Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Архитектура вычислительных систем

*К защите допустить*:

И.О. заведующего кафедрой информатики

\_\_\_\_\_\_\_\_\_\_\_\_ С.И. Сиротко

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту

на тему

**ИСПОЛЬЗОВАНИЕ *GPU* ДЛЯ ОПТИМИЗАЦИИ ВРЕМЕНИ ВЫПОЛНЕНИЯ КОДА**

БГУИР КП 1-40 04 01 01 024 ПЗ

Студент: гр.153502 Толстой Д.В.

Руководитель: старший преподаватель кафедры информатики

Марков А.Н.

Нормоконтролер: ассистент кафедры информатики Калиновская А.А

Минск 2023

**СОДЕРЖАНИЕ**

[Введение 5](#_Toc153122992)

[1 Архитектура вычислительной системы 6](#_Toc153122993)

[1.1 История развития графических процессоров 6](#_Toc153122994)

[1.2 Общие сведения об архитектуре современного *GPU* 8](#_Toc153122995)

[1.3 Области применения *GPU* 10](#_Toc153122996)

[2 Платформа программного обеспечения 14](#_Toc153122997)

[2.1 Операционная система 14](#_Toc153122998)

[2.2 Язык программирования 16](#_Toc153122999)

[3 Теоретическое обоснование разработки программного продукта 19](#_Toc153123000)

[3.1 Обоснование необходимости анализа 19](#_Toc153123001)

[3.2Перемножение матриц 20](#_Toc153123002)

[3.3Поиск всех простых чисел меньше заданного числа 20](#_Toc153123003)

[3.4Поиск НОП из 2 последовательностей 21](#_Toc153123004)

[3.5 *Taichi* 22](#_Toc153123005)

[3.6 *Numba* 23](#_Toc153123006)

[4 Проектирование функциональных возможностей программы 24](#_Toc153123007)

[4.1 Функции программного обеспечения 24](#_Toc153123008)

[5 Анализ оптимизации времени выполнения кода с помощью *GPU* 26](#_Toc153123009)

[5.1 Анализ полученных результатов 26](#_Toc153123010)

[Заключение 32](#_Toc153123011)

[Список литературных источников 33](#_Toc153123012)

[Приложение А (обязательное) Графики сравнения 35](#_Toc153123013)

[Приложение Б (обязательное) Функциональная схема алгоритма 37](#_Toc153123014)

[Приложение В (обязательное) Блок-схема алгоритма 39](#_Toc153123015)

[Приложение Г (обязательное) Листинг программного кода 41](#_Toc153123016)

[Приложение Д (обязательное) Ведомость документов 51](#_Toc153123017)

# **ВВЕДЕНИЕ**

Графические процессоры (*GPU*) стали неотъемлемой частью современных вычислительных систем и нашли широкое применение в различных областях, включая графику, научные исследования, искусственный интеллект и обработку данных. Однако, помимо их первоначального предназначения для обработки графики, *GPU* обладают огромным потенциалом для ускорения вычислений в других областях, в частности, в оптимизации времени выполнения программного кода.

Использование *GPU* для оптимизации времени выполнения кода становится всё более актуальной задачей, учитывая растущие требования к вычислительной мощности и скорости обработки данных в современных информационных технологиях. *GPU* обладают параллельной архитектурой и большим количеством ядер, что позволяет им выполнять множество вычислительных задач одновременно и существенно ускорять процессы обработки данных.

Цель данного курсового проекта состоит в исследовании и анализе методов и технологий, позволяющих использовать *GPU* для оптимизации времени выполнения кода. Будут рассмотрены основные принципы параллельных вычислений на *GPU*, а также различные подходы и инструменты для переноса и оптимизации программного кода на графический процессор. Кроме того, будут исследованы примеры практического применения *GPU* для ускорения вычислительных задач и рассмотрены результаты исследований в этой области.

В итоге, данное исследование поможет разработчикам и инженерам эффективно использовать *GPU* для оптимизации времени выполнения своих программных проектов, что является важным аспектом в современной вычислительной индустрии.

# **1 АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНОЙ СИСТЕМЫ**

## **1.1 История развития графических процессоров**

С самого начала, с 1951 года, Массачусетский технологический институт создал *Whirlwind*, авиационный симулятор для ВМФ. Хотя его можно считать первой *3D*-графической системой, основа современных *GPU* была заложена в середине 70-х с так называемыми видео-сдвигателями и видео-генераторами адресов. Они передавали информацию от центрального процессора к дисплею. Специализированные графические чипы широко использовались в аркадных игровых системах. В 1976 году *RCA* создала видеочип "*Pixie*", который мог выдавать видеосигнал разрешением 62×128. Графическое оборудование аркадной системы *Namco* *Galaxian* уже в 1979 году поддерживало *RGB*-цвета, спрайты множественных цветов и фон из плиточных карт.

В 1981 году *IBM* начала использовать монохромные и цветные адаптеры дисплея (*MDA/CDA*) в своих ПК. Хотя это был еще не современный *GPU*, это был отдельный компонент компьютера, предназначенный для одной цели: отображения видео. Сначала это было 80 столбцов по 25 строк текстовых символов или символов. В 1983 году *Intel* выпустила *ISBX* 275 *Video* *Graphics* *Controller* *Multimodule* *Board*, следующее революционное устройство. Оно могло показывать восемь цветов разрешением 256x256 или монохромное разрешением 512x512.

В 1985 году три иммигранта из Гонконга в Канаде создали компанию *Array* *Technology* *Inc*, вскоре переименованную в *ATI* *Technologies*. Эта компания на протяжении многих лет была лидером рынка с линейкой графических плат и чипов *Wonder*.

*S3* *Graphics* представила *S3* *86C911*, названную в честь *Porsche* *911*, в 1991 году. Название указывало на увеличение производительности. Эта карта породила множество подражателей: к 1995 году все крупные производители графических карт добавили поддержку *2D*-ускорения к своим чипам. В течение 1990-х годов уровень интеграции видеокарт значительно улучшился с добавлением дополнительных интерфейсов прикладного программирования (*API*).

В целом, начало 1990-х годов было временем создания множества компаний по производству графического оборудования, которые затем либо покупали, либо выводили из бизнеса. Среди победителей, основанных в это время, была компания *NVIDIA*. К концу 1997 года у этой компании было почти 25 процентов рынка графики.

История современных *GPU* начинается в 1995 году с появления первых *3D*-аддитивных карт, а затем принятием 32-битных операционных систем и доступных персональных компьютеров. Ранее отрасль сосредотачивалась на *2D* и не-ПК архитектуре, и графические карты были в основном известны по алфавитно-цифровым названиям и огромным ценникам.

Графическая карта *3DFx* *Voodoo*, запущенная в конце 1996 года, захватила около 85% рынка. Карты, способные только визуализировать *2D*, очень быстро устарели. *Voodoo1* полностью обходила *2D*-графику; пользователям приходилось использовать ее вместе с отдельной *2D*-картой. Но это все равно было открытием для геймеров. Следующий продукт компании, *Voodoo2* (1998), имел три встроенных чипа и был одной из первых видеокарт, поддерживающих параллельную работу двух карт в одном компьютере.

С развитием технологий производства видео, *2D* *GUI* ускорение и *3D*-функциональность были интегрированы в один чип. Чипсеты *Rendition* *Verite* были одними из первых, кто сделал это хорошо. Видеокарты с *3D*-ускорителями перестали быть просто растеризаторами.

Наконец, "первый в мире *GPU*" пришел в 1999 году. Именно так *Nvidia* рекламировала свою *GeForce* 256. *Nvidia* определила термин *GPU* как «однокристальный процессор с интегрированными трансформацией, освещением, настройкой/обрезкой треугольников и рендеринговыми двигателями, способный обрабатывать не менее 10 миллионов полигонов в секунду».

Соперничество между *ATI* и *Nvidia* было основным моментом начала 2000-х годов. Обе компании предлагали графические карты с функциями, которые теперь являются обыденными. Например, возможность выполнения зеркального отражения, объемного взрыва, волн, преломления, объемных теней, смешивания вершин, карты рельефа и карты высот.

Эра общего назначения *GPU* началась в 2007 году. Компании *Nvidia* и *ATI* (впоследствии приобретенная *AMD*) внедряли в свои графические карты все больше количество возможностей.

Однако, обе компании пошли по разным путям в отношении общего вычислительного использования *GPU* (*GPGPU*). В 2007 году *Nvidia* выпустила свою среду разработки *CUDA*, самую раннюю широко используемую модель программирования для вычислений на *GPU*. Два года спустя появилась широко поддерживаемая *OpenCL*. Этот фреймворк позволяет разрабатывать код как для *GPU*, так и для ЦП с акцентом на переносимость. Таким образом, *GPU* стали более обобщенным вычислительным устройством.

В 2010 году *Nvidia* сотрудничала с *Audi*. Они использовали *GPU* *Tegra* для управления панелью приборов автомобилей и увеличения систем навигации и развлечений. Эти достижения в графических картах в автомобилях способствовали технологии самоуправляемых автомобилей.

*Pascal* - самое новое поколение графических карт от *Nvidia*, выпущенное в 2016 году. Их 16-нм техпроцесс улучшает предыдущие микроархитектуры. *AMD* выпустила *GPU* *Polaris* 11 и *Polaris* 10 с техпроцессом 14 нм, что привело к значительному увеличению производительности на ватт. Однако энергопотребление современных *GPU* также увеличилось.

Сегодня графические процессоры не только для графики. Они нашли свое применение в таких разных областях, как машинное обучение, разведка нефти, научная обработка изображений, статистика, линейная алгебра, *3D*-реконструкция, медицинские исследования и даже определение цен на фондовых опционах. Технология *GPU* склонна добавлять еще больше программной настройки и параллелизма к базовой архитектуре, которая постоянно развивается в сторону ядра общего назначения, подобного ЦП. [1]

## **1.2 Общие сведения об архитектуре современного *GPU***

Первые компьютерные видеокарты имели лишь кадровый буфер: изображение формировалось центральным процессором компьютера и программным обеспечением, а карта отвечала за хранение кадров в буфере памяти и вывод их на монитор.

Однако повышение требований к качеству изображения привело к созданию специализированного процессора, который занимается исключительно расчетом и формированием изображения, освобождая от этих обязанностей центральный процессор. Современные графические процессоры по сложности не уступают центральным процессорам, и более того, во многих случаях в них используют технологии, опережающие применяемые в центральных процессорах. Как и центральные процессоры, графические процессоры характеризуются внутренней архитектурой, рабочей частотой графического ядра, технологическими нормами, по которым изготовлена микросхема. Тактовые частоты *GPU* ниже, чем у центрального процессора компьютера. Однако благодаря большому количеству потоковых процессоров производительность *GPU* весьма значительна, а если еще учесть, что существует возможность установки на одну машину двух графических карт, то это позволяет получить пиковую производительность до нескольких *TFloaps*.

Современные графические процессоры имеют довольно схожую архитектуру (рисунок 1), они содержат набор одинаковых вычислительных устройств – потоковых процессоров, (ПП, *Thread Processor*), работающих с общей памятью графического процессора (видеоОЗУ).

Число потоковых процессоров, а также размер видеоОЗУ может быть различным, в зависимости от модели *GPU*.

Все ПП синхронно исполняют одну и ту же команду, что позволяет отнести *GPU* к классу *SIMD* (англ. *Single instruction*, *multiple data* – одиночный поток команд, множественный поток данных, ОКМД – принцип компьютерных вычислений, позволяющий обеспечить параллелизм на уровне данных).

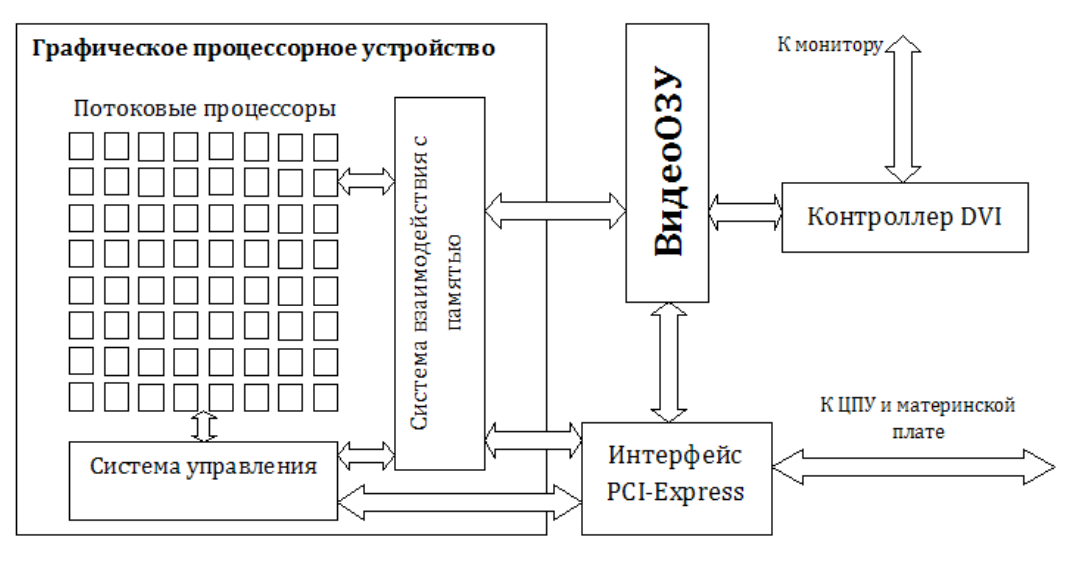


Рисунок 1 – Схема строения GPU

Основные элементы видеоадаптера: видео память (*VRAM*) и графический процессор (*GPU*).[2]

Графический процессор (*GPU*) предоставляет значительно более высокую пропускную способность инструкций и пропускную способность памяти, чем центральный процессор (*CPU*), при схожей цене и энергопотреблении. Множество приложений используют эти возможности для более быстрой работы на *GPU* по сравнению с *CPU*. Другие вычислительные устройства, такие как ПЛИС (программируемые логические интегральные схемы), также эффективны с точки зрения энергопотребления, но предлагают гораздо меньше гибкости в программировании по сравнению с *GPU*.

Эти различия в возможностях между *GPU* и *CPU* существуют потому, что они разработаны с разными целями. В то время как *CPU* спроектирован для максимально быстрого выполнения последовательности операций, называемой потоком, и может выполнить несколько десятков таких потоков параллельно, *GPU* спроектирован для выполнения тысяч таких потоков параллельно (амортизируя более медленную производительность одиночного потока для достижения большей пропускной способности).

*GPU* специализируется на высокопараллельных вычислениях и, следовательно, спроектирован так, что больше транзисторов уделяется обработке данных, а не кэшированию данных и управлению потоком.

Такая структура позволяет добиться не только параллельного выполнения алгоритма на разных мультипроцессорах, но и параллельной обработки данных в рамках одного мультипроцессора. [3]

## **1.3** **Области применения *GPU***

Мир вычислительной техники быстро меняется, и графические процессоры находятся в первых рядах этой революции. Их возможности параллельной обработки данных привели к тому, что некоторые эксперты предположили, что в некоторых задачах они могут заменить центральные процессоры.

К преимуществам использования видеокарт относится их способность параллельно обрабатывать огромные объемы данных, что делает их идеальными для таких специфических типов вычислений, как машинное обучение и искусственный интеллект. Высокая производительность и эффективность в этих областях делают их ценным активом в мире вычислений. Однако центральные процессоры более универсальны и могут решать более широкий круг задач благодаря своей более гибкой архитектуре и более быстрому переключению контекста.

В настоящее время в большинстве современных вычислительных систем *GPU* и *CPU* работают вместе. Центральные чипы выполняют задачи общего назначения, такие как запуск операционной системы и других приложений, а блоки графической обработки используются для более специализированных задач, таких как игры, рендеринг видео и научное моделирование. Однако по мере того, как вычислительные системы становятся все более сложными и мощными, вероятно, оба процессора будут использоваться все более сложным образом.

Заглядывая в будущее, становится ясно, что они будут продолжать играть важную роль в современных вычислительных системах. Хотя видеоускорители могут со временем заменить микропроцессоры для определенных типов вычислений, маловероятно, что они полностью вытеснят центральные процессоры. Напротив, более вероятно, что оба процессора будут работать вместе во все более сложных и мощных вычислительных системах.

Области применения графических процессоров:

1*GPU*-вычисления для программирования. Вычисления общего назначения на графических процессорах (*GPGPU*) произвели революцию в программировании, обеспечив более эффективный и быстрый способ обработки данных. Возможность выполнения параллельных расчётов делает идеальным выбором для задач программирования, требующих больших объемов данных и сложных вычислений. Это приводит к ускорению обработки данных и повышению производительности приложений.

Разработчики могут создавать более сложные и требовательные к данным приложения, используя для программирования видеопроцессоры. *GPU* обладают значительно большей вычислительной мощностью, чем *CPU*, что позволяет разработчикам создавать более эффективные и продвинутые приложения. Видеоускорители обладают значительно большей вычислительной мощностью, чем центральные чипы, что делает их идеальными для обработки сложных вычислений и приложений с большим объемом данных.

Существует несколько языков программирования, поддерживающих вычисления на графических картах, включая *C++, Python* и *CUDA*. Каждый язык имеет свои преимущества и недостатки, поэтому важно выбрать тот, который лучше всего подходит конкретных потребностей в программировании.

Инструменты и библиотеки для программирования на GPU предоставляют разработчикам необходимые ресурсы для использования мощности графических процессоров для решения задач программирования. К наиболее популярным инструментам и библиотекам относятся *CUDA*, *OpenCL, Numba, Taichi* и *TensorFlow*. Эти инструменты позволяют разработчикам создавать более эффективные и мощные приложения, способные обрабатывать большие объемы данных.

2 Искусственный интеллект и машинное обучение. В сфере ИИ и МО графические процессоры оказались незаменимыми благодаря своей способности обрабатывать параллельные вычисления с поразительной скоростью. Поскольку движущей силой этих областей являются глубинное обучение и нейронные сети, вычислительная мощность видеокарт сыграла важную роль в ускорении сложных вычислений, на которые в противном случае ушли бы дни или даже месяцы. Обработка естественного языка и распознавание речи также улучшаются благодаря графическим процессорам, что демонстрируют такие компании, как Google и Amazon, которые используют их для своих систем распознавания голоса.

3 Обработка изображений и компьютерная графика. *GPGPU* также добились значительных успехов в обработке изображений и компьютерной графике. Благодаря вычислительной мощности, необходимой для обработки больших объемов данных и сложных алгоритмов, видеоускорители незаменимы при обработке видео и аудио, сжатии изображений и видео, а также в компьютерном зрении. Фактически, киноиндустрия является одной из отраслей, которая получила наибольшую выгоду от вычислений на *GPU*, благодаря высококачественной и реалистичной графике для фильмов и видеоигр.

4 Научное моделирование и анализ данных. Научное моделирование и анализ данных также получили значительный подъем благодаря вычислениям на *GPU*. Вычислительная мощность, необходимая для выполнения сложных симуляций и быстрой обработки больших объемов данных, обеспечивается графическими картами. Например, моделирование вычислительной гидродинамики, используемое в таких областях, как аэрокосмическая и автомобильная техника, требует высокой степени параллельной обработки. Научные исследования в таких областях, как астрофизика, прогнозирование погоды и моделирование сворачивания белков, также выиграли от использования графических вычислений: время, необходимое для проведения моделирования, сократилось с недель или месяцев до нескольких дней или часов.

5 Медицинская визуализация. Медицинская визуализация, включая МРТ и КТ, – еще одна область, в которой широко используются вычисления на *GPU*. Вычислительная мощность, необходимая для обработки больших массивов данных и сложных алгоритмов, требуемых для медицинской визуализации, обеспечивается видеокартами. Например, блоки обработки видео могут ускорить реконструкцию *3D*-изображений из *2D*-сканов, позволяя врачам и исследователям быстрее и точнее анализировать данные. *GPGPU* также используются для разработки новых методов визуализации и повышения точности диагнозов.

6 Проектирование и дизайн. Вычисления на базе графических процессоров также широко используются в машиностроении и дизайне, включая проектирование автомобилей и структурный анализ. Видеокарты обеспечивают вычислительную мощность, необходимую для моделирования и анализа сложных систем, таких как поведение жидкостей или распределение напряжений в механической конструкции. В автомобильном дизайне они используются для моделирования аэродинамики автомобиля и оптимизации его конструкции для достижения максимальной эффективности. Структурный анализ, например, проектирование мостов и зданий, также выигрывает от вычислений на базе *GPU*, поскольку они позволяют моделировать поведение конструкции при различных нагрузках и условиях.

7 Разработка и открытие лекарств. Вычисления на *GPU* все чаще используются при проектировании и поиске лекарств. Моделирование молекулярной динамики и виртуальный скрининг требуют вычислительной мощности графических процессоров для моделирования поведения молекул и анализа больших наборов данных о потенциальных лекарственных препаратах. Исследователи могут быстро просеивать большое количество соединений и определять возможных кандидатов для дальнейшего изучения с помощью визуальных процессоров. *GPU*-вычисления также используются для моделирования поведения белков, что очень важно для разработки и открытия лекарств.

*8 GPU* в облачных вычислениях. Включение видеокарт в облачные вычисления – одно из самых многообещающих достижений в этой области. Эта мощная технология способна революционизировать наше представление о вычислениях, предлагая клиентам более быстрые и эффективные возможности, чем когда-либо прежде.

Одним из ключевых преимуществ использования графических процессоров в облачных вычислениях является их оптимизированная мощность параллельной обработки. Это делает их идеальным решением для задач, требующих больших вычислительных затрат, таких как машинное обучение, анализ данных и научное моделирование. Благодаря использованию экземпляров визуальных сопроцессоров клиенты могут получать более быстрые результаты и более высокую производительность даже при работе с большими массивами данных.

Но это еще не все. *GPU* также помогают снизить затраты за счет повышения эффективности использования ресурсов. Рабочие нагрузки, для которых в противном случае потребовалось бы несколько *CPU*, могут быть объединены в один чип видеоускорителя, что приводит к снижению затрат и повышению эффективности. А возможность арендовать экземпляры *GPU* по мере необходимости позволяет заказчикам увеличивать или уменьшать масштаб по мере изменения потребностей в вычислениях, что упрощает управление расходами и оптимизацию ресурсов.

Более того, графические процессоры в облачных вычислениях могут оказывать положительное влияние на окружающую среду. При выполнении определенных типов вычислений они более энергоэффективны, чем центральные процессоры, а это значит, что использование видеоускорителей может снизить общее энергопотребление центра обработки данных. Это беспроигрышная ситуация, поскольку клиенты выигрывают от повышения производительности и одновременно сокращают выбросы углекислого газа.

И последнее, но не менее важное: интеграция видеокарт в облачные вычисления открывает новые возможности для приложений и сценариев использования, которые были невозможны при традиционных вычислениях на базе центрального процессора. Например, рендеринг и транскодирование видео с графическим ускорением могут обеспечить более быструю и эффективную обработку мультимедиа, а базы данных с *GPU*-ускорением могут позволить проводить анализ больших массивов данных в режиме реального времени. [4]

Таким образом, видеокарта, также известная как графический процессор (*GPU*), отвечает за вычисление изображений в компьютере, которые затем могут быть отображены на мониторе. Она представляет собой интерфейс между вычислениями процессора и монитором. Однако развитие графических карт настолько продвинулось, что помимо этой функции они также могут поддерживать и разгружать ЦП во время вычислений и используются во многих сферах информационных технологий.[5]

# **2 ПЛАТФОРМА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ**

## **2.1 Операционная система**

*Linux* (или *GNU*/*Linux*) – семейство бесплатных *Unix*-подобных многопользовательских операционных систем, основанных на ядре *Linux* и на программном обеспечении *GNU*. Широкое, в том числе коммерческое, распространение стало возможным в 1992 году благодаря лицензированию ядра *Linux* по свободной лицензии *GPL*. Одним из инициаторов *Linux* был финский программист Линус Торвальдс. Он по-прежнему играет координирующую роль в дальнейшей разработке ядра *Linux* и известен как «Великодушный пожизненный диктатор».

Модульная операционная система дорабатывается разработчиками программного обеспечения по всему миру. В разработке участвуют компании, некоммерческие организации и множество волонтёров. При использовании на компьютерах обычно используются так называемые дистрибутивы *Linux*. Дистрибутив объединяет ядро *Linux* с различным программным обеспечением в операционную систему, подходящую для конечного пользователя. Многие распространители и опытные пользователи адаптируют ядро под свои нужды.

*Linux* широко и разнообразно используется, например, на рабочих станциях, серверах, мобильных телефонах, маршрутизаторах, ноутбуках, встроенных системах, мультимедийных терминалах и суперкомпьютерах. Система *Linux* прочно обосновалась на рынке серверов, а также в мобильном секторе, и в то время играет небольшую, но растущую роль на рынке настольных компьютеров и ноутбуков. *Linux* используется многочисленными пользователями, включая частных пользователей, правительства, организации и предприятия.[6]

Ядро *Linux* представляет собой монолитное ядро, написанное на языке программирования *C* с использованием некоторых расширений *GNU-C*. Однако важные подпрограммы и критичные модули программируются на языке ассемблера для конкретного процессора. Ядро позволяет использовать только драйверы, необходимые для соответствующего оборудования. Кроме того, ядро также берёт на себя выделение процессорного времени и ресурсов для отдельных программ. С технической точки зрения, дизайн *Linux* сильно основан на модели *Unix*.[7]

Ядро *Linux* было перенесено на очень большое количество аппаратных архитектур. Их репертуар варьируется от довольно экзотических операционных сред, таких как карманный компьютер *iPAQ*, навигационные устройства от *TomTom* или даже цифровые камеры, до мейнфреймов, таких как *IBM* *System* *z,* а с некоторых пор также мобильных телефонов, таких как *Motorola* *A780*, и смартфонов с операционными системами, такими как *Android* или *Sailfish*. Несмотря на модульную концепцию, монолитная базовая архитектура ядра сохраняется. Ориентация оригинальной версии на широко распространенные персональные компьютеры с процессором *x86* позволила обеспечить поддержку широкого спектра оборудования и доверить работу с драйверами даже неопытным программистам.

Все версии ядра *Linux* заархивированы на *kernel*.*org*. Версия, которую можно найти там, гарантированно является соответствующим эталонным ядром. На этом факте основаны так называемые дистрибутивные ядра, а дополнительные функции добавляются отдельными дистрибутивами *Linux*. Особенностью является схема нумерации версий, состоящая из четырех цифр, разделенных точками, например 2.6.14.1. Такая нумерация предоставляет информацию о точной версии и, таким образом, о возможностях соответствующего ядра. Из четырех чисел последнее меняется при исправлении ошибок и оптимизации кода, но не при введении новых функций или других серьезных изменениях. По этой причине его редко упоминают, например, при сравнении версий ядра. Предпоследнее, третье число меняется по мере добавления новых функций. То же самое относится к первым двум номерам, но для них изменения новые функции должны быть более радикальными. Начиная с версии 3.0 (август 2011 года) второе число опускается.

Несмотря на большую безопасность по сравнению с самой распространенной операционной системой *Windows*, возможность параллельной установки и большой выбор бесплатного программного обеспечения, Linux лишь изредка используется на настольных компьютерах. Хотя интерфейс наиболее популярных «сборок» *Linux* выглядит аналогично *Windows* или *macOS*, они отличаются различными системными функциями. Поэтому неопытному пользователю может потребоваться определённый период обучения.

Благодаря совместимости с другими *Unix*-подобными системами *Linux* особенно быстро зарекомендовал себя на рынке серверов. Поскольку множество часто используемых и необходимых серверных программ, таких как веб-серверы, серверы баз данных и групповое программное обеспечение, были доступны для *Linux* на раннем этапе, бесплатно и в основном без ограничений, доля рынка неуклонно росла. *Linux* считается стабильным и простым в обслуживании, он также отвечает особым требованиям, предъявляемым к серверной операционной системе. Модульная структура системы *Linux* также позволяет использовать компактные выделенные серверы. Кроме того, перенос Linux на самые разные аппаратные компоненты привел к тому, что *Linux* поддерживает все известные серверные архитектуры. В январе 2017 года не менее 34 % всех веб-сайтов были доступны с использованием сервера *Linux*. Поскольку не все серверы *Linux* идентифицируют себя как таковые, фактическая доля может быть значительно выше, до 65 %.[8]

Существуют специально оптимизированные дистрибутивы *Linux* для смартфонов и планшетов. В дополнение к функциям телефонии и *SMS* они предлагают различные функции *PIM*, навигации и мультимедиа. Работа обычно осуществляется с помощью мультитач или с помощью пера. *Android* также рассматривается как дистрибутив Linux, имея с ним много общего. С конца 2010 года системы *Linux* захватили лидерство на быстрорастущем рынке смартфонов, и в настоящее время их рыночная доля превышает 80 %.

Среди других областей применения Linux: автомобильные бортовые компьютерные системы и суперкомпьютеры.

Несмотря на большую безопасность по сравнению с самой распространенной операционной системой *Windows*, возможность параллельной установки и большой выбор бесплатного программного обеспечения, *Linux* лишь изредка используется на настольных компьютерах. Хотя интерфейс наиболее популярных «сборок» *Linux* выглядит аналогично *Windows* или *macOS*, они отличаются различными системными функциями. Поэтому неопытному пользователю может потребоваться определённый период обучения.[9]

## **2.2 Язык программирования**

*Python* – это высокоуровневый язык программирования, отличающийся эффективностью, простотой и универсальностью использования. Он широко применяется в разработке веб-приложений и прикладного программного обеспечения, а также в машинном обучении и обработке больших данных. За счет простого и интуитивно понятного синтаксиса является одним из распространенных языков для обучения программированию.

Язык программирования *Python* был создан в 1989 – 1991 годах голландским программистом Гвидо ван Россумом. Изначально это был любительский проект: разработчик начал работу над ним, просто чтобы занять себя на рождественских каникулах. Хотя сама идея создания нового языка появилась у него двумя годами ранее. Имя ему Гвидо взял из своей любимой развлекательной передачи «Летающий цирк Монти Пайтона». Язык программирования он и выбрал – *Python*, что это означало название комик-группы. Это шоу было весьма популярным среди программистов, которые находили в нем параллели с миром компьютерных технологий.[10]

История развития *Python* включает несколько этапов, каждый из которых заканчивался выходом новой версии.

В 1991 году Гвидо опубликовал первую версию (0.9.0) языка, включающую базовые возможности – в частности, работу с данными различных типов и корректировку ошибок.

Через три года вышла версия 1.0, в которой функционал был дополнен обработкой списков данных: систематизацией, фильтрацией, сокращением, сопоставлением.

Версия 2.0 была опубликована в 2000 году и отличалась исправленными недочетами прежних версий, а также новыми полезными функциями для программистов — в частности, поддержкой *Unicode* и облегченной методикой циклического просмотра списка.

В 2008 году представлена версия *Python 3*, включившая возможность печати, поддержку деления чисел и расширенное исправление ошибок.

Язык программирования «Питон», начавшийся как проект одного человека, сегодня развивается и поддерживается командой разработчиков. В 2001 году они объединились в некоммерческую организацию *Python* *Software* *Foundation*, целями которой стали популяризация и совершенствование языка в сообществе программистов, контроль над интеллектуальными правами, проведение тематических конференций и т.д.[11]

Уже в 2003 года *Python* входит в десятку самых популярных языков программирования по версии *TIOBE* *Programming* *Community* *Index*. По состоянию на сентябрь 2015 года он занимает пятую позицию. В 2007 и 2010 годах он был признан языком программирования года. Это третий по популярности язык, грамматический синтаксис которого не базируется преимущественно на языке *C*, как в случае с *C++* и *Objective*-*C*. К сведению, *C#* и *Java* имеют лишь частичное синтаксическое сходство с *C*, например, использование фигурных скобок, и больше похожи друг на друга, чем на *C*.

Практика показывает, что скриптовые языки, такие как Python, более производительны, чем традиционные (*C* и *Java*), при решении таких задач программирования, как работа со строками и поиск в словарях. При этом объём используемой памяти зачастую меньше, чем в *Java*, и не намного больше, чем в *C* и *C++*.

К числу крупных организаций, использующих Python, относятся [*Google*](https://znanierussia.ru/articles/Google_(компания)), *Yahoo*!, ЦЕРН, НАСА.

Python может использоваться в качестве языка сценариев для веб-приложений, например, через *mod\_wsg*i для *Apache*. С помощью интерфейса *Web* *Server* *Gateway* *Interface* стандартный интерфейс прикладного программирования расширяется для обслуживания таких приложений. Программные фреймворки для веб-приложений, такие как *Django*, *Pylons*, *Pyramid*, *TurboGears*, *web2py*, *Tornado*, *Flask*, *Bottle* и *Zope*, помогают разработчикам создавать и поддерживать более сложные приложения. *Pyjamas* и *IronPython* могут быть использованы при программировании приложений на основе Ajax на стороне клиента. *SQLAlchemy* может быть использован для сопоставления данных из аналогичных баз данных.

*Twisted* — программный фреймворк для программирования взаимодействия между компьютерами, используется в *Dropbox*.

Такие библиотеки, как *NumPy*, *SciPy* и *Matplotlib*, позволяют эффективно использовать *Python* для научных вычислений. *Sage* — математическое программное обеспечение, написанное преимущественно на языке *Python* и охватывающее многие аспекты математики, включая алгебру, комбинаторику, теорию чисел и многое другое.

*Python* успешно применяется в качестве скриптового языка в ряде программных продуктов, в том числе в программах, использующих метод конечных элементов, таких как *Abaqus*, *FreeCad*, программах и инструментах *3D*-анимации, таких как *3ds* *Max*, *Blender*, *Cinema* *4D*, *Lightwave*, программах обработки *2D*-изображений, таких как *GIMP*, *Inkscape*, *Scribus* и *Paint* *Shop* *Pro*. *GNU* *Debugger* (*GDB*) использует *Python* для визуализации сложных структур, таких как контейнеры элементов *C*++*.* *ESRI* представляет *Python* как лучший выбор языка сценариев для *ArcGIS*. Он также используется в видеоиграх и является одним из трех возможных языков программирования, принимаемых *Google* *App* *Engine*, двумя другими являются *Java* и [*Go*](https://znanierussia.ru/articles/Go).[12]

Как скриптовый язык с модульной архитектурой, простым синтаксисом и богатым набором средств обработки текста, *Python* используется в проектах по искусственному интеллекту (ИИ) и вычислениям на человеческом языке.

*Python* также широко используется в области информационной безопасности.

Большая часть программного обеспечения *Sugar* для проекта *One* *Laptop* *per* *Child* *XO*, разрабатываемого в *Sugar* *Labs*, написана на *Python*. Он также является основным языком программирования в проекте *Raspberry* *Pi*, разрабатывающем одноплатные компьютеры. *LibreOffice* включает в себя *Python* и намерен заменить им *Java*.[13]

Таким образом, *Python* - универсальный и простой язык программирования, применяемый в веб-разработке, машинном обучении и обработке данных. Созданный в 1989 году Гвидо ван Россумом, он прошел через несколько этапов развития, выходя на версии с улучшенным функционалом и исправленными ошибками. *Python* используется в различных областях, включая науку, разработку веб-приложений, научные вычисления и информационную безопасность. Популярность и удобство использования Python сделали его выбором для многих компаний, включая *Google*, *NASA* и *ESRI*. В данном курсовом проекте он будет использоваться в сочетании с *Linux,* который является многопользовательской ОС, основанной на ядре *Linux* и программном обеспечении GNU. Благодаря лицензированию ядра по свободной лицензии, начиная с 1992 года, *Linux* получил широкое распространение, используется на различных устройствах: от серверов до мобильных устройств и суперкомпьютеров. Он остается стабильным и удобным для серверов, с около 34% всех веб-сайтов, возможно, использующих сервера *Linux*, а фактическая доля может быть и выше - до 65%.

# **3 ТЕОРЕТИЧЕСКОЕ ОБОСНОВАНИЕ РАЗРАБОТКИ ПРОГРАММНОГО ПРОДУКТА**

## **3.1 Обоснование необходимости анализа**

Исследование использования графических процессоров (*GPU*) для оптимизации времени выполнения кода представляет собой важное направление в современной вычислительной технологии. Оно имеет свои веские основания.

1 *GPU* спроектированы с учетом параллельной обработки данных. Эти устройства обладают множеством вычислительных ядер, способных одновременно выполнять задачи. Это позволяет существенно ускорить выполнение вычислительно интенсивных операций.

*2 GPU* представляют собой дополнительные ресурсы в компьютерных системах. Их использование может оптимизировать распределение задач между центральным процессором (*CPU*) и графическим процессором. Эффективное использование ресурсов повышает производительность системы в целом.

3 Решение задач с большим объемом данных. Многие области, от обработки изображений и видео до научных вычислений и машинного обучения, требуют обработки огромных объемов данных. *GPU* способны эффективно справляться с этой задачей, что расширяет возможности исследования и разработки.

4 Сокращение времени ожидания. Ускорение выполнения кода на *GPU* может существенно уменьшить задержки в получении результатов. Это особенно важно в интерактивных приложениях, где скорость обработки данных играет критическую роль.

5 Экономия ресурсов. Использование *GPU* может сэкономить энергию и ресурсы компьютера, так как графические процессоры более эффективно обрабатывают параллельные задачи.

6 *GPU* для оптимизации времени выполнения кода способствует развитию этой технологии. Это означает создание более эффективных и быстрых приложений, что вносит вклад в развитие информационных технологий.

Использование *GPU* для оптимизации времени выполнения кода не только актуально, но и представляет собой поле исследований, которое может значительно улучшить производительность и возможности в различных областях вычислительных наук.

В данной работе поставлена цель проанализировать время работы кода с использованием *GPU*, без *GPU*, с использованием других оптимизационных методов. В данном курсовом проекте рассматриваются 3 типовые задачи: поиск простых чисел меньше заданного числа, нахождение наибольшей подпоследовательности из 2 последовательностей, перемножение матриц. Анализ основывается на времени выполнения этих задач с разной размерностью.

## **3.2****Перемножение матриц**

Матрица – это прямоугольная таблица, состоящая из строк и столбцов, содержащих числа.

Результатом перемножения матриц будет такая матрица, что элемент матрицы, стоящий в *i*-той строке и *j*-том столбце, равен сумме произведений элементов *i*-той строки матрицы *A* на соответствующие элементы *j*-того столбца матрицы *B*.

Следует отметить, что две матрицы можно перемножить между собой тогда и только тогда, когда количество столбцов первой матрицы равно количеству строк второй матрицы [14].

Операция перемножения матриц является фундаментальной в линейной алгебре и находит широкое применение в различных областях информационных технологий.

Эта операция используется в графике и компьютерных играх для визуализации трехмерных объектов и преобразования координат, в машинном обучении для вычисления параметров моделей и анализа данных, а также в вычислительных приложениях благодаря высокой параллелизуемости и возможности оптимизации для многопоточных устройств.

Скорость выполнения операции перемножения матриц имеет критическое значение в информационных технологиях. Эффективное и быстрое выполнение этой операции ускоряет обучение моделей машинного обучения, обеспечивает плавную визуализацию в графических приложениях и улучшает обработку потоковых данных в реальном времени.

## **3.3****Поиск всех простых чисел меньше заданного числа**

Поиск простых чисел представляет собой процесс, направленный на определение чисел, которые делятся только на единицу и самих себя. Одним из наиболее известных методов является "Решето Эратосфена", основанный на итеративном отсеивании кратных значений начиная с двойки и завершающийся на заданном пределе. Оставшиеся числа после этого процесса считаются простыми.

Простые числа используются в криптографии, где они служат основой для создания криптографических ключей в алгоритмах шифрования. В системах шифрования *RSA* они используются для создания ключей, обеспечивая надежную защиту данных.[15]

В математических исследованиях простые числа играют важную роль в алгоритмах и задачах, таких как факторизация больших чисел или анализ числовых последовательностей.

Простые числа также используются в оптимизации алгоритмов. Некоторые алгоритмы и структуры данных используют простые числа для улучшения производительности. Например, в хешировании для равномерного распределения данных применяются простые числа.

Скорость выполнения операции поиска простых чисел имеет ключевое значение в информационных технологиях по нескольким причинам:

1 Безопасность. Криптографические системы основаны на сложности факторизации больших простых чисел. Быстрый поиск простых чисел способствует созданию надежных ключей для шифрования данных.

2 Эффективность алгоритмов. Использование простых чисел в алгоритмах и структурах данных подразумевает высокую производительность. Быстрый поиск оптимизирует работу алгоритмов, зависящих от простых чисел.

3 Ускорение вычислений. Поиск простых чисел может быть частью сложных вычислительных задач, и оптимизация этого процесса улучшает общую скорость выполнения.

## **3.4****Поиск НОП из 2 последовательностей**

Поиск наибольшей общей подпоследовательности (НОП) представляет собой задачу нахождения самой длинной последовательности элементов, которая является подпоследовательностью как минимум в двух данных последовательностях. Важно отметить, что эти элементы не обязательно должны следовать друг за другом в исходных последовательностях, но они должны сохранять свой порядок.

Поиск НОП находит применение в таких областях как строковая обработка. Данная операция является важной задачей в обработке строк, особенно в анализе текста и поиске сходства между последовательностями символов или слов.

Данная операция также применяется в биоинформатике. В молекулярной биологии НОП используется для сравнения геномов, выявления общих областей между ДНК или белковыми последовательностями, что помогает в изучении структуры и функции молекул.

В обработке изображений НОП применяется для сопоставления и выявления схожих областей между изображениями, таких как шаблоны, формы или объекты.

Скорость выполнения операции поиска НОП имеет важное значение в информационных технологиях по нескольким причинам:

1 Эффективность алгоритмов. Быстрые и эффективные алгоритмы поиска НОП повышают производительность систем, основанных на сопоставлении данных, таких как поисковые системы или приложения обработки изображений.

2 Анализ данных. Высокая скорость поиска НОП позволяет быстро обрабатывать большие объемы данных, что важно для машинного обучения, анализа данных и других приложений, требующих вычислительных ресурсов.

3 Оптимизация вычислений. В контексте вычислительных задач, требующих сопоставления больших наборов данных, оптимизация поиска НОП помогает ускорить выполнение алгоритмов и улучшить их эффективность.[16]

## **3.5*****Taichi***

*Python* стал самым популярным языком во многих быстроразвивающихся областях, таких, как глубокое обучение и различные направления анализа и обработки данных. Но при этом за удобство работы с *Python*-кодом, за высокий уровень его читабельности, приходится платить производительностью. Конечно, все мы время от времени жалуемся на скорость работы программ, и *Python*, безусловно, не стоит винить во всех грехах. Несмотря на это, справедливым будет заявление о том, что природа *Python*, интерпретируемого языка, не способствует высокой производительности кода, особенно когда речь идёт о «тяжёлых» вычислениях (один из признаков таких вычислений – наличие в программе нескольких вложенных циклов).

Тайчи (*Taichi*) – это многоцелевой фреймворк для вычислительного программирования, созданный для эффективной работы с современными аппаратными ресурсами, включая многозадачные и многопроцессорные системы, а также графические процессоры (*GPU*). Этот фреймворк обеспечивает высокую производительность и параллелизм, что делает его подходящим для обработки вычислительно интенсивных задач.

Основные особенности *Taichi* включают в себя поддержку *GPU* для ускорения вычислений, интеграцию с *Python* для удобства программирования и динамическое определение вычислительных операций в процессе выполнения программы. Этот фреймворк предоставляет мощные инструменты для создания вычислительных ядер и может быть использован в различных областях, включая физическое моделирование, графическое программирование, машинное обучение и научные вычисления.

*Taichi* разработан с целью упростить создание высокопроизводительного программного обеспечения и обеспечить эффективное использование вычислительных ресурсов для различных научных и инженерных задач.[17]

Концепция декомпозиции дополняется также в системе типов. Поскольку ёмкость и пропускная способность памяти *GPU* становятся основными узкими местами сегодня, важно уметь упаковывать больше данных в одну единицу памяти. С 2021 года *Taichi* представил настраиваемые квантованные типы, позволяющие определять фиксированные точки или числа с плавающей запятой с произвольным числом битов (по-прежнему должно быть менее 64). Это позволило провести симуляцию *MPM* более чем 400 миллионов частиц на одном устройстве *GPU*.[18]

## **3.6 *Numba***

*Numba* поддерживает программирование для *CUDA* *GPU*, преобразуя ограниченное подмножество кода Python непосредственно в ядра *CUDA* и функции устройства, следуя модели выполнения *CUDA*. Ядра, написанные в *Numba* имеют прямой доступ к массивам *NumPy*. Массивы *NumPy* автоматически передаются между *CPU* и *GPU*.

*CUDA* имеет модель выполнения, отличную от традиционной последовательной модели, используемой для программирования *CPU*. В *CUDA* код, который вы пишете, будет выполнен несколькими потоками одновременно (часто сотнями или тысячами). Ваше решение будет моделироваться путем определения иерархии потоков, сеток и блоков.

Поддержка *CUDA* в *Numba* предоставляет средства для объявления и управления этой иерархией потоков. Эти средства во многом аналогичны тем, которые предоставляются языком *CUDA* C от *Nvidia*.

*Numba* также предоставляет три вида памяти *GPU*: глобальную память устройства (большая, относительно медленная внешняя память, подключенная к самому *GPU*), общую память на чипе и локальную память. Для всех, кроме самых простых алгоритмов, важно тщательно рассмотреть, как использовать и получать доступ к памяти, чтобы минимизировать требования к пропускной способности и конфликты. [19]

Таким образом, была обоснована необходимость анализа, а также рассмотрены библиотеки, которые позволят использовать *GPU* инструкции для оптимизации времени выполнения кода программ.

# **4 ПРОЕКТИРОВАНИЕ ФУНКЦИОНАЛЬНЫХ ВОЗМОЖНОСТЕЙ ПРОГРАММЫ**

## **4.1 Функции программного обеспечения**

**4.1.1** Поиск всех простых чисел меньше заданного числа. Программа выполняет поиск всех простых чисел до заданного числа *n* с использованием алгоритма «Решето Эратосфена» на *CPU* и *GPU* с использованием библиотеки *Numba*.

Программа начинается с определения класса ResearchData, который создает объект для хранения времени выполнения и результатов исследования. Далее следует реализация функций \_*sieve*\_forward и \_sieve\_forward\_cpu, осуществляющих процесс нахождения простых чисел с использованием алгоритма "Решето Эратосфена" на *GPU* и *CPU* соответственно.

Функция count\_*primes* запускает оба метода (*GPU* и *CPU*) для нахождения простых чисел до заданного предела n. Она измеряет время выполнения каждого метода и собирает результаты в объекты *ResearchData* для последующего анализа. После этого идет тестирование на разных размерах входных данных, где результаты каждого метода сравниваются между собой. Если результаты не совпадают, выводится сообщение об ошибке.

Кроме того, скрипт создает графики, отображающие время выполнения каждого метода (*GPU* и *CPU*) в зависимости от размера входных данных. Это позволяет визуально оценить производительность алгоритма на обеих платформах и выявить разницу во времени выполнения между ними.

**4.1.2** Умножение матриц. В коде используются библиотеки для работы с массивами на *GPU* (*CuPy*), для параллельных вычислений на *GPU* (*Numba*), а также *NumPy* для работы с массивами на *CPU*. Также используются библиотеки для измерения времени выполнения (*time*) и построения графиков (*matplotlib*.*pyplot*).

В коде определены функции для умножения матриц на *CPU*, с использованием *NumPy* и на *GPU* через *Numba*. Функция *start*\_mult инициализирует матрицы случайными значениями и запускает умножение для каждого из методов, замеряя время выполнения. Есть возможность использовать только методы *CPU* и *NumPy* или все три метода одновременно, в зависимости от параметра *need*\_*pure*\_*python*.

После измерения времени выполнения для разных размеров матриц скрипт создает файлы с результатами и построенные графики времени выполнения для каждого метода в зависимости от размера матрицы. Это позволяет визуально оценить производительность каждого метода и сравнить их эффективность при умножении матриц различных размеров.

**4.1.3** Поиск наибольшей подпоследовательности из 2 последовательностей. Определяются две функции для вычисления наибольшей общей подпоследовательности (*Longest* *Common* *Subsequence*, *LCS*).

Этот скрипт на Python предназначен для нахождения длины наибольшей общей подпоследовательности (*LCS*) двух последовательностей. Он использует алгоритм динамического программирования для реализации этой задачи на *CPU* и на *GPU* с использованием фреймворка *Taichi*. Задача состоит в поиске наибольшей общей подпоследовательности чисел от 0 до 99 случайной длины.

В коде определены три функции: *compute\_lcs\_gpu* для вычисления *LCS* на GPU с использованием *Taichi*, *compute\_lcs\_cpu* для вычисления на *CPU* с использованием библиотеки *NumPy* и *start*\_*computing* для запуска вычислений и замера времени выполнения для обоих методов.

Программа генерирует случайные последовательности целых чисел от 0 до 99 заданной длины и затем находит их *LCS*, замеряя время выполнения операций на *GPU* и на *CPU*. Для каждой длины последовательности проводится несколько тестов (заданное число *tests*\_*number*), чтобы получить средние значения времени выполнения.

Результаты тестов записываются в файл *finding*\_*LCS*\_*results*.*txt*, содержащий средние времена выполнения для каждого метода в зависимости от длины последовательности. Кроме того, строятся графики производительности методов (*GPU* и *CPU*) относительно размера входных данных. Это позволяет визуально оценить производительность алгоритмов на различных объемах данных.

Код программ, реализующие функции, описанные выше, представлен в приложении А, блок-схема программ представлены в приложении В. На функциональной схеме, которая представлена в приложении Б, изложено описание основных компонентов и их взаимодействия в разрабатываемых программах.

Таким образом, были перечислены функциональные возможности разрабатываемых программ в рамках курсового проекта.

# **5 АНАЛИЗ ОПТИМИЗАЦИИ ВРЕМЕНИ ВЫПОЛНЕНИЯ КОДА С ПОМОЩЬЮ *GPU***

## **5.1 Анализ полученных результатов**

**5.1.1** Поиск всех простых чисел меньше заданного числа.

Параметры тестирования:

1 Размер числа: 1000, 10000, 20000, 100000, 200000, 1000000, 2000000, 10000000, 20000000.

2 Тип данных: целочисленные (*int32*).

3 Аппаратное обеспечение: *NVIDIA* *GeForce* *940MX*, *Intel* *Core i3-7100U* (4 ядра), 8 ГБ *RAM*.

Таблица 5.1 – Результаты для поиска простых чисел (секунды) на *GPU*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Размер числа | Номер запуска | | | | | |
| 1 | 2 | 3 | 4 | 5 | Среднее |
| 1000 | 0.00666 | 0.00127 | 0.00118 | 0.00081 | 0.00136 | 0.00225 |
| 10000 | 0.0015 | 0.00121 | 0.00102 | 0.00088 | 0.00088 | 0.0011 |
| 20000 | 0.00132 | 0.00129 | 0.00128 | 0.00131 | 0.00131 | 0.0013 |
| 100000 | 0.00479 | 0.00483 | 0.0048 | 0.00486 | 0.00483 | 0.00482 |
| 200000 | 0.00921 | 0.00945 | 0.00918 | 0.00944 | 0.00922 | 0.0093 |
| 1000000 | 0.05026 | 0.04948 | 0.04965 | 0.04961 | 0.04985 | 0.04977 |
| 2000000 | 0.10146 | 0.09633 | 0.09584 | 0.09721 | 0.09582 | 0.09733 |
| 10000000 | 0.49377 | 0.49229 | 0.49766 | 0.49283 | 0.49857 | 0.49502 |
| 20000000 | 0.99921 | 0.99898 | 0.9887 | 1.00056 | 0.99759 | 0.99701 |

Таблица 5.2 – Результаты для поиска простых чисел (секунды) на *CPU*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Размер числа | Номер запуска | | | | | |
| 1 | 2 | 3 | 4 | 5 | Среднее |
| 1000 | 0.00826 | 0.00507 | 0.00574 | 0.00515 | 0.00666 | 0.00617 |
| 10000 | 0.01573 | 0.01932 | 0.01526 | 0.01413 | 0.01349 | 0.01558 |
| 20000 | 0.02022 | 0.02022 | 0.02026 | 0.02018 | 0.02013 | 0.0202 |
| 100000 | 0.06905 | 0.06095 | 0.06538 | 0.05939 | 0.05862 | 0.06268 |
| 200000 | 0.09761 | 0.09667 | 0.10238 | 0.10216 | 0.09808 | 0.09938 |
| 1000000 | 0.363 | 0.36516 | 0.36123 | 0.36084 | 0.43645 | 0.37733 |
| 2000000 | 0.66579 | 0.67584 | 0.68541 | 0.66054 | 0.6614 | 0.6698 |
| 10000000 | 2.95894 | 2.9732 | 2.95719 | 2.91245 | 2.92208 | 2.94477 |
| 20000000 | 5.66602 | 5.82502 | 5.70319 | 5.69336 | 5.69278 | 5.71607 |

При нахождении всех простых чисел меньших заданного числа *n* использовался алгоритм, реализующий так называемое «Решето Эратосфена». Вместо того, чтобы проверять каждое число на простоту, на ноль заменяются все числа не являющиеся простыми. Хоть алгоритм и так является относительно быстрым по сравнению с «наивным» решением, полученные результаты (таблицы 5.1 – 5.2) показывают, что с увеличением размера *n* время выполнения на *GPU* растет медленно (алгоритм с поддержкой *GPU* работает примерно в 6 раз быстрее), что указывает на высокую параллельную производительность при обработке больших данных.

Вместо обычного цикла, который проходит по всем элементам и изменяет исходную матрицу, исключая непростые числа, требуется описать только функцию, определяющую поведение при каждой итерации. Из-за декоратора *@jit.cuda* и специально переданных аргументов *[grid\_blocks, BLOCK\_SIZE]* функция будет вызываться для каждого потока в сетке потоков в *GPU*, и в итоге «итерации» будут проходить параллельно, уменьшая время выполнения.

График сравнения составлен из средних значений из 5 тестовых замеров для каждого тестового случая, и находится в приложении А на рисунке 1.

**5.1.2** Умножение матриц.

Параметры тестирования:

1 Размер матриц: 10x10, 25x25, 50x50, 75x75, 100x100, 150x150, 200x200, 300x300, 400x400.

2 Тип данных: числа с плавающей точкой (*float64*).

3 Аппаратное обеспечение: *NVIDIA* *GeForce* *940MX*, *Intel* *Core* *i3-7100U* (4 ядра), 8 ГБ *RAM*.

Таблица 5.3 – Результаты перемножения матриц (секунды) на чистом *Python*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Размер матрицы | Номер запуска | | | | | |
| 1 | 2 | 3 | 4 | 5 | Среднее |
| 10 | 0.00111 | 0.00114 | 0.00114 | 0.00113 | 0.00116 | 0.00114 |
| 25 | 0.02162 | 0.02088 | 0.01855 | 0.01738 | 0.0168 | 0.01905 |
| 50 | 0.15169 | 0.13466 | 0.13506 | 0.13478 | 0.13502 | 0.13824 |
| 75 | 0.53776 | 0.52435 | 0.53508 | 0.60249 | 0.53427 | 0.54679 |
| 100 | 1.15114 | 1.1303 | 1.13897 | 1.14126 | 1.13952 | 1.14024 |
| 150 | 3.6983 | 3.62058 | 3.59545 | 3.72677 | 3.6668 | 3.66158 |
| 200 | 8.79253 | 8.85763 | 8.60142 | 8.65253 | 8.54254 | 8.68933 |
| 300 | 29.00595 | 28.79546 | 28.51615 | 28.56061 | 28.73563 | 28.72276 |
| 400 | 68.54287 | 70.69293 | 68.42978 | 67.93258 | 67.65303 | 68.65024 |

Таблица 5.4 – Результаты перемножения матриц (секунды) на *Numpy*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Размер матрицы | Номер запуска | | | | | |
| 1 | 2 | 3 | 4 | 5 | Среднее |
| 10 | 0.00029 | 1.78814e-05 | 1.54972e-05 | 1.3113e-05 | 1.33514e-05 | 7.06673e-05 |
| 25 | 3.69549e-05 | 3.09944e-05 | 2.3365e-05 | 2.28882e-05 | 2.40803e-05 | 2.76566e-05 |

Продолжение таблицы 5.4

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Размер матрицы | Номер запуска | | | | | |
| 1 | 2 | 3 | 4 | 5 | Среднее |
| 50 | 4.19617e-05 | 3.93391e-05 | 3.91006e-05 | 3.8147e-05 | 4.05312e-05 | 3.98159e-05 |
| 75 | 0.00172 | 8.86917e-05 | 0.00012 | 0.00012 | 0.00013 | 0.00043 |
| 100 | 0.00014 | 0.00015 | 0.00015 | 0.00014 | 0.00016 | 0.00015 |
| 150 | 0.00028 | 0.00023 | 0.00025 | 0.00025 | 0.00025 | 0.00025 |
| 200 | 0.00066 | 0.00044 | 0.00044 | 0.00046 | 0.00045 | 0.00049 |
| 300 | 0.00166 | 0.00163 | 0.00121 | 0.00119 | 0.00119 | 0.00138 |
| 400 | 0.00322 | 0.00243 | 0.0032 | 0.00242 | 0.00321 | 0.0029 |

Таблица 5.5 – Результаты перемножения матриц (секунды) *GPU*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Размер матрицы | Номер запуска | | | | | |
| 1 | 2 | 3 | 4 | 5 | Среднее |
| 10 | 0.37485 | 0.0014 | 0.00141 | 0.00121 | 0.00122 | 0.07602 |
| 25 | 0.00154 | 0.00152 | 0.00138 | 0.00139 | 0.00137 | 0.00144 |
| 50 | 0.00173 | 0.00147 | 0.00146 | 0.00143 | 0.00141 | 0.0015 |
| 75 | 0.00164 | 0.00142 | 0.00157 | 0.00139 | 0.00143 | 0.00149 |
| 100 | 0.00154 | 0.00138 | 0.0014 | 0.00139 | 0.00141 | 0.00143 |
| 150 | 0.00163 | 0.00141 | 0.00139 | 0.0014 | 0.00146 | 0.00146 |
| 200 | 0.00142 | 0.00141 | 0.00143 | 0.00163 | 0.00137 | 0.00145 |
| 300 | 0.00141 | 0.00138 | 0.00137 | 0.00137 | 0.00143 | 0.00139 |
| 400 | 0.00141 | 0.00141 | 0.00137 | 0.00142 | 0.00142 | 0.00141 |

Как видно из результатов (таблицы 5.3 – 5.5) использовать самописную функцию на «чистом» *python* если размер матриц больше размера 50 на 50 не является хорошей идеей. И как можно заметить, *numpy* и функция, оптимизированная с помощью *numba* практически идентичны, и время их работы не изменяется при увеличении размеров матриц. Может возникнуть вопрос, почему *numpy*, который работает на *CPU* показывает настолько высокие показатели скорости. Дело заключается в том, что реализация операции умножения матриц использует технологию *BLAS* (библиотека элементарных линейных алгебраических преобразований), которая обеспечивает эффективное выполнение линейных алгебраических преобразований над матрицами и векторами. А данная библиотека в свою очередь использует *SIMD* (принцип компьютерных вычислений, позволяющий обеспечить параллелизм на уровне данных).

Функция, использующая *GPU,* показала очень высокую эффективность по сравнению с функцией, написанной без использования сторонних библиотек и помощи *GPU,* и теперь имеет смысл сделать сравнение *numpy* и функции, использующей *GPU* на более высоких показателях размера матриц.

Параметры тестирования:

1 Размер матриц: 100x100, 500x500, 1000x1000, 2000x2000, 3000x3000, 4000x4000.

2 Тип данных: числа с плавающей точкой (*float64*).

3 Аппаратноеобеспечение*: NVIDIA GeForce 940MX, Intel Core i3-7100U* (4ядра)*,* 8ГБ *RAM.*

Таблица 5.6 – Результаты перемножения матриц (секунды) *Numpy*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Размер матрицы | Номер запуска | | | | | |
| 1 | 2 | 3 | 4 | 5 | Среднее |
| 100 | 0.0022 | 0.00011 | 0.0001 | 0.00013 | 0.00013 | 0.00054 |
| 500 | 0.00715 | 0.00757 | 0.01202 | 0.0051 | 0.0051 | 0.00739 |
| 1000 | 0.0384 | 0.03695 | 0.03691 | 0.03511 | 0.03511 | 0.0365 |
| 2000 | 0.29244 | 0.2929 | 0.30259 | 0.28717 | 0.2905 | 0.29312 |
| 3000 | 0.96291 | 0.96688 | 0.96351 | 0.93762 | 0.96745 | 0.95967 |
| 4000 | 2.23928 | 2.2397 | 2.26158 | 2.26307 | 2.24613 | 2.24995 |

Таблица 5.7 – Результаты перемножения матриц (секунды) *GPU*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Размер матрицы | Номер запуска | | | | | |
| 1 | 2 | 3 | 4 | 5 | Среднее |
| 100 | 0.36441 | 0.00337 | 0.00349 | 0.00323 | 0.00312 | 0.07553 |
| 500 | 0.00336 | 0.00335 | 0.00337 | 0.00339 | 0.00343 | 0.00338 |
| 1000 | 0.00341 | 0.0014 | 0.0014 | 0.00143 | 0.00142 | 0.00181 |
| 2000 | 0.00137 | 0.00137 | 0.00145 | 0.00158 | 0.00143 | 0.00144 |
| 3000 | 0.00227 | 0.00226 | 0.00226 | 0.00227 | 0.00224 | 0.00226 |
| 4000 | 0.00382 | 0.0038 | 0.00379 | 0.00389 | 0.00379 | 0.00382 |

Как видно из результатов (таблицы 5.6 – 5.7), функция, использующая *GPU,* показывает хорошие результаты по сравнению с готовой, протестированной и постоянно улучшающейся библиотекой *Numpy,* которая выполняет операции в *CPU* параллельно. Стоит также учесть то, что, используя *GPU,* система использует ресурсы видеокарты, освобождая *CPU* от большой нагрузки (использование *CPU* и *numpy* при умножении матриц большой размерности может сильно нагрузить процессор). Такие результаты достигаются за счет того, что каждый элемент результатной матрицы высчитывается в отдельном потоке в *GPU*. Это позволяет выполнять большое количество операций одновременно, что значительно ускоряет процесс умножения матриц, особенно для больших матриц.

Также стоит отметить, что использование библиотеки *CuPy* для работы с массивами на *GPU* упрощает перенос кода между *CPU* и *GPU*, что делает процесс разработки и оптимизации более удобным. Это позволяет эффективно использовать ресурсы графического процессора для выполнения вычислений и ускоряет обработку больших объемов данных.

Графики сравнения составлены из средних значений из 5 тестовых замеров для каждого тестового случая, и находится в приложении А на рисунке 2 – 3.

**5.1.3** Нахождение длины наибольшей подпоследовательности из двух последовательностей.

Параметры тестирования:

1 Размер матрицы: 10, 50, 100, 250, 500, 750, 1000, 1500, 2000.

2 Тип данных: числа с плавающей точкой (*float64*).

3 Аппаратноеобеспечение*: NVIDIA GeForce 940MX, Intel Core i3-7100U* (4ядра)*,* 8ГБ *RAM.*

Таблица 5.8 – Результаты для поиска НОП (секунды) на *CPU*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Размер числа | Номер запуска | | | | | |
| 1 | 2 | 3 | 4 | 5 | Среднее |
| 10 | 0.00018 | 0.00012 | 0.00013 | 0.00012 | 0.00013 | 0.00014 |
| 50 | 0.00264 | 0.00273 | 0.00271 | 0.00266 | 0.00264 | 0.00268 |
| 100 | 0.01045 | 0.01077 | 0.01091 | 0.01105 | 0.01571 | 0.01178 |
| 250 | 0.06635 | 0.06967 | 0.06896 | 0.06732 | 0.0714 | 0.06874 |
| 500 | 0.288 | 0.27234 | 0.27457 | 0.27041 | 0.27549 | 0.27616 |
| 750 | 0.64347 | 0.63627 | 0.63295 | 0.63662 | 0.6494 | 0.63974 |
| 1000 | 1.1454 | 1.14973 | 1.14639 | 1.11462 | 1.13959 | 1.13914 |
| 1500 | 2.5912 | 2.5315 | 2.64197 | 2.57435 | 2.5852 | 2.58485 |
| 2000 | 4.70274 | 4.83698 | 4.70803 | 4.72724 | 4.76129 | 4.74726 |

Таблица 5.9 – Результаты для поиска НОП (секунды) на *GPU*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Размер числа | Номер запуска | | | | | |
| 1 | 2 | 3 | 4 | 5 | Среднее |
| 10 | 0.1064 | 0.00063 | 0.00057 | 0.00055 | 0.00057 | 0.02174 |
| 50 | 0.00156 | 0.00156 | 0.00157 | 0.00156 | 0.00157 | 0.00157 |
| 100 | 0.00469 | 0.00473 | 0.00471 | 0.00469 | 0.00469 | 0.0047 |
| 250 | 0.02604 | 0.02604 | 0.02604 | 0.02604 | 0.02604 | 0.02621 |
| 500 | 0.10207 | 0.10261 | 0.1004 | 0.09999 | 0.09991 | 0.101 |
| 750 | 0.22308 | 0.21758 | 0.21654 | 0.21653 | 0.21756 | 0.21826 |
| 1000 | 0.38592 | 0.38407 | 0.38415 | 0.38596 | 0.38595 | 0.38521 |
| 1500 | 0.86284 | 0.867 | 0.86707 | 0.86297 | 0.86293 | 0.86456 |
| 2000 | 1.54015 | 1.54021 | 1.54006 | 1.53287 | 1.5401 | 1.53868 |

Как видно из результатов (таблицы 5.8 – 5.9), время работы функции, использующей *GPU,* значительно меньше при любых значениях *n.* Дело в том, что библиотека *NumPy* специализируется на операциях, направленных на обработку массивов, а *Taichi* поддерживает более высокий уровень «детализации», напрямую работая с элементами контейнеров данных. *Taichi* предоставляет инструменты для явного задания параллелизма, что позволяет использовать мощности GPU для обработки данных. Здесь функция *compute*\_*lcs*\_*gpu* запускается как *@ti*.*kernel*, что позволяет *Taichi* эффективно распараллеливать вычисления на *GPU*. Такая параллельная обработка данных на *GPU* может быть более эффективной по сравнению с последовательной обработкой на CPU.

Также стоит упомянуть, что *Taichi* использует оптимизированные алгоритмы и структуры данных для работы на *GPU*, что может значительно ускорять выполнение подобных вычислений по сравнению с *CPU*.

Кроме того, *Taichi* обеспечивает высокую производительность за счет оптимизации доступа к памяти, что также может способствовать более эффективной работе с данными на *GPU*.[20]

График сравнения составлен из средних значений из 5 тестовых замеров для каждого тестового случая, и находится в приложении А на рисунке 4.

Таким образом, мы проанализировали результаты, полученные при тестировании алгоритмов и их оптимизированных с помощью *GPU* версий.

# **ЗАКЛЮЧЕНИЕ**

Использование *GPU* для оптимизации времени выполнения кода привело к существенному ускорению вычислений в различных задачах. Параллельные вычисления на графическом процессоре оказались эффективными для операций, которые могут быть разделены на параллельные задачи, таких как поиск простых чисел, нахождение наибольшей общей подпоследовательности (*LCS*) и умножение матриц больших размеров.

Применение *GPU* для поиска простых чисел дает выгоду при генерации больших списков простых чисел для криптографических операций, таких как генерация ключей в криптографических алгоритмах *RSA*. В случае поиска наибольшей общей подпоследовательности (*LCS*) параллельные вычисления на GPU ускоряют обработку больших объемов данных, что полезно, например, в биоинформатике при анализе последовательностей ДНК или РНК. Также, оптимизированные вычисления на *GPU* для перемножения матриц могут применяться в задачах машинного обучения, где требуется обработка больших матриц данных.

Однако, в случае задач, где сложно параллелизовать вычисления или требуется сложная логика управления потоками, многопоточность на CPU может оказаться более эффективной. Например, для задач, связанных с сложными операциями обработки данных, которые не могут быть эффективно разделены на параллельные задачи, могут быть предпочтительным выбором многопоточные вычисления на *CPU*.

Таким образом, оптимизации, реализованные с использованием *GPU*, могут быть полезны в вычислительных задачах, которые могут быть эффективно параллелизованы и требуют обработки больших объемов данных. В то же время, в зависимости от характеристик задачи и доступного оборудования, многопоточные вычисления на *CPU* могут оказаться предпочтительными для задач, где сложно достичь параллелизации или требуется более сложная логика управления потоками данных.

# **СПИСОК ЛИТЕРАТУРНЫХ ИСТОЧНИКОВ**

[1] A Brief History of GPU. [Электронный ресурс]. – Режим доступа: <https://medium.com/altumea/a-brief-history-of-gpu-47d98d6a0f8a>. – Дата доступа: 05.10.2023.

[2] Архитектура GPU. [Электронный ресурс]. – Режим доступа: <http://optic.cs.nstu.ru/files/CC/CompGraph/G2_arch.pdf>. – Дата доступа: 05.10.2023.

[3] Exploring the GPU Architecture. [Электронный ресурс]. – Режим доступа: https://core.vmware.com/resource/exploring-gpu-architecture. – Дата доступа: 05.10.2023.

[4] Вычисления на GPU: Области применения [Электронный ресурс]. – Режим доступа: <https://introserv.com/ru/blog/vyichisleniya-na-gpu-oblasti-primeneniya>. – Дата доступа: 05.10.2023.

[5] The Power of GPUs: Revolutionizing Computing and Unlocking New Frontiers [Электронный ресурс]. – Режим доступа: https://towardsdatascience.com/why-does-a-graphics-card-help-in-machine-learning-8f365593b22. – Дата доступа: 05.10.2023.

[6] Importance of Linux OS [Электронный ресурс]. – Режим доступа: https://www.sevenmentor.com/importance-of-linux-os. – Дата доступа: 05.10.2023.

[7] Architecture of linux operating system [Электронный ресурс]. – Режим доступа: https://www.geeksforgeeks.org/architecture-of-linux-operating-system/. – Дата доступа: 05.10.2023.

[8] What is Linux Used For? [Электронный ресурс]. – Режим доступа: <https://www.lenovo.com/za/en/faqs/operating-systems/what-is-linux-used-for>. – Дата доступа: 05.10.2023.

[9] Linux. [Электронный ресурс]. – Режим доступа: <https://znanierussia.ru/articles/Linux>. – Дата доступа: 05.10.2023

[10] History of Python. [Электронный ресурс]. – Режим доступа: https://www.geeksforgeeks.org/history-of-python/. – Дата доступа: 05.10.2023

[11] Python. [Электронный ресурс]. – Режим доступа: [https://blog.skillfactory.ru/glossary/python/](https://blog.skillfactory.ru/glossary/python/%20) – Дата доступа: 05.10.2023

[12] The Best Python Frameworks in 2023. [Электронный ресурс]. – Режим доступа: https://hackr.io/blog/python-frameworks. – Дата доступа: 05.10.2023

[13] Python. [Электронный ресурс]. – Режим доступа: https://znanierussia.ru/articles/Python – Дата доступа: 05.10.2023

[14] Умножение матриц [Электронный ресурс]. – Режим доступа: https://ru.onlinemschool.com/math/library/matrix/multiply/. – Дата доступа: 31.10.2023.

[15] Why should we care about prime numbers? [Электронный ресурс]. – Режим доступа: https://mashable.com/article/why-should-we-care-about-prime-numbers. – Дата доступа: 31.10.2023.

[16] The Longest Common Subsequence Problem: An Introduction and Solution in Ruby. [Электронный ресурс]. – Режим доступа: https://patrickkarsh.medium.com/the-longest-common-subsequence-problem-an-introduction-and-solution-in-ruby-d7076e277e5b— Дата доступа: 31.10.2023

[17] Taichi. [Электронный ресурс]. – Режим доступа: <https://habr.com/ru/companies/wunderfund/articles/688134/> — Дата доступа: 31.10.2023

[18] Why a New Programming Language. [Электронный ресурс]. — Режим доступа: https://docs.taichi-lang.org/docs/overview — Дата доступа: 31.10.2023

[19] Numba for CUDA GPUs. [Электронный ресурс]. — Режим доступа: https://numba.readthedocs.io/en/stable/cuda/ — Дата доступа: 31.11.2023

[20] Compatibility, Maintainability, and Portability Improvements to the Taichi Graphics Programming Framework. [Электронный ресурс]. — Режим доступа: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-112.pdf. — Дата доступа: 31.11.2023

# **Приложение А**

**(обязательное)**

**Графики сравнения**

# **Приложение Б**

**(обязательное)**

**Функциональная схема алгоритма**

# **Приложение В**

(**обязательное)**

**Блок-схема алгоритма**

# **Приложение Г**

**(обязательное)**

**Листинг программного кода**

Листинг 1 **–** Код программы перемножения матриц

import multiprocessing

import sys

import numba

from numba import cuda, float32

import cupy as cp

import numpy as np

import time

import matplotlib.pyplot as plt

import psutil

class ResearchData:

def \_\_init\_\_(self, time, answer):

self.time = time

self.answer = answer

def matrix\_multiplication\_cpu(A, B, C):

'''Перемножение матриц на чистом python'''

for i in range(A.shape[0]):

for j in range(B.shape[0]):

for k in range(C.shape[0]):

C[i][j] += A[i][k] \* B[k][j]

def matrix\_multiplication\_numpy(A, B):

'''Перемножение матриц с помощью библиотеки numpy'''

return np.dot(A, B)

@cuda.jit

def matrix\_multiplication\_gpu(A, B, C):

'''Перемножение матриц с помощью библиотеки numba'''

i, j = cuda.grid(2)

if i < C.shape[0] and j < C.shape[1]:

tmp = 0.

for k in range(A.shape[1]):

tmp += A[i, k] \* B[k, j]

C[i, j] = tmp

# sA = cuda.shared.array(shape=(TPB, TPB), dtype=float32)

# sB = cuda.shared.array(shape=(TPB, TPB), dtype=float32)

# x, y = cuda.grid(2)

# tx = cuda.threadIdx.x

# ty = cuda.threadIdx.y

# bpg = cuda.gridDim.x

# if x >= C.shape[0] and y >= C.shape[1]:

# return

# tmp = 0.

# for i in range(bpg):

# sA[tx, ty] = A[x, ty + i \* TPB]

# sB[tx, ty] = B[tx + i \* TPB, y]

# cuda.syncthreads()

# for j in range(TPB):

# tmp += sA[tx, j] \* sB[j, ty]

# cuda.syncthreads()

# C[x, y] = tmp

def start\_mult(SIZE: int, TPB: int, need\_pure\_python: bool):

A\_cupy = cp.random.uniform(1, 10, size=(SIZE,SIZE), dtype=np.float64)

B\_cupy = cp.random.uniform(1, 10, size=(SIZE, SIZE), dtype=np.float64)

A\_numpy = cp.asnumpy(A\_cupy)

B\_numpy = cp.asnumpy(B\_cupy)

C\_gpu = cp.zeros((SIZE, SIZE), dtype=np.float64)

C\_pure\_python = np.zeros((SIZE, SIZE))

C\_numpy = np.zeros((SIZE, SIZE))

threadsperblock = (TPB, TPB)

blockspergrid = int(np.ceil(SIZE / threadsperblock[0]))

blockspergrid = (blockspergrid, blockspergrid)

start\_gpu = time.time()

matrix\_multiplication\_gpu[blockspergrid, threadsperblock](A\_cupy, B\_cupy, C\_gpu)

end\_gpu = time.time() - start\_gpu

research\_data\_gpu = ResearchData(end\_gpu, C\_gpu)

start\_numpy = time.time()

C\_numpy = matrix\_multiplication\_numpy(A\_numpy, B\_numpy)

end\_numpy = time.time() - start\_numpy

research\_data\_numpy = ResearchData(end\_numpy, C\_numpy)

if need\_pure\_python:

start\_pure\_python = time.time()

matrix\_multiplication\_cpu(A\_numpy, B\_numpy, C\_pure\_python)

end\_pure\_python = time.time() - start\_pure\_python

research\_data\_pure\_python = ResearchData(end\_pure\_python, C\_pure\_python)

return research\_data\_gpu, research\_data\_numpy, research\_data\_pure\_python

else:

return research\_data\_gpu, research\_data\_numpy

TPB = 16

tests\_number = 5

# research\_data\_gpu, research\_data\_numpy, research\_data\_pure\_python = start\_mult(SIZE, TPB)

# #check\_answer = np.allclose(cp.asnumpy(research\_data\_gpu.answer), research\_data\_numpy.answer, atol=1e-4, rtol=1e-3) and np.allclose(research\_data\_numpy.answer, research\_data\_pure\_python.answer, atol=1e-4, rtol=1e-3)

# check\_answer = True

need\_pure\_python = False

if need\_pure\_python:

list\_of\_sizes = [10, 25, 50, 75, 100, 150, 200, 300, 400]

list\_of\_times\_gpu = []

list\_of\_times\_numpy = []

list\_of\_times\_pure\_python = []

# Сохранение стандартного вывода в переменную

original\_stdout = sys.stdout

# Открытие файла для записи вывода

with open('matrix\_results\_1.txt', 'w') as file:

# Перенаправление вывода в файл

sys.stdout = file

for i in list\_of\_sizes:

test\_time\_gpu = []

test\_time\_numpy = []

test\_time\_pure\_python = []

for t in range(0, tests\_number):

research\_data\_gpu, research\_data\_numpy, research\_data\_pure\_python = start\_mult(i, TPB, True)

test\_time\_gpu.append(research\_data\_gpu.time)

test\_time\_numpy.append(research\_data\_numpy.time)

test\_time\_pure\_python.append(research\_data\_pure\_python.time)

list\_of\_times\_gpu.append(test\_time\_gpu)

list\_of\_times\_numpy.append(test\_time\_numpy)

list\_of\_times\_pure\_python.append(test\_time\_pure\_python)

print(f'size - {i}:')

print(f'gpu - {test\_time\_gpu}')

print(f'numpy - {test\_time\_numpy}')

print(f'pure python - {test\_time\_pure\_python}')

# Восстановление стандартного вывода

sys.stdout = original\_stdout

new\_list\_of\_times\_gpu = []

new\_list\_of\_times\_numpy = []

new\_list\_of\_times\_pure\_python = []

for l in list\_of\_times\_gpu:

sum = 0

for t in l:

sum += t

new\_list\_of\_times\_gpu.append((sum / tests\_number))

for l in list\_of\_times\_numpy:

sum = 0

for t in l:

sum += t

new\_list\_of\_times\_numpy.append((sum / tests\_number))

for l in list\_of\_times\_pure\_python:

sum = 0

for t in l:

sum += t

new\_list\_of\_times\_pure\_python.append((sum / tests\_number))

with open('matrix\_results\_1.txt', 'a') as file:

# Перенаправление вывода в файл

sys.stdout = file

print(new\_list\_of\_times\_gpu)

print(new\_list\_of\_times\_numpy)

print(new\_list\_of\_times\_pure\_python)

# Восстановление стандартного вывода

sys.stdout = original\_stdout

# График для первого вида

plt.plot(list\_of\_sizes, new\_list\_of\_times\_gpu, label='gpu perf', linestyle='-', color='blue')

# График для второго вида

plt.plot(list\_of\_sizes, new\_list\_of\_times\_numpy, label='numpy BLAS perf', linestyle='--', color='green')

# График для третьего вида

plt.plot(list\_of\_sizes, new\_list\_of\_times\_pure\_python, label='pure python perf', linestyle=':', color='red')

plt.xlabel('size of matrix') # Подпись оси x

plt.ylabel('time, s') # Подпись оси y

plt.title('multiplying matrix perfomance') # Заголовок графика

plt.legend() # Отображение легенды

# Отображение графика

plt.grid(True) # Отображение сетки

plt.show()

list\_of\_sizes = [100, 500, 1000, 2000, 3000, 4000]

list\_of\_times\_gpu = []

list\_of\_times\_numpy = []

# Сохранение стандартного вывода в переменную

original\_stdout = sys.stdout

# Открытие файла для записи вывода

with open('matrix\_results\_2.txt', 'w') as file:

# Перенаправление вывода в файл

sys.stdout = file

for i in list\_of\_sizes:

test\_time\_gpu = []

test\_time\_numpy = []

for t in range(0, tests\_number):

research\_data\_gpu, research\_data\_numpy = start\_mult(i, TPB, False)

test\_time\_gpu.append(research\_data\_gpu.time)

test\_time\_numpy.append(research\_data\_numpy.time)

# answer\_gpu = cp.asnumpy(research\_data\_gpu.answer)

# answer\_numpy = research\_data\_numpy.answer

# comparison = np.allclose(answer\_gpu, answer\_numpy, atol=1e-4, rtol=1e-3)

# if not comparison:

# print(answer\_gpu)

# print("------------")

# print(answer\_numpy)

# print("\*\*\*\*\*\*\*\*\*\*\*\*\*")

# raise Exception("wrong answer")

list\_of\_times\_gpu.append(test\_time\_gpu)

list\_of\_times\_numpy.append(test\_time\_numpy)

print(f'size - {i}:\n')

print(f'gpu - {test\_time\_gpu}')

print(f'numpy - {test\_time\_numpy}')

# Восстановление стандартного вывода

sys.stdout = original\_stdout

new\_list\_of\_times\_gpu = []

new\_list\_of\_times\_numpy = []

for l in list\_of\_times\_gpu:

sum = 0

for t in l:

sum += t

new\_list\_of\_times\_gpu.append((sum / tests\_number))

for l in list\_of\_times\_numpy:

sum = 0

for t in l:

sum += t

new\_list\_of\_times\_numpy.append((sum / tests\_number))

with open('matrix\_results\_2.txt', 'a') as file:

# Перенаправление вывода в файл

sys.stdout = file

print(new\_list\_of\_times\_gpu)

print(new\_list\_of\_times\_numpy)

# Восстановление стандартного вывода

sys.stdout = original\_stdout

# График для первого вида

plt.plot(list\_of\_sizes, new\_list\_of\_times\_gpu, label='gpu perf', linestyle='-', color='blue')

# График для второго вида

plt.plot(list\_of\_sizes, new\_list\_of\_times\_numpy, label='numpy BLAS perf', linestyle='--', color='green')

plt.xlabel('size of matrix (n \* n)') # Подпись оси x

plt.ylabel('time, s') # Подпись оси y

plt.title('multiplying matrix perfomance') # Заголовок графика

plt.legend() # Отображение легенды

# Отображение графика

plt.grid(True) # Отображение сетки

plt.show()

Листинг 2 **–** Код программы нахождения всех простых чисел меньше заданного.

import sys

import cupy as cp

from numba import cuda

import numpy as np

import time

import matplotlib.pyplot as plt

class ResearchData:

def \_\_init\_\_(self, time, answer):

self.time = time

self.answer = answer

BLOCK\_SIZE = 8

@cuda.jit("void(uint32[:], int32)")

def \_sieve\_forward(arr, bound) -> None:

i = cuda.grid(1)

if i < arr.size and i < bound:

if arr[i] == 1:

arr[i \* i :: i] = 0

def \_sieve\_forward\_cpu(arr, i, bound) -> None:

if i < arr.size and i < bound:

if arr[i] == 1:

arr[i \* i :: i] = 0

def count\_primes(n: int):

bound = int(cp.sqrt(n) + 1)

grid\_blocks = (bound + (BLOCK\_SIZE - 1))

primes = cp.full(n, 1, dtype=cp.uint32)

primes[0], primes[1] = 0, 0

start\_gpu = time.time()

with cuda.defer\_cleanup():

\_sieve\_forward[grid\_blocks, BLOCK\_SIZE](primes, bound)

result\_gpu = cp.flatnonzero(primes)

end\_gpu = time.time() - start\_gpu

research\_data\_gpu = ResearchData(end\_gpu, result\_gpu)

start\_cpu = time.time()

for i in range(0, n):

\_sieve\_forward\_cpu(primes, i, bound)

result\_cpu = cp.flatnonzero(primes)

end\_cpu = time.time() - start\_cpu

research\_data\_cpu = ResearchData(end\_cpu, result\_cpu)

return research\_data\_gpu, research\_data\_cpu

list\_of\_sizes = [1000, 10000, 20000, 100000, 200000, 1000000, 2000000, 10000000, 20000000]

list\_of\_times\_gpu = []

list\_of\_times\_cpu = []

tests\_number = 5

# Сохранение стандартного вывода в переменную

original\_stdout = sys.stdout

# Открытие файла для записи вывода

with open('primary\_numbers\_results.txt', 'w') as file:

# Перенаправление вывода в файл

sys.stdout = file

for i in list\_of\_sizes:

test\_time\_gpu = []

test\_time\_cpu = []

for t in range(0, tests\_number):

research\_data\_gpu, research\_data\_cpu = count\_primes(i)

test\_time\_gpu.append(research\_data\_gpu.time)

test\_time\_cpu.append(research\_data\_cpu.time)

answer\_gpu = cp.asnumpy(research\_data\_gpu.answer)

answer\_cpu = cp.asnumpy(research\_data\_cpu.answer)

comparison = np.allclose(answer\_gpu, answer\_cpu, atol=1e-4, rtol=1e-3)

if not comparison:

print(answer\_gpu)

print("------------")

print(answer\_cpu)

print("\*\*\*\*\*\*\*\*\*\*\*\*\*")

raise Exception("wrong answer")

list\_of\_times\_cpu.append(test\_time\_cpu)

list\_of\_times\_gpu.append(test\_time\_gpu)

print(f'size - {i}:')

print(f'gpu - {test\_time\_gpu}')

print(f'cpu - {test\_time\_cpu}')

# Восстановление стандартного вывода

sys.stdout = original\_stdout

new\_list\_of\_times\_gpu = []

new\_list\_of\_times\_cpu = []

for l in list\_of\_times\_gpu:

sum = 0

for t in l:

sum += t

new\_list\_of\_times\_gpu.append((sum / tests\_number))

for l in list\_of\_times\_cpu:

sum = 0

for t in l:

sum += t

new\_list\_of\_times\_cpu.append((sum / tests\_number))

with open('primary\_numbers\_results.txt', 'a') as file:

# Перенаправление вывода в файл

sys.stdout = file

print(new\_list\_of\_times\_gpu)

print(new\_list\_of\_times\_cpu)

# Восстановление стандартного вывода

sys.stdout = original\_stdout

# График для первого вида

plt.plot(list\_of\_sizes, new\_list\_of\_times\_gpu, label='gpu perf', linestyle='-', color='blue')

# График для второго вида

plt.plot(list\_of\_sizes, new\_list\_of\_times\_cpu, label='cpu perf', linestyle='--', color='green')

plt.xlabel('size of number') # Подпись оси x

plt.ylabel('time, s') # Подпись оси y

plt.title('finding all prime numbers before n') # Заголовок графика

plt.legend() # Отображение легенды

# Отображение графика

plt.grid(True) # Отображение сетки

plt.show()

Листинг 3 **–** Код программы нахождения общей подпоследовательности из двух последовательностей.

import sys

import taichi as ti

import numpy as np

import time

import matplotlib.pyplot as plt

ti.init(arch=ti.gpu)

class ResearchData:

def \_\_init\_\_(self, time, answer):

self.time = time

self.answer = answer

N = 2000

f = ti.field(dtype=ti.i32, shape=(N + 1, N + 1))

@ti.kernel

def compute\_lcs\_gpu(a: ti.types.ndarray(), b: ti.types.ndarray()) -> ti.i32:

len\_a, len\_b = a.shape[0], b.shape[0]

ti.loop\_config(serialize=True) # To forbid automatic parallelism

for i in range(1, len\_a + 1):

for j in range(1, len\_b + 1):

if a[i-1] == b[j-1]:

f[i, j] = f[i-1, j-1] + 1

else:

f[i, j] = max(f[i-1, j], f[i, j-1])

return f[len\_a, len\_b]

def compute\_lcs\_cpu (a, b):

"""Length of LCS using a dynamic program."""

# Initialize matrix of substring LCSs.

lcs\_mat = np.zeros((len(a)+ 1, len(b) + 1), dtype=int)

# Build matrix, left to right, row by row

for i in range(1, len(a)+1):

for j in range(1, len(b)+1):

if a[i-1] == b[j-1]:

lcs\_mat[i, j] = lcs\_mat[i-1, j-1] + 1

else:

lcs\_mat[i, j] = max(lcs\_mat[i-1, j], lcs\_mat[i, j-1])

return lcs\_mat[-1,-1]

def start\_computing(n: int):

a = np.random.randint(0, 100, n, dtype=np.int32)

b = np.random.randint(0, 100, n, dtype=np.int32)

start\_gpu = time.time()

result\_gpu = compute\_lcs\_gpu(a, b)

end\_gpu = time.time() - start\_gpu

research\_data\_gpu = ResearchData(end\_gpu, result\_gpu)

start\_cpu = time.time()

result\_cpu = compute\_lcs\_cpu(a, b)

end\_cpu = time.time() - start\_cpu

research\_data\_cpu = ResearchData(end\_cpu, result\_cpu)

return research\_data\_gpu, research\_data\_cpu

list\_of\_sizes = [10, 50, 100, 250, 500, 750, 1000, 1500, 2000]

list\_of\_times\_gpu = []

list\_of\_times\_cpu = []

tests\_number = 5

# Сохранение стандартного вывода в переменную

original\_stdout = sys.stdout

with open('finding\_LCS\_results.txt', 'w') as file:

# Перенаправление вывода в файл

sys.stdout = file

for i in list\_of\_sizes:

test\_time\_gpu = []

test\_time\_cpu = []

for t in range(0, tests\_number):

research\_data\_gpu, research\_data\_cpu = start\_computing(i)

test\_time\_gpu.append(research\_data\_gpu.time)

test\_time\_cpu.append(research\_data\_cpu.time)

if (research\_data\_gpu.answer != research\_data\_cpu.answer):

print(f"gpu answer: {research\_data\_gpu.answer}")

print(f"cpu answer: {research\_data\_cpu.answer}")

raise Exception("wrong answers")

list\_of\_times\_cpu.append(test\_time\_cpu)

list\_of\_times\_gpu.append(test\_time\_gpu)

print(f'size - {i}:')

print(f'gpu - {test\_time\_gpu}')

print(f'cpu - {test\_time\_cpu}')

# Восстановление стандартного вывода

sys.stdout = original\_stdout

new\_list\_of\_times\_gpu = []

new\_list\_of\_times\_cpu = []

for l in list\_of\_times\_gpu:

sum = 0

for t in l:

sum += t

new\_list\_of\_times\_gpu.append((sum / tests\_number))

for l in list\_of\_times\_cpu:

sum = 0

for t in l:

sum += t

new\_list\_of\_times\_cpu.append((sum / tests\_number))

with open('finding\_LCS\_results.txt', 'a') as file:

# Перенаправление вывода в файл

sys.stdout = file

print(new\_list\_of\_times\_gpu)

print(new\_list\_of\_times\_cpu)

# Восстановление стандартного вывода

sys.stdout = original\_stdout

# График для первого вида

plt.plot(list\_of\_sizes, new\_list\_of\_times\_gpu, label='gpu perf', linestyle='-', color='blue')

# График для второго вида

plt.plot(list\_of\_sizes, new\_list\_of\_times\_cpu, label='cpu perf', linestyle='--', color='green')

plt.xlabel('size of number n') # Подпись оси x

plt.ylabel('time, s') # Подпись оси y

plt.title('finding LCS of 2 sequences of size n') # Заголовок графика

plt.legend() # Отображение легенды

# Отображение графика

plt.grid(True) # Отображение сетки

plt.show()

# **Приложение Д**

**(обязательное)**

**Ведомость документов**