенность – данные процессоры широко используются в современных высокопроизводительных ноутбуках и рабочих станциях, что делает их сравнение практически значимым для разработчиков и исследователей, работающих с задачами параллельного поиска и обработки

данных.

4 Доступность для тестирования – наличие доступного оборудования на базе этих процессоров позволяет провести натурный эксперимент и получить реальные данные о производительности при различных сценариях нагрузки, что особенно важно для задач с параллелизацией вычислений.

5 Репрезентативность архитектур – сравнение процессоров разных архитектурных поколений (*Zen 3* у *AMD* и *Alder Lake* у *Intel*) позволяет оценить эволюцию подходов к параллельной обработке данных и сделать выводы об эффективности различных архитектурных решений для задач поиска.

Таким образом, выбор данных процессоров обусловлен их современными архитектурными особенностями, поддержкой технологий параллельных вычислений и практической значимостью для задач оптимизации алгоритмов поиска в многопоточных средах.

1.4 Анализ выбранной вычислительной системы для написания программы

Для сравнительного анализа производительности процессоров *Intel Core i7-12700H* и *AMD Ryzen 7 5800H* были использованы данные синтетических бенчмарков *Cinebench R23*, *Geekbench 6* и *PassMark*, а также специализированные тесты производительности. Эти тесты позволяют оценить вычислительные способности процессоров в различных режимах работы, что особенно важно для анализа алгоритмов бинарного и линейного поиска с параллелизацией вычислений [4].

Согласно данным тестирования, процессор *Intel Core i7-12700H* демонстрирует значительное преимущество в многопоточных рабочих нагрузках благодаря гибридной архитектуре, сочетающей 6 производительных *P-ядер* и 8 энергоэффективных *E-ядер*. В тесте *Cinebench R23* многоядерная производительность *i7-12700H* превышает показатели *Ryzen 7 5800H* на 25-30%, что составляет 17500 баллов против 13500 баллов у *AMD*. Это преимущество особенно важно для задач параллельной обработки данных, включая реализацию параллельных алгоритмов поиска.

В однопоточных тестах *Intel Core i7-12700H* также показывает лидерство, опережая *Ryzen 7 5800H* на 10-15% *в Geekbench 6 Single-Core*. Это объясняется более высокой тактовой частотой *P-ядер* и улучшенной архитектурой *Golden Cove*.

Тесты *PassMark* подтверждают общее преимущество *Intel* процессора: в многопоточном режиме *i7-12700H* набирает на 28% больше баллов (35000 против 27300), а в однопоточном – на 12% (3900 против 3480).

Таким образом, можно предположить, что *Intel Core i7-12700H* будет демонстрировать лучшую производительность в задачах параллельного поиска благодаря превосходству в многопоточных тестах и улучшенной работе с памятью. Однако архитектурные особенности *Ryzen 7 5800H*, в частности однородность ядер, могут обеспечить более предсказуемую производительность в определенных сценариях параллелизации.

2 ПЛАТФОРМА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

2.1 Структура и архитектура платформы

Для выполнения данного курсового проекта в качестве программной платформы была выбрана операционная система *Linux*. Далее будут рассмотрены основные характеристики, структура и архитектура ОС.

Операционная система *Linux* – это гибкая, масштабируемая и широко используемая платформа как в серверных, так и в пользовательских средах. Чтобы понимать, как она работает, важно разобраться в её архитектуре и ключевых компонентах. *Linux* построен на модульной архитектуре, где каждая часть выполняет свою функцию. Это позволяет системе быть гибкой и подстраиваться под конкретные задачи [5]. Вот основные компоненты и архитектурные особенности платформы *Linux*:

1 Ядро *Linux* – это основной компонент, управляющий аппаратным обеспечением и взаимодействием с другими компонентами ОС. Задачи включают управление памятью, процессами и устройствами. Ядро *Linux* представляет собой микроядро, которое обеспечивает базовые функции, включая планирование задач, управление памятью и взаимодействие с устройствами [6]. Особенностью *Linux* является модульное ядро. Это означает, что можно загружать функции в ядро "на лету", не перекомпилируя его. Модули – это отдельные части ядра (*.ko* файлы), которые загружаются при необходимости:

- *lsmod* – просмотр загруженных модулей;
- *modprobe* – загрузка нового модуля;
- *rmmod* – удаление модуля;
- *insmod* – ручная загрузка модуля из файла.

2 Системные библиотеки – это специальные программы, которые предоставляют интерфейс для доступа приложений к функциям ядра операционной системы. В *Linux* это наборы предварительно написанного кода, обеспечивающие необходимую функциональность для взаимодействия пользовательских приложений с операционной системой без прямого доступа к пространству ядра. Библиотека *C* (*glibc*) – наиболее часто используемая системная библиотека в *Linux*, предлагающая такие основные функции, как распределение памяти, операции ввода/вывода и манипуляция строками. Математическая библиотека (*libm*) – библиотека, предоставляющая математические функции, такие как тригонометрические, логарифмические и экспоненциальные операции. Библиотека потоков (*libpthread*) – библиотека, которая поддерживает возможности многопоточности в приложениях [7].

3 Утилиты и инструменты. Под системными утилитами понимают программы, отвечающие за выполнение отдельных специализированных задач управления. *Linux* поставляется с множеством утилит и инструментов командной строки, которые позволяют управлять системой, настраивать её, администрировать и разрабатывать приложения. Примеры таких утилит: *bash*, *grep*, *sed*, *awk*, *gcc*, *make* и многие другие [8].

4 Файловая система. Файловая система – это не просто способ хранения данных, а одна из основ архитектуры *Linux*. В отличие от *Windows, Linux* использует единую древовидную структуру каталогов, начинающуюся с корня ("/"). Основные уровни файловой иерархии:

- */bin* – основные двоичные файлы;
- */etc* – конфигурационные файлы;
- */home* – пользовательские данные;
- */var* – лог-файлы, динамические данные;
- */boot* – загрузочные файлы (*vmlinuz*, *initrd*);
- */usr* – второстепенные программы и библиотеки.

Linux поддерживает широкий спектр файловых систем: *ext4*, *XFS*, *Btrfs*, *ZFS* и другие. Современные дистрибутивы используют systemd-tmpfiles или такие технологии как *tmpfs* для управления временными директориями [9].

5 Драйверы устройств. Ядро предоставляет абстракции для работы с оборудованием. Гибкость происходит за счет модульной инфраструктуры, которая позволяет подгружать драйверы без перезагрузки ОС.

Самый общий взгляд на архитектуру *Unix* позволяет увидеть двухуровневую модель системы, состоящую из пользовательской и системной части (ядра), рассмотреть можно на рисунке 2.1. Ядро непосредственно взаимодействует с аппаратной частью компьютера, изолируя прикладные программы (процессы в пользовательской части операционной системы) от особенностей ее архитектуры. Ядро имеет набор услуг, предоставляемых прикладным программам посредством системных вызовов. Таким образом, в системе можно выделить два уровня привилегий: уровень системы (привилегии специального пользователя *root*) и уровень пользователя (привилегии всех остальных пользователей).

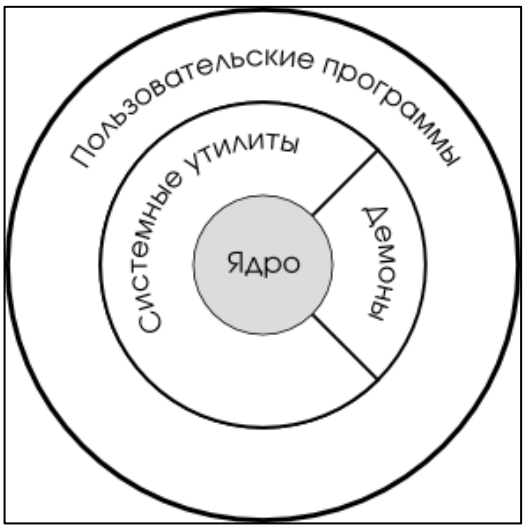

Рисунок 2.1 – Архитектура операционной системы Unix

Операционная система *Unix* обладает классическим монолитным ядром, в котором можно выделить следующие основные части. Это можно увидеть на рисунке 2.2.

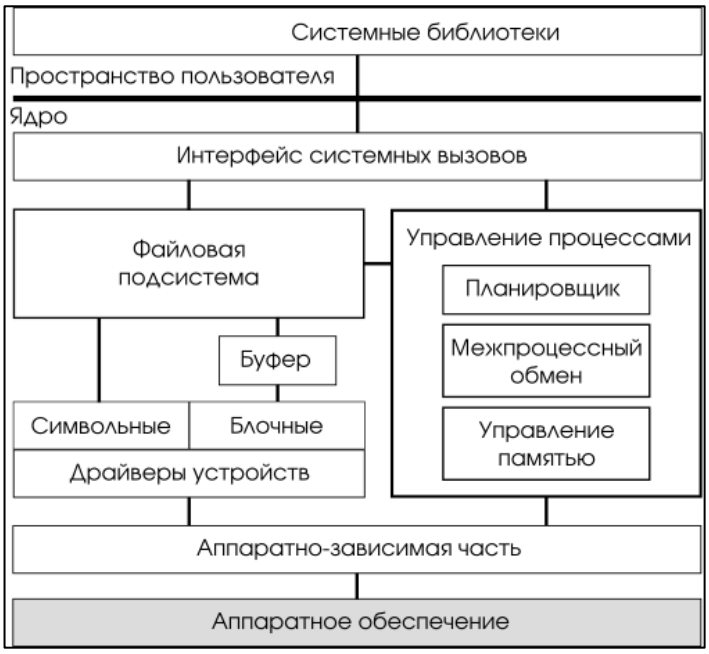

Рисунок 2.2 – Традиционная архитектура ядра *Unix*-системы

Использование общепринятых имен основных файлов и структуры каталогов существенно облегчает работу в операционной системе, её администрирование и переносимость. Некоторые из этих структур используются при запуске системы, некоторые – во время работы, но все они имеют большое значение для ОС в целом, а нарушение этой структуры может привести к неработоспособности системы или ее отдельных компонентов. Структура файловой системы проиллюстрирована на рисунке 2.3.

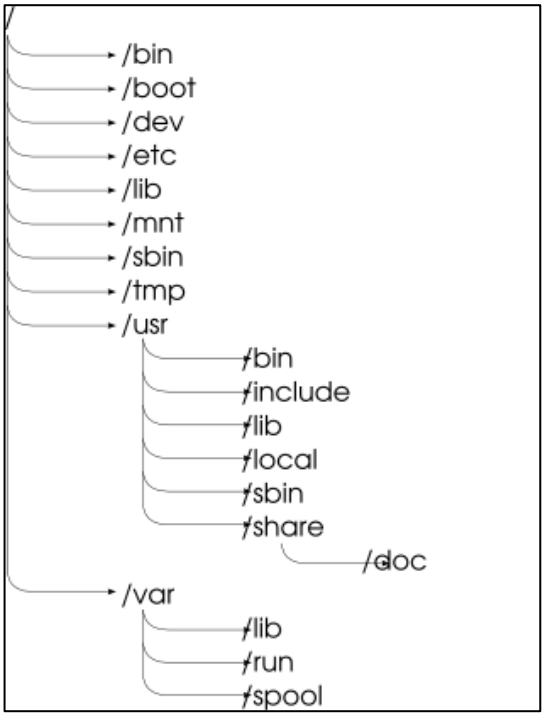

Рисунок 2.3 – Стандартные каталоги файловой системы в *Unix*

Операционная система *Linux* построена на надежной и гибкой архитектуре, что позволяет ей эффективно работать на всем, от настольных компьютеров до серверов. Ее модульная конструкция с мощным ядром и хорошо структурированным пользовательским пространством обеспечивает масштабируемость и безопасность.

Понимая компоненты архитектуры *Linux*, пользователи могут лучше настраивать, устранять неполадки в системе, в конечном итоге делая *Linux* идеальным выбором для широкого спектра вычислительных задач.

2.2 История, версии и достоинства

Unix начал свое развитие в 1969 году, в стенах американской компании *Bell Labs*, где группа программистов заложила его основу [10]. Чтобы сделать разработку *Unix* более независимой от конфигурации компьютера, Денис Ритчи и Кен Томпсон создали язык программирования *C*. Этот язык по-прежнему используется для написания ключевых компонентов большинства операционных систем.

К середине 70-х годов *Unix* широко распространился, хотя его использование было ограничено крупными предприятиями и университетами, так как персональные компьютеры в те времена были редкостью. *Linux* распространялся бесплатно, и каждая организация могла адаптировать систему под свои нужды, в результате чего возникли множество вариаций *Unix*, включая коммерческие.

В начале 80-х годов *Unix* выбрали как основную платформу для развития сетевого протокола *TCP/IP*, что значительно поспособствовало её укреплению на серверном рынке.

Тем не менее, коммерческие версии *Unix* замедляли прогресс, так как фирмы, производившие свои дистрибутивы, не позволяли распространять исходные коды, что препятствовало другим разработчикам пользоваться уже написанным кодом.

В 1982 году Ричард Столлман столкнулся с ограничением лицензий программного обеспечения, мешавшим обмену исходниками в научных кругах. Покинув рабочее место, в 1983 году он инициировал проект *GNU*, целью которого стало создание системы с открытым исходным кодом.

В 1987 году Эндрю Таненбаум создал учебную *Unix*-подобную систему *Minix*. Она служила учащимся инструментом для освоения основ ОС и состава её компонентов, распространявшаяся по свободной лицензии для изучения исходного кода.

Minix стал важным этапом для появления *Linux*, так как большинство студентов учились на её основе.

В начале 90-х, Линус Торвальдс, обучавшись на базе *Minix*, ощутил нехватку возможностей этой учебной системы и начал работу над собственной ОС, начиная с разработки системных вызовов. К лету 1991 года Линус Торвальдс, чрезвычайно увлёкся идеей написать совместимое с *Unix* ядро операционной системы для своего персонального компьютера с процессором

ставшей очень широко распространённой архитектуры *Intel 80386*. Прототипом для будущего ядра стала операционная система *Minix*. Название своему ядру он дал *freax*, но позже оно было изменено хозяином *ftp* сервера на *Linux* – гибрид имени создателя и слова *Unix*.

Важно отметить, что ядро операционной системы не является полной ОС с точки зрения пользователя. Оно служит для взаимодействия программ с оборудованием, управляет распределением памяти и процессорного времени.

Linux стал активно развиваться благодаря открытому доступу и поддержке сообщества программистов со всего мира.

Поскольку проекту *GNU* недоставало только ядра, роль *Linux* оказалась весьма кстати. Это ядро интегрировалось с ПО проекта *GNU*, формируя полноценную операционную систему, известную как *GNU/Linux*, а часто просто называемую *Linux*.

Если изначально *Linux* ориентировался на специалистов, то впоследствии дистрибутивы стали более доступными для обычных пользователей.

Дистрибутивы различаются не столько по графическому интерфейсу, сколько по области применения – для серверов или настольных компьютеров, а также по способу управления пакетами и другим функциям.

С течением времени *Linux* была адаптирована для различных архитектур и теперь используется в самых разнообразных электронных устройствах.

На текущий момент системы *GNU/Linux* чаще всего применяются на серверах, включая веб-серверы и суперкомпьютеры, и в меньшей степени на настольных ПК.

Версии ядра *Linux* [11]:

Версия *1.x* (1991 – 1995): версия ядра *Linux*, созданная Линусом Торвальдсом. Включала базовые функции, такие как управление процессами, файловой системой и базовое взаимодействие с оборудованием.

Версия *2.x* (1996 – 2011): поддержка многопроцессорных систем и улучшенное управление памятью, улучшение сетевого стека и масштабируемости, поддержка новых файловых систем, таких как *ext3*.

Версия *3.x* (2011 – 2015): объединение ряда нововведений, включая поддержку улучшенных драйверов, более быстрые файловые системы и новые механизмы планирования задач.

Версия *4.x* (2015 – 2018): повышенная энергоэффективность, поддержка файловой системы *Btrfs*, развитие технологий сетей и драйверов.

Версия *5.x* (2019 – 2022): улучшения для *ARM, RISC-V*, расширение функций безопасности, оптимизация файловых систем и производительности.

Версия *6.x* (2022 – настоящее время): Оптимизация для серверов и встраиваемых систем, улучшенная поддержка современных аппаратных платформ и виртуализации.

2.3 Обоснование выбора платформы

В рамках данного курсового проекта было принято решение

использовать дистрибутив *Manjaro Linux* [12]. Данный выбор обоснован следующими факторами:

1 Доступ к современному программному обеспечению: Модель *rolling release* обеспечивает постоянное обновление системы и предоставляет доступ к самым последним версиям ядра *Linux*, компиляторов (*GCC, Clang*) и системных библиотек. Это критически важно для обеспечения максимальной производительности и совместимости с новейшим аппаратным обеспечением во время проведения вычислительных экспериментов.

2 Стабильность и надежность: Несмотря на постоянное обновление, стабильность системы достигается за счет тщательного тестирования пакетов перед их попаданием в основные репозитории. Это позволяет минимизировать количество ошибок и сторонних сбоев, которые могли бы повлиять на чистоту и достоверность результатов исследований.

3 Легковесность и производительность: Дистрибутив известен своей оптимизированностью и низким потреблением системных ресурсов по умолчанию. Это обеспечивает максимальную прозрачность управления вычислительной мощностью и позволяет получить более точные результаты тестирования производительности процессоров.

4 Гибкость настройки и контроль: Расширенные возможности для тонкой настройки операционной системы под конкретные задачи. Это включает компиляцию ядра с нужными параметрами и точный контроль над фоновыми процессами, что является ключевым преимуществом для проведения ресурсоемких вычислений.

5 Широкий выбор инструментов разработки: Благодаря доступу к *Arch User Repository* (*AUR*) доступен практически любой инструмент для разработки и отладки. Это значительно упрощает процесс установки и настройки всего необходимого программного стека для работы с кодом на *C++*.

2.4 Анализ операционной системы для написания программы

Manjaro Linux – это современный и высокопроизводительный дистрибутив, основанный на *Arch Linux*, ориентированный на предоставление пользователю последних версий программного обеспечения при сохранении стабильности системы. Его ключевыми преимуществами являются модель *Rolling Release*, обеспечивающая постоянное обновление пакетов, и доступ к обширным репозиториям, включая *Arch User Repository* (*AUR*). Это делает *Manjaro* идеальным выбором для разработчиков и исследователей, требующих максимального контроля над средой и доступа к новейшим инструментам оптимизации.

Для проведения сравнительного анализа производительности процессоров, такого как сравнение *Intel i7-12700H* и *AMD Ryzen 7 5800H*, важным является доступ к самым последним версиям компиляторов (*GCC*, *Clang*) и низкоуровневых системных библиотек. *Manjaro*, в отличие от дистрибутивов с фиксированными циклами выпуска, предоставляет эти

инструменты немедленно, что позволяет в полной мере задействовать все архитектурные новшества и инструкции каждого тестируемого процессора для алгоритмов бинарного и линейного поиска.

Хотя *Manjaro* не основан на *Ubuntu*, он обладает исключительно широкой совместимостью с программным обеспечением благодаря поддержке форматов пакетов *Snap* и *Flatpak*, а также наличию *AUR* – крупнейшего сообщественного репозитория. Это гарантирует простоту установки всего необходимого стека разработки и средств профилирования для точного измерения производительности.

Manjaro использует новейшие стабильные версии ядра *Linux*, которые включают оптимизированные и актуальные драйверы для чипсетов и всех компонентов современных ноутбуков. Это обеспечивает максимально эффективное и прямое взаимодействие с аппаратным обеспечением, минимизируя накладные расходы операционной системы. Такая точность и прозрачность управления ресурсами являются фундаментальным требованием для чистого и объективного бенчмаркинга, где любые фоновые помехи могут исказить результаты.

Благодаря легковесности и возможности тонкой настройки, вплоть до компиляции собственного ядра, *Manjaro* позволяет создать идеально оптимизированную среду для тестирования. Это дает возможность изолировать производительность исключительно самих процессоров и реализованных алгоритмов, а не фоновых процессов ОС. Поддержка современных стандартов многозадачности и многопоточности в ядре делает *Manjaro* превосходной платформой для проведения интенсивных вычислений, связанных с параллелизацией.

3 ТЕОРЕТИЧЕСКОЕ ОБОСНОВАНИЕ РАЗРАБОТКИ ПРОГРАММНОГО ПРОДУКТА

3.1 Обоснование необходимости разработки

В условиях стремительного развития вычислительной техники производительность центральных процессоров остается ключевым фактором для эффективного решения широкого круга задач, от повседневных приложений до сложных научных вычислений. Современные мобильные процессоры, такие как *Intel Core i7-12700H* и *AMD Ryzen 7 5800H*, обладают мощной многоядерной архитектурой, что позволяет реализовывать высокопроизводительные вычисления в портативных устройствах. Однако выбор оптимального процессора для конкретных классов задач при ограниченном бюджете представляет собой сложную проблему, требующую объективных данных сравнительного тестирования в контролируемых условиях.

Анализ и сравнение производительности современных многоядерных процессоров на основе алгоритмов бинарного и линейного поиска с использованием параллельных вычислений является актуальной задачей. Это обосновано следующими ключевыми факторами:

1 Фундаментальность алгоритмов поиска. Алгоритмы линейного и бинарного поиска являются одними из базовых в компьютерных науках и лежат в основе множества более сложных приложений, включая системы управления базами данных, обработку больших данных и искусственный интеллект. Исследование их эффективности на современном оборудовании имеет большую практическую ценность.

2 Оптимизация параллельных вычислений. Реализация этих алгоритмов с использованием параллелизма позволяет максимально задействовать вычислительные ресурсы многоядерных процессоров. Сравнение производительности *Intel i7-12700H* (архитектура *hybrid* с *Performance*- и *Efficient*-ядрами) и *AMD Ryzen 7 5800H* (архитектура с однородными ядрами) при параллельном выполнении поиска позволяет выявить сильные и слабые стороны каждой архитектуры для задач, связанных с операциями поиска и обработки данных.

3 Практическое применение. Эффективный поиск данных критически важен в таких областях, как IT-инфраструктура, аналитика в реальном времени, игровые движки и финансовые технологии. Разработка оптимизированного программного продукта для тестирования и сравнения процессоров позволяет получить данные, необходимые для выбора оптимального аппаратного обеспечения под конкретные рабочие нагрузки.

4 Конкурентный анализ на рынке. В условиях высокой конкуренции между *Intel* и *AMD* объективное сравнение производительности их флагманских мобильных процессоров в контексте классических алгоритмов представляет значительный интерес для технических специалистов, энтузиастов и компаний при формировании парка вычислительной техники.

Таким образом, разработка специализированного программного продукта для сравнения производительности процессоров *Intel Core i7-12700H* и *AMD Ryzen 7 5800H* на основе параллельных версий алгоритмов бинарного и линейного поиска не только актуальна с научно-исследовательской точки зрения, но и имеет четкую практическую направленность, помогая определить наиболее эффективное аппаратное решение для задач, интенсивно использующих операции поиска.

3.2 Технологии программирования, используемые для решения поставленных задач

Для реализации программного продукта, направленного на сравнительный анализ производительности процессоров, был выбран язык программирования *C++*. Данный выбор обоснован следующими факторами, связанными с требованиями к эффективности и контролю над вычислительными ресурсами, необходимыми для реализации алгоритмов поиска с параллелизацией.

C++ обеспечивает высокую производительность и детализированное управление памятью, что критически важно для точного измерения временных характеристик алгоритмов. Низкоуровневые возможности языка позволяют минимизировать накладные расходы во время выполнения, обеспечивая чистоту эксперимента при сравнении производительности алгоритмов линейного и бинарного поиска.

Для реализации параллельных вычислений была использована технология *OpenMP*, которая предоставляет удобный интерфейс для распараллеливания циклов на уровне директив компилятора. В коде применялись директивы *#pragma omp parallel for* с редукционными операциями для безопасного подсчета количества найденных элементов [13].

Библиотека *<chrono>* из стандарта *C++* была использована для высокоточного измерения времени выполнения алгоритмов. Применение *std::chrono::high_resolution_clock* и *std::chrono::duration_cast* обеспечило точность измерений до микросекунд, что необходимо для корректного сравнения производительности алгоритмов поиска.

Генератор *std::mt19937* реализует алгоритм "Вихрь Мерсенна" – детерминированный алгоритм генерации псевдослучайных чисел, основанный на свойствах простых чисел Мерсенна. Его математическая модель использует рекуррентные соотношения в конечном поле *GF(2)*, где каждое последующее состояние вычисляется через линейную связь с предыдущими состояниями с использованием битовых операций сдвига и маскирования. Ключевые особенности алгоритма включают использование примитивного многочлена степени 19937 над полем *GF(2)*, что обеспечивает максимальный период последовательности 2^{19937}-1. Структура генератора оптимизирована через так называемое "скручивание" (*twisting*) – процедуру, которая равномерно распределяет значения в многомерном пространстве, устраняя корреляции между последовательными состояниями. Массив

значений поиска формировался по смешанной стратегии: 50% существующих элементов из основного массива и 50% случайных чисел для тестирования различных сценариев поиска.

Библиотека <*fstream*> обеспечила сохранение результатов тестирования в формате *CSV*, что позволило организовать структурированное хранение данных для последующего анализа. Формат вывода включал количество потоков, время выполнения для каждого алгоритма, коэффициент ускорения и количество найденных элементов. Для визуализации полученных данных использовалась библиотека *matplotlib* в среде *Python*, которая предоставила инструменты для построения сравнительных графиков производительности и анализа масштабируемости алгоритмов.

Библиотека hwloc обеспечила детальное управление процессорной топологией, что позволило реализовать точную привязку вычислительных потоков к физическим ядрам процессоров. Функционал библиотеки предоставил инструменты для обнаружения иерархической структуры процессоров, включая физические ядра, логические процессоры и кэш-память разных уровней.

Библиотека <*algorithm*> использовалась для сортировки данных перед выполнением бинарного поиска с помощью функции *std::sort*, реализующей гибридный алгоритм сортировки, что гарантирует временную сложность в худшем случае *O(N log N)* и является обязательным условием корректной работы бинарного поиска.

Таким образом, комбинация технологий *C++*, *OpenMP* и стандартных библиотек в совокупности с инструментами визуализации *matplotlib* предоставила необходимый инструментарий для создания надежной тестовой системы, позволяющей объективно сравнивать производительность процессоров *Intel Core i7-12700H* и *AMD Ryzen 7 5800H* при выполнении параллельных версий алгоритмов линейного и бинарного поиска.

3.3 Связь архитектуры вычислительной системы с разрабатываемым программным обеспечением

Разрабатываемый программный продукт для сравнительного анализа алгоритмов поиска тесно интегрирован с архитектурными особенностями тестируемых процессоров *Intel Core i7-12700H* и *AMD Ryzen 7 5800H*. Архитектурные различия этих процессоров непосредственно влияют на эффективность параллельного выполнения алгоритмов линейного и бинарного поиска.

Процессор *Intel Core i7-12700H* реализует гибридную архитектуру, сочетающую 6 производительных ядер (*P-cores*) и 8 энергоэффективных ядер (*E-cores*). Такая структура требует оптимизации распределения потоков между разнородными ядрами для достижения максимальной производительности. В отличие от него, *AMD Ryzen 7 5800H* обладает однородной архитектурой с 8 идентичными вычислительными ядрами, что упрощает балансировку нагрузки при параллельных вычислениях.

На уровне программной реализации взаимодействие с процессорными ресурсами обеспечивается средствами *OpenMP* и операционной системы. Директивы *OpenMP* позволяют явно распределять вычислительную нагрузку между доступными потоками, используя механизмы динамического планирования и статического распределения итераций циклов.

Особенностью тестируемых алгоритмов является различная вычислительная сложность: бинарный поиск демонстрирует логарифмическую сложность $O(\log N)$, в то время как линейный поиск имеет линейную сложность $O(N)$. Это различие по-разному проявляется на многоядерных архитектурах, где эффективность распараллеливания зависит от способности алгоритма к декомпозиции задачи.

Для понимания взаимодействия программного продукта с вычислительной системой следует рассмотреть основные понятия, такие как физическое и логическое ядро процессора, процесс, поток, а также роль операционной системы в управлении распределением вычислительных задач между этими сущностями.

Физическое ядро процессора – это аппаратная единица, способная выполнять вычисления. В современных многоядерных процессорах каждое физическое ядро имеет собственные вычислительные блоки и может выполнять независимые задачи. Логическое ядро создается путем виртуализации ресурсов физического ядра, и, в процессорах с технологией *Hyper-Threading* и *SMT*, одно физическое ядро может обрабатывать несколько потоков одновременно.

Процесс – изолированная программа, работающая в своем адресном пространстве, которая не имеет доступа к памяти других процессов. Поток – это базовая единица, которой операционная система выделяет процессорное время. Поток может выполнять любую часть кода процесса, включая те части, которые в настоящее время выполняются другим потоком [14].

Программное обеспечение использует механизмы синхронизации OpenMP, включая редукционные операции для безопасного подсчета количества найденных элементов. Это обеспечивает корректность выполнения при одновременной работе множества потоков на разных ядрах процессора.

Проведенное исследование позволяет выявить оптимальные стратегии распараллеливания для каждой архитектуры процессора и оценить влияние аппаратных особенностей на эффективность выполнения фундаментальных алгоритмов поиска.

4 ПРОЕКТИРОВАНИЕ ФУНКЦИОНАЛЬНЫХ ВОЗМОЖНОСТЕЙ ПРОГРАММЫ

4.1 Общая цель программы

Цель программы заключается в проведении сравнительного анализа производительности двух современных мобильных процессоров – *Intel Core i7-12700H* и *AMD Ryzen 7 5800H* при выполнении алгоритмов линейного и бинарного поиска с использованием параллельных вычислений. Программа позволяет оценить эффективность распараллеливания вычислительной нагрузки на многоядерных архитектурах и выявить оптимальные конфигурации потоков для каждого типа процессора.

Программа обеспечивает измерение временных характеристик алгоритмов поиска при различном количестве потоков выполнения, что позволяет проанализировать масштабируемость производительности в зависимости от задействованных вычислительных ядер. Особое внимание уделяется сравнению эффективности параллельного выполнения для алгоритмов с разной вычислительной сложностью – *O(N)* для линейного поиска и *O(log N)* для бинарного поиска.

Дополнительной целью является оценка влияния архитектурных особенностей процессоров на производительность параллельных вычислений. Гибридная архитектура *Intel Core i7-12700H* (*P-cores* и *E-cores*) сравнивается с однородной архитектурой *AMD Ryzen 7 5800H* в контексте задач поиска данных, что позволяет выявить преимущества и недостатки каждой архитектуры для данного класса алгоритмов.

4.2 Теоретические сведения об алгоритмах поиска

Алгоритмы поиска данных в вычислительных системах подразделяются на несколько категорий по различным критериям. По способу организации данных различают алгоритмы для неупорядоченных и упорядоченных массивов. По стратегии поиска выделяют последовательные, интервальные и хэш-ориентированные методы. Для целей сравнительного анализа производительности процессоров были выбраны два фундаментальных алгоритма – линейный и бинарный поиск, как наиболее репрезентативные представители разных классов.

Линейный поиск был выбран по следующим причинам:

1 Простота и минимальные накладные расходы – алгоритм не требует дополнительной памяти и предварительной обработки данных, что позволяет измерить "чистую" производительность процессора при последовательном доступе к памяти.

2 Предсказуемость доступа к памяти – последовательный обход элементов массива обеспечивает оптимальное использование кэш-памяти процессора через механизм пространственной локальности.

3 Идеальная распараллеливаемость – независимость проверок

отдельных элементов позволяет эффективно распределять нагрузку между множеством ядер без необходимости синхронизации.

4 Репрезентативность для реальных задач – множество прикладных задач (поиск в небольших наборах данных, обработка потоковых данных) используют аналогичные последовательные алгоритмы.

Бинарный поиск представляет особый интерес благодаря:

1 Демонстрации эффективности предварительной обработки – необходимость сортировки данных перед поиском иллюстрирует компромисс между временем предобработки и временем выполнения.

2 Сложному шаблону доступа к памяти – случайный доступ к различным участкам массива создает нагрузку на подсистему памяти и позволяет оценить эффективность кэширования.

3 Ограниченной параллелизуемости – алгоритм демонстрирует сложности при распараллеливании, что позволяет исследовать эффективность процессоров при работе с последовательными алгоритмами.

4 Практической значимости – бинарный поиск является основой для более сложных структур данных (деревья, *B*-деревья) и алгоритмов вычислительной геометрии

Сравнительные характеристики для линейного поиска:

– временная сложность: *O(n)* в худшем случае;

– пространственная сложность*: O(1)*;

– требования к данным: отсутствуют;

– параллельная сложность: *O(n/p)*.

Сравнительные характеристики для бинарного поиска:

– временная сложность: *O(log n);*

– пространственная сложность: *O(1);*

– требования к данным: обязательная сортировка *(O(n log n));*

– параллельная сложность: *O(log n)* с ограниченным ускорением.

Выбранные алгоритмы по-разному нагружают компоненты процессора. Линейный поиск создает равномерную нагрузку на арифметико-логическое устройство и обеспечивает последовательный доступ к памяти, что оптимально для предсказателя переходов и кэш-памяти. Бинарный поиск характеризуется непредсказуемыми переходами и случайным доступом к памяти, что позволяет оценить эффективность предугадывания ветвлений и иерархии кэш-памяти.

Для многопоточных систем актуальны параллельные версии алгоритмов. Линейный поиск легко распараллеливается разделением массива между потоками. Бинарный поиск допускает параллелизацию через одновременный поиск в нескольких сегментах или использование параллельных деревьев поиска. Особый интерес представляет возможность векторизации операций в линейном поиске с использованием *SIMD*-инструкций (*Single Instruction, Multiple Data*). Технологии *SSE* и *AVX*, доступные в обоих тестируемых процессорах, позволяют одновременно сравнивать с целевым значением несколько элементов массива – 4 при использовании 128-битных регистров *SSE* или 8 при задействовании 256-

битных регистров *AVX2*. Это теоретически может обеспечить пропорциональное ускорение выполнения алгоритма. Однако эффективность *SIMD*-векторизации сильно зависит от правильного выравнивания данных в памяти и отсутствия зависимостей между операциями.

Для бинарного поиска ключевым аспектом является поведение предсказателя переходов процессора. Из-за высокого уровня ветвления и их непредсказуемости характерного для этого алгоритма может происходить значительное количество ошибок предсказания что негативно сказывается на конвейеризации исполнения команд. Различные реализации предсказателей переходов в *Intel Core i7-12700H* и *AMD Ryzen 7 5800H* могут проявлять разную эффективность на данном алгоритме.

Также важно учитывать влияние иерархии кэш-памяти. Линейный поиск демонстрирует последовательный доступ к памяти, что оптимально для механизмов предварительной выборки и обеспечивает высокий процент попаданий в кэш. В отличие от него, бинарный поиск с его случайным доступом создает интенсивную нагрузку на подсистему памяти, и его производительность в большей степени зависит от латентности и пропускной способности оперативной памяти, а также от размеров и ассоциативности кэшей *L1*, *L2* и *L3*.

Выбор именно этих алгоритмов позволяет комплексно оценить производительность процессоров *Intel Core i7-12700H* и *AMD Ryzen 7 5800H* при различных типах вычислительных нагрузок и шаблонах доступа к памяти.

4.3 Описание функциональных возможностей

Программа предназначена для сравнительного анализа производительности алгоритмов линейного и бинарного поиска на многоядерных процессорах. Ее функциональные возможности включают:

1 Генерация тестовых данных – программа создает основной массив и массив целевых значений, которое следует найти в основном массиве. Для формирования данных используется генератор псевдослучайных чисел *std::mt19937*, обеспечивающий равномерное распределение значений.

2 Смешанная стратегия тестирования – массив целевых значений формируется таким образом, что 50% элементов гарантированно присутствуют в основном массиве, а остальные 50% являются случайными числами. Это позволяет тестировать алгоритмы в условиях, приближенных к реальным задачам.

3 Многоуровневое тестирование производительности – программа выполняет сравнительный анализ трех конфигураций:

– последовательное выполнение (search_sequential_timed);

– параллельное выполнение с OpenMP (search_openmp_timed);

– выполнение на одном физическом ядре (search_single_core_timed);

– параллельное выполнение на *iGPU* (search_gpu _timed).

4 Точное измерение времени – для хронометража используется библиотека <*chrono*>, обеспечивающая замеры времени с точностью до

микросекунд. Измерения выполняются отдельно для каждого алгоритма поиска.

5 Статистическая достоверность – каждый тест выполняется в 10 итераций (*ITERATIONS*) с последующим усреднением результатов, что обеспечивает высокую точность измерений и исключает влияние случайных факторов.

6 Сравнительный анализ производительности – программа вычисляет коэффициент ускорения (*speed_ratio*) как отношение времени выполнения линейного поиска к времени бинарного поиска. Это позволяет наглядно оценить эффективность каждого алгоритма при различном количестве потоков.

7 Верификация результатов – осуществляется подсчет количества успешно найденных элементов для обоих алгоритмов, что обеспечивает контроль корректности выполнения поисковых операций.

8 Многоплатформенный вывод результатов – программа предоставляет два формата представления данных: форматированная таблица в консоли с использованием манипуляторов ввода-вывода и структурированный *CSV*-файл с разделителями-запятыми для последующего анализа.

9 Автоматическое сохранение данных – все результаты тестирования сохраняются в файл *CSV* с четкой структурой заголовков, включающей количество потоков, время выполнения алгоритмов, коэффициент ускорения и статистику найденных элементов.

10 Анализ процессорной топологии – программа использует библиотеку *hwloc* для обнаружения иерархической структуры процессора, включая физические ядра, логические процессоры и их распределение. Функция *setAffinityToPhysicalCore* обеспечивает точную привязку потоков к конкретным физическим ядрам процессора с использованием библиотеки *hwloc* для последующего запуска в режиме одного ядра, что исключает влияние планировщика ОС на результаты измерений.

11 Автоматическое обнаружение *GPU* – система самостоятельно находит доступные *GPU* устройства, с последующей инициализацией *OpenCL* контекста и созданием командных очередей.

Программа обеспечивает комплексную оценку производительности процессоров *Intel Core i7-12700H* и *AMD Ryzen 7 5800H*, позволяя выявить оптимальные конфигурации потоков для каждого типа алгоритмов поиска и проанализировать эффективность использования многоядерных архитектур.

5 СРАВНЕНИЕ ПРОИЗВОДИТЕЛЬНОСТИ ПРОЦЕССОРОВ

5.1 Общая структура программы

Программа представляет собой систему для сравнительного анализа производительности алгоритмов линейного и бинарного поиска на гетерогенных вычислительных системах, включая *CPU* и *GPU*. Архитектура программы построена по модульному принципу и включает несколько ключевых компонентов.

Основу программы составляют четыре тестовых массива данных: массив для линейного поиска размером 100000 элементов, массив для бинарного поиска размером 10000000 элементов, а также соответствующие массивы целевых значений для каждого типа поиска. Генерация тестовых данных осуществляется с использованием генератора псевдослучайных чисел std::mt19937 с равномерным распределением. Особенностью формирования целевых значений является смешанная стратегия: 50% элементов гарантированно присутствуют в основном массиве, а остальные 50% представляют собой случайные числа.

Модуль *GPU*-вычислений реализован в классе *GPUSearcher*, который обеспечивает автоматическое обнаружение доступных *GPU* устройств.

Центральным элементом архитектуры является структура *SearchResults*, которая инкапсулирует результаты тестирования, включая время выполнения линейного и бинарного поиска, количество найденных элементов для каждого алгоритма и отношение производительности между алгоритмами. Программа реализует три основных режима тестирования: последовательное выполнение поиска, параллельное выполнение с использованием технологии *OpenMP* и выполнение на одном физическом ядре процессора с контролируемой привязкой потоков.

Модуль управления аппаратной топологией использует библиотеку *hwloc* для анализа процессорной архитектуры и предоставляет функции для точной привязки потоков к физическим ядрам. Функция *initTopology* инициализирует систему обнаружения аппаратных ресурсов, *setAffinityToPhysicalCore* обеспечивает привязку к конкретным физическим ядрам, а printTopologyInfo выводит детальную информацию о конфигурации процессора, включая распределение логических процессоров по физическим ядрам.

Модуль алгоритмов поиска содержит реализации линейного и бинарного поиска. Линейный поиск реализован через последовательный перебор элементов массива, в то время как бинарный поиск использует алгоритм деления пополам на отсортированном массиве. Функции *search_sequential_timed* и *search_openmp_timed* обеспечивают временные измерения для последовательного и параллельного выполнения соответственно.

Система тестирования построена на шаблонной функции *TestSearchTimed*, которая обеспечивает многократное выполнение тестов с

последующим усреднением результатов для повышения статистической достоверности Программа выполняет прогрессивное тестирование на восьми различных размерах данных: от 100 до 50000 целей для линейного поиска и от 1000 до 500000 целей для бинарного поиска. Каждый тест выполняется в 10 итераций для минимизации влияния случайных факторов.

Модуль вывода результатов предоставляет два формата представления данных: форматированные таблицы в консоли с использованием манипуляторов ввода-вывода и структурированный *CSV*-файл для последующего анализа. Функция *printResultsTable* формирует наглядные таблицы с временными характеристиками, отношениями производительности и статистикой найденных элементов, а *saveResultsToFile* сохраняет полный набор результатов в файл для дальнейшей обработки.

Параллельная реализация использует директивы *OpenMP #pragma omp parallel for* с редукционными операциями для безопасного параллельного выполнения поисковых операций. Программа автоматически определяет максимальное количество доступных потоков через *omp_get_max_threads* и динамически настраивает параллельное выполнение. Для обеспечения точности временных измерений используется библиотека *chrono* с высокоточными часами *std::chrono::high_resolution_clock* и преобразованием в микросекунды.

Управление ресурсами осуществляется через систему инициализации и освобождения топологии *hwloc*, что гарантирует корректную работу на различных аппаратных платформах и предотвращает утечки памяти. Программа поддерживает тестирование на различных конфигурациях вычислительных систем и предоставляет инструменты для анализа эффективности использования *CPU* и *GPU* ресурсов при выполнении поисковых операций. Исходный код программы представлен в приложении Б.

5.2 Описание функциональной схемы программы

Функциональная схема программы представлена в приложении В. В ходе разработки функциональной схемы выделены следующие ключевые блоки:

1 Генерация тестовых данных. На этом этапе выполняется создание массивов данных для линейного поиска (100000 элементов) и бинарного поиска (10000000 элементов), а также соответствующих массивов целевых значений. Для генерации используется алгоритм "Вихрь Мерсенна" (std::mt19937) с равномерным распределением. Массивы целевых значений формируются по смешанной стратегии: 50% элементов гарантированно присутствуют в основном массиве, а остальные 50% являются случайными числами. Массив для бинарного поиска предварительно сортируется.

2 Инициализация графических ускорителей. Программа выполняет автоматическое сканирование доступных *OpenCL*-платформ для обнаружения *GPU*-устройств. Для выбранного устройства создается вычислительный контекст, компилируются специализированные ядра поисковых алгоритмов,

формируются командные очереди.

3 Анализ топологии процессора. Программа инициализирует библиотеку *hwloc* для обнаружения аппаратной конфигурации системы. Выполняется анализ иерархической структуры процессора, включая определение количества физических ядер, логических процессоров и их распределения. Полученная информация о топологии выводится в консоль и используется для привязки потоков к конкретным физическим ядрам процессора для последующего запуска в режиме одного ядра.

4 Выполнение алгоритмов поиска. Тестирование включает измерение времени выполнения трех конфигураций алгоритмов поиска: последовательного выполнения (*search_sequential_timed*), параллельного выполнения с использованием *OpenMP* (*search_openmp_timed*), выполнения на одном физическом ядре (*search_single_core_timed*) и параллельное выполнение на *GPU* с использованием *OpenCL (search_gpu_timed)*. Для каждой конфигурации выполняются линейный и бинарный поиск на различных объемах данных. Измерение времени выполнения осуществляется с использованием библиотеки <*chrono*>, которая предоставляет высокоточные инструменты для хронометража.

5 Анализ результатов. На этом этапе производится обработка полученных данных: вычисление среднего времени выполнения для каждого алгоритма, определение количества найденных элементов и расчет отношения производительности между линейным и бинарным поиском. Анализ включает сравнение эффективности различных конфигураций выполнения и оценку масштабируемости алгоритмов при увеличении объема данных.

6 Вывод и сохранение результатов. Программа предоставляет два формата представления результатов: форматированный вывод в консоль в виде таблицы с детализацией времени выполнения, отношений производительности и статистики найденных элементов для всех тестируемых конфигураций, а также сохранение полного набора результатов в *CSV*-файл для последующего анализа и визуализации. Файл содержит структурированные данные по всем размерам тестов и конфигурациям выполнения.

5.3 Описание блок схемы алгоритма программы

Блок-схема алгоритма программы представлена в приложении Г. Схема включает следующие ключевые функции:

– *main*: начальная точка программы, отвечает за инициализацию *hwloc* топологии, генерацию тестовых данных, вызов функций поиска и сохранение результатов;

– *search_sequential_timed*: измеряет время выполнения последовательного линейного и бинарного поиска для заданного набора данных;

– *search_single_core_timed*: выполняет привязку вычислений к конкретному ядру процессора и измеряет время поиска на выделенном

вычислительном ресурсе;

– search_openmp_timed: выполняет параллельный поиск с использованием технологии *OpenMP* с автоматическим распределением нагрузки между доступными потоками;

– *GPUSearcher*: класс для управления *GPU* вычислениями, обеспечивающий автоматическое обнаружение графических ускорителей, инициализацию *OpenCL* контекста;

– search_gpu_timed: измеряет время выполнения линейного и бинарного поиска на *GPU*;

– *linear_search*: реализует алгоритм последовательного линейного поиска в массиве данных;

– *binary_search*: реализует алгоритм бинарного поиска в отсортированном массиве;

– *generate_test_data*: генерирует тестовые данные для проведения сравнительного анализа алгоритмов поиска;

– *sort_data*: выполняет сортировку данных для последующего использования в бинарном поиске.

Программа использует библиотеку *hwloc* для анализа топологии вычислительной системы и технологию *OpenMP* для распараллеливания операций поиска. Для каждого размера массива данных выполняется настройка количества потоков *OpenMP*, после чего производится сравнительное измерение времени выполнения линейного и бинарного поиска. Результаты измерений сохраняются в табличном формате и экспортируются в *CSV*-файл для последующего анализа.

5.4 Подготовка к тестированию

Для анализа производительности алгоритмов поиска (линейного и бинарного) будет проведено тестирование на процессорах в различных режимах загрузки. Основной акцент будет сделан на сравнении скорости работы при выполнении на одном ядре по сравнению с полной загрузкой процессора. Испытания проводятся в два этапа: с включенной *HyperThreading/SMT* и с отключенной, чтобы оценить влияние этого фактора на эффективность вычислений. Для получения дополнительных данных о поведении процессоров также будут собраны ключевые аппаратные метрики производительности.

Для мониторинга распределения нагрузки на вычислительные ядра в реальном времени использовалась утилита *htop*, которая предоставила визуальное представление о загрузке процессорных ядер и использовании памяти во время выполнения тестов. Параллельно для детального анализа топологии процессора и управления привязкой процессов к конкретным ядрам применялась библиотека *hwloc*. Эта библиотека позволила точно определить физическую структуру процессора и обеспечить корректную привязку вычислительных потоков к целевым ядрам.

Размеры тестовых данных, на которых проводились измерения,

варьировались от 10^2 до 10^6 элементов с экспоненциальным шагом. Каждое измерение времени выполнения запускается по 10 раз для минимизации влияния случайных системных факторов. В качестве метрики для анализа выбрано среднее время выполнения тестов, которое позволяет объективно оценить, как различные режимы загрузки влияют на итоговую производительность при выполнении поисковых операций.

Для более глубокого анализа производительности для каждого типа решения использовалась утилита *perf* [15], которая позволила собрать следующие аппаратные метрики:

– общее количество тактов процессора (*cycles*);
– количество исполненных инструкций (*instructions*);
– количество обращений к кэш-памяти (*cache-references*);
– количество кэш-промахов (*cache-misses*);
– количество переходов в коде (*branches*);
– количество промахов при предсказании переходов (*branch-misses*);
– количество страниц, выгруженных из памяти (*page-faults*);
– контекстные переключения (*context-switches*);
– застои на этапе выборки инструкций (*stalled-cycles-frontend*).

Сбор метрик для каждого режима проводился с использованием отдельного запуска тестов

Сравнение проводилось в следующих режимах:

– последовательный алгоритм без использования *OpenMP*;
– параллельный алгоритм с использованием *OpenMP*;
– последовательный алгоритм с привязкой к одному ядру процессора
– параллельное выполнение на *iGPU* с использованием *OpenCL*.

Исследования начались с проведения тестов на обоих устройствах с включенной *HyperThreading/SMT*. После этого функции *HyperThreading/SMT* были отключены, и тесты были повторно запущены для получения следующего набора данных.

5.5 Результаты измерений с включенным Hyper-Threading/SMT

Таблица 5.1 представляет замеры времени выполнения (в микросекундах) алгоритма линейного поиска на процессоре *AMD Ryzen 7 5800H* с включенной функцией *SMT*. В таблице представлены результаты для линейного поиска при различных размерах тестового массива данных.

Колонка "Размер" указывает количество целевых элементов для поиска. В колонке "Последовательный" представлено время выполнения стандартного алгоритма линейного поиска без использования параллельных технологий. Колонка "Параллельный (*OpenMP*)" отображает результаты параллельной реализации с распределением нагрузки между всеми доступными ядрами процессора. Колонка "На одном ядре" демонстрирует время выполнения последовательного алгоритма с привязкой к конкретному ядру процессора с использованием библиотеки *hwloc*. Колонка "На *iGPU*" показывает время выполнения на интегрированной графике через *OpenCL*.

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

Размер	Последовательный	На одном ядре	Параллельный (*OpenMP*)	На *iGPU*
100	3539,6	4010,3	456,8	9807,4
500	17648,6	17982,8	1980,1	9838,1
1000	35301,6	39060,8	3879,4	9441,7
2000	70647,3	86794,8	7437,7	9425,6
5000	176791,7	221557,9	17928,5	12932,9
10000	353709,8	379604,9	37854,2	19236,7
25000	887316,1	909689,6	80449,4	43528,6
50000	1781289,1	1803696,7	151819,3	79691,9

Таблица 5.2 представляет замеры времени для программы для линейного поиска на процессоре *Intel Core i7 12700H* с включенной функцией *Hyper-Threading*.

Таблица 5.2 – Замеры времени линейного поиска на *Intel Core i7 12700H*

Размер	Последовательный	На одном ядре	Параллельный (*OpenMP*)	На *iGPU*
100	1552,8	4010,3	339	22664,2
500	8126,9	17982,8	1046,3	20866,4
1000	16559,2	39060,8	2183,2	20817,7
2000	33132,6	86794,8	5067,5	20025,9
5000	82350	221557,9	15277,2	21607
10000	164447	379604,9	32134,9	22689,7
25000	410692,8	909689,6	82369,3	63256,2
50000	820053	1803696,7	235833,5	126746,7

Оба процессора демонстрируют схожую тенденцию в поведении алгоритмов, однако с заметными различиями в абсолютных показателях производительности. Прежде всего, наблюдается общая закономерность для обеих платформ: параллельная реализация с использованием *OpenMP* обеспечивает значительное ускорение по сравнению с последовательным выполнением. Коэффициент ускорения составляет от 7.7 до 11.7 раз для процессора *AMD* и от 4.6 до 7.5 раз для процессора *Intel*.

Из нагрузки на ядра, которую можно рассмотреть в Приложении Е следует, что оба процессора демонстрируют схожую картину на

последовательном, параллельном (OpenMP) и одном ядре.

Реализация на *iGPU* показывает ожидаемые результаты: для небольших размеров данных (100-2000 элементов) время выполнения на i*GPU* значительно превышает *CPU* реализации, что связано с накладными расходами на передачу данных и запуск ядер iGPU. Однако с увеличением объема данных до 50000 элементов преимущество *iGPU* становится более заметным, особенно на платформе *AMD*, где iGPU демонстрирует в 22.3 раза лучшее время по сравнению с последовательной *CPU* реализацией.

Что касается ключевых различий между процессорами, то процессор *Intel Core i7 12700H* демонстрирует более высокую производительность в абсолютных значениях для *CPU* реализаций. Например, при размере целевого массива в 50000 элементов разница в производительности последовательной реализации составляет примерно 2.17 раз в пользу процессора *Intel*. Однако в сегменте *iGPU AMD* показывает значительно лучшую эффективность – при 50000 элементов время выполнения на *AMD iGPU* в 1.59 раза лучше, чем на *Intel iGPU*.

Наибольший выигрыш от параллелизации на *CPU* наблюдается при обработке больших массивов данных, где вычислительная нагрузка эффективно распределяется между всеми доступными ядрами процессора. *GPU* реализация становится эффективной только при работе с большими объемами данных, компенсируя накладные расходы массовым параллелизмом. С графиком можно ознакомиться в приложении Д.

Таблица 5.3 представляет замеры времени выполнения (в микросекундах) бинарного поиска на процессоре *AMD Ryzen 7 5800H* с включенной функцией *SMT*.

Таблица 5.3 – Замеры времени бинарного поиска на *AMD Ryzen 7 5800H*

Размер	Последовательный	На одном ядре	Параллельный (*OpenMP*)	На *iGPU*
1000	153,9	137,4	19,9	305,4
5000	868,2	780,5	100,8	604,1
10000	2183,6	2215,6	242,2	739
20000	4901,5	5583,7	485,8	1196,2
50000	12557,1	14921,9	1377,3	2016,3
100000	26233	26764,6	5523	3066,8
250000	62783,5	62040,6	8512,6	6738,4
500000	121714,7	121875,3	14384,1	13180,8

Таблица 5.4 представляет замеры времени для программы для бинарного поиска на процессоре *Intel Core i7 12700H* с включенной функцией *Hyper-Threading*.

Таблица 5.4 – Замеры времени бинарного поиска на *Intel Core i7 12700H*

Размер	Последовательный	На одном ядре	Параллельный (*OpenMP*)	На *iGPU*
1000	155,2	176,8	60,8	203,1
5000	822,4	801,1	125,6	324,9
10000	1814,7	1780,5	251,7	459,6
20000	3792,2	3762,9	693,4	751,5
50000	9464,2	9322,7	1691,6	1338,9
100000	18713,4	18664,1	2894,3	2072,8
250000	46501,1	46191,3	6415	3661,5
500000	93334,7	107593,3	19650,2	5439,7

Анализ результатов бинарного поиска выявляет существенные отличия в эффективности различных вычислительных платформ в зависимости от объема данных. Наибольшее ускорение обеспечивает параллельная реализация на *CPU* с использованием *OpenMP* – на платформе AMD ускорение составляет от 7.7 до 8.5 раз, на Intel – от 2.5 до 4.8 раз по сравнению с последовательным выполнением.

Особый интерес представляют результаты выполнения на *iGPU*. В отличие от линейного поиска, где *GPU* становился эффективным только на больших объемах данных, для бинарного поиска iGPU демонстрирует конкурентоспособную производительность уже на средних размерах массивов, но следует учесть, что и размеры целевых значений для бинарного выше в связи с асимптотикой выполения алгоритмов.

Реализация на одном ядре демонстрирует результаты, сопоставимые с последовательным выполнением, что подтверждает корректность методики измерений и отсутствие существенного влияния миграции потоков между ядрами на производительность.

Что касается *OpenMP* на *CPU*, здесь *AMD* стабильно обгоняет *Intel* на обоих типах поиска на больших объёмах данных. Это происходит потому, что ядра *AMD* эффективнее работают с памятью и лучше распределяют нагрузку между потоками. Архитектура *Zen 3* просто более сбалансирована для параллельных вычислений.

Для анализа производительности были проведены измерения аппаратных метрик с использованием утилиты *perf*. Эти метрики отражают различные аспекты работы процессора, такие как количество тактовых циклов, выполненных инструкций, промахов кэша, переключений контекста и другие параметры. В таблице 5.5 представлены результаты измерений для каждого из исследуемых вариантов решения линейного поиска на процессоре *AMD Ryzen 7 5800H*.

Таблица 5.5 – Замеры метрик линейного поиска на *AMD Ryzen 7 5800H*

N	Последовательный	На одном ядре	Параллельный (*OpenMP*)
cycles	128 709 419 200	125 548 432 505	256 023 511 285
instructions	434 922 941 255	433 954 310 266	436 308 736 660
insn per cycle	3,38	3,46	1,70
cache-references	549 409 263	596 576 554	568 335 709
cache-misses	11 197 958	10 555 272	9 460 685
branches	55 918 159 976	55 793 885 711	56 296 660 395
branch-misses	978 092	1 007 017	1 200 335
seconds time elapsed	30,141217524	29,790667822	4,571793237
seconds user	30,134509000	29,771086000	69,156161000
seconds sys	0,002999000	0,005998000	0,327834000

Аналогичные измерения метрик при выполнении задач были проведены и на процессоре *Intel Core i7 12700H*. Первый параметр в таблице соответствует энергоэффективным ядрам (до знака ‘+’ или ‘/’), а второй производительным. С замерами метрик линейного поиска можно ознакомиться в таблице 5.6.

Таблица 5.6 – Замеры метрик линейного поиска на *Intel Core i7 12700H*

N	Последовательный	На одном ядре	Параллельный (*OpenMP*)
cycles	47 833 835 622 + 82 152 553 728	16 595 418 652 + 80 523 379 053	123 957 291 864 + 149 648 984 491
instructions	137 422 248 682 + 430 865 351 534	39 964 974 701 + 428 820 162 488	476 775 533 810 + 404 854 029 638
insn per cycle	2,87/5,24	2,41/5,33	3,85/2,71
cache-references	41 683 085 + 108 715	70 128 124 + 288 938	1 628 428 + 5 457 848
cache-misses	21 952 662 + 17 605	42 156 430 + 20 716	22 774 + 43 869
branches	17 570 508 841 + 55 396 563 997	5 325 028 770 + 55 133 720 641	61 299 597 356 + 52 052 521 694

Продолжение таблицы 5.6

branch-misses	284 093 855 + 604 374	119 462 738 + 832 680	332 275 + 1 253 492
seconds time elapsed	18,069033468	17,577858295	4,580543570
seconds user	18,052042000	17,549208000	80,549363000
seconds sys	0,003328000	0,016586000	0,066524000

Для линейного поиска при анализе работы с памятью оба процессора демонстрируют низкий уровень кэш-промахов (менее 2%), однако *Intel* показывает значительно большее количество обращений к кэшу в параллельном режиме, что связано с особенностями работы гибридной архитектуры. Метрики ветвлений выявляют преимущество *AMD* в предсказании переходов – количество промахов составляет менее 0,002% против 0,5-1,7% у *Intel*.

В таблице 5.7 представлены результаты измерений для каждого из исследуемых вариантов решения бинарного поиска на процессоре *AMD Ryzen 7 5800H*.

Таблица 5.7 – Замеры метрик бинарного поиска на *AMD Ryzen 7 5800H*

N	Последовательный	На одном ядре	Параллельный (*OpenMP*)
cycles	8 785 057 094	8 919 055 379	9 583 717 583
instructions	15 317 947 164	15 413 812 259	15 227 400 131
insn per cycle	1,74	1,73	1,59
cache-references	47 303 328	49 125 138	42 175 805
cache-misses	12 129 836	12 341 077	9 165 665
branches	1 873 589 814	1 885 255 246	1 868 543 899
branch-misses	112 487 653	111 170 340	112 974 566
seconds time elapsed	2,085306622	2,154123976	1,853453531
seconds user	2,075003000	2,130084000	2,322115000
seconds sys	0,010000000	0,016000000	0,011010000

Аналогичные измерения метрик при выполнении задач были проведены и на процессоре *Intel Core i7 12700H*. Напомним, что первый параметр в таблице соответствует энергоэффективным ядрам (до знака ‘+’ или ‘/’), а второй производительным. Полученные результаты продемонстрированы в таблице 5.8.

Таблица 5.8 – Замеры метрик бинарного поиска на *Intel Core i7 12700H*

N	Последовательный	На одном ядре	Параллельный (*OpenMP*)
cycles	1 092 684 187 + 7 520 285 181	4 016 352 280 + 7 486 207 773	3 123 208 656 + 8 433 810 764
instructions	1 757 428 711 + 15 399 544 579	11 542 083 261 + 15 393 697 010	5 833 729 527 + 17 006 563 440
insn per cycle	1,61/2,05	2,87/2,06	1,87/2,02
cache-references	12 317 540 + 43 294 469	9 265 593 + 45 288 762	107 623 814 + 27 130 934
cache-misses	8 622 803 + 7 994 108	2 349 380 + 8 246 428	16 059 638 + 5 066 830
branches	298 668 144 + 1 883 235 895	1 482 609 075 + 1 881 750 399	742 666 362 + 2 075 818 123
branch-misses	8 962 353 + 109 687 204	19 217 110 + 111 170 694	26 692 528 + 124 575 806
seconds time elapsed	1,664550522	1,714084094	1,449087862
seconds user	1,643314000	1,648468000	2,194536000
seconds sys	0,019892000	0,023231000	0,019077000

Для бинарного поиска наблюдаются иные закономерности. Важным отличием является уровень промахов предсказателя переходов при бинарном поиске: 6-12% на *AMD* против 3-6% на *Intel*, что свидетельствует о лучшей работе алгоритмов предсказания ветвлений в гибридной архитектуре *Intel*.

Временные характеристики подтверждают преимущество параллельных реализаций: для линейного поиска ускорение составляет 6,6 раза на *AMD* и 3,9 раза на Intel, для бинарного поиска – 1,12 и 1,15 раза соответственно. Меньшая эффективность параллелизации бинарного поиска обусловлена его алгоритмическими особенностями. Время выполнения бинарного поиска у *Intel* имеет меньшее отставание от *AMD*, что связано с распределением ресурсов больше на производительные ядра, тогда как в линейной реализации нагрузка между энергоэффективными и производительными ядрами распределена равномерно.

Распределение нагрузки показывает, что в последовательных режимах на *Intel* нагрузка неравномерно распределяется между ядрами разного типа, а в параллельных режимах более сбалансированное распределение.

5.6 Результаты измерений с выключенным Hyper-Threading/SMT

Результаты измерений производительности с выключенной *Hyper-Threading/SMT* на процессорах *Intel Core i7 12700H* и *AMD Ryzen 7 5800H*. Ниже будут представлены аналогичные замеры, как и при включенной функции *Hyper-Threading/SMT*.

Следует выделить, что выключение функции *Hyper-Threading/SMT* не влияет на работу *iGPU*, потому что это совершенно независимые компоненты процессора. *iGPU* – это отдельный блок со своими вычислительными ядрами, который физически не использует технологию многопоточности *CPU*. *Hyper-Threading/SMT* работает только на *CPU*-ядрах, создавая виртуальные потоки внутри физических ядер, в то время как *iGPU* имеет собственную архитектуру и ресурсы. Они могут работать одновременно, но не пересекаются – отключение *Hyper-Threading/SMT* на *CPU* никак не затрагивает графические процессоры, которые продолжают выполнять свои задачи в обычном режиме.

В таблице 5.9 представлены результаты для линейного поиска при различных размерах тестового массива данных на процессоре *AMD Ryzen 7 5800H* с выключенной функцией *SMT*.

Таблица 5.9 – Замеры времени линейного поиска на *AMD Ryzen 7 5800H*

Размер	Последовательный	На одном ядре	Параллельный (*OpenMP*)	На *iGPU*
100	3718,1	3640,5	428,9	10008,6
500	17554,4	17718,1	1821,5	9848,5
1000	34964,2	35819,4	3103,5	9675,7
2000	69788,1	74128,8	6220,6	9642,7
5000	175551,3	187741,5	14974,1	12874,7
10000	352280,8	365513,5	29496,5	19429,9
25000	883806,4	900451,3	72842,9	43390,5
50000	1775705,2	1777875,9	141628,6	79680,4

Таблица 5.10 представляет замеры времени для программы для линейного поиска на процессоре *Intel Core i7 12700H* с выключенной функцией *Hyper-Threading*.

Таблица 5.10 – Замеры времени линейного поиска на *Intel Core i7 12700H*

Размер	Последовательный	На одном ядре	Параллельный (*OpenMP*)	На *iGPU*
100	1627,2	1931,2	350,2	22621,9
500	8280,5	8362,7	1490,6	20851,1

Продолжение таблицы 5.10

1000	16645,6	17850,1	2516,5	20764,6
2000	32780,5	34753,5	5320	20315,7
5000	82699,4	85111,3	15926,8	21530
10000	166741	169721,3	32679,8	22332,1
25000	419363,2	422872,8	83643,8	62160,7
50000	895596,2	1006823,1	237672,2	124482,5

Анализ результатов измерений с отключенной функцией *Hyper-Threading/SMT* выявил различные эффекты на процессорах *AMD* и *Intel*. На платформе *AMD* отключение не привело к снижению производительности параллельной реализации, время выполнения осталось практически неизменным. Это говорит о более эффективном распределении вычислительной нагрузки между физическими ядрами.

На процессоре Intel деактивация *Hyper-Threading*, напротив, вызвала снижение производительности. Последовательные версии алгоритмов стали выполняться примерно на 9% медленнее, а параллельная реализация *OpenMP* также показала незначительное ухудшение показателей. Это демонстрирует зависимость архитектуры *Intel* от технологии виртуальных потоков для эффективной обработки вычислительных нагрузок.

Наиболее значимым результатом является подтверждение полной независимости работы интегрированной графики от конфигурации многопоточности *CPU*. На обоих процессорах производительность *iGPU* осталась практически неизменной с отклонениями в пределах 1-2%, что находится в пределах статистической погрешности измерений.

Таблица 5.11 представляет замеры времени выполнения (в микросекундах) бинарного поиска на процессоре *AMD Ryzen 7 5800H* с выключенной функцией *SMT*.

Таблица 5.11 – Замеры времени бинарного поиска на *AMD Ryzen 7 5800H*

Размер	Последовательный	На одном ядре	Параллельный (*OpenMP*)	На *iGPU*
1000	159,4	139,6	32,8	281,5
5000	911,3	788,7	236,2	531,5
10000	2157,6	1975,3	253,2	804,6
20000	4929,7	5137,2	556,2	1251,9
50000	13541,2	14948,8	1501,1	1992,3
100000	27322,6	26847,5	3059,8	3549

Продолжение таблицы 5.11

250000	62866,8	62962,7	7512,9	6937
500000	122446,5	121778,2	17036,7	13174,4

Аналогичные измерения проведены на процессоре *Intel Core i7 12700H*. Результаты приведены в таблице 5.12

Таблица 5.12 – Замеры времени бинарного поиска на *Intel Core i7 12700H*

Размер	Последовательный	На одном ядре	Параллельный (*OpenMP*)	На *iGPU*
1000	153,1	170,5	62,2	185,9
5000	828,2	778,5	192,3	304,3
10000	1824,3	1649,3	348,8	438,4
20000	3656,4	3574	690,7	701,5
50000	9232	9152,8	2170,4	1313,3
100000	18288,6	18056,7	4051,6	2075,4
250000	45319,3	45094,1	8792,6	3648,9
500000	96912,8	104515,1	26893,5	5519,7

На *AMD Ryzen 7 5800H* наблюдается заметное снижение производительности параллельной реализации – время выполнения немного увеличилось.

На процессоре *Intel Core i7 12700H* отключение *Hyper-Threading* привело к снижению производительности параллельных вычислений для большинства размеров массивов. Наибольшее ухудшение наблюдается при обработке 500000 элементов, где время выполнения возросло примерно в 1.4 раза.

Таким образом, основным выводом является то, что отключение SMT/Hyper-Threading не приносит существенного преимущества для производительности *CPU* при выполнении бинарного поиска, а в большинстве случаев даже ухудшает показатели параллельных реализаций.

В таблице 5.13 представлены результаты измерений метрик линейного поиска на процессоре *AMD Ryzen 7 5800H*.

Таблица 5.13 – Замеры метрик линейного поиска на *AMD Ryzen 7 5800H*

N	Последовательный	На одном ядре	Параллельный (*OpenMP*)
cycles	128 037 688 388	124 835 559 563	136 853 832 144
instructions	432 325 021 040	434 219 181 691	432 979 277 800

Продолжение таблицы 5.13

insn per cycle	3,38	3,46	3,16
cache-references	556 849 982	550 696 334	592 934 871
cache-misses	10 395 929	12 249 179	12 631 092
branches	55 584 135 721	55 827 903 806	55 766 323 545
branch-misses	972 753	975 684	1 062 796
seconds time elapsed	30,619558520	30,816811539	4,854175939
seconds user	30,601850000	30,808281000	35,968073000
seconds sys	0,001999000	0,004999000	0,068698000

Аналогичные измерения были проведены и на тестируемом процессоре *Intel Core i7 12700H*, результаты этих измерений продемонстрированы в таблице 5.14.

Таблица 5.14 – Замеры метрик линейного поиска на *Intel Core i7 12700H*

N	Последовательный	На одном ядре	Параллельный (*OpenMP*)
cycles	28 480 752 515 + 82 453 409 804	15 022 743 418 + 80 579 032 676	87 579 766 162 + 105 080 800 206
instructions	76 190 330 642 + 429 116 369 796	31 460 255 267 + 424 412 456 148	336 914 693 864 + 555 364 798 548
insn per cycle	2,68/5,20	2,09/5,27	3,85/5,29
cache-references	72 473 381 + 57 875	64 419 615 + 151 590	1 061 985 + 11 975 458
cache-misses	21 952 662 + 17 605	7 411 599 + 31 777	49 673 + 19 721
branches	9 766 223 903 + 55 171 591 628	4 162 008 676 + 54 567 051 274	43 318 041 476 + 71 403 103 508
branch-misses	193 790 377 + 794 030	71 051 046 + 880 143	244 812 + 1 641 175
seconds time elapsed	17,990191395	17,607709340	4,548561683
seconds user	17,979236000	17,589247000	51,353225000

Продолжение таблицы 5.14

seconds sys	0,003333000	0,009959000	0,046632000

В таблице 5.15 представлены результаты измерений метрик бинарного поиска на процессоре *AMD Ryzen 7 5800H*. Напомним, что первый параметр в таблице соответствует энергоэффективным ядрам (до знака ‘+’ или ‘/’), а второй производительным.

Таблица 5.15 – Замеры метрик бинарного поиска на *AMD Ryzen 7 5800H*

N	Последовательный	На одном ядре	Параллельный (*OpenMP*)
cycles	8 850 746 373	9 121 072 023	9 008 321 506
instructions	15 245 390 581	15 333 890 582	15 380 604 366
insn per cycle	1,72	1,68	1,71
cache-references	46 666 332	47 505 830	46 941 004
cache-misses	12 089 537	12 156 116	11 739 405
branches	1 864 964 739	1 875 743 266	1 884 015 747
branch-misses	113 229 346	111 952 302	111 484 979
seconds time elapsed	2,111268148	2,174788083	1,888432994
seconds user	2,101866000	2,167976000	2,197793000
seconds sys	0,007995000	0,006002000	0,009994000

Аналогичные измерения были проведены и на тестируемом процессоре *Intel Core i7 12700H*, результаты этих измерений продемонстрированы в таблице 5.16.

Таблица 5.16 – Замеры метрик бинарного поиска на *Intel Core i7 12700H*

N	Последовательный	На одном ядре	Параллельный (*OpenMP*)
cycles	2 522 347 984 + 7 519 139 549	1 211 658 264 + 7 454 463 950	3 933 522 546 + 8 278 185 725
instructions	6 407 601 277 + 15 347 327 760	2 578 667 904 + 15 349 315 222	7 116 504 954 + 17 124 275 224
insn per cycle	2,54/2,04	2,13/2,06	1,81/2,07
cache-references	11 231 241 + 41 214 808	8 037 788 + 42 703 561	138 737 624 + 26 925 108

Продолжение таблицы 5.16

cache-misses	3 136 911 + 7 248 589	3 319 937 + 7 051 690	23 839 177 + 5 267 248
branches	932 705 891 + 1 876 305 365	391 865 356 + 1 877 339 123	905 016 892 + 2 087 796 992
branch-misses	3 226 086 + 111 786 274	4 250 050 + 110 819 710	33 376 416 + 125 595 438
seconds time elapsed	1,687054910	1,695869784	1,512173344
seconds user	1,672462000	1,661839000	1,936618000
seconds sys	0,013281000	0,029890000	0,056309000

Анализ аппаратных метрик с выключенной технологией *SMT/Hyper-Threading* показывает отсутствие значительных изменений в производительности обоих процессоров. Наблюдается лишь незначительное снижение эффективности, которое носит системный характер и не оказывает существенного влияния на общие результаты тестирования.

ЗАКЛЮЧЕНИЕ

На основе проведенной работы можно сделать следующие выводы. Оба процессора демонстрируют значительное ускорение при использовании параллельных версий алгоритмов поиска. 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 i7-12700H* демонстрирует высокую производительность в многопоточных задачах благодаря гибридной архитектуре, но эффективность зависит от оптимизации распределения нагрузки между *Performance*- и *Efficient*-ядрами.

Для алгоритма линейного поиска параллельная реализация с использованием *OpenMP* обеспечивает наибольшее ускорение на обоих процессрах. При этом *Intel i7-12700H* показывает лучшее абсолютное время выполнения для *CPU*-реализаций, в то время как *AMD iGPU* демонстрирует более высокую эффективность при работе с большими объемами данных.

Технология *Hyper-Threading/SMT* оказывает различное влияние на производительность. На платформе *Intel* ее отключение может приводить к снижению производительности, в то время как на *AMD* изменение производительности чаще незначительное.

Анализ аппаратных метрик показал, что *AMD Ryzen 7 5800H* демонстрирует лучшее предсказание переходов и более эффективную работу с памятью. *Intel Core i7-12700H* показывает лучшие результаты в управлении ветвлениями для алгоритма бинарного поиска.

Интегрированная графика 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 i7-12700H* демонстрирует превосходство над *AMD Ryzen 7 5800H*. Это преимущество особенно заметно в задачах, где эффективно задействуются производительные *P*-ядра гибридной архитектуры.

СПИСОК ЛИТЕРАТУРНЫХ ИСТОЧНИКОВ

[1] Сравнение процессоров *Amd Ryzen 7 5800H* и *Intel Core i7-12700H* [Электронный ресурс]. – Режим доступа: https://technical.city/ru/cpu/Ryzen-7-5800H-protiv-Core-i7-12700H. – Дата доступа: 20.09.2025.

[2] Обзор процессора *Amd Ryzen 7 5800H* [Электронный ресурс]. – Режим доступа: https://www.techpowerup.com/cpu-specs/ryzen-7-5800h.c2368 – Дата доступа: 20.09.2025.

[3] Обзор процессора *Intel Core i7-12700H* [Электронный ресурс]. – Режим доступа: https://www.techpowerup.com/cpu-specs/core-i7-12700h.c2537 – Дата доступа: 20.09.2025.

[4] Результаты в бенчмарках [Электронный ресурс]. – https://gadgetversus.com/processor/amd-ryzen-7-5800h-vs-intel-core-i7-12700h/. – Дата доступа: 20.09.2025.

[5] Компоненты *Linux*: структура и назначение [Электронный ресурс]. – Режим доступа: https://www.cleverence.ru/articles/auto-busines/-komponenty-linux-struktura-i-naznachenie/. – Дата доступа: 20.09.2025.

[6] Ядро ОС *Linux* [Электронный ресурс]. – Режим доступа: https://younglinux.info/bash/kernellinux. – Дата доступа: 20.09.2025.

[7] Системные библиотеки Linux [Электронный ресурс]. – Режим доступа: https://www.youstable.com/ru/blog/architecture-of-the-linux-operating-system/. – Дата доступа: 20.09.2025

[8] Архитектура ОС Linux [Электронный ресурс]. – Режим доступа: http://opensourcerules.net/arhitektura-linux.html. – Дата доступа: 20.09.2025.

[9] Файловые системы в Linux: их структура и типы [Электронный ресурс]. – Режим доступа: https://timeweb.com/ru/community/articles/struktura-i-tipy-faylovyh-sistem-v-linux. – Дата доступа: 20.09.2025.

[10] История операционной системы Linux [Электронный ресурс]. – Режим доступа: https://timeweb.com/ru/community/articles/kratkaya-istoriya-linux-1?ysclid=mfs6r15l7a176645606. – Дата доступа: 20.09.2025.

[11] Версии ядра Linux [Электронный ресурс]. – Режим доступа: https://kernelnewbies.org/LinuxVersions. – Дата доступа: 20.09.2025.

[12] Причины выбрать, в качестве дистрибутива Linux Manjaro [Электронный ресурс]. – Режим доступа: https://lifehacker.ru/pochemu-manjaro-luchshe-ubuntu/?ysclid=mfs791et30636469569. – Дата доступа: 20.09.2025.

[13] Директивы OpenMP [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/cpp/parallel/openmp/reference/. – Дата доступа: 22.10.2025.

[14] Microsoft: Processes and threads [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/procthread/processes-and-threads. – Дата доступа: 22.10.2025.

[15] Профилирование и трейсинг с perf [Электронный ресурс]. – Режим доступа: https://habr.com/ru/companies/first/articles/442738/. – Дата доступа: 27.10.2025.

ПРИЛОЖЕНИЕ А
(обязательное)
Справка о проверке на заимствования

Рисунок А.1 – Справка о проверке на заимствования

ПРИЛОЖЕНИЕ Б
(обязательное)
Листинг программного кода

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <chrono>
#include <random>
#include <omp.h>
#include <cstring>
#include <iomanip>
#include <fstream>
#include <sched.h>
#include <pthread.h>
#include <hwloc.h>
#include </usr/include/gegl-0.4/opencl/opencl.hpp>
#include <numeric>

#define ITERATIONS 10
#define LINEAR_DATA_SIZE 100000
#define BINARY_DATA_SIZE 10000000

// Глобальная топология hwloc
hwloc_topology_t topology;

struct SearchResults
{
    long double linear_time;
    long double binary_time;
    size_t linear_found;
    size_t binary_found;
};

// Класс для GPU
class GPUSearcher
{
private:
    cl::Context context;
    cl::CommandQueue queue;
    cl::Program program;
    cl::Kernel linear_search_kernel;
    cl::Kernel binary_search_kernel;
    bool available;

public:
    GPUSearcher() : available(false)
    {
        try
        {
            std::vector<cl::Platform> platforms;
            cl::Platform::get(&platforms);

            if (platforms.empty())
            {
                std::cerr << "OpenCL платформы не найдены" << std::endl;
                return;
            }

            cl::Platform selected_platform;
            cl::Device selected_device;
            bool found_gpu = false;
```

```
            for (auto &platform : platforms)
            {
                std::vector<cl::Device> devices;
                platform.getDevices(CL_DEVICE_TYPE_GPU, &devices);

                for (auto &device : devices)
                {
                    std::string                  device_name                  =
device.getInfo<CL_DEVICE_NAME>();
                    std::string                 platform_name                 =
platform.getInfo<CL_PLATFORM_NAME>();

                    std::cout << "Найдено GPU устройство: " << device_name
                              <<  "  на  платформе:  "  <<  platform_name  <<
std::endl;

                    // Предпочтение Intel iGPU
                    if (platform_name.find("Intel") != std::string::npos)
                    {
                        selected_platform = platform;
                        selected_device = device;
                        found_gpu = true;
                        std::cout << "Выбрана Intel iGPU" << std::endl;
                        break;
                    }
                    // Или любая другая GPU
                    else if (!found_gpu)
                    {
                        selected_platform = platform;
                        selected_device = device;
                        found_gpu = true;
                        std::cout  <<  "Выбрана  GPU:  "  <<  device_name  <<
std::endl;
                    }
                }

                if (found_gpu)
                    break;
            }

            if (!found_gpu)
            {
                std::cerr << "GPU устройства не найдены" << std::endl;
                return;
            }

            context = cl::Context(selected_device);
            queue = cl::CommandQueue(context, selected_device);

            std::string kernel_source = R"(
                __kernel void linear_search(__global const int* data,
                                            __global const int* targets,
                                            __global int* results,
                                            int data_size,
                                            int targets_size) {
                    int tid = get_global_id(0);
                    if (tid >= targets_size) return;

                    int target = targets[tid];
                    int found = 0;

                    for (int i = 0; i < data_size; i++) {
                        if (data[i] == target) {
```

```
                            found = 1;
                            break;
                        }
                    }
                    results[tid] = found;
                }

                __kernel void binary_search(__global const int* sorted_data,
                                            __global const int* targets,
                                            __global int* results,
                                            int data_size,
                                            int targets_size) {
                    int tid = get_global_id(0);
                    if (tid >= targets_size) return;

                    int target = targets[tid];
                    int left = 0;
                    int right = data_size - 1;
                    int found = 0;

                    while (left <= right) {
                        int mid = left + (right - left) / 2;
                        if (sorted_data[mid] == target) {
                            found = 1;
                            break;
                        } else if (sorted_data[mid] < target) {
                            left = mid + 1;
                        } else {
                            right = mid - 1;
                        }
                    }
                    results[tid] = found;
                }
            )";

            cl::Program::Sources sources;
            sources.push_back({kernel_source.c_str(),
kernel_source.length()});

            program = cl::Program(context, sources);
            program.build();

            linear_search_kernel = cl::Kernel(program, "linear_search");
            binary_search_kernel = cl::Kernel(program, "binary_search");
            available = true;

            std::cout << "OpenCL инициализирован успешно" << std::endl;
        }
        catch (const std::exception &e)
        {
            std::cerr << "Ошибка инициализации GPU: " << e.what() << std::endl;
            available = false;
        }
    }

    bool isAvailable() const { return available; }

    SearchResults search_gpu_timed(const std::vector<int> &linear_data,
                                   const std::vector<int> &binary_data,
                                   const std::vector<int> &linear_targets,
                                   const std::vector<int> &binary_targets)
    {
        if (!available)
        {
```

```
            std::cerr << "GPU недоступен" << std::endl;
            return {0, 0, 0, 0};
        }

        size_t linear_data_size = linear_data.size();
        size_t binary_data_size = binary_data.size();
        size_t linear_targets_size = linear_targets.size();
        size_t binary_targets_size = binary_targets.size();

        SearchResults results = {0, 0, 0, 0};

        if (linear_targets_size <= 50000)
        {
            cl::Buffer     linear_data_buf(context,     CL_MEM_READ_ONLY     |
CL_MEM_COPY_HOST_PTR,
                                           linear_data_size     *     sizeof(int),
const_cast<int *>(linear_data.data()));
            cl::Buffer    linear_targets_buf(context,    CL_MEM_READ_ONLY    |
CL_MEM_COPY_HOST_PTR,
                                              linear_targets_size * sizeof(int),
const_cast<int *>(linear_targets.data()));
            cl::Buffer linear_results_buf(context, CL_MEM_WRITE_ONLY,
                                              linear_targets_size              *
sizeof(int));

            auto linear_start = std::chrono::high_resolution_clock::now();

            linear_search_kernel.setArg(0, linear_data_buf);
            linear_search_kernel.setArg(1, linear_targets_buf);
            linear_search_kernel.setArg(2, linear_results_buf);
            linear_search_kernel.setArg(3,
static_cast<int>(linear_data_size));
            linear_search_kernel.setArg(4,
static_cast<int>(linear_targets_size));

            queue.enqueueNDRangeKernel(linear_search_kernel, cl::NullRange,
                                       cl::NDRange(linear_targets_size),
cl::NullRange);
            queue.finish();

            std::vector<int> linear_results(linear_targets_size);
            queue.enqueueReadBuffer(linear_results_buf, CL_TRUE, 0,
                                    linear_targets_size     *     sizeof(int),
linear_results.data());

            auto linear_end = std::chrono::high_resolution_clock::now();
            results.linear_time                                                =
std::chrono::duration_cast<std::chrono::microseconds>(
                                       linear_end - linear_start)
                                       .count();
            results.linear_found = std::accumulate(linear_results.begin(),
                                                   linear_results.end(), 0);
        }

        cl::Buffer      binary_data_buf(context,      CL_MEM_READ_ONLY      |
CL_MEM_COPY_HOST_PTR,
                                        binary_data_size       *       sizeof(int),
const_cast<int *>(binary_data.data()));
        cl::Buffer     binary_targets_buf(context,     CL_MEM_READ_ONLY     |
CL_MEM_COPY_HOST_PTR,
                                          binary_targets_size   *   sizeof(int),
const_cast<int *>(binary_targets.data()));
        cl::Buffer binary_results_buf(context, CL_MEM_WRITE_ONLY,
                                          binary_targets_size * sizeof(int));
```

```
        auto binary_start = std::chrono::high_resolution_clock::now();

        binary_search_kernel.setArg(0, binary_data_buf);
        binary_search_kernel.setArg(1, binary_targets_buf);
        binary_search_kernel.setArg(2, binary_results_buf);
        binary_search_kernel.setArg(3, static_cast<int>(binary_data_size));
        binary_search_kernel.setArg(4, static_cast<int>(binary_targets_size));

        queue.enqueueNDRangeKernel(binary_search_kernel, cl::NullRange,
                                   cl::NDRange(binary_targets_size),
cl::NullRange);
        queue.finish();

        std::vector<int> binary_results(binary_targets_size);
        queue.enqueueReadBuffer(binary_results_buf, CL_TRUE, 0,
                                binary_targets_size       *       sizeof(int),
binary_results.data());

        auto binary_end = std::chrono::high_resolution_clock::now();
        results.binary_time                                                    =
std::chrono::duration_cast<std::chrono::microseconds>(
                                   binary_end - binary_start)
                                   .count();
        results.binary_found = std::accumulate(binary_results.begin(),
                                               binary_results.end(), 0);

        return results;
    }
};

// Глобальный экземпляр GPU поисковика
GPUSearcher g_gpu_searcher;

// Инициализация hwloc
void initTopology()
{
    hwloc_topology_init(&topology);
    hwloc_topology_load(topology);
}

// Освобождение ресурсов hwloc
void cleanupTopology()
{
    hwloc_topology_destroy(topology);
}

// Установка аффинити к конкретному физическому ядру
void setAffinityToPhysicalCore(int physicalCoreId)
{
    int depth = hwloc_get_type_or_below_depth(topology, HWLOC_OBJ_CORE);
    hwloc_obj_t core = hwloc_get_obj_by_depth(topology, depth, physicalCoreId);

    if (core == NULL)
    {
        std::cerr << "Ошибка: физическое ядро " << physicalCoreId << " не
найдено" << std::endl;
        return;
    }

    if (hwloc_set_cpubind(topology, core->cpuset, HWLOC_CPUBIND_THREAD) != 0)
    {
        std::cerr << "Ошибка установки аффинити к физическому ядру " <<
physicalCoreId << std::endl;
```

```
        char *str;
        hwloc_bitmap_asprintf(&str, core->cpuset);
        std::cerr << "CPUSet: " << str << std::endl;
        free(str);
    }
}

size_t linear_search(const std::vector<int> &data, int target)
{
    for (size_t i = 0; i < data.size(); ++i)
    {
        if (data[i] == target)
        {
            return i;
        }
    }
    return data.size();
}

size_t binary_search(const std::vector<int> &data, int target)
{
    size_t left = 0;
    size_t right = data.size();
    while (left < right)
    {
        size_t mid = left + (right - left) / 2;
        if (data[mid] == target)
        {
            return mid;
        }
        else if (data[mid] < target)
        {
            left = mid + 1;
        }
        else
        {
            right = mid;
        }
    }
    return data.size();
}

SearchResults search_sequential_timed(const std::vector<int> &linear_data,
const std::vector<int> &binary_data,
                                      const std::vector<int> &linear_targets,
const std::vector<int> &binary_targets)
{
    size_t linear_found = 0;
    size_t binary_found = 0;

    // Линейный поиск
    auto linear_start = std::chrono::high_resolution_clock::now();
    for (size_t i = 0; i < linear_targets.size(); ++i)
    {
        if       (linear_search(linear_data,       linear_targets[i])       !=
linear_data.size())
        {
            linear_found++;
        }
    }
    auto linear_end = std::chrono::high_resolution_clock::now();
    auto                                linear_time                                =
std::chrono::duration_cast<std::chrono::microseconds>(linear_end            -
linear_start).count();
```

```
    // Бинарный поиск
    auto binary_start = std::chrono::high_resolution_clock::now();
    for (size_t i = 0; i < binary_targets.size(); ++i)
    {
        if       (binary_search(binary_data,       binary_targets[i])       !=
binary_data.size())
        {
            binary_found++;
        }
    }
    auto binary_end = std::chrono::high_resolution_clock::now();
    auto                              binary_time                              =
std::chrono::duration_cast<std::chrono::microseconds>(binary_end             -
binary_start).count();

    return     {static_cast<long     double>(linear_time),     static_cast<long
double>(binary_time), linear_found, binary_found};
}

SearchResults search_openmp_timed(const std::vector<int> &linear_data, const
std::vector<int> &binary_data,
                                  const    std::vector<int>    &linear_targets,
const std::vector<int> &binary_targets)
{
    size_t linear_found = 0;
    size_t binary_found = 0;

    auto linear_start = std::chrono::high_resolution_clock::now();
#pragma omp parallel for reduction(+ : linear_found)
    for (size_t i = 0; i < linear_targets.size(); ++i)
    {
        if       (linear_search(linear_data,       linear_targets[i])       !=
linear_data.size())
        {
            linear_found++;
        }
    }
    auto linear_end = std::chrono::high_resolution_clock::now();
    auto                              linear_time                              =
std::chrono::duration_cast<std::chrono::microseconds>(linear_end             -
linear_start).count();

    auto binary_start = std::chrono::high_resolution_clock::now();
#pragma omp parallel for reduction(+ : binary_found)
    for (size_t i = 0; i < binary_targets.size(); ++i)
    {
        if       (binary_search(binary_data,       binary_targets[i])       !=
binary_data.size())
        {
            binary_found++;
        }
    }
    auto binary_end = std::chrono::high_resolution_clock::now();
    auto                              binary_time                              =
std::chrono::duration_cast<std::chrono::microseconds>(binary_end             -
binary_start).count();

    return     {static_cast<long     double>(linear_time),     static_cast<long
double>(binary_time), linear_found, binary_found};
}

SearchResults  search_gpu_timed(const  std::vector<int>  &linear_data,  const
std::vector<int> &binary_data,
```

```
                                const std::vector<int> &linear_targets, const
std::vector<int> &binary_targets)
{
    return      g_gpu_searcher.search_gpu_timed(linear_data,      binary_data,
linear_targets, binary_targets);
}

SearchResults search_single_core_timed(const std::vector<int> &linear_data,
const std::vector<int> &binary_data,
                                       const std::vector<int> &linear_targets,
const std::vector<int> &binary_targets, int coreId)
{
    setAffinityToPhysicalCore(coreId);
    return search_sequential_timed(linear_data, binary_data, linear_targets,
binary_targets);
}

template <typename TimeUnits>
SearchResults        TestSearchTimed(SearchResults        (*search_func)(const
std::vector<int> &, const std::vector<int> &,
                                                                    const
std::vector<int> &, const std::vector<int> &),
                                const   std::vector<int>   &linear_data,   const
std::vector<int> &binary_data,
                                const std::vector<int> &linear_targets, const
std::vector<int> &binary_targets,
                                int iterations)
{
    long double linear_sum = 0;
    long double binary_sum = 0;
    size_t linear_found = 0;
    size_t binary_found = 0;

    for (int i = 0; i < iterations; i++)
    {
        auto results = search_func(linear_data, binary_data, linear_targets,
binary_targets);
        linear_sum += results.linear_time;
        binary_sum += results.binary_time;
        linear_found = results.linear_found;
        binary_found = results.binary_found;
    }

    return {linear_sum / iterations, binary_sum / iterations, linear_found,
binary_found};
}

SearchResults      TestSingleCoreTimed(SearchResults      (*search_func)(const
std::vector<int> &, const std::vector<int> &,
                                                                         const
std::vector<int> &, const std::vector<int> &),
                                    const std::vector<int> &linear_data, const
std::vector<int> &binary_data,
                                    const   std::vector<int>   &linear_targets,
const std::vector<int> &binary_targets,
                                    int iterations, int coreId)
{
    long double linear_sum = 0;
    long double binary_sum = 0;
    size_t linear_found = 0;
    size_t binary_found = 0;

    for (int i = 0; i < iterations; i++)
    {
```

```
        setAffinityToPhysicalCore(coreId);
        auto results = search_func(linear_data, binary_data, linear_targets,
binary_targets);
        linear_sum += results.linear_time;
        binary_sum += results.binary_time;
        linear_found = results.linear_found;
        binary_found = results.binary_found;
    }

    return {linear_sum / iterations, binary_sum / iterations, linear_found,
binary_found};
}

void printTopologyInfo()
{
    int depth = hwloc_get_type_or_below_depth(topology, HWLOC_OBJ_CORE);
    int num_cores = hwloc_get_nbobjs_by_depth(topology, depth);

    std::cout                                                                <<
"========================================================================
==================\n";
    std::cout << "ИНФОРМАЦИЯ О ТОПОЛОГИИ ПРОЦЕССОРА\n";
    std::cout                                                                <<
"========================================================================
==================\n";
    std::cout << "Обнаружено физических ядер: " << num_cores << std::endl;

    for (int i = 0; i < num_cores; i++)
    {
        hwloc_obj_t core = hwloc_get_obj_by_depth(topology, depth, i);
        char *cpuset_str;
        hwloc_bitmap_asprintf(&cpuset_str, core->cpuset);
        std::cout << "Физическое ядро " << i << ": логические CPU " << cpuset_str
<< std::endl;
        free(cpuset_str);
    }
    std::cout                                                                <<
"========================================================================
==================\n\n";
}

void printResultsTable(int linear_targets_size, int binary_targets_size,
                       const SearchResults &seq_results, const SearchResults
&omp_results,
                       const     SearchResults     &single_results,     const
SearchResults &gpu_results)
{
    double seq_ratio = (seq_results.binary_time > 0) ? seq_results.linear_time
/ seq_results.binary_time : 0;
    double omp_ratio = (omp_results.binary_time > 0) ? omp_results.linear_time
/ omp_results.binary_time : 0;
    double    single_ratio    =    (single_results.binary_time    >    0)    ?
single_results.linear_time / single_results.binary_time : 0;
    double gpu_ratio = (gpu_results.binary_time > 0) ? gpu_results.linear_time
/ gpu_results.binary_time : 0;

    std::cout                                                                <<
"========================================================================
==================\n";
    std::cout << "СРАВНЕНИЕ ПРОИЗВОДИТЕЛЬНОСТИ ЛИНЕЙНОГО И БИНАРНОГО ПОИСКА\n";
    std::cout                                                                <<
"========================================================================
==================\n";
```

```
    std::cout << "Размер данных для линейного поиска: " << LINEAR_DATA_SIZE <<
" элементов\n";
    std::cout << "Размер данных для бинарного поиска: " << BINARY_DATA_SIZE <<
" элементов\n";
    std::cout   <<   "Количество   целей   для   линейного   поиска:   "   <<
linear_targets_size << "\n";
    std::cout   <<   "Количество   целей   для   бинарного   поиска:   "   <<
binary_targets_size << "\n";
    std::cout << "Потоков OpenMP: " << omp_get_max_threads() << "\n";
    std::cout << "GPU доступен: " << (g_gpu_searcher.isAvailable() ? "Да" :
"Нет") << "\n\n";

    std::cout                                                                 <<
"=============================================================================
==================\n";
    std::cout << "ТАБЛИЦА ПРОИЗВОДИТЕЛЬНОСТИ (время в микросекундах)\n";
    std::cout                                                                 <<
"=============================================================================
==================\n";
    std::cout << "-------------------------------------------------------------
-----------------------------------\n";
    std::cout << "| " << std::setw(13) << "Потоки" << " | "
              << std::setw(15) << "Линейный(мкс)" << " | "
              << std::setw(15) << "Бинарный(мкс)" << " | "
              << std::setw(10) << "Найдено_Л" << " | "
              << std::setw(10) << "Найдено_Б" << " |\n";
    std::cout << "-------------------------------------------------------------
-----------------------------------\n";

    std::cout << "| " << std::setw(7) << "1(seq)" << " | "
              <<          std::setw(15)          <<          static_cast<long
long>(seq_results.linear_time) << " | "
              <<          std::setw(15)          <<          static_cast<long
long>(seq_results.binary_time) << " | "
              << std::setw(10) << seq_results.linear_found << " | "
              << std::setw(10) << seq_results.binary_found << " |\n";

    std::cout << "| " << omp_get_max_threads() << "(omp)" << " | "
              <<          std::setw(15)          <<          static_cast<long
long>(omp_results.linear_time) << " | "
              <<          std::setw(15)          <<          static_cast<long
long>(omp_results.binary_time) << " | "
              << std::setw(10) << omp_results.linear_found << " | "
              << std::setw(10) << omp_results.binary_found << " |\n";

    std::cout << "| " << std::setw(6) << "1(core)" << " | "
              <<          std::setw(15)          <<          static_cast<long
long>(single_results.linear_time) << " | "
              <<          std::setw(15)          <<          static_cast<long
long>(single_results.binary_time) << " | "
              << std::setw(10) << single_results.linear_found << " | "
              << std::setw(10) << single_results.binary_found << " |\n";

    if (g_gpu_searcher.isAvailable())
    {
        std::cout << "| " << std::setw(6) << "GPU" << " | "
                  <<         std::setw(15)         <<         static_cast<long
long>(gpu_results.linear_time) << " | "
                  <<         std::setw(15)         <<         static_cast<long
long>(gpu_results.binary_time) << " | "
                  << std::setw(10) << gpu_results.linear_found << " | "
                  << std::setw(10) << gpu_results.binary_found << " |\n";
    }
```

```
    std::cout << "-----------------------------------------------------------
-----------------------------------\n\n";
}

// Сохранение в файл
void saveResultsToFile(const std::vector<std::tuple<int, int, SearchResults,
SearchResults, SearchResults, SearchResults>> &results)
{
    std::ofstream file("search_results_with_openCL.csv");
    if (!file.is_open())
    {
        std::cerr << "Ошибка открытия файла для записи!" << std::endl;
        return;
    }

    file                                                                    <<
"Линейные_Цели,Бинарные_Цели,Линейный_Посл,Бинарный_Посл,Линейный_OMP,Бинарны
й_OMP,Линейный_Ядро,Бинарный_Ядро,Линейный_GPU,Бинарный_GPU,"
         << "Отношение_Посл,Отношение_OMP,Отношение_Ядро,Отношение_GPU\n";

    for (const auto &result : results)
    {
        int linear_size, binary_size;
        SearchResults seq, omp, single, gpu;
        std::tie(linear_size, binary_size, seq, omp, single, gpu) = result;

        file << linear_size << "," << binary_size << ","
             << std::fixed << std::setprecision(2) << seq.linear_time << ","
             << std::fixed << std::setprecision(2) << seq.binary_time << ","
             << std::fixed << std::setprecision(2) << omp.linear_time << ","
             << std::fixed << std::setprecision(2) << omp.binary_time << ","
             << std::fixed << std::setprecision(2) << single.linear_time << ","
             << std::fixed << std::setprecision(2) << single.binary_time << ","
             << std::fixed << std::setprecision(2) << gpu.linear_time << ","
             << std::fixed << std::setprecision(2) << gpu.binary_time << ","
             << std::fixed << std::setprecision(2) << (seq.binary_time > 0 ?
seq.linear_time / seq.binary_time : 0) << ","
             << std::fixed << std::setprecision(2) << (omp.binary_time > 0 ?
omp.linear_time / omp.binary_time : 0) << ","
             << std::fixed << std::setprecision(2) << (single.binary_time > 0
? single.linear_time / single.binary_time : 0) << ","
             << std::fixed << std::setprecision(2) << (gpu.binary_time > 0 ?
gpu.linear_time / gpu.binary_time : 0) << "\n";
    }
    file.close();
}

void    generateTestData(std::vector<int>    &linear_data,    std::vector<int>
&binary_data,
                         std::vector<int>  &all_linear_targets,  std::vector<int>
&all_binary_targets)
{
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<int> dist(1, BINARY_DATA_SIZE * 10);

    std::cout << "Генерация данных для линейного поиска (" << LINEAR_DATA_SIZE
<< " элементов)...\n";
    linear_data.resize(LINEAR_DATA_SIZE);
    for (size_t i = 0; i < LINEAR_DATA_SIZE; ++i)
    {
        linear_data[i] = dist(gen);
    }
```

```
    std::cout << "Генерация данных для бинарного поиска (" << BINARY_DATA_SIZE
<< " элементов)...\n";
    binary_data.resize(BINARY_DATA_SIZE);
    for (size_t i = 0; i < BINARY_DATA_SIZE; ++i)
    {
        binary_data[i] = dist(gen);
    }
    std::sort(binary_data.begin(), binary_data.end());

    int max_linear_targets = 50000;
    int max_binary_targets = 500000;

    std::cout << "Генерация целей для линейного поиска (" << max_linear_targets
<< " целей)...\n";
    all_linear_targets.resize(max_linear_targets);
    for (size_t i = 0; i < max_linear_targets; ++i)
    {
        if (i % 2 == 0)
        {
            all_linear_targets[i] = linear_data[dist(gen) % LINEAR_DATA_SIZE];
        }
        else
        {
            all_linear_targets[i] = dist(gen);
        }
    }

    std::cout << "Генерация целей для бинарного поиска (" << max_binary_targets
<< " целей)...\n";
    all_binary_targets.resize(max_binary_targets);
    for (size_t i = 0; i < max_binary_targets; ++i)
    {
        if (i % 2 == 0)
        {
            all_binary_targets[i] = binary_data[dist(gen) % BINARY_DATA_SIZE];
        }
        else
        {
            all_binary_targets[i] = dist(gen);
        }
    }

    std::cout << "Генерация данных завершена!\n\n";
}

int main()
{
    initTopology();
    printTopologyInfo();

    int max_threads = omp_get_max_threads();
    omp_set_num_threads(max_threads);

    std::vector<int>     linear_data,     binary_data,     all_linear_targets,
all_binary_targets;
    generateTestData(linear_data,       binary_data,       all_linear_targets,
all_binary_targets);

    std::vector<int> linear_test_sizes = {100, 500, 1000, 2000, 5000, 10000,
25000, 50000};
    std::vector<int> binary_test_sizes = {1000, 5000, 10000, 20000, 50000,
100000, 250000, 500000};
```

```
    std::vector<std::tuple<int,     int,     SearchResults,     SearchResults,
SearchResults, SearchResults>> results;

    for (size_t i = 0; i < linear_test_sizes.size(); ++i)
    {
        int linear_size = linear_test_sizes[i];
        int binary_size = binary_test_sizes[i];

        std::vector<int>            linear_targets(all_linear_targets.begin(),
all_linear_targets.begin() + linear_size);
        std::vector<int>            binary_targets(all_binary_targets.begin(),
all_binary_targets.begin() + binary_size);

        std::cout << "Тестирование с " << linear_size << " целями для линейного
поиска и "
                  << binary_size << " целями для бинарного поиска...\n";

        auto seq_results = TestSearchTimed<std::chrono::microseconds>(
            search_sequential_timed, linear_data, binary_data, linear_targets,
binary_targets, ITERATIONS);

        auto omp_results = TestSearchTimed<std::chrono::microseconds>(
            search_openmp_timed,  linear_data,  binary_data,  linear_targets,
binary_targets, ITERATIONS);

        auto single_results = TestSingleCoreTimed(
            search_sequential_timed, linear_data, binary_data, linear_targets,
binary_targets, ITERATIONS, 0);

        auto gpu_results = TestSearchTimed<std::chrono::microseconds>(
            search_gpu_timed,   linear_data,   binary_data,   linear_targets,
binary_targets, ITERATIONS);

        results.emplace_back(linear_size,      binary_size,      seq_results,
omp_results, single_results, gpu_results);
        printResultsTable(linear_size, binary_size, seq_results, omp_results,
single_results, gpu_results);
    }

    saveResultsToFile(results);
    std::cout << "Результаты сохранены в search_results_with_openCL.csv" <<
std::endl;

    cleanupTopology();

    return 0;
}
```

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

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

ПРИЛОЖЕНИЕ Д
(обязательное)
Графики сравнений производительности процессоров

ПРИЛОЖЕНИЕ Е
(обязательное)
Графическое представление нагрузки на ядра процессоров

ПРИЛОЖЕНИЕ Ж
(обязательное)
Ведомость курсового проекта