Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет   
информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина «Архитектуры вычислительных систем»

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

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту

на тему

**РАСПРЕДЕЛЕННЫЕ ВЫЧИСЛЕНИЯ НА НЕСКОЛЬКИХ ПРОЦЕССОРАХ (CPU, GPU)**

БГУИР КП 1-40 04 01 011 ПЗ

Студент            Е. А. Киселёва

Руководитель            А. Н. Марков

Нормоконтролер            А. А. Калиновская

Минск 2023

**СОДЕРЖАНИЕ**

[Введение 6](#_Toc152329604)

[1 Архитектура вычислительной системы 8](#_Toc152329605)

[1.1 Структура и архитектура вычислительной системы 8](#_Toc152329606)

[1.2 История, версии и достоинства 13](#_Toc152329607)

[1.3 Анализ выбранной вычислительной системы и обоснование выбора 17](#_Toc152329608)

[2 Платформа программного обеспечения 19](#_Toc152329609)

[2.1 Структура и архитектура платформы 19](#_Toc152329610)

[2.2 История, версии и достоинства 21](#_Toc152329611)

[2.3 Анализ программного обеспечения для написания программы и   
          обоснование выбора платформы 24](#_Toc152329612)

[3 теоретическое обоснование разработки    программного продукта 25](#_Toc152329613)

[3.1 Обоснование необходимости разработки 25](#_Toc152329614)

[3.2 Технологии программирования, используемые для решения  
          поставленных задач 25](#_Toc152329615)

[3.3 Связь архитектуры вычислительной системы с разрабатываемым  
          программным обеспечением 29](#_Toc152329616)

[4 Проектирование функциональных возможностей программы 31](#_Toc152329617)

[4.1 Функциональные возможности программы 31](#_Toc152329618)

[4.2 Обоснования и описание функций программы 32](#_Toc152329619)

[5 Архитектура разрабатываемой программы 36](#_Toc152329620)

[5.1 Общая структура программы 36](#_Toc152329621)

[5.2 Описание функциональной схемы программы 36](#_Toc152329622)

[5.3 Описание блок схемы алгоритма программы 38](#_Toc152329623)

[Заключение 39](#_Toc152329624)

[Список использованных источников 40](#_Toc152329625)

[Приложение А (обязательное) Ведомость документов 42](#_Toc152329626)

[Приложение Б (обязательное) Листинг исходного кода 43](#_Toc152329627)

[Приложение В (обязательное) Функциональная схема алгоритма,   
                                    реализующего программное средство 47](#_Toc152329628)

[Приложение Г (обязательное) Блок схема алгоритма, реализующего   
                                 программное средство 48](#_Toc152329629)

[Приложение Д (обязательное) Графический интерфейс пользователя 49](#_Toc152329630)

****ВВЕДЕНИЕ****

Курсовая работа на заданную тему предоставляет возможность глубокого погружения в современные тенденции развития вычислительных систем. Распределенные вычисления на нескольких процессорах, включая CPU и GPU, становятся все более важными в контексте растущей сложности задач, требующих высокой производительности.

Причины актуальности курсовой работы:

1 С ростом объемов данных и сложности задач возрастает потребность в эффективных методах их обработки. Распределенные вычисления позволяют использовать ресурсы нескольких процессоров, ускоряя вычисления во много раз.

2 В современных вычислительных системах наряду с многоядерными процессорами все чаще используются графические ускорители (GPU). Эффективное распределение задач между этими устройствами является важной задачей, которая позволяет повысить производительность вычислительных систем.

3 Распределенные вычисления являются одной из важнейших технологий в современном мире. Они активно применяются в различных областях, таких как машинное обучение, научные исследования, симуляции, обработка больших данных, финансы, медицина. Из этого следует, что понимание того, как эффективно использовать ресурсы различных процессоров, очень важно.

4 Распределенные вычисления позволяют решать сложные задачи, которые невозможно или слишком дорого решать на одном процессоре.

Цель данной курсовой работы заключается в том, чтобы разработать и реализовать приложение, использующее распределенные вычисления на нескольких процессорах (*CPU*, *GPU*) в контексте современных вычислительных систем, для повышения эффективности, производительности и оптимизации использования ресурсов. Для достижения данной цели были поставлены следующие задачи:

1 Изучить актуальные научные статьи, публикации и ресурсы по распределенным вычислениям на *CPU* и *GPU*. Определить основные технологические и методологические подходы к распределенным вычислениям в современных системах.

2 Проанализировать архитектуры *CPU* и *GPU*. Исследовать особенности многоядерных процессоров и графических ускорителей. Оценить преимущества и ограничения каждого типа процессора в контексте параллельных вычислений.

3 Изучить стратегии распределения задач и данных между *CPU* и *GPU*. Оценить эффективность механизмов управления ресурсами в распределенных вычислительных системах.

4 Разработать структуру приложения, использующего распределенные вычисления на нескольких процессорах (*CPU*, *GPU*) в контексте современных вычислительных систем, для повышения эффективности, производительности и оптимизации использования ресурсов.

5 Создать удобный и информативный интерфейс пользователя для приложения, чтобы он был простым, понятным и доступным для обширного круга пользователей.

Таким образом данная курсовая работа ориентирована на разработку и внедрение приложения, использующего принципы и технологии распределенных вычислений на нескольких процессорах (*CPU*, *GPU*). Выбранное приложение для решения систем линейных алгебраических уравнений (СЛАУ) методом *LU*-разложения служит примером задачи, которая может быть эффективно решена с применением распределенных вычислений.

# 1 Архитектура вычислительной системы

## 1.1 Структура и архитектура вычислительной системы

Вычислительная система – это совокупность аппаратных и программных компонентов, предназначенных для обработки информации и выполнения вычислительных задач. Эта система включает в себя несколько ключевых элементов:

1 Аппаратная часть: процессор, также известный как центральный процессор (*CPU*), является основным вычислительным мозгом системы, ответственным за выполнение инструкций и обработку данных. Оперативная память (*RAM*) представляет собой временное хранилище данных и инструкций, обеспечивающее быстрый доступ для процессора. Хранение данных, включая жесткие диски, *SSD* и другие устройства, служит постоянным хранилищем данных, используемым для сохранения операционной системы, программ и файлов.

2 Ввод-вывод (*I*/*O*) устройства: Клавиатура, мышь, монитор: Устройства ввода и вывода, обеспечивающие взаимодействие пользователя с системой. Принтеры, сканеры, сетевые карты: Устройства для обработки и передачи данных вне системы.

3 Системная плата и шина: Материнская плата: Основная плата, на которой располагаются основные компоненты системы, такие как процессор, память и другие. Программное обеспечение: Операционная система (ОС):

4 Программное обеспечение, управляющее ресурсами компьютера и обеспечивающее взаимодействие между аппаратной частью и пользователями или приложениями. Прикладное программное обеспечение (ПО): Программы и приложения, выполняющие конкретные задачи, такие как текстовые редакторы, браузеры, графические редакторы и т.д. Драйверы: Специальные программы, обеспечивающие взаимодействие с аппаратными устройствами.

5 Сетевые соединения: Сетевые адаптеры: Устройства, обеспечивающие подключение к сети для обмена данными между компьютерами.

Вычислительные системы могут быть различными по своей сложности и назначению, от персональных компьютеров до высокопроизводительных серверов. Они играют ключевую роль в обработке информации, решении задач и обеспечении взаимодействия человека с технологией.

Архитектура вычислительной системы относится к структуре и организации её аппаратных и программных компонентов. Эта концепция включает в себя различные уровни абстракции, от общей организации системы до деталей функционирования её отдельных элементов. [1]

**1.1.1** Выбранная вычислительная система

Курсовой проект разрабатывается на основе операционной системы *Windows* 10 и аппаратного обеспечения ноутбука изготовителя *Acer*, модели *Swift* *SF*314-59. Эта вычислительная система включает в себя следующие ключевые компоненты:

1 Процессор *Intel* *Core* *i*5: Центральный процессор представляет собой главный вычислительный двигатель компьютера, обеспечивая высокую производительность и играя важную роль в общей архитектуре системы.

2 Оперативная память (*RAM*): Оперативная память служит для временного хранения данных, используемых программами в процессе выполнения. Этот компонент играет ключевую роль в архитектуре системы, влияя на её производительность в зависимости от объема и скорости оперативной памяти.

3 Жесткий диск (*HDD*) или *SSD*: Жесткий диск или твердотельный накопитель предназначены для хранения операционной системы, приложений и данных, обеспечивая устойчивость и быстрый доступ к информации.

4 Графический процессор *Intel® Iris® Xe Graphics*: Интегрированный графический процессор *Intel® Iris® Xe Graphics* предоставляет выдающуюся графическую производительность в составе вычислительной системы. Спроектированный с учетом современных графических требований, этот графический процессор обеспечивает плавное воспроизведение графики, быстрый рендеринг и поддержку новейших графических технологий.

Эти компоненты в совокупности формируют архитектуру вычислительной системы, обеспечивая её функциональность и производительность для успешного выполнения поставленных задач.

**1.1.2** Центральный процессор

Центральный процессор – это центральное устройство компьютера, которое выполняет операции по обработке данных и управляет периферийными устройствами компьютера. У компьютеров четвёртого поколения и старше функции центрального процессора выполняет микропроцессор на основе СБИС, содержащей несколько миллионов элементов, конструктивно созданный на полупроводниковом кристалле путём применения сложной микроэлектронной технологии.

В комплектацию центрального процессора входят устройство управления (УУ), арифметико-логическое устройство (АЛУ), запоминающее устройство (ЗУ) на основе регистров процессорной и кэш-памяти, а также генератор тактовой частоты (ГТЧ). Устройство управления организует выполнение программ и координирует взаимодействие всех устройств ЭВМ в процессе работы.

Арифметико-логическое устройство выполняет арифметические и логические операции над данными, включая сложение, вычитание, умножение и деление.

Запоминающее устройство – это внутренняя память процессора. Регистры служит промежуточной быстрой памятью, используя которые, процессор выполняет расчёты и сохраняет промежуточные результаты. Для ускорения работы с оперативной памятью используется кэш-память, в которую с опережением подкачиваются команды и данные из оперативной памяти, необходимые процессору для последующих операций.

Генератор тактовой частоты создает синхронизирующие импульсы, регулирующие работу всех компонентов компьютера. Тактовая частота, измеряемая в мегагерцах (МГц), отражает уровень технологии процессора и его производительность. Разрядность процессора определяет максимальное количество бит информации, которое может обрабатываться одновременно.

Основные характеристики процессора:

1 Быстродействие (вычислительная мощность) – это среднее число операций процессора в секунду.

2 Тактовая частота равна количеству тактов в секунду. Такт – это промежуток времени между началом подачи текущего импульса ГТЧ и началом подачи следующего.

3 Разрядность – это максимальное количество бит информации, которые могут обрабатываться и передаваться процессором одновременно. Разрядность процессора определяется разрядностью регистров, в которые помещаются обрабатываемые данные. [2]

**1.1.3** Оперативная память

Оперативная память – это рабочая область для процессора компьютера. В ней во время работы хранятся программы и данные. Оперативная память часто рассматривается как временное хранилище, потому что данные и программы в ней сохраняются только при включенном компьютере или до нажатия кнопки сброса. Перед выключением или нажатием кнопки сброса все данные, подвергнутые изменениям во время работы, необходимо сохранить на запоминающем устройстве, которое может хранить информацию постоянно (обычно это жесткий диск). При новом включении питания сохраненная информация вновь может быть загружена в память.

За несколько лет определение *RAM* (*Random Access Memory*) превратилось из обычной аббревиатуры в термин, обозначающий основное рабочее пространство памяти, создаваемое микросхемами динамической оперативной памяти (*Dynamic* *RAM* – *DRAM*) и используемое процессором для выполнения программ. Одним из свойств микросхем *DRAM* (и, следовательно, оперативной памяти в целом) является динамическое хранение данных, что означает, во-первых, возможность многократной записи информации в оперативную память, а во-вторых, необходимость постоянного обновления данных (т.е., в сущности, их перезапись) примерно каждые 15 мс. Также существует так называемая статическая оперативная память (*Static* *RAM* – *SRAM*), не требующая постоянного обновления данных. Следует заметить, что данные сохраняются в оперативной памяти только при включенном питании. [3, 499-572]

**1.1.3** Вспомогательная память – *HDD* и *SSD*

Жесткий диск (ЖД, англ. *Hard Disk Drive, HDD*) — это устройство для хранения данных внутри компьютера. Он представляет собой механическое устройство, состоящее из нескольких дисков, покрытых магнитным материалом, и манипулятора, который может перемещаться по поверхности дисков для чтения и записи информации.

Жесткие диски широко применяются для хранения операционных систем, приложений и пользовательских файлов. Однако, с появлением твердотельных накопителей (*SSD*), которые обеспечивают более высокую скорость чтения/записи и обладают надежностью за счет отсутствия подвижных частей, *HDD* стали постепенно уступать свои позиции в сфере высокопроизводительного хранения данных.

*SSD*, или твердотельный накопитель (англ. *Solid State Drive*), представляет собой электронное устройство для хранения данных. В отличие от традиционных жестких дисков (*HDD*), *SSD* не содержит подвижных частей, таких как вращающиеся диски и магнитные головки. Вместо этого *SSD* использует флэш-память (например, *NAND*-флэш) для хранения информации.

*SSD* широко применяются в современных компьютерах, ноутбуках и серверах из-за своей высокой производительности, надежности и отсутствия подвижных частей, что делает их более устойчивыми к механическим повреждениям. Они являются эффективным решением для улучшения общей производительности системы и ускорения загрузки операционной системы и приложений. [3, 653-700]

**1.1.4** Графический процессор

*GPU* – графический процессор, который предназначен для ускорения рендеринга графики и параллельных вычислений. Он произвел настоящую революцию в том, как мы взаимодействуем с цифровым контентом. Будь то потрясающие визуальные эффекты в видеоиграх, сложные вычисления в алгоритмах машинного обучения или плавная анимация в высококлассных программах для 3*D*-дизайна.

Роль *GPU* выходит за рамки рендеринга графики. С появлением вычислений общего назначения на графических процессорах (*GPGPU*) эти процессоры нашли применение в областях, где требуются высокопроизводительные вычисления. Например, ускорение задач искусственного интеллекта, глубокое обучение для высокоскоростной аналитики данных и научных симуляций.

Графический процессор (*GPU* или *Graphics Processing Unit*) — это один из видов микропроцессоров. Он управляет памятью видеокарт и ускоряет вывод графики на экран устройства. Его архитектура отличается от архитектуры *CPU*:

1 В *GPU* содержатся несколько сотен вычислительных ядер, с помощью которых сложные расчеты выполняются очень быстро. Энергии при таких вычислениях потребляется значительно меньше, чем при работе *CPU*.

2 В *GPU* обработка графики в видеороликах, графических программах, играх выполняется быстрее и эффективнее благодаря разделению процессов. Это позволяет разгрузить *CPU* для других задач.

3 *GPU* может взять на себя некоторые вычисления вместо процессора. Он позволяет выполнять расчеты с плавающей точкой или расчеты с одинаковой или сходной формулой.

*GPU* и видеокарта – это не одно устройство. Видеокарта – это устройство, на котором расположены *GPU*, *VRAM*, питание, линия обмена информацией с *CPU*, видеовыходы. Графический процессор – составляющая видеокарты. Он нужен для обработки графики, 3*D*-моделирования и работы с любыми другими данными, которые требуют сложных математических действий. Графические процессоры разделяют на встроенные и дискретные. От типа *GPU* зависит стоимость и мощность устройства.

Общее назначение ядер *GPU* заключается в ускорении параллельных вычислений, таких как обработка графики, научные вычисления и расчеты в машинном обучении. Их архитектура оптимизирована для обработки массивов данных параллельно, что делает их мощным инструментом в контексте вычислений больших объемов данных. [4]

## 1.2 История, версии и достоинства

**1.2.1** Архитектура x64

Архитектура x64, известная также как x86-64 или *AMD64*, представляет собой 64-битное расширение архитектуры x86, начавшейся с процессора *Intel* 8086 в конце 1970-х годов. После внедрения 32-битной эры x86 с процессором *Intel* 80386 в 1985 году и последующими улучшениями, *AMD* в конце 1990-х годов разработала расширение x86-64, изначально известное как *AMD*64.

Это расширение внесло поддержку 64-битных вычислений, позволяя адресовать более 4 гигабайт памяти и предоставляя новые регистры и команды. *AMD*64 была затем принята индустрией, включая *Intel*, что привело к тому, что x86-64 стал универсальным стандартом для 64-битных вычислений на процессорах x86.

С течением времени архитектура x86-64 продолжала эволюционировать. Новые расширения, такие как *SSE* и *AVX*, были внедрены для повышения производительности в области векторных вычислений. Последние поколения процессоров *Intel* и *AMD* продолжают улучшать архитектуру, добавляя новые технологии, оптимизируя энергопотребление и увеличивая вычислительную мощность. В настоящее время архитектура x86-64 широко используется в компьютерах и серверах, обеспечивая поддержку 64-битных операционных систем и приложений.

Архитектура x64 применяется в разнообразных областях информационных технологий и вычислений. Эта архитектура широко используется в современных компьютерах и ноутбуках, обеспечивая поддержку 64-битных операционных систем и эффективное использование больших объемов оперативной памяти. Серверы также в значительной степени оснащены процессорами с архитектурой x64, что обеспечивает высокую производительность и способность обрабатывать множество одновременных запросов.

Разработчики программного обеспечения ориентированы на создание приложений, совместимых с 64-битными системами, учитывая их высокую производительность. В области научных исследований и инженерных расчетов архитектура x64 используется для выполнения сложных вычислений, моделирования и симуляций благодаря своей вычислительной мощности и поддержке 64-битных регистров.

Мультимедийные приложения, обрабатывающие графику, аудио и видео, также ориентированы на архитектуру x64 для эффективной работы с большими объемами мультимедийных данных. Системы управления базами данных (СУБД) часто используют x64 для обеспечения быстрого выполнения запросов и обработки данных.

Технологии виртуализации, такие как *VMware* и *Hyper-V*, широко используют архитектуру x64 для создания виртуальных машин, обеспечивая изоляцию и эффективное управление ресурсами. Все эти примеры подчеркивают универсальность и важность архитектуры x64 в различных областях, обеспечивая выдающуюся производительность и совместимость с современными технологиями. [5]

**1.2.2**Оперативная память (*RAM*), 16Гб

История оперативной памяти пронизана стремлением к постоянному увеличению скорости и емкости памяти в компьютерах. От первых попыток использования электронных трубок и резисторов в *ENIAC*, представляющих из себя весьма медленные и объемом ограниченные технологии, к магнитным барабанам в 1950-60-х. Последние представляли собой вращающиеся магнитные цилиндры с возможностью чтения и записи данных на различных дорожках, что улучшало как емкость, так и скорость.

Переход к использованию полупроводников в конце 1960-х годов принес с собой новый этап в эволюции памяти. Это включало в себя магнитные сердечники, магнитные полосы, а затем интегральные схемы (*IC*). В 1970-х годах появилась динамическая оперативная память (*DRAM*), использующая конденсаторы для хранения данных. Однако требование периодического обновления данных сделало ее динамичной.

В 1980-90-е годы привнесли в развитие оперативной памяти статическую оперативную память (*SRAM*) и технологии *Flash Memory*. *SRAM* предоставляла более высокую скорость за счет повышенной сложности и стоимости. *Flash Memory*, способная сохранять данные без подачи энергии, стала неотъемлемой для мобильных устройств и карт памяти.

В начале 2000-х годов технологии *DDR* (*Double Data Rate*) внесли значительный вклад в увеличение скорости передачи данных. Последующие поколения (*DDR*2, *DDR*3, *DDR*4) принесли повышенную пропускную способность и энергоэффективность, что стало особенно важным с развитием мобильных устройств и вычислительной техники в целом.

С 2010 года исследования активно направлены на технологии 3*D* *NAND*, которые позволяют увеличивать емкость памяти на одном чипе. Кроме того, появляются новые концепции, такие как *Magnetoresistive Random Access Memory* (*MRAM*), с акцентом на увеличении емкости, улучшении скорости доступа и энергоэффективности. Развитие оперативной памяти продолжается, нацеленное на инновации в увеличении ее емкости, улучшении характеристик доступа и хранения данных, что подчеркивает важность этого компонента в технологическом прогрессе. [6]

**1.2.3** SSD-диск *NVMe* *SAMSUNG* 1024Гб (1Тб)

История создания *SSD*-диска *NVMe* *Samsung* объемом 1024 Гб является частью эволюции накопителей и технологии хранения данных.

1 Этап развития *SSD*-технологии: в начале 2000-х годов началось внедрение *SSD* (*Solid* *State* *Drive*) как альтернативы традиционным механическим жестким дискам (*HDD*). *SSD* использовали флэш-память для хранения данных, что обеспечивало более высокую скорость доступа и отсутствие подвижных частей.

2 Внедрение *NVMe* (*Non-Volatile Memory Express*): *NVMe* стало ответом на растущие потребности в более эффективном использовании *SSD*. Этот протокол обеспечивает низкую задержку (лаг) и высокую пропускную способность, оптимизированные для работы с флэш-памятью. Это привело к появлению *SSD* с интерфейсом *NVMe*, который предоставляет выдающуюся производительность.

3 Участие *Samsung* в разработке *SSD*: *Samsung* является одним из крупнейших производителей полупроводников и компонентов памяти. Компания активно участвует в инновационных решениях для хранения данных. Запуск 1-терабайтного *SSD* с интерфейсом *NVMe* был следующим естественным шагом в развитии их продуктового портфеля.

4 Технологические характеристики 1-Терабайтного *NVMe* *SSD* *Samsung*: Подобные накопители обычно оснащаются последними технологиями флэш-памяти, такими как 3*D* *NAND*, которые позволяют увеличить емкость и производительность. Также важными характеристиками являются скорость чтения/записи, надежность, энергоэффективность и поддержка современных стандартов безопасности данных.

5 Применение и распространение: *SSD Samsung NVMe* 1 Тб нашел применение в различных сферах, включая персональные компьютеры, ноутбуки, рабочие станции и сервера, благодаря своей высокой производительности и надежности. Общий контекст создания *SSD* Samsung *NVMe* 1 Тб включает в себя эволюцию *SSD*-технологии, внедрение протокола *NVMe*, активное участие *Samsung* в разработке и производстве *SSD*, а также постоянные инновации и обновления продуктового портфеля для соответствия требованиям современных пользователей и бизнес-приложений.

*SSD*-диски *NVMe* *Samsung* обладают рядом преимуществ, которые делают их популярными среди пользователей и профессиональных систем:

1 *SSD NVMe Samsung* обеспечивают высокую скорость чтения и записи данных. Использование технологий *NVMe* и флэш-памяти с трехмерной структурой (3*D NAND*) позволяет достигнуть выдающейся производительности.

2 Протокол *NVMe* оптимизирован для работы с флэш-памятью, что обеспечивает низкую задержку при доступе к данным. Это особенно важно для приложений, где требуется высокая отзывчивость, таких как игры и профессиональные задачи.

3 *SSD NVMe* *Samsung* часто более энергоэффективны по сравнению с традиционными механическими жесткими дисками (*HDD*). Они не имеют подвижных частей, что снижает энергопотребление и повышает долговечность.

4 Современные *SSD*-диски *NVMe* *Samsung* предоставляют большие объемы хранения, включая модели емкостью в терабайты. Это делает их отличным выбором для хранения больших объемов данных, таких как мультимедийные файлы и профессиональные проекты.

5 *SSD*-диски, в том числе и от *Samsung*, обычно более надежны и долговечны по сравнению с *HDD*. Отсутствие подвижных частей снижает риск механических повреждений, а также увеличивает срок службы.

6 Многие *SSD*-диски *Samsung* поддерживают современные технологии безопасности данных, такие как аппаратное шифрование и защита от несанкционированного доступа. Это обеспечивает безопасность важных данных.

7 *SSD NVMe Samsung* легко устанавливаются в современные системы, поддерживающие интерфейс *NVMe*. Это делает их удобными для обновления существующих систем или встраивания в новые. [7]

**1.2.4** Графический процессор *Intel*® *Iris*® *Xe* *Graphics*

История создания графического процессора *Intel*® *Iris*® *Xe Graphics* представляет собой увлекательный путь эволюции компании в области графических технологий. Перед появлением *Iris Xe, Intel* долгое время предоставляла встроенные графические решения, такие как *Intel HD Graphics* и *Intel UHD Graphics*, ориентированные, в основном, на повседневные графические задачи.

Однако, с ростом конкуренции в индустрии графических технологий со стороны гигантов, таких как *NVIDIA* и *AMD*, *Intel* решила предпринять решительные шаги в направлении более мощных и конкурентоспособных графических решений.

Запуск инициативы *Xe*, изначально известной как *Project* *Xe*, стал ключевым этапом. Эта архитектурная инициатива охватывала как встроенные, так и дискретные графические решения. *Iris Xe Graphics* вступила в этот игровой процесс, предназначенная для встроенного использования в процессорах *Intel* и обещая улучшенную графику и производительность.

Процесс развития *Iris Xe* включал в себя ориентацию не только на основные графические задачи, но и на требования геймеров и профессионалов в области создания контента. В этом контексте, компания представила дискретные графические карты под брендом *Iris Xe MAX*, призванные удовлетворить потребности геймерского сообщества и творческих профессионалов.

Технологические характеристики *Iris Xe* включают в себя поддержку аппаратного ускорения трассировки лучей, интеграцию с процессорами *Intel* через технологию *Intel Deep Link*, а также соответствие современным *API* графики, таким как *DirectX* 12.

С течением времени и с каждым новым поколением процессоров, ожидается, что *Iris Xe Graphics* будет продолжать свою эволюцию, предоставляя пользователям ещё более широкий выбор возможностей и повышенную производительность. Этот графический процессор стал неотъемлемой частью стратегии *Intel* в области графики, обеспечивая инновационные решения для различных потребностей пользователей – от гейминга до профессиональных приложений. [8]

## 1.3 Анализ выбранной вычислительной системы и обоснование       выбора

Выбор данной вычислительной системы, предназначенной для реализации темы курсовой, обоснован не только общей производительностью и функциональностью компонентов, но и их способностью эффективно справляться с распределенными вычислениями.

Процессор *Intel Core i*5 был выбран не только как вычислительный двигатель системы, но и как основной *CPU* для управления вычислительными задачами. Его многозадачность и поддержка многопоточности обеспечивают хорошую базу для параллельных вычислений.

Оперативная память (*RAM*) с оптимальным объемом и высокой скоростью является ключевым элементом для эффективного обмена данными между процессором и *GPU* в условиях распределенных вычислений.

Жесткий диск (*HDD*) или твердотельный накопитель (*SSD*) обеспечивают необходимое хранилище данных для распределенных задач, где эффективное чтение и запись данных являются важными аспектами.

Графический процессор *Intel*® *Iris*® *Xe Graphics*, помимо обеспечения выдающейся графической производительности, также может быть задействован в параллельных вычислениях, особенно в контексте *GPU*-вычислений. Его интеграция в архитектуру системы позволяет распределенным вычислениям эффективно использовать графические вычислительные ресурсы.

Следует отметить, что немалую роль в выборе вычислительной системы играет ее наличие, что значительно облегчает начало работы над курсовым проектом. Наличие устройства в личном распоряжении предоставляет удобство и оперативность в проведении экспериментов и тестировании конкретных распределенных вычислительных сценариев. Это также снижает временные затраты на поиск и адаптацию нового оборудования, позволяя более эффективно использовать ресурсы и фокусироваться на ключевых аспектах исследования.

Таким образом, данная вычислительная система выбрана с учетом её пригодности для реализации распределенных вычислений, обеспечивая оптимальное взаимодействие между *CPU* и *GPU*. Комбинация высокопроизводительного процессора, оперативной памяти и графического процессора создает подходящую платформу для проведения исследований и разработки, связанных с темой курсовой работы.

# 2 Платформа программного обеспечения

**2.1** **Структура и архитектура платформы**

**2.1.1** Выбранная платформа

В рамках выполнения курсовой работы была принята решающая роль в выборе определенных компонентов разработочной платформы. Операционная система *Windows* 10 была выбрана в качестве основной, обеспечивая широкий спектр совместимости с различными программными и аппаратными решениями. Это также предоставляет удобную интеграцию с инструментарием разработки и обеспечивает стабильную среду для проведения экспериментов и тестирования.

Интегрированная среда разработки *Visual* *Studio* была предпочтительным выбором для написания кода и выполнения проекта. *Visual* *Studio* обеспечивает удобный интерфейс, богатые возможности отладки, а также поддержку различных языков программирования. Это средство разработки обладает широким функционалом и является стандартом в индустрии для разработки программного обеспечения под *Windows*.

Язык программирования *C/C*++ был выбран в связи с его эффективностью, близким уровнем доступа к аппаратным ресурсам и поддержкой параллельных вычислений. *C/C*++ предоставляет высокую производительность и контроль над памятью, что особенно важно при реализации распределенных вычислений на нескольких процессорах (*CPU*, *GPU*).

Таким образом, выбранная платформа, объединяющая *Windows* 10, *Visual Studio* и язык программирования *C/C*++, обеспечивает оптимальные условия для эффективной разработки, тестирования и анализа распределенных вычислений в рамках поставленных задач курсовой работы. [9]

**2.1.2** Операционная система

Операционная система – это программное обеспечение, управляющее аппаратными и программными ресурсами компьютера. Она предоставляет интерфейс для взаимодействия пользователя с компьютером, обеспечивает выполнение прикладных программ и эффективное управление аппаратурой, такой как процессор, память, устройства ввода-вывода и хранения данных. Операционная система выполняет ряд основных функций, таких как загрузка системы, управление процессами, обеспечение безопасности данных, файловая система и другие.

С точки зрения программиста операционная система – это программное окружение, которое предоставляет набор абстракций и сервисов для управления аппаратными ресурсами компьютера. Программист взаимодействует с операционной системой для выполнения своих программных задач. [10]

**2.1.3** Интегрированная среда разработки

Интегрированная среда разработки – это программное средство, предоставляющее программисту интегрированный набор инструментов и функций для удобного создания, редактирования, компиляции, отладки и управления программным кодом. *IDE* обычно включает в себя следующие ключевые элементы:

1 Текстовый редактор: для написания и редактирования исходного кода. Обеспечивает подсветку синтаксиса, автодополнение и другие удобства.

2 Компилятор/Интерпретатор: позволяет преобразовывать исходный код в исполняемый файл или промежуточный код.

3 Отладчик: предоставляет средства для пошагового выполнения кода, выявления ошибок и отслеживания значений переменных в процессе выполнения программы.

4 Система управления версиями: инструменты для отслеживания изменений в коде, совместной работы нескольких разработчиков и восстановления предыдущих версий кода.

5 Панели инструментов: сокращают количество действий для выполнения основных операций, таких как компиляция или запуск программы.

6 Менеджер проектов: позволяет организовать файлы и ресурсы проекта, управлять зависимостями и настройками проекта.

7 Интеграция с отладочными и профилирующими инструментами: для более глубокого анализа и оптимизации кода. [11]

**2.1.4** Язык программирования

Язык программирования представляет собой формальный инструмент, используемый для написания компьютерных программ и обеспечивающий взаимодействие между человеком и компьютером. Он предоставляет программистам средства для выражения идей и логики работы программы, используя читаемый синтаксис. Эти инструкции, созданные на языке программирования, являются понятными для компьютера и предоставляют ему путь выполнения задач.

Язык программирования выполняет описательную функцию, позволяя программистам выражать свои концепции и логику работы программы так, чтобы было понятно человеку. Он также служит для написания инструкций, которые могут быть выполнены компьютером, предоставляя необходимые средства для создания программного кода.

Основной целью языка программирования является создание абстракции, позволяющей программистам работать на более высоком уровне, избегая прямого взаимодействия с аппаратными ресурсами компьютера. Это уровень абстракции делает возможным более удобное и эффективное программирование.

Кроме того, язык программирования обеспечивает переносимость программ между различными платформами и системами, предоставляя возможность запускать одинаковый код на разных устройствах. [12]

**2.2 История, версии и достоинства**

**2.2.1** Операционная система *Windows* 10

Операционная система *Windows* 10 была представлена *Microsoft* в июле 2015 года. Это было важное обновление для предыдущих версий *Windows*, таких как *Windows* 8 и *Windows* 8.1, и представляло собой попытку объединения функциональности для ПК, планшетов и мобильных устройств в одну универсальную платформу.

*Windows* 10 изначально выпущена как одна общая версия, но *Microsoft* предоставляет регулярные обновления функций два раза в год. Эти обновления включают в себя улучшения безопасности, новые функции и улучшения производительности.

Достоинства операционной системы, выбранной в курсовом проекте:

1 Универсальность: *Windows* 10 предназначена для работы на широком спектре устройств, включая ПК, планшеты, ноутбуки и мобильные устройства.

2 Интерфейс: Введение стартового меню возвращает привычную среду для пользователей, привыкших к более ранним версиям *Windows*.

3 Безопасность: *Windows* 10 включает улучшенные средства безопасности, такие как *Windows Defender* и система обновлений для борьбы с вирусами и вредоносным программным обеспечением.

4 Обновления: Регулярные обновления обеспечивают улучшения функциональности, новые возможности и патчи безопасности.

5 Совместимость с приложениями: *Windows* 10 обеспечивает поддержку широкого спектра приложений, включая множество классических приложений *Windows* и современных *Universal Windows Platform* (*UWP*) приложений.

6 Облако и интеграция с онлайн-сервисами: *Windows* 10 включает интеграцию с облачными службами *Microsoft*, такими как *OneDrive*, обеспечивая удобный доступ к данным из любой точки мира. [9]

**2.2.2** Интегрированная среда разработки *Visual Studio*

Интегрированная среда разработки (*IDE*) *Visual Studio* была представлена компанией *Microsoft* в 1997 году и с тех пор прошла долгий путь эволюции и усовершенствования. Она стала одним из основных инструментов для разработки программного обеспечения под платформу *Windows* и другие технологии. *Visual Studio* имеет множество версий, каждая из которых вносит улучшения в среду разработки и предоставляет новые инструменты для программистов. Каждая последующая версия вносила новые функциональности, интегрировала последние технологии и улучшала процесс разработки.

Достоинства выбранной интегрированной среды разработки:

1 Многозадачность: *Visual Studio* обеспечивает разработчиков множеством инструментов и функций, улучшающих процесс создания программного обеспечения, включая редактор кода, отладчик, дизайнер форм, и многие другие.

2 Языковая поддержка: Поддержка различных языков программирования, включая *C*++, *C*#, *Visual Basic*, *F*#, *Python*, и другие, что делает его универсальным инструментом для разработчиков.

3 Отладка: Мощные инструменты отладки, включая возможность пошагового выполнения кода, анализа переменных и состояния приложения.

4 Графические средства разработки: Интегрированные средства для создания графических интерфейсов приложений, веб-сайтов и других проектов.

5 Тестирование: Инструменты для автоматизации тестирования, включая модульное, функциональное и нагрузочное тестирование.

6 Совместная разработка: Возможности совместной разработки, поддержка систем контроля версий и средства для работы в команде.

7 Расширяемость: Возможность установки и использования различных плагинов и расширений для адаптации среды разработки под конкретные потребности разработчика.

8 Обновления: Регулярные обновления и поддержка последних технологий, что позволяет разработчикам оставаться в курсе современных требований и тенденций в программировании. [13]

**2.2.3** Язык программирования *C/C*++

*C/C*++ – это один из наиболее влиятельных и широко используемых языков программирования. Их история, версии и достоинства формировались на протяжении десятилетий.

История:

1 Язык программирования *C* был создан в 1972 году в лабораториях *Bell Telephone* (*Bell Labs*) Деннисом Ритчи. Он был разработан как язык программирования для написания операционной системы *UNIX*. C был отмечен своей эффективностью и низкоуровневым характером, что делало его предпочтительным выбором для системного программирования.

2 В 1983 году Бьёрн Страуструп расширил язык *C*, создав *C*++. *C*++ добавил объектно-ориентированные возможности, что сделало его более гибким и удобным для разработки сложных программных систем.

Версии:

1 Язык C не имеет формальных версий, но стандарты были выпущены. Один из самых важных – *ANSI C* (*C*89), затем был выпущен стандарт *C*99, а затем C11.

2 *C*++ также прошел через несколько версий. Одной из первых была *C*++98, затем *C*++03. Существенные изменения внесены в стандарт *C*++11, который внедрил множество новых функций и улучшений. Последующие стандарты включают *C*++14, *C*++17 и *C*++20.

Достоинства:

1 Эффективность: *C*/*C*++ известны своей высокой производительностью и эффективностью выполнения.

2 Близость к аппаратуре: Эти языки предоставляют низкоуровневый доступ к памяти и аппаратным ресурсам, что важно для системного программирования.

3 Обширные библиотеки: Оба языка обладают обширными библиотеками, предоставляя разработчикам множество функций и инструментов.

4 Переносимость: Код на *C*/*C*++ легко переносится между различными платформами.

5 Объектно-ориентированное программирование (*C*++): *C*++ добавил объектно-ориентированные концепции, что делает его подходящим для создания сложных и масштабируемых систем.

6 Широкое использование в индустрии: *C*/*C*++ широко используются в разработке операционных систем, встроенных систем, игр, высокопроизводительных приложений и других областях.

7 Огромное сообщество: Оба языка имеют огромное сообщество разработчиков, что обеспечивает поддержку и ресурсы для программистов.

8 Использование *C*/*C*++ остается актуальным в различных областях программирования, обеспечивая мощные инструменты для разработки высокопроизводительных приложений. [14]

**2.3 Анализ программного обеспечения для написания программы и   
      обоснование выбора платформы**

Анализ программного обеспечения для написания программы важен в контексте темы курсовой работы. Выбор платформы, языка программирования и интегрированной среды разработки играют ключевую роль в успешной реализации курсового проекта. Рассмотрим обоснование выбора *C/C*++, *Visual Studio* и *Windows* 10.

Язык программирования *C/C*++ выбран для обеспечения высокой производительности и низкоуровневого управления ресурсами, что является критичным аспектом для эффективных распределенных вычислений. Его близость к аппаратному обеспечению позволяет оптимизированно использовать вычислительные ресурсы.

Интегрированная среда разработки *Visual Studio* предоставляет полный инструментарий для написания, отладки и тестирования кода, обеспечивая эффективную разработку сложных программных проектов. Многозадачность и средства совместной разработки делают ее удобным выбором для командной работы.

Операционная система *Windows* 10 была выбрана из-за широкой поддержки пользователей и совместимости с разнообразными вычислительными системами. Драйвера и библиотеки обеспечивают необходимый доступ к ресурсам, что важно при работе с различными процессорами, включая *CPU* и *GPU*.

Можно сказать, что комбинация *C/C*++ обеспечивает эффективность и контроль над ресурсами, Visual Studio предоставляет удобную среду разработки, а *Windows* 10 обеспечивает совместимость с большинством вычислительных систем. Также этот выбор соответствует требованиям задачи по распределенным вычислениям на различных процессорах (*CPU*, *GPU*).

# 3 теоретическое обоснование разработки             программного продукта

**3.1 Обоснование необходимости разработки**

Распределённые вычисления на нескольких процессорах являются неотъемлемой частью современных вычислительных парадигм. Они находят широкое применение по нескольким важным причинам:

1 Распределение вычислений между несколькими процессорами позволяет параллельно выполнять задачи, что существенно увеличивает общую производительность системы. Это особенно важно при работе с большими объемами данных или выполнении сложных вычислительных операций.

2 Распределение задач между процессорами позволяет более эффективно использовать вычислительные ресурсы системы. Каждый процессор может заниматься своей частью работы, что способствует оптимальному распределению нагрузки.

3 Некоторые задачи требуют огромных вычислительных мощностей, одиночный процессор может не справиться с их обработкой. Распределение задач позволяет объединить вычислительные ресурсы нескольких процессоров для решения сложных проблем.

4 Распределённые вычисления обеспечивают высокую степень скалируемости. При увеличении вычислительных ресурсов системы можно эффективно масштабировать процессы, что особенно важно в условиях изменяющихся требований.

5 В случае отказа одного из процессоров, другие могут продолжить выполнение задач, что обеспечивает более высокую отказоустойчивость системы.

6 Распределённые вычисления широко применяются при обработке и анализе больших объемов данных, например, в области исследований, научных расчетов, анализа данных и машинного обучения.

Таким образом, распределённые вычисления на нескольких процессорах являются эффективным инструментом для решения разнообразных задач, обеспечивая более быстрое, эффективное и гибкое функционирование вычислительных систем. [15]

**3.2 Технологии программирования, используемые для решения поставленных задач**

Работа посвящена разработке эффективной программы для распределенных вычислений на нескольких процессорах *CPU* и *GPU*. Для достижения этой цели рассматривается использование языков программирования *C/C*++ в сочетании с библиотекой *OpenCL*. *C/C*++ выбран для обеспечения высокой производительности и полного контроля над ресурсами, в то время как *OpenCL* позволяет нам реализовать распределенные вычисления на нескольких процессорах, включая как *CPU*, так и *GPU*. В дополнение к этому, мы рассмотрим использование *OpenMP* для дополнительного распараллеливания вычислений на *CPU* и достижения большей эффективности программы. Теперь подробно рассмотрим технологии программирования, которые будут использоваться для успешной реализации данного проекта.

**3.2.1** Язык программирования C/C++. Многозадачность и многопоточность

Языки программирования C/C++ выбраны в качестве основных языков разработки. Они предоставляют высокую производительность и непосредственный контроль над ресурсами, что существенно для эффективных распределенных вычислений.

В проекте будет использоваться многозадачность для разделения вычислительных задач на подзадачи и многопоточность для параллельного выполнения задач на различных ядрах CPU.

Многозадачность (*multitasking*) – свойство операционной системы или среды выполнения обеспечивать возможность параллельной (или псевдопараллельной) обработки нескольких задач.

Многопоточность (*multithreading*) – свойство платформы (например, операционной системы, виртуальной машины и т. д.) или приложения, состоящее в том, что процесс, порождённый в операционной системе, может состоять из нескольких потоков, выполняющихся «параллельно», то есть без предписанного порядка во времени. При выполнении некоторых задач такое разделение может достичь более эффективного использования ресурсов вычислительной машины.

В тандеме с многозадачностью, многопоточность будет активно применяться для разделения задач на потоки и эффективного использования вычислительных ресурсов каждого процессора.

В контексте *CPU*, будут использоваться стандартные библиотеки многопоточности в *C*++, такие как *std*::*thread*, для создания потоков, которые будут выполнять части вычислений. Эти потоки будут синхронизированы для обеспечения правильного взаимодействия и обмена данными.

Кроме того, учитывая, что современные *GPU* также поддерживают многопоточность, планируется использовать соответствующие возможности *OpenCL* для создания многопоточных вычислений и максимального использования параллельности на *GPU*.

Такой подход позволит эффективно реализовать распределенные вычисления на нескольких процессорах, гармонично использовать ресурсы каждого устройства и максимизировать общую производительность разрабатываемой программы. [16]

**3.2.2** Библиотека *OpenCL*

*OpenCL* (*Open Computing Language*) представляет собой открытый стандарт для параллельных вычислений на гетерогенных системах, что делает ее мощным инструментом для решения задач распределенных вычислений на *CPU* и *GPU*.

Гетерогенные системы – это компьютерные системы, в которых используются различные типы вычислительных устройств, такие как центральные процессоры (*CPU*), графические процессоры (*GPU*), физические процессоры (*FPGA*), и/или другие специализированные ускорители. В отличие от гомогенных систем, где все устройства имеют однородную архитектуру, гетерогенные системы объединяют разнообразные вычислительные ресурсы под единым управлением.

На уровне *CPU*, *OpenCL* позволяет создавать параллельные вычисления, используя несколько ядер процессора. Это означает, что можно эффективно распараллеливать задачи и использовать вычислительные ресурсы *CPU* на полную мощность.

Основное преимущество *OpenCL* заключается в его способности работать с *GPU*. Создание ядер *OpenCL* для *GPU* позволяет параллельно выполнять вычисления на тысячах ядер графического процессора. Это обеспечивает значительное ускорение при выполнении вычислительно интенсивных задач, таких как обработка изображений, научные вычисления и многие другие. [17]

Таким образом, *OpenCL* предоставляет единый интерфейс для программирования как *CPU*, так и *GPU*, что делает его идеальным выбором для распределенных вычислений. Он позволяет эффективно использовать параллелизм обоих устройств, максимизируя общую производительность системы.

**3.2.3** Библиотека *OpenMP*

*OpenMP,* или *Open Multi-Processing*, представляет собой набор стандартов и спецификаций для упрощения разработки параллельных программ. Этот набор инструментов предназначен для использования на языках программирования *C, C++,* и *Fortran*. Основная цель *OpenMP* – обеспечить эффективное создание программ, способных выполняться параллельно на многопроцессорных и многоядерных системах.

Основной механизм, предоставляемый *OpenMP* – это директивы препроцессора, которые встраиваются в исходный код программы. Эти директивы сообщают компилятору, какие участки кода следует выполнять параллельно. Так, например, директива *#pragma omp parallel* начинает параллельный блок кода.

Одной из ключевых характеристик *OpenMP* является управление потоками. Он предоставляет механизмы создания и управления потоками внутри параллельных секций кода, что делает возможным эффективное использование ресурсов многопроцессорных систем.

Синхронизация – важный аспект параллельного программирования, *OpenMP* предоставляет механизмы для координации работы потоков. Например, директивы *critical* и *barrier* позволяют избегать гонок данных и обеспечивают контроль над выполнением потоков.

*OpenMP* также предоставляет специальные средства для работы с переменными в параллельных участках кода. Директива *private*, например, обеспечивает каждому потоку свою собственную копию переменной, избегая таким образом конфликтов при обращении к данным.

В итоге можно сказать, что *OpenMP* облегчает процесс создания параллельных программ, предоставляя программистам удобные средства управления потоками, синхронизации и работой с данными. Это особенно важно в условиях, когда необходимо эффективно использовать вычислительные ресурсы современных многопроцессорных систем. [18]

**3.2.4** Инструменты разработки и отладка среды разработки

При разработке программы для распределенных вычислений на нескольких процессорах, включая использование *OpenCL* и *OpenMP*, важно выбирать подходящие инструменты разработки и отладки.

В контексте разработки программы для распределенных вычислений на нескольких процессорах с использованием *Visual Studio*, следует подробнее рассмотреть инструменты, которые обеспечивают эффективное программирование и отладку в гетерогенной среде.

*Visual Studio* выделяется интегрированной средой разработки, предоставляя широкий спектр возможностей для создания, отладки и оптимизации кода. В контексте распределенных вычислений, *Visual Studio* позволяет интегрировать не только *C/C*++ код, но и обеспечивает удобство работы с библиотеками *OpenCL* и *OpenMP*. Это интуитивное и мощное средство программирования.

Компилятор *Visual Studio* является ключевым инструментом для оптимизации и компиляции кода. Он обеспечивает совместимость с используемыми библиотеками, а также позволяет настроить проект для работы с технологиями параллельного программирования, такими как *OpenCL* и *OpenMP*.

Интеграция с *Git* в *Visual Studio* предоставляет средства для эффективного управления версиями кода. Разработчик может комфортно отслеживать изменения, возвращаться к предыдущим версиям и вести работу с историей проекта.

*AMD CodeXL* и *NVIDIA Nsight* – это инструменты, интегрированные в *Visual Studio*, которые предоставляют средства для профилирования и отладки *OpenCL*-кода. Они позволяют анализировать выполнение ядер на *GPU*, выявлять узкие места и оптимизировать код для конкретных графических ускорителей.

*Visual Studio* интегрирует поддержку *OpenMP* непосредственно в свою среду разработки. Это означает, что нет нужды устанавливать дополнительные плагины или настраивать компиляторы отдельно для поддержки *OpenMP.*

Таким образом, выбор *Visual Studio* и связанных с ним инструментов обеспечивает полноценную интегрированную среду для разработки, отладки и оптимизации программы для распределенных вычислений на различных процессорах, включая *CPU* и *GPU*. [19]

**3.3 Связь архитектуры вычислительной системы с   
      разрабатываемым программным обеспечением**

Выбранная вычислительная система, основанная на операционной системе *Windows* 10 и ноутбуке *Acer Swift SF*314-59, имеет спецификации, которые имеют важное значение для разработки программы, ориентированной на распределенные вычисления, включая использование процессоров *CPU* и *GPU*.

Прежде всего, центральный процессор *Intel Core i*5 играет ключевую роль в выполнении вычислительных задач. В данном случае, его характеристики, такие как количество ядер, тактовая частота и поддержка технологий параллельного программирования, становятся факторами, которые необходимо учитывать. Технологии, такие как *Hyper-Threading* или *Turbo Boost*, могут повлиять на распараллеливание и общую производительность.

*Turbo Boost* – эта технология позволяет переводить некоторые ядра на более высокую частоту работы (выше номинальной), если текущее энергопотребление процессора свидетельствует о том, что он «недогружен» работой. Таким образом, по идее, *Turbo Boost* должен оказывать положительное влияние на скорость работы в основном старого, однопоточного ПО: именно в этом случае велика вероятность простоя «лишних» ядер.

*Hyper-Threading* (*HT*) позволяет эмулировать два логических (видимых операционной системой) ядра на базе одного физического за счёт отправки на исполнение физическому ядру команд из двух параллельно исполняемых потоков. Основная идея состоит в том, что некоторое количество исполнительных устройств в ядре почти всегда простаивает, т.к. для них не находится нужных команд. Если же на одном ядре исполнять сразу два потока – шансы на то, что все исполнительные устройства будут загружены работой, увеличатся, и общая производительность системы возрастёт. Однако при практической реализации у *Intel* не все блоки удалось сделать распределяемыми между двумя виртуальными ядрами динамически. В частности, *load/store/reorder* буферы при включении *HT* просто делятся между двумя виртуальными ядрами пополам. Таким образом, технические характеристики виртуального ядра всё-таки хуже, чем у одиночного физического ядра при выключенной *HT*, и производительность его в некоторых случаях может быть меньше. [20]

Оперативная память (*RAM*) объемом 16Гб является важным компонентом, особенно при работе с распределенными вычислениями, где доступ к большому объему памяти может повысить эффективность программы. Необходимо эффективно управлять памятью, особенно при параллельном выполнении задач на разных процессорах.

*SSD*-диск *NVMe SAMSUNG* емкостью 1024Гб обеспечивает высокую скорость чтения и записи, что может быть критичным для операций ввода-вывода при распределенных вычислениях. Это особенно важно, когда данные обрабатываются на различных процессорах.

Графический процессор *Intel Iris Xe Graphics*, хотя и является встроенным, но важен при использовании *GPU* для параллельных вычислений. Программное обеспечение должно быть настроено для оптимального использования возможностей *GPU* при выполнении конкретных вычислительных задач.

Таким образом, связь между архитектурой вычислительной системы и программой распределенных вычислений заключается в адаптации программы к характеристикам процессоров *CPU* и *GPU*, управлении оперативной памятью и оптимизации операций ввода-вывода на *SSD*-диске, что обеспечивает максимальную производительность в данной вычислительной среде.

# 4 Проектирование функциональных             возможностей программы

**4.1 Функциональные возможности программы**

Разработанная программа представляет собой мощное средство для выполнения численных вычислений, сфокусированное на эффективном использовании как графического процессора (*GPU*), так и центрального процессора (*CPU*). Программа реализует решение системы линейных уравнений методом *LU*-разложения матрицы.

Метод решения системы линейных алгебраических уравнений (СЛАУ) с использованием *LU*-разложения основан на математическом преобразовании матрицы коэффициентов системы в произведение двух треугольных матриц: нижней треугольной (*L*) и верхней треугольной (*U*).

Когда матрица коэффициентов системы представляется в виде *LU*-разложения, система линейных уравнений принимает вид *LUx = B*, где L и U – это матрицы *LU*-разложения, *x* – вектор неизвестных, и *B* – вектор правой части.

Процесс разложения матрицы *A* (коэффициентов системы) на L и U выполняется таким образом, что *L* – это нижнетреугольная матрица с единичной диагональю, а *U* – это верхнетреугольная матрица. Такое представление позволяет более эффективно решать системы линейных уравнений.

Основная идея заключается в том, что после *LU*-разложения, решение системы сводится к последовательному решению двух систем: первая система – *Lc* = *B*, где *c* – это новый вектор неизвестных, а вторая система – *Ux = c.* Решение этих систем легче, так как матрицы *L* и *U* обладают треугольной структурой.

Преимуществом использования *LU*-разложения является повышение эффективности численного решения СЛАУ, особенно при использовании распределенных и параллельных вычислений для повышения эффективности алгоритма.

Ниже перечислены ключевые функциональные возможности программы, подчеркивающие её гибкость, оптимизированный распределенный характер вычислений и способность взаимодействия с различными вычислительными ресурсами.

1 Интерфейс ввода данных. Программа предоставляет пользователю удобный интерфейс для ввода размера матрицы и её элементов, а также ввода вектора правых частей. Интерфейс является интуитивно понятным и удобным для взаимодействия.

2 *LU*-разложение и решение СЛАУ. Пользователь имеет возможность выполнить *LU*-разложение матрицы и решение системы линейных уравнений. Программа автоматически определяет, является ли матрица невырожденной, и только в этом случае предоставляет результаты *LU*-разложения и решения СЛАУ.

3 Визуализация результатов. Результаты *LU*-разложения (матрицы *L* и *U*) и решение системы линейных уравнений выводятся на экран для визуального анализа. Информация о матрицах и решении представлена в читаемой форме, что облегчает восприятие.

4 Измерение времени выполнения. Программа измеряет и отображает время выполнения *LU*-разложения на *GPU* и решения системы линейных уравнений на *CPU*. Пользователь получает информацию о производительности и эффективности вычислений на различных устройствах.

5 Управление ресурсами *OpenCL*. Программа автоматически управляет ресурсами *OpenCL*, такими как буферы, ядро, программа, командная очередь и контекст, обеспечивая корректное их освобождение после выполнения вычислений.

6 Интерактивность и обратная связь. Пользователь взаимодействует с программой через консольный интерфейс, вводя данные и получая результаты. В случае вырожденной матрицы, программа сообщает об этом и предостерегает пользователя от решения невозможной системы линейных уравнений.

7 Гибкость в выборе размера матрицы. Программа позволяет пользователю выбирать размер матрицы, что делает её универсальной для различных задач.

8 Простота использования. Интерфейс программы ориентирован на простоту использования, что позволяет пользователям без лишних трудностей проводить LU-разложение и решать системы линейных уравнений.

Такое проектирование обеспечит создание гибкой, производительной, интуитивно понятной и удобной для работы программы, способной эффективно использовать ресурсы *CPU* и *GPU* для *LU*-разложения и решения систем линейных уравнений методом, использующим это разложение.

**4.2 Обоснования и описание функций программы**

Программа представляет собой интегрированное решение для выполнения численных вычислений, сфокусированное на решении систем линейных уравнений (СЛАУ) через *LU*-разложение с использованием как графического процессора (*GPU*), так и центрального процессора (*CPU*). Рассмотрим основные функции программы:

1 Интерфейс ввода данных: интуитивно понятный пользовательский интерфейс позволяет пользователям легко взаимодействовать с программой. Предоставление удобного способа ввода размера матрицы, её элементов и вектора правых частей делает использование программы доступным для широкого круга пользователей. Ввод размера матрицы предоставлен на рисунке 4.1, ввод матрицы – на рисунке 4.2, ввод вектора – на рисунке 4.3.



Рисунок 4.1 – Ввод размера матрицы

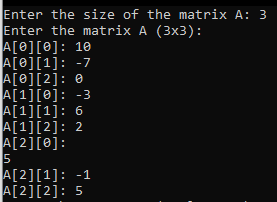


Рисунок 4.2. – Ввод матрицы

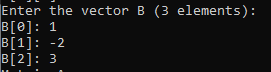


Рисунок 4.3 – Ввод вектора

2 *LU*-разложение на *GPU* и решение СЛАУ: распределение *LU*-разложения на *GPU* позволяет эффективно использовать мощности графического процессора и ускорить процесс разложения матрицы. Решение системы линейных уравнений с использованием распараллеливания алгоритма на *OpenMP* и уже разложенной матрицы обеспечивает точные и быстрые результаты.

3 Визуализация результатов: визуализация матриц *L* и *U*, а также решения СЛАУ, предоставляет пользователю наглядную картину выполненных вычислений. Это важно для анализа результатов и проверки корректности выполненных операций. Визуализация результатов *LU*-разложения предоставлена на рисунке 4.4, а решения СЛАУ – на рисунке 4.5.

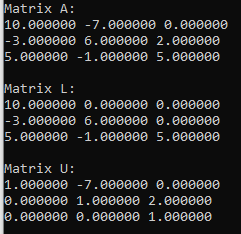


Рисунок 4.4 – Результаты *LU*-разложения матрицы *A*

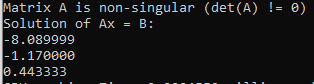


Рисунок 4.5 – Подтверждение, что матрица является невырожденной и результаты решения СЛАУ

4 Измерение времени выполнения: измерение времени выполнения *LU*-разложения на *GPU* и решения системы линейных уравнений на *CPU* предоставляет пользователям информацию о производительности программы на различных вычислительных устройствах. Это обеспечивает возможность выбора наилучшего подхода в зависимости от характеристик оборудования. На рисунке 4.6 предоставлены замеры времени выполнения *LU*-разложения на *GPU* и решения системы линейных уравнений на *CPU.*



Рисунок 4.6 – Замеры времени выполнения *LU*-разложения на *GPU* и решения системы линейных уравнений на *CPU*

5 Управление ресурсами *OpenCL*: эффективное управление ресурсами *OpenCL*, такими как буферы, ядро, программа, командная очередь и контекст, гарантирует оптимальное использование памяти и вычислительных ресурсов. Это способствует стабильной работе программы и предотвращает утечки памяти.

6 Гибкость и адаптивность: возможность выбора размера матрицы обеспечивает гибкость программы и позволяет пользователям адаптировать её под конкретные задачи, что важно для обеспечения универсальности программы в различных контекстах использования.

7 Простота использования: простота использования программы достигается через интуитивно понятный интерфейс, автоматическую проверку на вырожденность матрицы и четкую визуализацию результатов. Это позволяет пользователям с разным уровнем опыта эффективно взаимодействовать с программой.

Таким образом, разработанная, использующая распределенные вычисления на процессорах *CPU* и *GPU* для решения СЛАУ, программа представляет собой инструмент, который сочетает в себе высокую производительность, гибкость в настройке параметров и простоту использования, а также эффективное средство для ускоренного и оптимизированного выполнения вычислений, что особенно важно в контексте современных вычислительных архитектур.

# 5 Архитектура разрабатываемой программы

**5.1 Общая структура программы**

Программа для решения систем линейных алгебраических уравнений создана для эффективного выполнения математических вычислений с использованием распределенных вычислений на процессорах *CPU* и *GPU*. Позволяет пользователям легко решать системы линейных уравнений, выполняя *LU*-разложение на графическом процессоре (*GPU*) и последующее решение системы на центральном процессоре (*CPU*).

Основные компоненты программы включают в себя:

1 Ввод данных: это часть интерфейса, позволяющая пользователю вводить свои данные, для которых и будет выполняться программа.

2 Вывод матрицы *А* вместе с матрицами разложения: часть интерфейса, позволяющая пользователю проверить, верно ли ввелись данные в матрицу *А,* а также верно ли произошло *LU*-разложение матрицы.

3 Результат вычислений: пользователь может видеть проверку на то, является ли исходная матрица невырожденной, а также решение системы линейных алгебраических уравнений.

4 Время, затраченное на выполнение *LU*-разложения на *GPU* и решение системы линейных уравнений на *CPU*: пользователь может видеть время, затраченное процессорами на выполнение поставленных задач при распределении на них вычислений.

Программа структурирована с использованием функций для вывода матриц, умножения матриц, решения системы линейных уравнений, инициализации *OpenCL* и освобождения ресурсов. Она обеспечивает взаимодействие с пользователем, эффективное использование *GPU* для вычислений, и обработку решения системы линейных уравнений на *CPU* с использованием параллельных вычислений *OpenMP*.

Таким образом, общая структура программы ориентирована на предоставление пользователю возможности быстро посчитать решение системы линейных алгебраических уравнений в удобном и интуитивно понятном виде.

**5.2 Описание функциональной схемы программы**

Описание функциональной схемы программы, реализующей решение СЛАУ методом *LU*-разложения, включает следующие ключевые компоненты и функции:

– ввод данных;

– выполнение *LU*-разложения исходной матрицы на *GPU*;

– проверка невырожденности исходной матрицы;

– решение СЛАУ на *CPU* с использованием параллельных вычислений;

– вывод результатов;

– освобождение ресурсов.

Программа начинает свое выполнение с интерактивного взаимодействия с пользователем, где запрашиваются необходимые входные данные. Пользователь предоставляет размер матрицы *A*, элементы этой матрицы и вектор *B*, представляющий правую часть системы линейных уравнений (СЛАУ). Эта информация необходима для построения исходной матрицы системы.

После успешного ввода данных, программа переходит к следующему этапу, который включает в себя выполнение *LU*-разложения на графическом процессоре (*GPU*). Для этого производится передача матрицы *A* на устройство *GPU*, где выполняется специальное *OpenCL*-ядро для проведения *LU*-разложения. Результатом являются матрицы *L* и *U*, представляющие собой разложение исходной матрицы.

Далее программа проверяет вырожденность исходной матрицы, вычисляя определитель матрицы *U*. Этот шаг необходим для обеспечения возможности дальнейшего решения СЛАУ на центральном процессоре (*CPU*). Если определитель не равен нулю, что свидетельствует о невырожденности матрицы, то программа продолжает выполнение. В противном случае, выводится сообщение о том, что матрица является вырожденной, и решение СЛАУ невозможно.

После проверки вырожденности, происходит решение системы линейных уравнений на *CPU*. Этот этап включает в себя параллельные вычисления, которые обеспечивают более эффективное выполнение алгоритма.

В конце работы программы, пользователю предоставляется возможность просмотреть всю необходимую информацию через пользовательский интерфейс, что может включать в себя вывод матриц *L* и *U*, результатов умножения этих матриц, решения СЛАУ, проверки на вырожденность, а также времени выполнения на *GPU* и *CPU*. Функциональная схема разработанного проекта предоставлена в приложении Б.

**5.3 Описание блок схемы алгоритма программы**

Блок схема алгоритма, реализующей решение СЛАУ методом *LU*-разложения, представляет основные шаги и этапы алгоритма программы для решения систем линейных алгебраических уравнений методом *LU*-разложения. Блок схема алгоритма представлена в приложении В.

Программа начинается с динамического выделения памяти под объекты (используемые матрицы), после чего идет создание OpenCL контекста, командной очереди и буферов для использования *GPU* для распределенных вычислений. Сразу после происходит выполнение *LU*-разложения исходной матрицы с использованием *GPU*, чтение результата с графического процессора в общую память программы, проверяет, является ли матрица невырожденной и если да, то решает СЛАУ с использованием параллельных вычислений на *CPU*.

Взаимодействие с пользователем, интерфейс и взаимодействие с операционной системой дополняют основной алгоритм. Так же в блок схеме алгоритма есть условие невырожденности исходной матрицы, где программа проверяет, невырожденная матрица или нет. Если матрица является вырожденной, то программа сразу выводит сообщение об этом, освобождает ресурсы и завершается.

В итоге, разработанная программа представляет собой интуитивно понятный, функциональный и настраиваемый инструмент, который удовлетворяет потребности разнообразной аудитории пользователей. Сочетание простоты использования с богатой функциональностью делает эту программу неотъемлемой частью опыта вычисления систем линейных алгебраических выражений пользователей.

ЗАКЛЮЧЕНИЕ

В рамках данного курсового проекта была разработана и реализована программа, реализующая решение СЛАУ методом *LU*-разложения с использованием распределенных вычислений для достижения наибольшей эффективности. В процессе разработки использовались современные технологии, такие как язык программирования *C/C++*, среда разработки *VisualStudio*, библиотека *OpenCL* для использования *GPU,* библиотека *OpenMP* для использования параллельных вычислений, библиотека *iostream* для вывода результатов, библиотека *chrono* для отслеживания времени работы процессоров *CPU* и *GPU* для распределенных вычислений.

Интерфейс приложения включает в себя информацию о введенном размере матрицы, введенной матрице, введенном векторе, матрицах, полученных вследствие *LU*-разложения, сведения о невырожденности исходной матрицы, результаты решения СЛАУ, время, затраченное на *LU*-разложение на *GPU* и затраченное на решение СЛАУ с помощью параллельных вычислений на *CPU*.

Общая структура программы ориентирована на предоставление пользователю возможности быстро посчитать решение системы линейных алгебраических уравнений в удобном и интуитивно понятном виде.

Таким образом, созданная программа предоставляет пользователю удобный и информативный инструмент для решения СЛАУ методом *LU*-разложения с использованием распределенных вычислений для достижения наибольшей эффективности.

СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

[1] Архитектура вычислительных систем [Электронный ресурс]: учебное пособие – Эл. изд. - Электрон. текстовые дан. (1 файл pdf: 77 с.). – Грейбо С.В., Новосёлова Т.Е., Пронькин Н.Н., Семёнычева И.Ф. 2019. – Режим доступа: <http://scipro.ru/conf/computerarchitecture.pdf>

[2] Каптерев А.И. Электронный учебник по информатике [Электронный ресурс]. – Режим доступа: http://www.mediagnosis.ru/Autorun/Page6/5\_3\_.htm

[3] Скотт Мюллер Модернизация и ремонт ПК 17-е изд. М. Вильямс, 2007, 499 – 572, 653 – 700 c.

[4] Яндекс Глоссарий [Электронный ресурс]. – Режим доступа: https://cloud.yandex.ru/docs/glossary/gpu.

[5] Microsoft Documents [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/windows-hardware/drivers/debugger/x64-architecture

[6] История развития оперативной памяти [Электронный ресурс]. – Режим доступа: https://m-i-kuznetsov.livejournal.com/213674.html

[7] NVMe-накопитель Samsung [Электронный ресурс]. – Режим доступа: https://3dnews.ru/1021227/obzor-nvmenakopitelya-samsung-980-pro

[8] История развития графики Intel [Электронный ресурс]. – Режим доступа: https://club.dns-shop.ru/blog/t-100-protsessoryi/78375-istoriya-razvitiya-grafiki-intel-chast-2/

[9] Microsoft Windows 10 – полный обзор [Электронный ресурс]. – Режим доступа: https://xn--80aa0aebnilejl.xn--p1ai/%D0%9F%D0%BE%D0%BB%D0%B5%D0%B7%D0%BD%D0%B0%D1%8F\_%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%86%D0%B8%D1%8F/Microsoft\_Windows\_10/

[10] Что такое операционная система [Электронный ресурс]. – Режим доступа: https://help.reg.ru/support/servery-vps/oblachnyye-servery/ustanovka-programmnogo-obespecheniya/chto-takoye-operatsionnaya-sistema

[11] Интегрированная среда разработки [Электронный ресурс]. – Режим доступа: https://blog.skillfactory.ru/glossary/ide/

[12] Язык программирования [Электронный ресурс]. – Режим доступа: https://blog.skillfactory.ru/glossary/yazyk-programmirovaniya/

[13] Microsoft Learn [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/visualstudio/get-started/visual-studio-ide?view=vs-2022

[14] Web proger C/C++ [Электронный ресурс]. – Режим доступа: http://web.spt42.ru/index.php/chto-takoe-c-plus-plus

[15] Преимущества распределенных вычислений [Электронный ресурс]. – Режим доступа: https://aws.amazon.com/ru/what-is/distributed-computing/

[16] Многозадачность и многопоточность. Основные понятия [Электронный ресурс]. – Режим доступа: https://radioprog.ru/post/1402

[17] OpenCL. Открытый язык для параллельных программ [Электронный ресурс]: учебное пособие – Эл. изд. - Электрон. текстовые дан. (1 файл pdf: 88 с.). – Антонюк В.А. 2017. – Режим доступа: https://cmp.phys.msu.ru/sites/default/files/OpenCL.pdf

[18] Introduction of OpenMP [Электронный ресурс]. – Режим доступа: https://carleton.ca/rcs/rcdc/introduction-to-openmp/

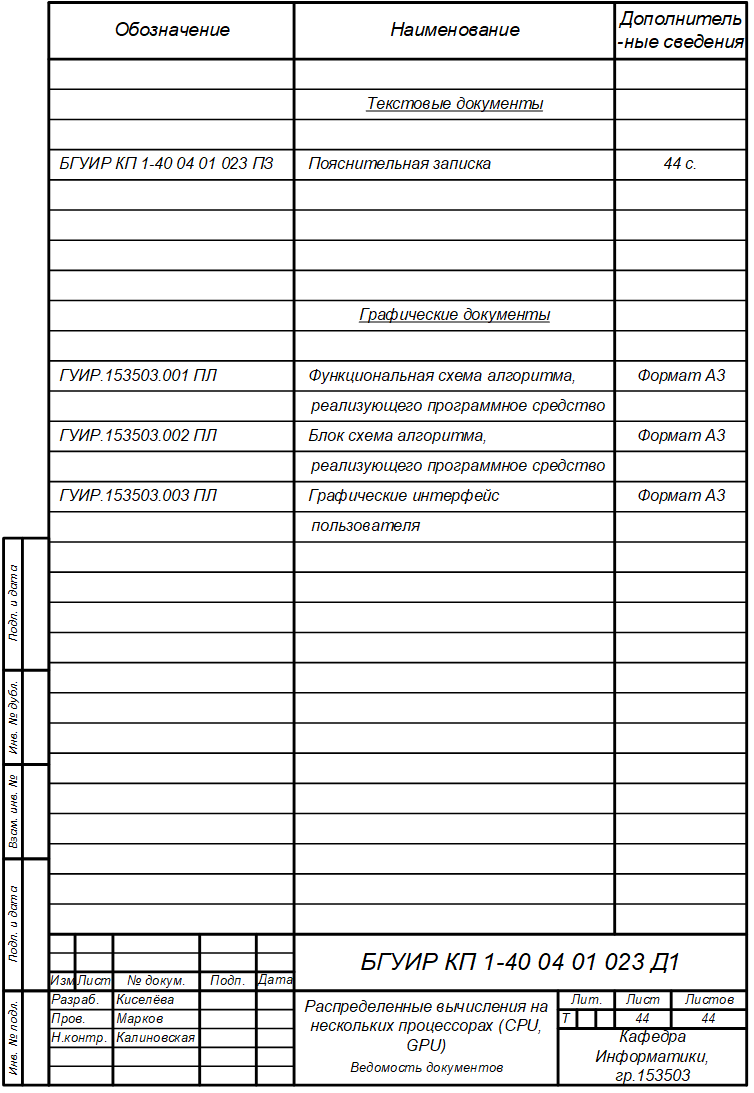
[19] Методы и средства отладки Visual Studio [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/visualstudio/debugger/write-better-code-with-visual-studio?view=vs-2022

[20] Влияние различных характеристик на быстродействие процессоров современных архитектур. [Электронный ресурс]. – Режим доступа: https://ixbt.com/cpu/archspeed-2009-3.shtml

# ПРИЛОЖЕНИЕ А

**(обязательное)**

**Ведомость документов**

****

# ПРИЛОЖЕНИЕ Б

**(обязательное)**

**Листинг программного кода**

**Main.cs**

#include <CL/cl.h>

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#include <chrono>

#include <iostream>

const char\* luDecompositionKernelSource =

"\_\_kernel void luDecomposition(\_\_global float\* A, \_\_global float\* L, \_\_global float\* U, const int N) {\n"

" int i, j, k;\n"

"\n"

" for (k = 0; k < N; k++) {\n"

" float Akk = A[k \* N + k];\n"

" L[k \* N + k] = 1.0f;\n"

" U[k \* N + k] = Akk;\n"

"\n"

" for (i = k + 1; i < N; i++) {\n"

" L[i \* N + k] = A[i \* N + k] / Akk;\n"

" U[k \* N + i] = A[k \* N + i];\n"

" }\n"

"\n"

" for (i = k + 1; i < N; i++) {\n"

" for (j = k + 1; j < N; j++) {\n"

" A[i \* N + j] -= L[i \* N + k] \* U[k \* N + j];\n"

" }\n"

" }\n"

" }\n"

"}\n";

void printMatrix(const char\* name, float\* matrix, int rows, int cols) {

printf("%s:\n", name);

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

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

printf("%f ", matrix[i \* cols + j]);

}

printf("\n");

}

printf("\n");

}

void matrixMultiply(float\* L, float\* U, float\* result, int rows, int cols) {

#pragma omp parallel for collapse(3)

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

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

{

float sum = 0;

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

sum += L[i \* cols + k] \* U[k \* cols + j];

result[i \* cols + j] = sum;

}

}

void solveLinearSystem(float\* L, float\* U, float\* B, float\* result, int size) {

float\*\* y\_thread = (float\*\*)malloc(sizeof(float\*) \* omp\_get\_max\_threads());

#pragma omp parallel

{

int thread\_id = omp\_get\_thread\_num();

y\_thread[thread\_id] = (float\*)malloc(sizeof(float) \* size);

#pragma omp for

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

float sum = 0.0f;

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

sum += L[i \* size + j] \* y\_thread[thread\_id][j];

}

y\_thread[thread\_id][i] = (B[i] - sum) / L[i \* size + i];

}

#pragma omp barrier

#pragma omp for

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

float sum = 0.0f;

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

sum += U[i \* size + j] \* result[j];

}

result[i] = (y\_thread[thread\_id][i] - sum) / U[i \* size + i];

}

free(y\_thread[thread\_id]);

}

free(y\_thread);

}

// Функция для вычисления определителя на основе матрицы U с распараллеливанием

long double determinantFromLU(float\* matrixU, int size) {

long double det = 1.0f;

#pragma omp parallel for reduction(\*:det) num\_threads(omp\_get\_max\_threads())

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

det \*= matrixU[i \* size + i];

}

return det;

}

// Функция для вычисления определителя матрицы последовательно

float determinant(float\* matrix, int size) {

if (size == 1) {

return matrix[0];

}

if (size == 2) {

return matrix[0] \* matrix[3] - matrix[1] \* matrix[2];

}

float det = 0;

float sign = 1;

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

// Используем динамический массив для минора

float\* minor = (float\*)calloc((size - 1) \* (size - 1), sizeof(float));

for (int j = 1; j < size; j++) {

for (int k = 0, col = 0; k < size; k++) {

if (k != i) {

minor[(j - 1) \* (size - 1) + col++] = matrix[j \* size + k];

}

}

}

det += sign \* matrix[i] \* determinant(minor, size - 1);

// Освобождение памяти, выделенной под минор

free(minor);

sign = -sign;

}

return det;

}

void fillMatrixRandom(float\* matrix, int size) {

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

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

// Генерация случайного числа в диапазоне от 0 до 1

matrix[i \* size + j] = (float)rand() / RAND\_MAX;

}

}

}

void fillVectorRandom(float\* vector, int size) {

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

vector[i] = rand() % 10;

}

}

int main() {

cl\_platform\_id platform;

clGetPlatformIDs(1, &platform, NULL);

cl\_device\_id device;

clGetDeviceIDs(platform, CL\_DEVICE\_TYPE\_GPU, 1, &device, NULL);

cl\_context context = clCreateContext(NULL, 1, &device, NULL, NULL, NULL);

cl\_command\_queue queue = clCreateCommandQueueWithProperties(context, device, NULL, NULL);

// Ввод размера матрицы

int MATRIX\_SIZE;

do {

printf("Enter the size of the matrix A: ");

if (scanf\_s("%d", &MATRIX\_SIZE) != 1 || MATRIX\_SIZE <= 0) {

printf("Invalid input. Please enter a positive integer.\n");

while (getchar() != '\n'); // Очистка буфера ввода

}

} while (MATRIX\_SIZE <= 0);

// Динамическое выделение памяти для матрицы A и вектора B

float\* matrixA = (float\*)calloc(MATRIX\_SIZE \* MATRIX\_SIZE, sizeof(float));

float\* matrixB = (float\*)calloc(MATRIX\_SIZE, sizeof(float));

// Ввод матрицы A

//fillMatrixRandom(matrixA, MATRIX\_SIZE);

printf("Enter the matrix A (%dx%d):\n", MATRIX\_SIZE, MATRIX\_SIZE);

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

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

printf("A[%d][%d]: ", i, j);

while (scanf\_s("%f", &matrixA[i \* MATRIX\_SIZE + j]) != 1) {

printf("Invalid input. Please enter a valid floating-point number.\n");

while (getchar() != '\n');

}

}

}

// Ввод вектора B

//fillVectorRandom(matrixB, MATRIX\_SIZE);

printf("Enter the vector B (%d elements):\n", MATRIX\_SIZE);

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

printf("B[%d]: ", i);

// Проверка ввода на float

while (scanf\_s("%f", &matrixB[i]) != 1) {

printf("Invalid input. Please enter a valid floating-point number.\n");

// Очистка буфера ввода

while (getchar() != '\n');

printf("B[%d]: ", i);

}

}

printMatrix("Matrix A", matrixA, MATRIX\_SIZE, MATRIX\_SIZE);

int matrixSize = MATRIX\_SIZE;

cl\_mem bufferA = clCreateBuffer(context, CL\_MEM\_READ\_ONLY | CL\_MEM\_COPY\_HOST\_PTR,

sizeof(float) \* MATRIX\_SIZE \* MATRIX\_SIZE, matrixA, NULL);

cl\_mem bufferMatrixSize = clCreateBuffer(context, CL\_MEM\_READ\_ONLY | CL\_MEM\_COPY\_HOST\_PTR,

sizeof(int), &matrixSize, NULL);

cl\_program luDecompositionProgram = clCreateProgramWithSource(context, 1, &luDecompositionKernelSource, NULL, NULL);

clBuildProgram(luDecompositionProgram, 1, &device, NULL, NULL, NULL);

cl\_kernel kernelLU = clCreateKernel(luDecompositionProgram, "luDecomposition", NULL);

// Выделение памяти для L и U матриц

float\* matrixL = (float\*)calloc(MATRIX\_SIZE \* MATRIX\_SIZE, sizeof(float));

float\* matrixU = (float\*)calloc(MATRIX\_SIZE \* MATRIX\_SIZE, sizeof(float));

cl\_mem bufferL = clCreateBuffer(context, CL\_MEM\_READ\_WRITE,

sizeof(float) \* MATRIX\_SIZE \* MATRIX\_SIZE, NULL, NULL);

cl\_mem bufferU = clCreateBuffer(context, CL\_MEM\_READ\_WRITE,

sizeof(float) \* MATRIX\_SIZE \* MATRIX\_SIZE, NULL, NULL);

// Выполнение LU-разложения матрицы на GPU

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

clSetKernelArg(kernelLU, 0, sizeof(cl\_mem), &bufferA);

clSetKernelArg(kernelLU, 1, sizeof(cl\_mem), &bufferL);

clSetKernelArg(kernelLU, 2, sizeof(cl\_mem), &bufferU);

clSetKernelArg(kernelLU, 3, sizeof(int), &matrixSize);

size\_t globalSizeLU[2] = { MATRIX\_SIZE, MATRIX\_SIZE };

clEnqueueNDRangeKernel(queue, kernelLU, 2, NULL, globalSizeLU, NULL, 0, NULL, NULL);

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

double GPUworkingTime = std::chrono::duration<double, std::milli>(endGPU - startGPU).count();

clEnqueueReadBuffer(queue, bufferL, CL\_TRUE, 0, sizeof(float) \* MATRIX\_SIZE \* MATRIX\_SIZE, matrixL, 0, NULL, NULL);

clEnqueueReadBuffer(queue, bufferU, CL\_TRUE, 0, sizeof(float) \* MATRIX\_SIZE \* MATRIX\_SIZE, matrixU, 0, NULL, NULL);

printMatrix("Matrix L", matrixL, MATRIX\_SIZE, MATRIX\_SIZE);

printMatrix("Matrix U", matrixU, MATRIX\_SIZE, MATRIX\_SIZE);

float\* matrixResult = (float\*)calloc(MATRIX\_SIZE \* MATRIX\_SIZE, sizeof(float));

//Умножение матриц

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

matrixMultiply(matrixL, matrixU, matrixResult, MATRIX\_SIZE, MATRIX\_SIZE);

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

double CPUParallelWorkingTimemMultiply = std::chrono::duration<double, std::milli>(endCPUmMultiply - startCPUmMultiply).count();

printMatrix("Matrix Result (L \* U)", matrixResult, MATRIX\_SIZE, MATRIX\_SIZE);

std::cout << "CPU parallel matrix multiply time: " << CPUParallelWorkingTimemMultiply / 1000 << " milliseconds" << std::endl;

// Вычисление определителя

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

long double det = determinantFromLU(matrixU, MATRIX\_SIZE);

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

//Последовательное вычисление определителя

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

float detP = determinant(matrixA, MATRIX\_SIZE);

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

printf("Determinant: %f\n", det);

double CPUParallelWorkingTimeDet = std::chrono::duration<double, std::milli>(endCPUdet - startCPUdet).count();

double TimeDet = std::chrono::duration<double, std::milli>(enddet - startdet).count();

std::cout << "CPU parallel calculating determinant time: " << CPUParallelWorkingTimeDet / 1000 << " milliseconds" << std::endl;

std::cout << "Sequentially calculating determinant time: " << TimeDet / 1000 << " milliseconds" << std::endl;

// Решение системы линейных уравнений Ax = B

float\* solution = (float\*)calloc(MATRIX\_SIZE, sizeof(float));

double CPUParallelWorkingTime;

// Проверка невырожденности матрицы A

if (det != 0.0f) {

printf("Matrix A is non-singular (det(A) != 0)\n");

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

solveLinearSystem(matrixL, matrixU, matrixB, solution, MATRIX\_SIZE);

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

CPUParallelWorkingTime = std::chrono::duration<double, std::milli>(endCPU - startCPU).count();

printf("Solution of Ax = B:\n");

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

printf("%f\n", solution[i]);

}

std::cout << "CPU parallel solution of SoLE working time: " << CPUParallelWorkingTime / 1000 << " milliseconds" << std::endl;

}

else {

printf("Matrix A is singular (det(A) = 0), cannot solve the system.\n");

}

std::cout << "GPU LU-decomposition working time: " << GPUworkingTime / 1000 << " milliseconds" << std::endl;

// Освобождение памяти

clReleaseMemObject(bufferL);

clReleaseMemObject(bufferU);

clReleaseKernel(kernelLU);

clReleaseProgram(luDecompositionProgram);

clReleaseCommandQueue(queue);

clReleaseContext(context);

free(matrixA);

free(matrixB);

free(matrixL);

free(matrixU);

free(matrixResult);

free(solution);

return 0;

}

# ПРИЛОЖЕНИЕ В

**(обязательное)**

**Функциональная схема алгоритма, реализующего программное средство**

# ПРИЛОЖЕНИЕ Г

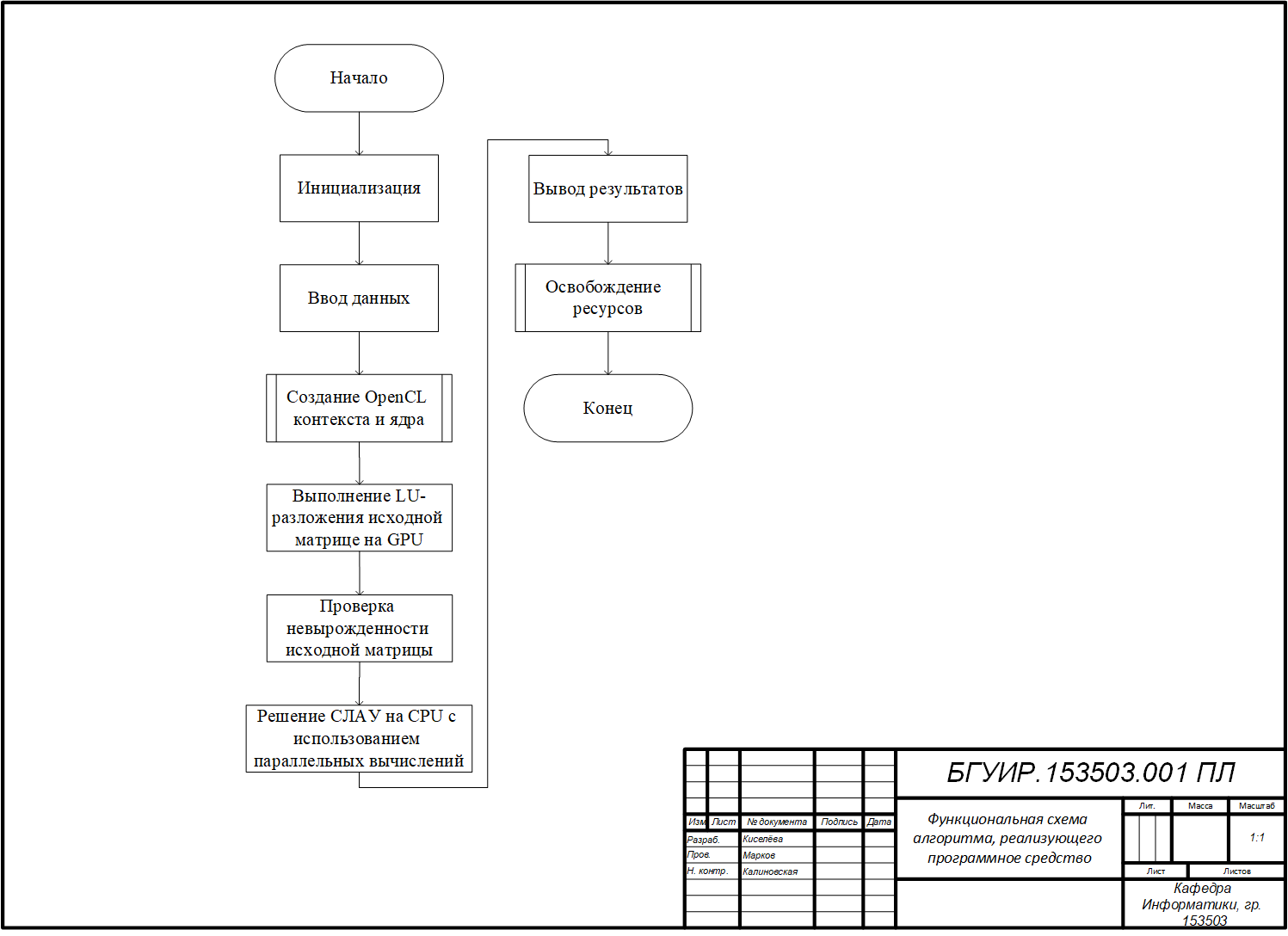
**(обязательное)**

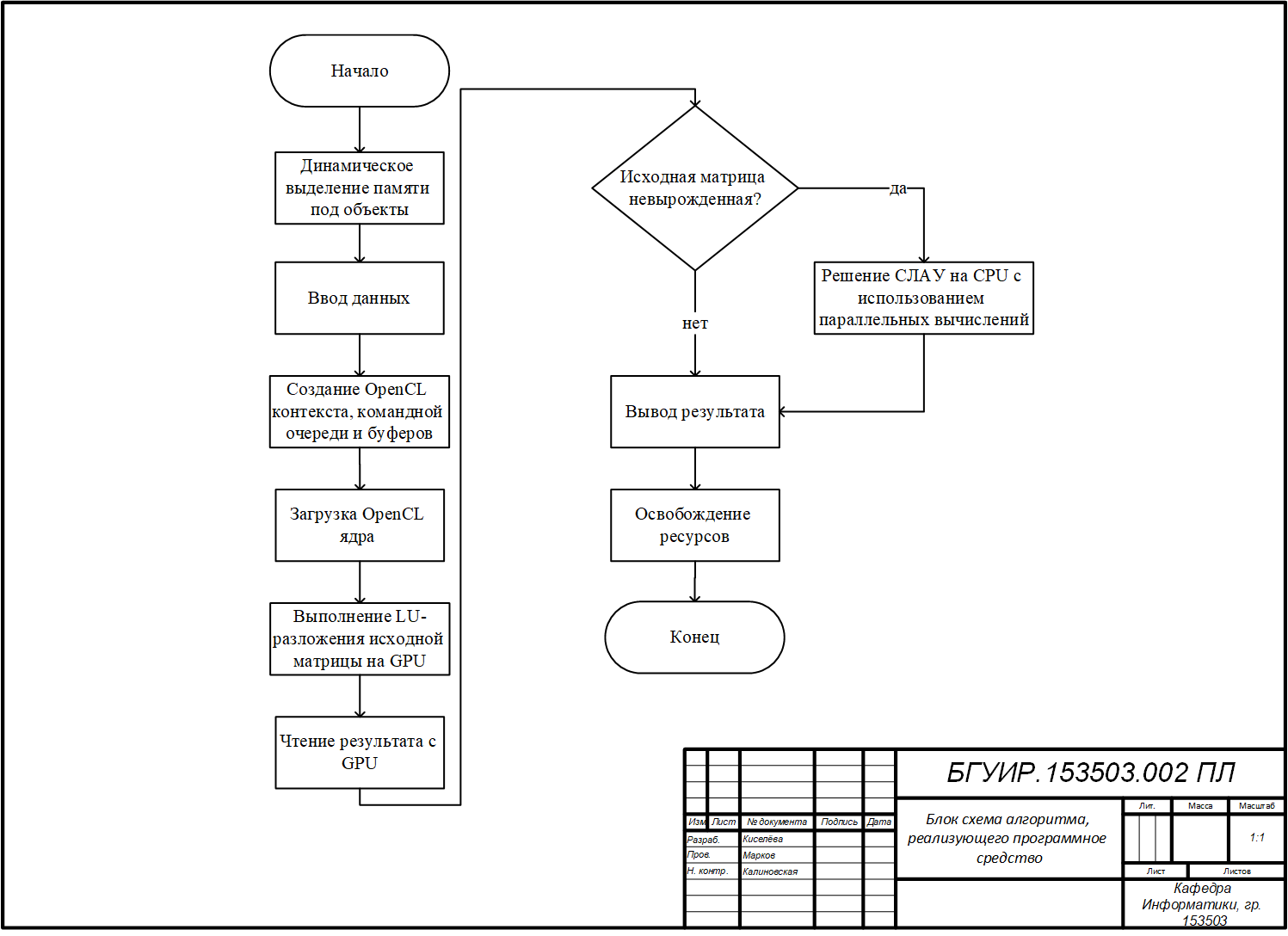
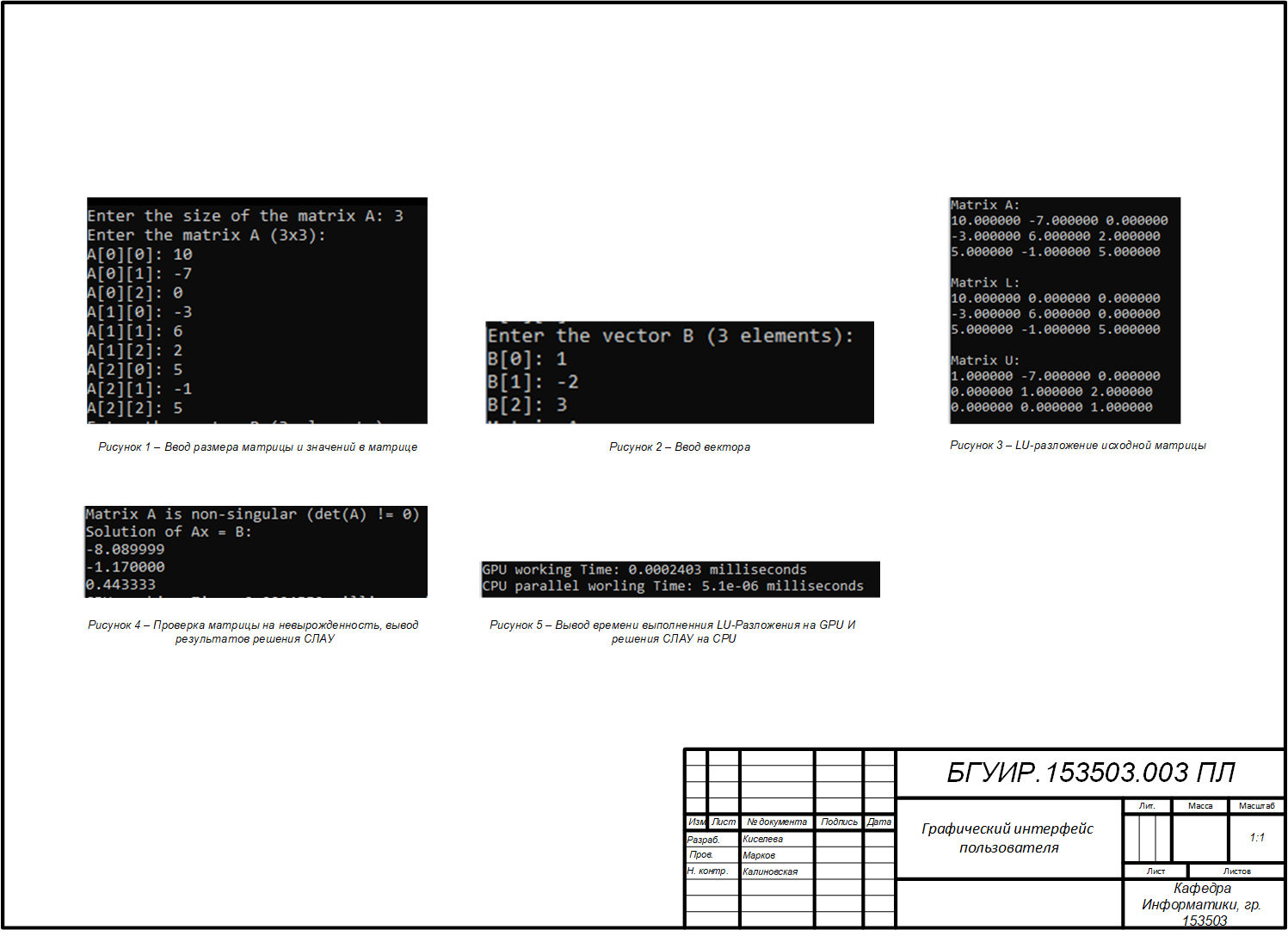
**Блок схема алгоритма, реализующего программное средство**

# ПРИЛОЖЕНИЕ Д

**(обязательное)**

**Графический интерфейс пользователя**

****

**  
**