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» |
|  | Руководитель курсового проекта  ассистент кафедры информатики  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_А. А. Калиновская |
|  | \_\_\_.\_\_\_\_.2023 |

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

к курсовому проекту

на тему:

**«Сравнение эффективности распараллеливания CPU и GPU с CUDA»**

БГУИР КП 1-40 04 01 021 ПЗ

|  |  |
| --- | --- |
|  | Выполнил студент группы 153503  Вергасов Вадим Михайлович  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись студента) |
|  | Курсовой проект представлен на проверку \_\_\_.\_\_\_\_.2023  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись студента) |

Минск 2023

**СОДЕРЖАНИЕ**

[Введение 3](#_Toc149126013)

[1 Архитектура вычислительной системы 5](#_Toc149126014)

[2 Платформа программного обеспечения 10](#_Toc149126015)

[3 Теоретическое обоснование разработки программного продукта 14](#_Toc149126016)

[4 Проектирование функциональных возможностей программы 16](#_Toc149126017)

[5 Сравнение эффективности распараллеливания 17](#_Toc149126018)

[Заключение 18](#_Toc149126019)

[Список литературы 19](#_Toc149126020)

[Приложение А 20](#_Toc149126021)

[Приложение Б 21](#_Toc149126022)

[Приложение В 22](#_Toc149126023)

[Приложение В 23](#_Toc149126024)

[Приложение Г 25](#_Toc149126025)

# ВВЕДЕНИЕ

Высокопроизводительные вычисления (*High Performance Computing, HPC*) являются важной составляющей многих областей науки и техники, таких как разработка новых материалов, моделирование климата, обработка больших данных и многих других. Распространенность *HPC* зависит от множества факторов, включая доступность оборудования, стоимость вычислений, уровень развития технологий и потребности пользователей.

В настоящее время высокопроизводительные вычисления становятся все более распространенными благодаря развитию технологий и увеличению доступности оборудования. Многие научные организации и промышленные предприятия используют *HPC* для решения различных задач, требующих большого объема вычислений. Например, многие университеты и исследовательские центры имеют свои собственные суперкомпьютеры, которые используются для проведения научных исследований и обучения студентов.

Также стоит отметить, что распространение высокопроизводительных вычислений не ограничивается только научными и образовательными учреждениями. В последнее время многие компании начинают использовать HPC для оптимизации своих бизнес-процессов, анализа больших данных и разработки новых продуктов. Это связано с тем, что высокопроизводительные вычисления позволяют компаниям быстрее обрабатывать большие объемы данных, что в свою очередь позволяет им принимать более обоснованные решения и улучшать качество своих продуктов.

Распараллеливание вычислений становится все более распространенным в различных областях науки и техники. Это связано с развитием технологий и увеличением доступности оборудования, которое позволяет выполнять параллельные вычисления.

Многие научные организации и промышленные предприятия используют распараллеливание для решения различных задач, требующих большого объема вычислений. Университеты и исследовательские центры также активно используют параллельные вычисления для проведения научных исследований и обучения студентов.

Кроме того, многие компании начинают использовать распараллеливание для оптимизации своих бизнес-процессов и анализа больших данных. Это позволяет им быстрее обрабатывать большие объемы информации и принимать более обоснованные решения.

Однако, чтобы полностью использовать потенциал распараллеливания, необходимо продолжать разрабатывать новые технологии и создавать более эффективное оборудование.

В современном мире, где вычислительные задачи становятся все более сложными и ресурсоемкими, вопрос оптимизации производительности вычислений является ключевым для многих научных, инженерных и коммерческих приложений. Одним из перспективных направлений в этой области является использование параллельных вычислений, которые позволяют одновременно выполнять множество операций на нескольких процессорах или устройствах.

Два основных типа процессоров, используемых в современных вычислительных системах, — это центральный процессор (*CPU*) и графический процессор (*GPU*). *GPU*, изначально разработанный для обработки графики в игровых консолях и ПК, стал одним из наиболее эффективных инструментов для вычислений с высоким параллелизмом благодаря своей архитектуре с тысячами ядер и высокой пропускной способности памяти.

*CUDA (Compute Unified Device Architecture)* – это программная архитектура, разработанная компанией *NVIDIA* для параллельных вычислений на графических процессорах. Она предоставляет набор инструментов и библиотек, упрощающих программирование на *GPU* и позволяющих разработчикам использовать всю мощь параллельной обработки для решения сложных вычислительных задач.

Цель данной курсовой работы – провести сравнение эффективности распараллеливания на *CPU* и *GPU* с использованием *CUDA* для вычислительных задач и определить, в каких случаях *GPU* с поддержкой *CUDA* может обеспечить более высокую эффективность распараллеливания по сравнению с традиционным *CPU*.

В рамках курсовой работы будут рассмотрены различные метрики производительности, включая время выполнения, энергопотребление, стоимость оборудования и другие факторы, влияющие на выбор оптимальной платформы для параллельных вычислений.

Результаты курсового проекта могут быть полезны для разработчиков, занимающихся созданием высокопроизводительных приложений, а также для ученых и инженеров, работающих в областях, где требуется решение сложных вычислительных задач в короткие сроки.

# 1 Архитектура вычислительной системы

Архитектура *CPU* может быть различной в зависимости от производителя и модели. Например, *Intel* использует архитектуру x86, которая имеет набор инструкций, позволяющих выполнять различные операции над данными. *AMD* также использует архитектуру x86, но имеет свои особенности в дизайне процессора. Кроме того, существуют процессоры, основанные на других архитектурах, таких как *ARM* и *POWER*.

*X86* и *x86-64* — это две архитектуры процессоров. Обе архитектуры используют набор инструкций x86 для выполнения различных операций над данными. Однако есть некоторые различия между этими двумя архитектурами.

*X86* — это 32-битная архитектура, которая была разработана в начале 1980-х годов. Она использовалась в первых компьютерах *IBM PC* и стала основой для многих других процессоров. *X86* до сих пор используется в некоторых старых компьютерах и в некоторых новых, более дешевых моделях.

*X86-64*, также известная как *AMD64* или *x64*, является 64-битной версией архитектуры *x86*. Она была разработана компанией *AMD* в начале 2000-х годов и стала стандартом для современных компьютеров. *X86-64* позволяет использовать больше оперативной памяти и выполнять более сложные операции, чем x86.

Для того, чтобы программа, написанная на *C++*, работала, необходимо её скомпилировать. Схема работы компилятора изображена на рисунке 1.1.

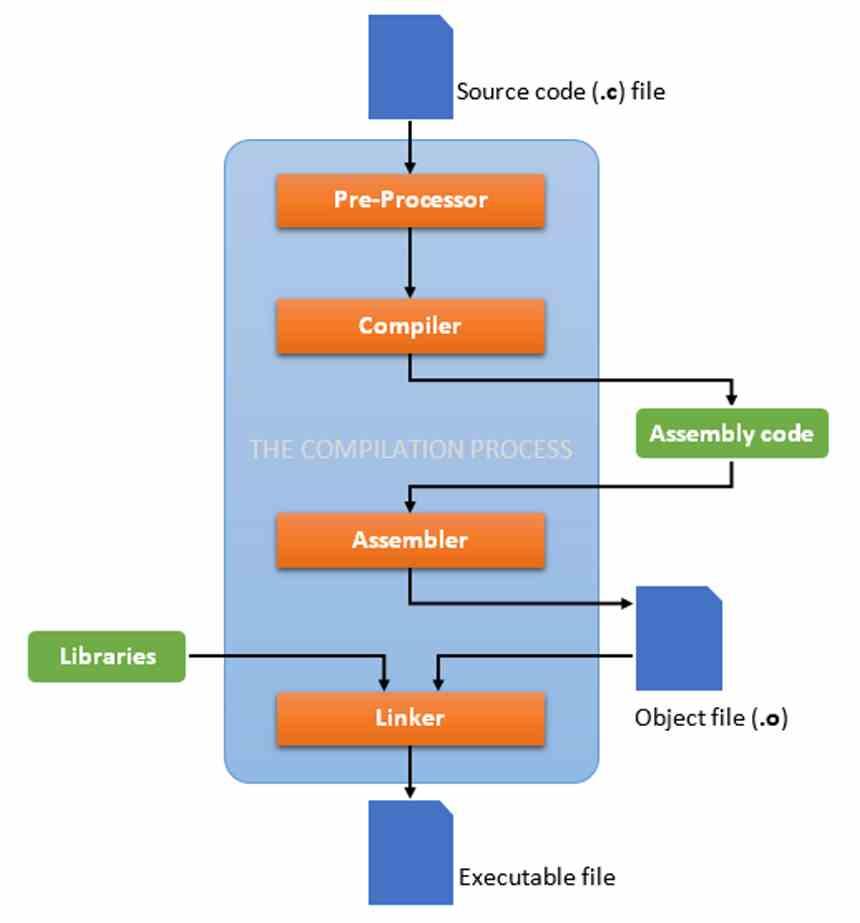


Рисунок 1.1 – процесс компиляции программы.

Обе архитектуры используются в современных компьютерах, но *x86-64* является более популярной из-за своих преимуществ. Однако, некоторые приложения и операционные системы до сих пор используют *x86* из-за своей совместимости с более старыми системами.

*GPU (Graphics Processing Unit)* — это специализированный процессор, предназначенный для обработки графической информации. Он используется в компьютерах, ноутбуках, игровых консолях и других устройствах для рендеринга *3D*-графики и обработки видео, а также исполнения других задач, поддающихся распараллеливанию.

Архитектура *GPU* отличается от архитектуры *CPU* тем, что она оптимизирована для выполнения большого количества параллельных вычислений и имеет множество ядер, каждый из которых может выполнять свою часть работы независимо от других. Это позволяет *GPU* обрабатывать огромные объемы данных и выполнять сложные вычисления быстрее, чем *CPU*.

Одним из главных преимуществ *GPU* является его высокая производительность при обработке графических данных. Однако, его использование в других областях, таких как научные вычисления и машинное обучение, может быть ограничено из-за отсутствия поддержки некоторых типов данных и операций.

Тем не менее, *GPU* продолжают развиваться и становятся все более универсальными, что делает их привлекательными для широкого круга задач, включая научные вычисления, обработку изображений и видео, а также искусственный интеллект.

*CUDA (Compute Unified Device Architecture)* была создана компанией *NVIDIA* в 2007 году как способ использования графических процессоров (*GPU*) для ускорения научных и технических расчетов. Она была первой архитектурой такого рода, и с тех пор многие другие компании, включая *AMD*, *Intel* и *ARM*, создали свои собственные аналоги.

С тех пор *CUDA* стала одной из наиболее широко используемых архитектур для *GPU*, и она продолжает развиваться, чтобы поддерживать новые возможности и улучшения производительности. *NVIDIA* продолжает улучшать свои GPU, чтобы сделать их еще более мощными и эффективными для научных и инженерных расчетов. Схема взаимодействия ПО, написанного с помощью библиотеки CUDA и исполнительным устройством изображено на рисунке 1.2.

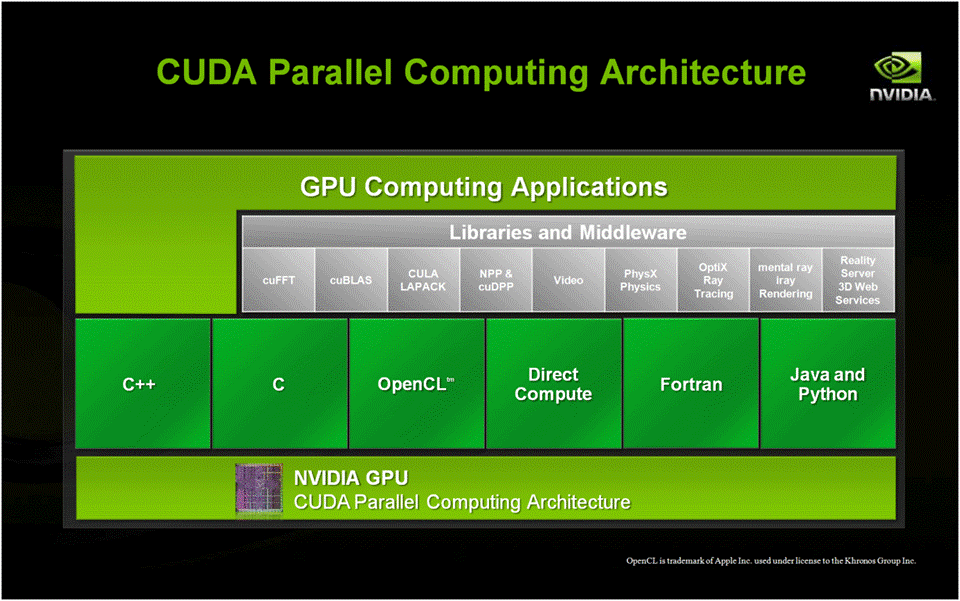


Рисунок 1.2 – схема взаимодействия ПО с GPU.

*Intel Core i7-8750H* — это процессор *Intel* восьмого поколения, разработанный для ноутбуков и настольных компьютеров. Он имеет 6 ядер, 12 потоков и работает на частоте до 4,1 ГГц. Процессор оснащен встроенной графикой *Intel UHD Graphics* 630 и поддерживает до 64 ГБ оперативной памяти *DDR4-2666*. *i7-8750H* также имеет встроенный контроллер *Thunderbolt* 3 и технологию *Intel Optane* для ускорения работы с твердотельными накопителями. Схема чипа изображена на рисунке 1.3.

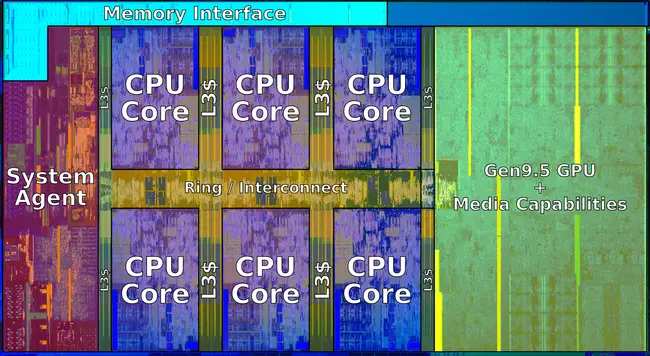


Рисунок 1.3 – Схема чипа *i7-8750H*.

Основные характеристики:

1. Частота: 2200 - 4100 МГц.
2. Кэш L1: 384 KB.
3. Кэш L2: 1.5 MB.
4. Кэш L3: 9 MB.
5. Ядер/потоков: 6/12.
6. Теплопакет (*TDP*): 45 Вт.
7. Техпроцесс: 14нм.
8. Макс. Температура: 100 °C.
9. Разъём (сокет): *FCBGA1440*.
10. Особенности: *Dual-Channel DDR4 Memory Controller, Hyperthreading, AVX, AVX2, Virtualization, AES-NI*.
11. Встроенная графика: *Intel UHD Graphics* 630 (350 - 1100 MHz).
12. Разрядность: 64 бита.
13. Архитектура: x86-64.

*RTX 2060* — это видеокарта от *NVIDIA*, выпущенная в 2019 году. Она предназначена для игровых компьютеров и оснащена технологией трассировки лучей, которая позволяет создавать более реалистичное освещение и тени в играх. *RTX 2060* также имеет поддержку *DLSS (Deep Learning Super Sampling)*, которая улучшает качество изображения.

*RTX* 2060 имеет в своем основании чип *TU106*, однако в нем отключены некоторые *SM*-блоки. Вид микроархитектуры чипа можно увидеть на рисунке 1.4.

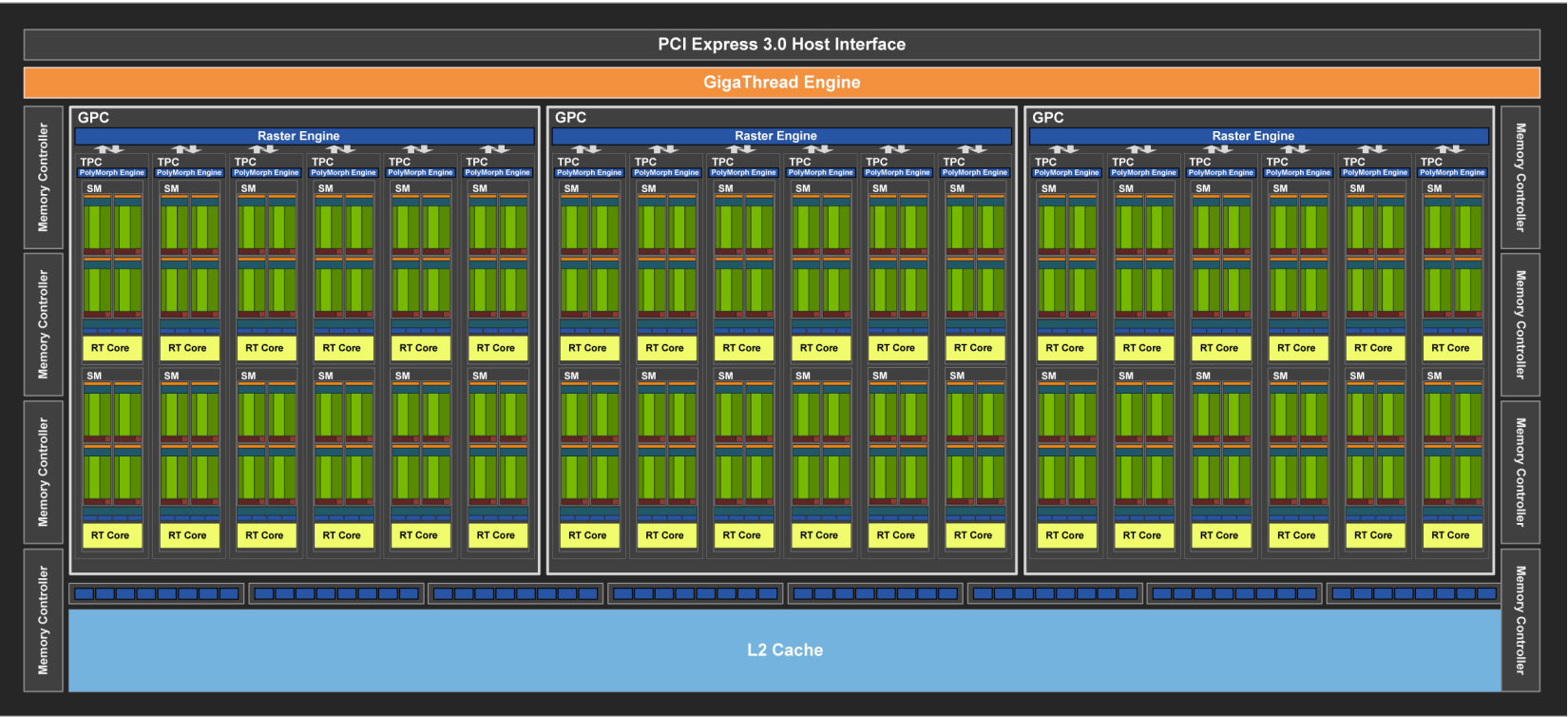


Рисунок 1.4 – Микроархитектура чипа TU106

Основные характеристики:

1. Кодовое имя: *N18E-G1 TU106*.
2. Архитектура: *Turing*.
3. Конвейеров: 1920 – унифицированные.
4. Частота видеопроцессора: 960 - 1200 (*Boost*) МГц.
5. Частота памяти: 14000 МГц.
6. Шина видеопамяти: 192 бит.
7. Тип памяти: *GDDR6*.
8. Количество памяти: 6 GB.
9. *API: DirectX* 12.1, *OpenGL* 4.6.
10. Энергопотребление: 80-90 Вт.
11. Технология производства: 12нм.
12. Особенности и возможности: *DLSS, Raytracing*.

Поколение видеокарт *NVIDIA Turing* (также известное как *RTX*) представляет собой серию графических процессоров, разработанных и произведенных компанией *NVIDIA*. Это поколение видеокарт включает в себя архитектуру *Turing*, которая была представлена в марте 2018 года. Основными особенностями архитектуры *Turing* являются использование технологии трассировки лучей в реальном времени (*real-time ray tracing*) и аппаратного ускорения искусственного интеллекта (*AI*).

В основе архитектуры *Turing* лежит улучшенный 12-нм техпроцесс *FinFET* от компании *Samsung*. Этот процесс обеспечивает более высокую производительность, энергоэффективность и плотность транзисторов по сравнению с предыдущими поколениями видеокарт. *NVIDIA* также представила технологию «*Tensor Cores*», которая обеспечивает аппаратное ускорение *AI* и поддерживает новые функции, такие как *DLSS (Deep Learning Super Sampling)* и *NVIDIA Broadcast*.

Поколение видеокарт *NVIDIA Turing* включает в себя несколько моделей, таких как *GeForce RTX 2060*, *RTX 2070*, *RTX 2080* и *RTX 2080Ti*.

*NVIDIA* также выпустила видеокарты на основе архитектуры *Turing* для рабочих станций, такие как *Quadro RTX* *5000*. Эта модель обеспечивает хорошую производительность в профессиональных приложениях, а также поддерживают трассировку лучей и *DLSS*.

В целом, поколение видеокарт *NVIDIA Turing* представляет собой значительный шаг вперед в области графических процессоров благодаря использованию трассировки лучей и аппаратного ускорения *AI*.

# 2 Платформа программного обеспечения

*CUDA (Compute Unified Device Architecture)* — это технология параллельных вычислений, разработанная компанией *NVIDIA* для использования на графических процессорах (*GPU*) для ускорения работы с массивами данных. *CUDA* предоставляет программистам возможность использовать графический процессор для выполнения сложных вычислений, таких как обработка изображений, моделирование физических процессов, научные вычисления и другие.

*CUDA* использует модель грид-вычислений, которая позволяет разделить задачу на множество потоков, каждый из которых работает на своем участке данных. Потоки могут взаимодействовать друг с другом, используя разделяемую память, что обеспечивает высокую производительность и эффективность использования ресурсов *GPU*.

Для программирования на *CUDA* используются языки программирования, такие как *C*, *C++* и *CUDA-C*. *CUDA-C* — это расширение языка C, которое предоставляет дополнительные функции и возможности для работы с *GPU*.

С помощью *CUDA* можно создавать графические приложения, которые используют преимущества *GPU* для ускорения работы. Например, можно использовать *CUDA* для обработки видео, создания *3D*-графики, симуляции физических процессов и многого другого.

Visual Studio — это интегрированная среда разработки (*IDE*) от компании *Microsoft*, предназначенная для создания программного обеспечения на языках программирования *C++, C#, Visual Basic, F#* и других. *Visual Studio* предоставляет разработчикам инструменты для создания, отладки, тестирования и развертывания приложений, а также для совместной работы над проектами.

Основные версии *Visual Studio*:

1. *Visual Studio.NET* (2002): выпущена в феврале 2002 года, включает *.NET Framework* 1.0.
2. *Visual Studio.NET* (2003): выпущена в апреле 2003 года, включает *.NET* Framework 1.1.
3. *Visual Studio* 2005: выпущена в конце октября 2005 года, включает *.NET Framewor*k 2.0.
4. *Visual Studio* 2008: выпущена 19 ноября 2007 года, включает.NET Framework 3.5.
5. *Visual Studio* 2010: выпущена 12 апреля 2010 года, включает.NET Framework 4.0.
6. *Visual Studio* 2012: выпущена 11 апреля 2012 года, включает *.NET* Framework 4.5.
7. *Visual Studio* 2013: выпущена 17 сентября 2013 года, включает *.NET* Framework 4.5.1.
8. *Visual Studio* 2015: выпущена 20 июля 2015 года, включает *.NET* Framework 4.6.
9. *Visual Studio* 2017: выпущена 7 марта 2017 года, включает *.NET* Framework 4.6.1.
10. *Visual Studio* 2019: выпущена 2 апреля 2019 года, включает *.NET* Framework 4.7.2.
11. *Visual Studio* 2022: выпущена 8 ноября 2021 года, включает *.NET* Framework 4.8.1.

*Visual Studio* также доступна в различных редакциях:

* *Visual Studio Community*: бесплатная версия для разработки программного обеспечения, подходящая для небольших проектов и обучения.
* *Visual Studio Professional*: коммерческая версия для разработки коммерческого программного обеспечения.
* *Visual Studio Enterprise*: версия для крупных компаний и организаций, предоставляющая расширенные возможности для совместной работы и управления проектами.

Помимо основных версий, *Visual Studio* также предлагает ряд дополнительных инструментов и функций, таких как отладчик, профилировщик, инструменты анализа кода, инструменты для работы с базами данных.

*Ubuntu Live CD* — это загрузочный компакт-диск, содержащий операционную систему *Ubuntu*, которая позволяет пользователям протестировать и установить *Ubuntu* на свой компьютер без необходимости форматирования жесткого диска. *Live CD* предоставляет возможность работать с системой, не затрагивая данные на жестком диске, что делает его удобным инструментом для тестирования и отладки программного обеспечения.

*Ubuntu Live CD* обычно содержит следующие компоненты:

1. Загрузочный образ диска: содержит операционную систему и необходимые драйверы для работы с различными типами оборудования.
2. Файловая система: содержит все необходимые файлы и каталоги для работы системы.
3. Инструменты для создания и работы с образами дисков: позволяют создавать образы дисков для резервного копирования и восстановления системы.
4. Инструменты для работы с образами дисков и сжатия файлов: позволяют создавать и сжимать образы дисков, что уменьшает их размер и упрощает передачу.
5. Инструменты для создания *ISO* образов: позволяют создать *ISO* образ диска для создания загрузочного компакт-диска или *DVD*.

Для работы с *Ubuntu Live CD* необходимо загрузить образ диска на жесткий диск и запустить его с помощью загрузочного меню. После запуска системы, пользователь может работать с ней, устанавливать и удалять программы, а также тестировать различные функции операционной системы.

В данной курсовой работе будет использоваться для создания более предсказуемых условий, чтобы сравнение было более объективным.

Так же будет использоваться профилирование программ, написанных под *CUDA* с помощью *Nvidia Nsight Compute*, которое позволяет оценить использование GPU, а именно: использование памяти, в том числе и скорость передачи данных, использование регистров исполнительных блоков, а также программа может давать рекомендации, которые могут повысить эффективность работы ПО.

# 3 Теоретическое обоснование разработки программного продукта

Сейчас любой персональный компьютер использует чипы, которые могут обрабатывать параллельно данные. Для того, чтобы приложения работали быстрее, необходимо поддержать работу ПО с несколькими потоками. Однако очевидно, что использование кратного количества исполнительных блоков не даст такое же увеличение производительности. Сегодня наиболее актуален вопрос эффективности работы многопоточных программ на CPU и GPU, как наиболее распространенных вычислительных устройств.

Разработка системы перемножения матриц на *GPU* с *CUDA* необходима из-за следующих преимуществ:

1. Увеличение производительности вычислений: благодаря большому количеству ядер и особенностям архитектуры, использование *GPU* позволяет значительно увеличить производительность для данной задачи.
2. Перспективы применения вычислений на *GPU*: разработка оптимального для графических вычислительных устройств параллельного алгоритма требует много ресурсов, однако полученный результат может сэкономить большое количество времени, что нивелирует время, затраченное на разработку и реализацию данного алгоритма.
3. Уменьшение времени выполнения умножений: использование *GPU* позволяет серьезно уменьшить время выполнения умножений, что приводит к уменьшению времени работы программы, решающей СЛАУ.

Однако, особенности работы *GPU* затрудняют применение данного подхода, поэтому необходимо определить критерии направления операций на *GPU* и оптимизировать работу с памятью.

Для написания программы, решающей СЛАУ на *CUDA*, используются следующие технологии программирования:

1. *CUDA API*: расширение языка *С*, позволяющее разработчикам не тратить много времени на изучение данной технологии.
2. Драйвер устройства: обеспечивает взаимодействие между хостом и графическим процессором.
3. Программный интерфейс (*API*): предоставляет доступ к функциям и ресурсам графического процессора.
4. Математические библиотеки: *CUBLAS* и *CUFFT*, содержащие основные операции линейной алгебры и быстрое преобразование Фурье.
5. Библиотека *Thrust*, позволяющая работать с *CUDA API* в парадигме *RAII*, которая является основной в языке *C++*.

Эти технологии позволяют разработчикам писать эффективные и параллельные программы для решения СЛАУ на *GPU* с использованием *CUDA*.

Связь архитектуры вычислительной системы с разрабатываемым ПО заключается в том, что архитектура определяет возможности и ограничения системы, а также влияет на выбор и разработку программного обеспечения. Архитектура определяет, какие типы вычислительных операций могут быть выполнены на данной системе, а также какие ресурсы и алгоритмы могут быть использованы для решения конкретных задач.

Например, если система имеет несколько ядер, то это может повлиять на выбор алгоритма, который будет использоваться для решения задачи. Если система имеет многоядерный процессор, то можно использовать параллельные алгоритмы, которые будут распределять вычисления между ядрами, что приведет к увеличению скорости решения задачи.

В данном случае будут заметны особенности работы *CPU* и *GPU* с памятью, а также то, что *CPU* – более универсальное вычислительное устройство, которое может иметь возможность исполнения более сложных инструкций, в то время как *GPU* имеет большое количество простых исполнительных устройств.

*CUDA API* – набор программ, позволяющий компилировать программу, написанную на расширенном *C++*. В состав входят: компилятор *NVCC*, система профилирования *NVIDIA Nsight Compute*, а также библиотеки, реализующие готовые функции. Компилятор преобразует код, написанный на *C++* в специальный диалект *Assembler*’а.

# 4 Проектирование функциональных возможностей программы

Перемножение матриц – это процесс умножения элементов (чисел или символов) в строках одной матрицы на элементы в столбцах другой матрицы и сложения полученных произведений. Для выполнения умножения матриц нужно выполнить следующие шаги:

1. Убедиться, что размеры матриц равны, то есть число строк первой матрицы равно числу столбцов второй матрицы.
2. Обозначить размеры конечной матрицы, чтобы правильно расположить результаты умножения.
3. Найти первое скалярное произведение, умножая первый элемент первой строки на первый элемент первого столбца и складывая полученные значения. Повторить для всех элементов строки и столбца.
4. Найти второе скалярное произведение, умножая элементы второй строки первой матрицы на элементы первого столбца второй матрицы и складывая результаты.
5. Продолжать находить оставшиеся скалярные произведения, умножая и складывая элементы соответствующих строк и столбцов.
6. Результат произведения двух матриц имеет столько же строк, сколько первая матрица, и столько же столбцов, сколько вторая матрица.

Реализованная программа будет работать в несколько этапов:

1. Получает входные данные с диапазоном матриц и шагом, а также имя выходного файла и признак использования многопоточности.
2. Далее программа записывает время, чтобы узнать затраченное время на выделение массива.
3. Выделяет память, нужную для исходных матриц, а также матрицу, в которой будет помещен результат.
4. Сохраняет время, затраченное на выделение памяти.
5. Опять записывает текущее время, чтобы в дальнейшем узнать время, затраченное на заполнение матриц исходными данными.
6. Заполняет две исходные матрицы значениями. Происходит это в одном потоке в случае работы программы на CPU и в многопоточном режиме, в случае работы на GPU (если программа использует многопоточность).
7. Сохраняет время, которое было затрачено на заполнение исходных матриц.
8. Вновь запишет текущее время, чтобы узнать скорость перемножения матриц нашей программой.
9. Перемножает матрицы, используя указанный режим работы.
10. Вычисляет время, затраченное на перемножение матриц, записывает эти данные в вывод.

В программе будет реализована возможность выполнять умножения матриц с различным размером. Диапазон размеров задается при запуске программы, а также шаг, с которым программа должна идти слева направо.

На выходе будет получен *csv* файл, где будет время, затраченное на выделение памяти, заполнение её и само умножение и так для каждого значения размеров матрицы.

Далее можно загрузить эти данные и построить графики, а т.к. есть возможность указать как выполнять вычисления: с распараллеливанием или без, то мы сможем одной программой получить данные для двух случаев.

Так же особый интерес представляет работа *GPU* с памятью, т.к. при нехватке видеопамяти, будет выделяться память в ОЗУ, что негативно повлияет на скорость работы с памятью, поэтому будет написана программа, которая будет выделять память, но не будет работать с ней. Так мы поставим контроллер памяти и чип в условия, где им нужно понять и оценить использование памяти, чтобы нужные для вычислений данные находились в видеопамяти, а ненужные выгрузились в ОЗУ.

# 5 Сравнение эффективности распараллеливания

При запуске в многопоточном режиме, мы получили ожидаемую асимптотику от размера стороны матрицы – . Стоит отметить, что заполняли матрицы мы всегда в однопоточном режиме, при запуске на CPU. Графики зависимости времени выполнения от размера стороны матрицы представлен на рисунке 5.1.

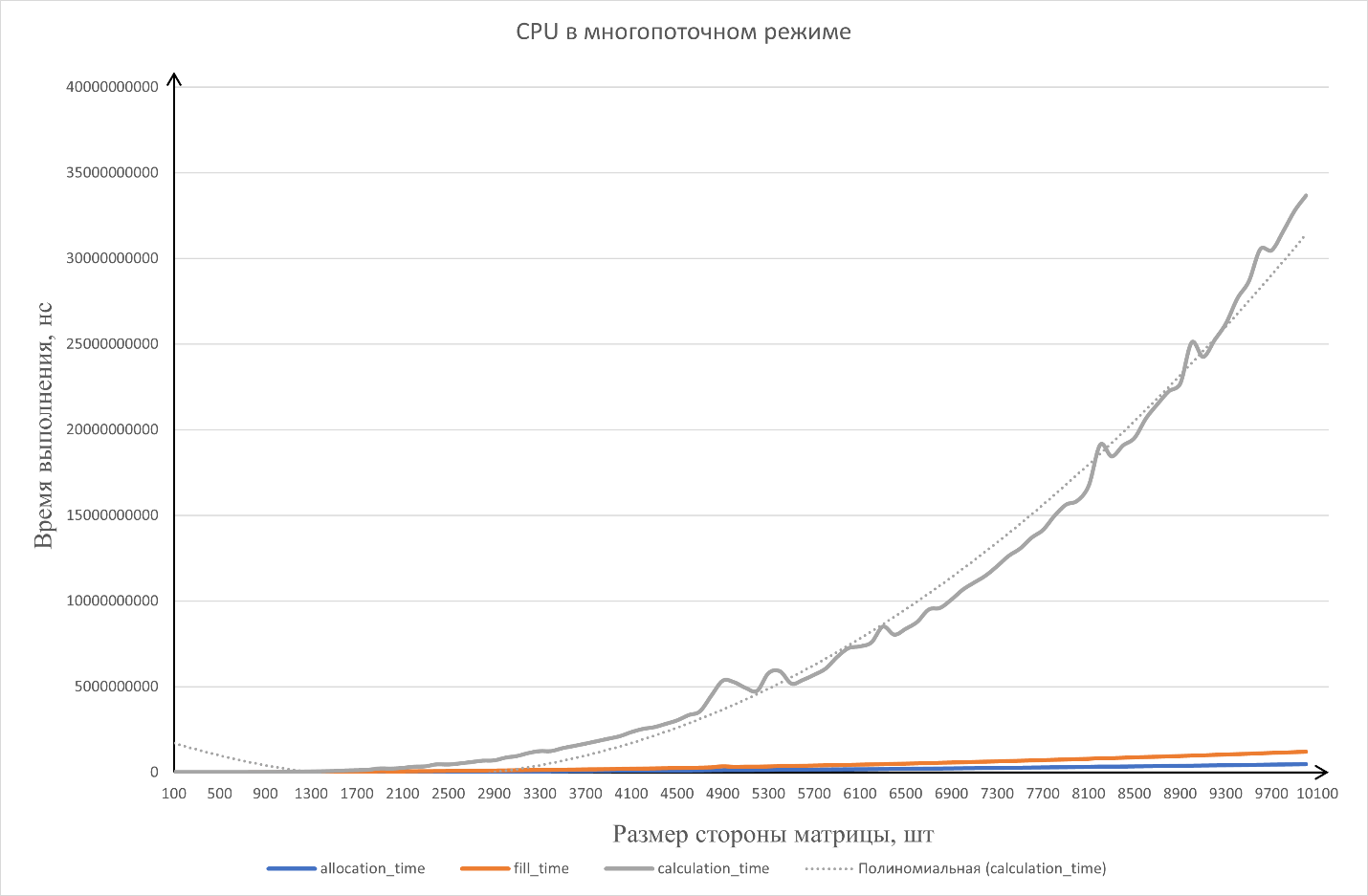


Рисунок 5.1 – График времени работы программы в многопоточном режиме на CPU

Далее была запущена эта же программа на таком же наборе входных данных, только выполнялись вычисления в одном потоке. Стоит упомянуть, что распараллеливание выполнялось за счет библиотеки *OpenMP*, т.к. здесь нет гонки данных, запись происходит только один раз в каждый элемент матрицы-ответа. А вот чтение одного и того же элемента исходных матриц происходит несколько раз, но т.к. это чтение, а не запись, то мы не блокируем другие задачи. Если же задача требовала более сложной работы с данными, то необходимо было бы менять способ распараллеливания со *static* на *dynamic*. Так библиотеке мы сообщаем, что операции внутри цикла могут иметь разную вычислительную сложность.

Результаты запуска, а именно время исполнения в зависимости от размера стороны матрицы изображена на рисунке 5.2.

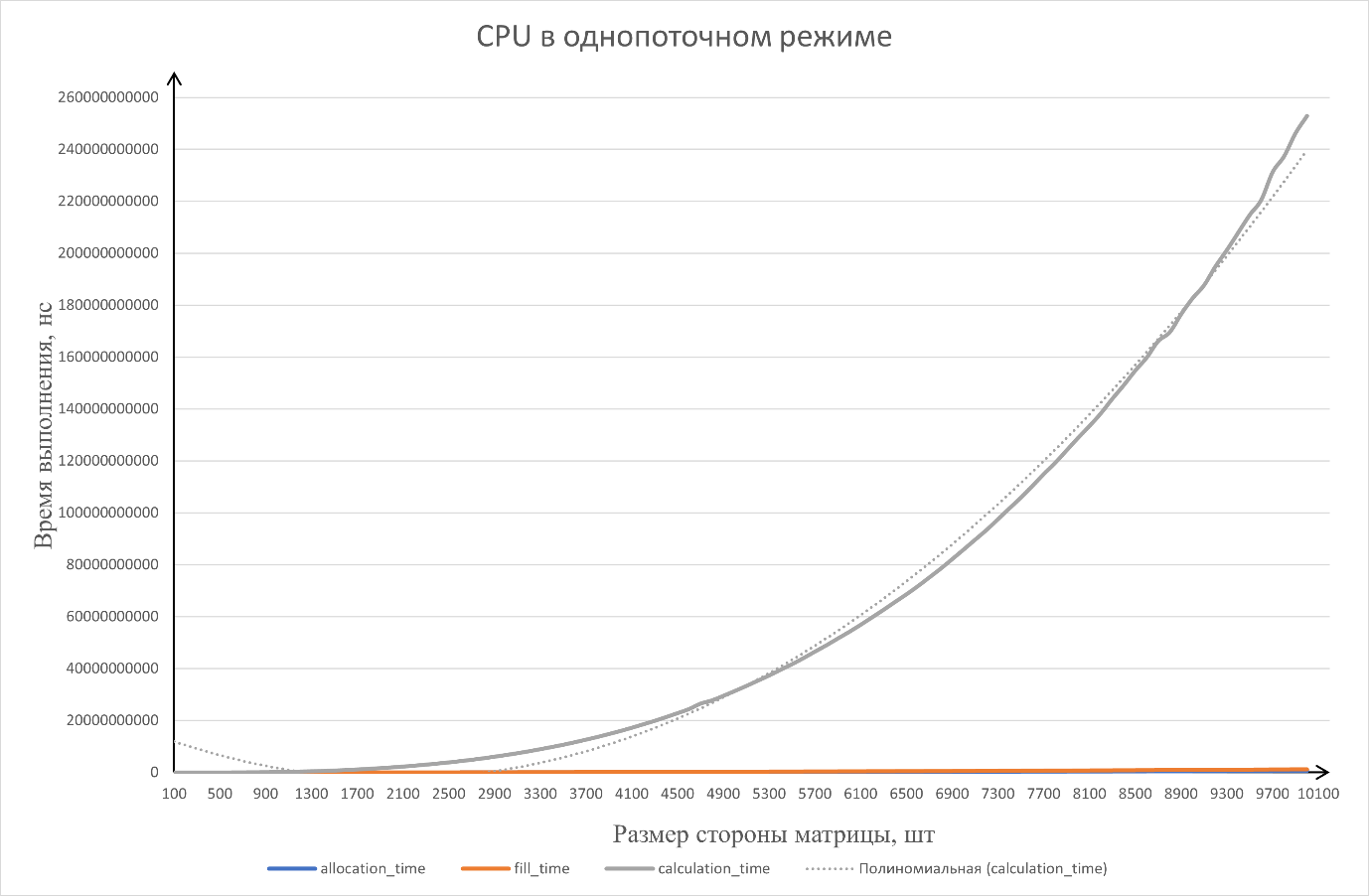


Рисунок 5.2 – График времени работы программы в однопоточном режиме на CPU

Далее была написана аналогичная программа, только использующая *GPU* с *CUDA*, схема взаимодействия будет аналогичной, только выполняться умножение будет на другом чипе. Стоит отметить, что в данном случае многопоточность – использование всех *SM* блоков, доступных на *GPU*, а в роли однопоточной работы будет исполнение на одном *SM* блоке, т.к. SM блок можно назвать независимой единицей в рамках ГПУ (он имеет свой контроллер для работы с памятью, свой кэш). В данном случае видеочип имеет 30 *SM* блоков.

Можно было бы провести сравнение скорости, при выполнении умножения в одной нити, но это можно назвать нерелевантным сравнением, т.к. мы не можем выполнять операции только на одном исполнительном устройством ЦПУ. Поэтому запуск в одной нити в этой работе не выполнялось.

Результаты запуска на *GPU* с распараллеливание изображены на рисунке 5.3.

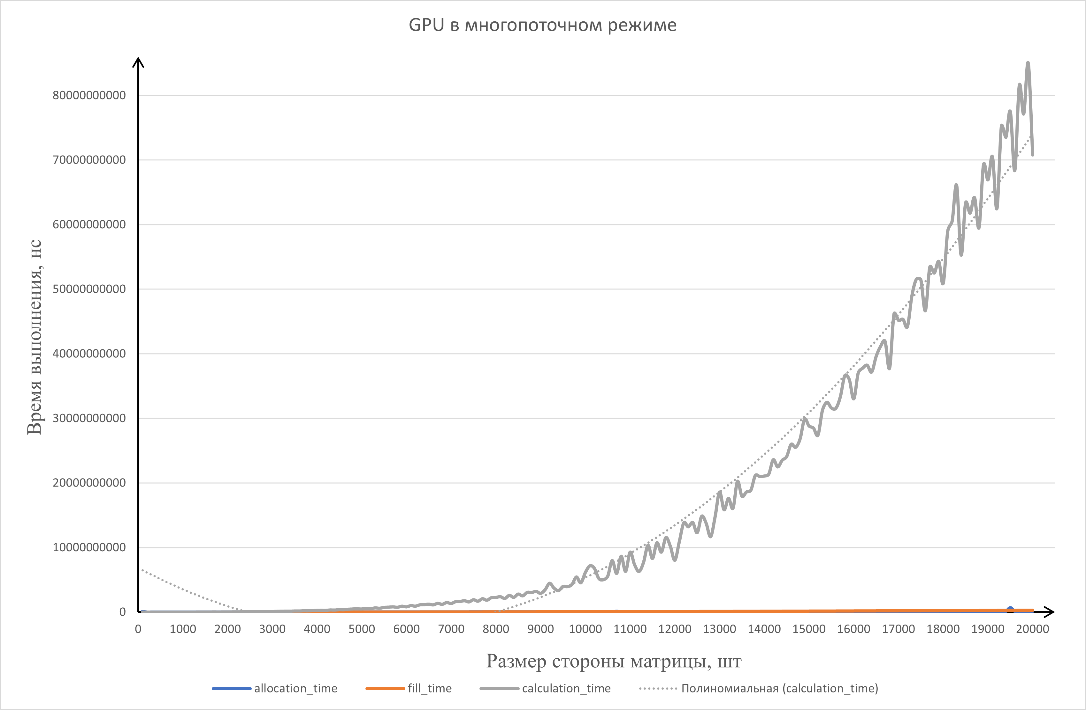


Рисунок 5.3 – График времени работы программы в многопоточном режиме на GPU

Результаты запуска на GPU на одном SM блоке, а именно зависимость времени работы от размера стороны матрицы изображены на рисунке 5.4.

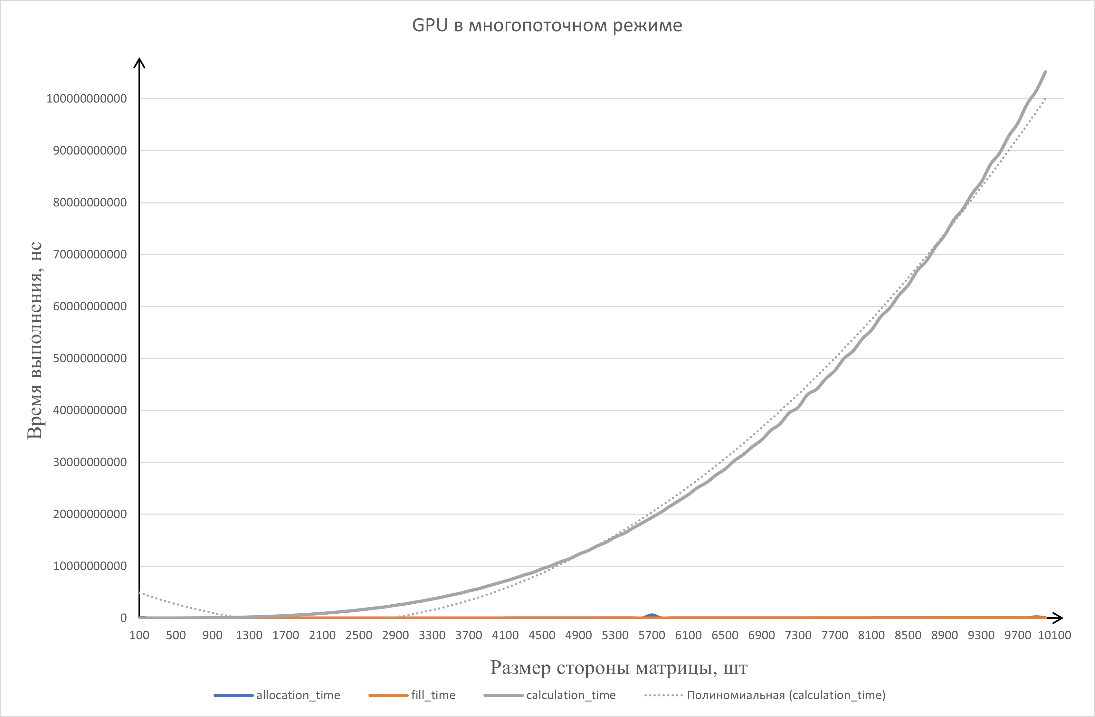


Рисунок 5.4 – График времени работы программы в однопоточном режиме на GPU

Так же сравним разницу между однопоточным и многопоточным режимом для ЦПУ – рисунок 5.5.

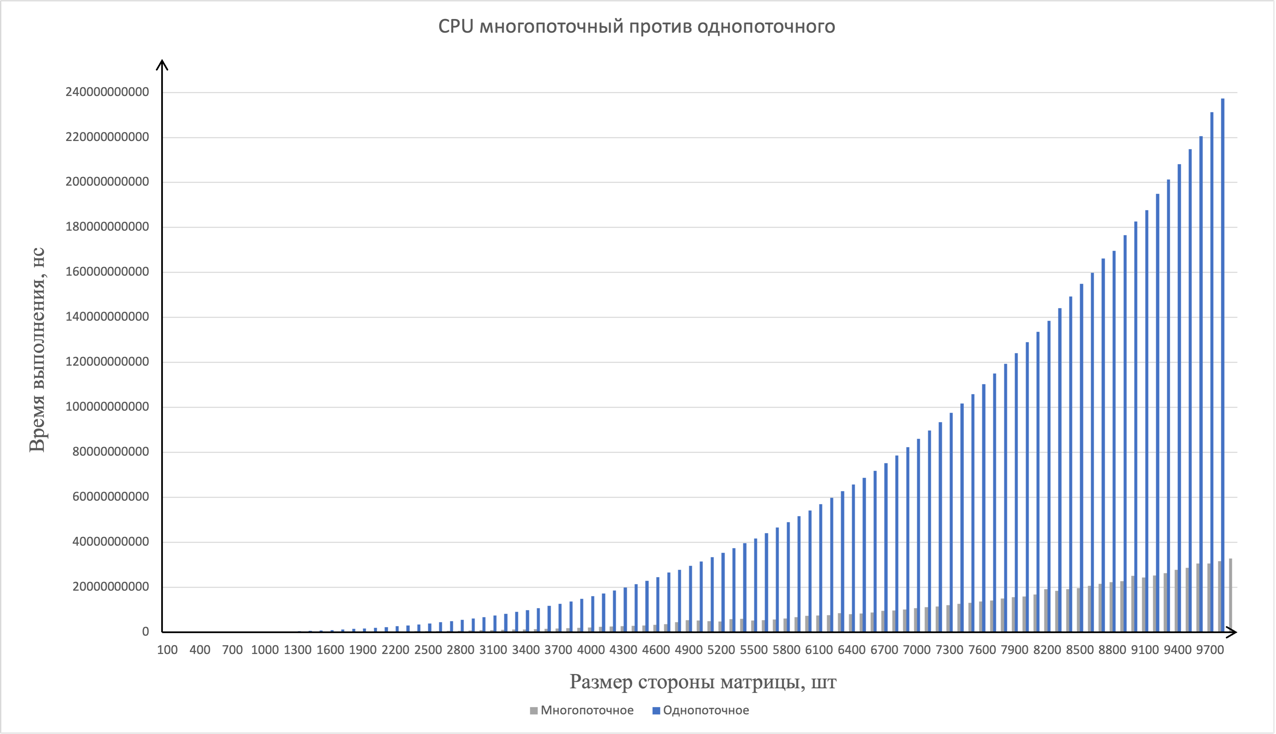


Рисунок 5.5 – Графики времени перемножения матриц в многопоточном и однопоточном режиме работы на *CPU*

Аналогичный график построим для сравнения разных режимов вычисления на GPU.

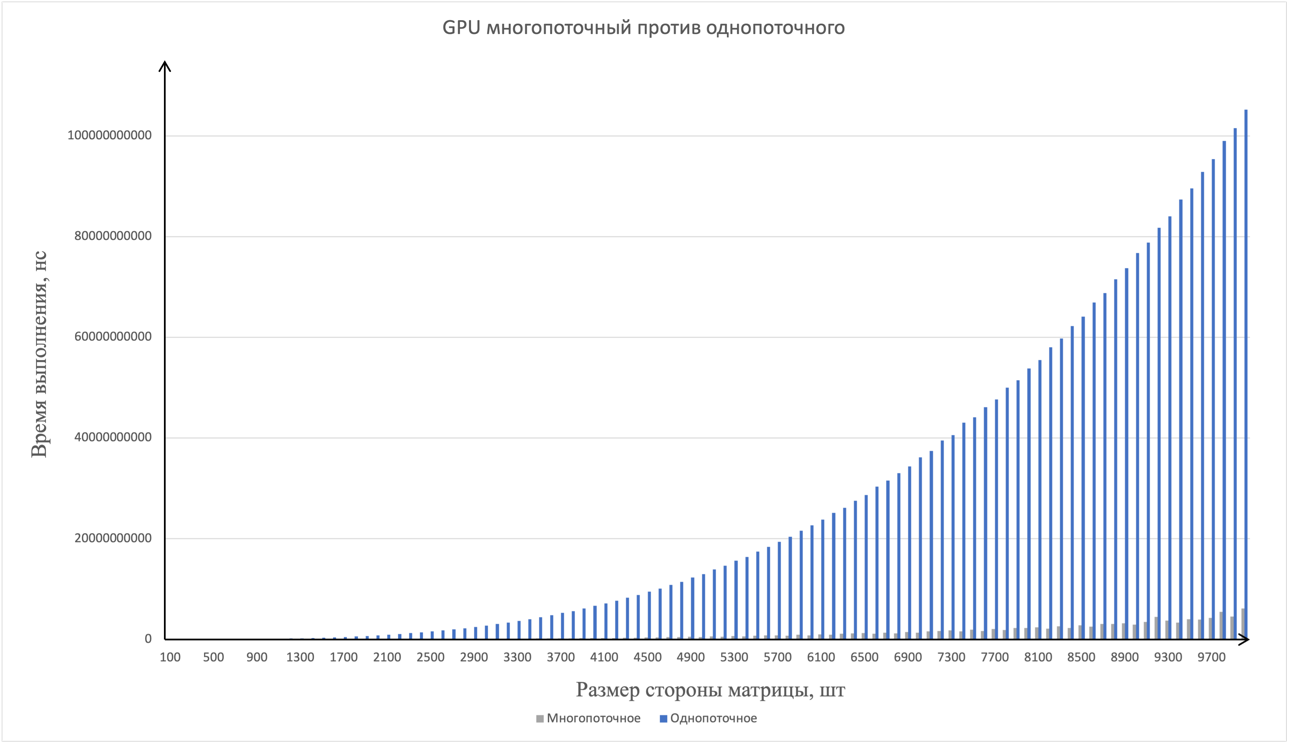


Рисунок 5.6

В заключении работы можно сделать вывод о том, что *GPU*, такие как *NVIDIA CUDA*, являются более эффективными для параллельных вычислений по сравнению с *CPU*. Благодаря специализированной архитектуре и большому количеству параллельных потоков, *GPU* способны обрабатывать огромные объемы данных, что делает их идеальными для таких задач как анализ изображений, обработка сигналов, симуляция физических процессов и систем искусственного интеллекта.

Так же можно отметить, что эффективность распараллеливания на ГПУ – выше, чем на ЦПУ. Это можно объяснить более простым устройством ГПУ, меньшим количеством конвейеров и переупорядочивания очереди исполнения. Это накладывает ограничения на написанный код, но при эффективном использовании это даёт значительное увеличение производительности.

В том числе было изучено, что контроллер памяти на видеокарте *NVIDIA* умеет анализировать использование памяти и выгружать ненужную видеопамять в ОЗУ. Это позволяет активным задачам меньше ожидать данных и быстрее исполнить инструкции.

Были отмечены особенности архитектуры современных центральных процессоров, как наличие многоуровневых конвейеров, использование нескольких исполнительных устройств в одном ядре. А также особенности строения архитектуры *GPU*, такие как: масштабирование блоков, которые сами по себе являются независимыми устройствами, использование разных исполнительных устройств для разных типов вычислений.

# ЗАКЛЮЧЕНИЕ

В заключении работы можно сделать вывод о том, что *GPU*, такие как *NVIDIA CUDA*, являются более эффективными для параллельных вычислений по сравнению с *CPU*. Благодаря специализированной архитектуре и большому количеству параллельных потоков, *GPU* способны обрабатывать огромные объемы данных, что делает их идеальными для таких задач как анализ изображений, обработка сигналов, симуляция физических процессов и систем искусственного интеллекта.

Так же можно отметить, что эффективность распараллеливания на ГПУ – выше, чем на ЦПУ. Это можно объяснить более простым устройством ГПУ, меньшим количеством конвейеров и переупорядочивания очереди исполнения. Это накладывает ограничения на написанный код, но при эффективном использовании это даёт значительное увеличение производительности.

В том числе было изучено, что контроллер памяти на видеокарте *NVIDIA* умеет анализировать использование памяти и выгружать ненужную видеопамять в ОЗУ. Это позволяет активным задачам меньше ожидать данных и быстрее исполнить инструкции.

Были отмечены особенности архитектуры современных центральных процессоров, как наличие многоуровневых конвейеров, использование нескольких исполнительных устройств в одном ядре. А также особенности строения архитектуры *GPU*, такие как: масштабирование блоков, которые сами по себе являются независимыми устройствами, использование разных исполнительных устройств для разных типов вычислений.

# СПИСОК ЛИТЕРАТУРЫ

1. C++ Reference [Электронный ресурс]. – Режим доступа: https://en.cppreference.com/w/ – Дата доступа: 10.09.2023.
2. NVIDIA TURING GPU ARCHITECTURE [Электронный ресурс]. – Режим доступа: https://images.nvidia.com/aem-dam/Solutions/design-visualization/technologies/turing‑architecture/NVIDIA‑Turing‑Architecture‑Whitepaper.pdf – Дата доступа: 15.09.2023.
3. NVIDIA CUDA Docs [Электронный ресурс]. – Режим доступа: https://docs.nvidia.com/cuda/doc/index.html – Дата доступа: 22.09.2023.
4. Базовые конструкции OpenMP [Электронный ресурс]. – Режим доступа: https://habr.com/ru/companies/intel/articles/85273/ – Дата доступа: 01.10.2023.
5. Базовые конструкции OpenMP [Электронный ресурс]. – Режим доступа: https://habr.com/ru/companies/intel/articles/85273/ – Дата доступа: 01.10.2023.
6. Valgrind Documentation [Электронный ресурс]. – Режим доступа: https://valgrind.org/docs/manual/quick-start.html – Дата доступа: 05.10.2023.
7. The user manual for NVIDIA Nsight Compute [Электронный ресурс]. – Режим доступа: https://docs.nvidia.com/nsight-compute/NsightCompute/index.html – Дата доступа: 14.10.2023.
8. Боресков, А. В. Основы работы с технологией CUDA / А. В. Боресков, А. А. Харламов. – Москва : ДМК Пресс, 2010. – 232 с.

# Приложение А

**(обязательное)**

**Ведомость**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *Перв. примен.* | ГУИР. ГУИР.153503.003 ПЗ | *Зона* | | *Обозначение* | | | | | *Наименование* | | | *Дополнительные*  *сведения* | | | |
|  | |  | | | | |  | | |  | | | |
|  | |  | | | | | *Текстовые документы* | | |  | | | |
|  | |  | | | | |  | | |  | | | |
|  | | *ГУИР КИ 1–40 04 01 003 ПЗ* | | | | | *Пояснительная записка* | | | *24 с.* | | | |
|  | |  | | | | |  | | |  | | | |
| *Справочный №* |  |  | |  | | | | |  | | |  | | | |
|  | |  | | | | |  | | |  | | | |
|  | |  | | | | | *Графические документы* | | |  | | | |
|  | |  | | | | |  | | |  | | | |
|  | | *ГУИР.153503.003.01* | | | | | *Функциональная схема* | | | *Формат А3* | | | |
|  | |  | | | | |  | | |  | | | |
|  | | *ГУИР.153503.003.02* | | | | | *Схема алгоритма* | | | *Формат А3* | | | |
|  | |  | |  | | | | |  | | |  | | | |
|  | | *ГУИР.153503.003.03 ПЛ* | | | | | *Графики сравнения* | | | *Формат А3* | | | |
|  | |  | | | | | *производительности* | | |  | | | |
|  | |  | | | | |  | | |  | | | |
|  | |  | | | | |  | | |  | | | |
| *Подпись и дата* |  |  | |  | | | | |  | | |  | | | |
|  | |  | | | | |  | | |  | | | |
|  | |  | | | | |  | | |  | | | |
|  | |  | | | | |  | | |  | | | |
| *Инв. № дубл.* |  |  | |  | | | | |  | | |  | | | |
|  | |  | | | | |  | | |  | | | |
|  | |  | | | | |  | | |  | | | |
| *Взам. Инв. №* |  |  | |  | | | | |  | | |  | | | |
|  | |  | | | | |  | | |  | | | |
|  | |  | | | | |  | | |  | | | |
| *Подпись и дата* |  |  | |  | | | | |  | | |  | | | |
|  | |  | | | | |  | | |  | | | |
|  |  | |  |  |  | *ГУИР КИ 1-40 04 01 003 ПЗ* | | | | | | | |
|  |  | |  |  |  |
| *Изм.* | *Лист* | | *№ докум.* | *Подп.* | *Дата* |
| *Инв. № подл.* |  | *Разраб.* | | | *Вергасов* |  |  | *Сравнение эффективности распараллеливания на CPU и GPU с CUDA*  *Ведомость курсового проекта* | | *Лит.* | | | | *Лист* | *Листов* |
| *Пров.* | | | *Марков* |  |  |  | *Т* | |  | *24* | *24* |
| *Рец.* | | |  |  |  | *Кафедра Информатики*  *группа 153503* | | | | | |
| *Н.контр.* | | | *Калиновская* |  |  |
| *Утв.* | | |  |  |  |

*Формат А4*

# Приложение А

**(обязательное)**

**Исходный код программы**

Листинг кода 1

#include <thrust/device\_vector.h>

#include <thrust/host\_vector.h>

#include <thrust/transform.h>

#include <thrust/functional.h>

#include <thrust/iterator/counting\_iterator.h>

#include <thrust/random.h>

#include <iostream>

#include <fstream>

#include <chrono>

#include <string>

#include <sstream>

#include <random>

struct GenRand {

private:

int seed;

public:

GenRand(int seed)

: seed(seed) {

}

\_\_device\_\_ float operator()(int idx) {

thrust::default\_random\_engine randEng(seed);

thrust::uniform\_int\_distribution<int> uniDist(1, 10);

randEng.discard(idx);

return uniDist(randEng);

}

};

\_\_global\_\_ void matrixMultiplicationKernel(const float\* A, const float\* B, float\* C, size\_t N) {

int row = blockIdx.y \* blockDim.y + threadIdx.y;

int col = blockIdx.x \* blockDim.x + threadIdx.x;

if (row < N && col < N) {

float sum = 0.0f;

for (int k = 0; k < N; ++k) {

sum += A[row \* N + k] \* B[k \* N + col];

}

C[row \* N + col] = sum;

}

}

void matrixMultiplication(const thrust::device\_vector<float>& A, const thrust::device\_vector<float>& B,

thrust::device\_vector<float>& C, int N, bool mp) {

dim3 threadsPerBlock, blocksPerGrid;

if (mp) {

threadsPerBlock = dim3(16, 16, 1);

blocksPerGrid =

dim3((N + threadsPerBlock.x - 1) / threadsPerBlock.x, (N + threadsPerBlock.y - 1) / threadsPerBlock.y);

} else {

threadsPerBlock = dim3(1, 1, 1);

blocksPerGrid =

dim3((N + threadsPerBlock.x - 1) / threadsPerBlock.x, (N + threadsPerBlock.y - 1) / threadsPerBlock.y);

}

matrixMultiplicationKernel<<<blocksPerGrid, threadsPerBlock>>>(

thrust::raw\_pointer\_cast(A.data()), thrust::raw\_pointer\_cast(B.data()), thrust::raw\_pointer\_cast(C.data()), N);

}

void printProgress(double progress) {

static int barWidth = 70;

std::cout << "\r[";

int pos = barWidth \* progress;

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

if (i < pos) {

std::cout << "=";

} else if (i == pos) {

std::cout << ">";

} else {

std::cout << " ";

}

}

std::cout << "] " << int(progress \* 100.0) << " %";

std::cout.flush();

}

int main(int argc, const char\*\* argv) {

if (argc != 6) {

std::cout << "You should specify start size, end size, step size of matrix and output filename and use "

"parallel or not!"

<< std::endl;

return -1;

}

srand(time(0));

std::ostringstream output;

output << "count,allocation\_time,fill\_time,calculation\_time\n";

size\_t left = std::atoi(argv[1]), right = std::atoi(argv[2]), step = std::atoi(argv[3]);

size\_t block\_count = 0, current = 0;

double progress = 0;

for (size\_t N = left; N <= right; N += step) {

block\_count += N \* N;

}

for (size\_t N = left; N <= right; N += step) {

progress = static\_cast<double>(current) / static\_cast<double>(block\_count);

printProgress(progress);

auto start = std::chrono::system\_clock::now();

thrust::device\_vector<float> A(N \* N);

thrust::device\_vector<float> B(N \* N);

thrust::device\_vector<float> C(N \* N);

auto end = std::chrono::system\_clock::now();

auto elapsed = std::chrono::duration\_cast<std::chrono::nanoseconds>(end - start).count();

output << N << ',' << elapsed << ',';

start = std::chrono::system\_clock::now();

thrust::transform(thrust::make\_counting\_iterator(0ULL), thrust::make\_counting\_iterator(N \* N), A.begin(),

GenRand(rand()));

thrust::transform(thrust::make\_counting\_iterator(0ULL), thrust::make\_counting\_iterator(N \* N), B.begin(),

GenRand(rand()));

end = std::chrono::system\_clock::now();

elapsed = std::chrono::duration\_cast<std::chrono::nanoseconds>(end - start).count();

output << elapsed << ',';

bool use\_mp = std::atoi(argv[5]);

start = std::chrono::system\_clock::now();

matrixMultiplication(A, B, C, N, use\_mp);

cudaDeviceSynchronize();

end = std::chrono::system\_clock::now();

elapsed = std::chrono::duration\_cast<std::chrono::nanoseconds>(end - start).count();

output << elapsed << '\n';

current += N \* N;

}

printProgress(progress);

std::cout << std::endl;

std::ofstream output\_file(std::string(argv[4]) + ".csv");

output\_file << output.str();

return 0;

}

Листинг кода 2.

#include <iostream>

#include <vector>

#include <chrono>

#include <random>

#include <fstream>

#include <sstream>

#include <omp.h>

void multiplyOMP(const std::vector<std::vector<float>>& A, const std::vector<std::vector<float>>& B,

std::vector<std::vector<float>>& C) {

#pragma omp parallel for num\_threads(12)

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

for (int k = 0; k < B.size(); k++) {

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

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

}

}

}

}

void multiply(const std::vector<std::vector<float>>& A, const std::vector<std::vector<float>>& B,

std::vector<std::vector<float>>& C) {

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

for (int k = 0; k < B.size(); k++) {

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

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

}

}

}

}

void printProgress(double progress) {

static int barWidth = 70;

std::cout << "\r[";

int pos = barWidth \* progress;

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

if (i < pos) {

std::cout << "=";

} else if (i == pos) {

std::cout << ">";

} else {

std::cout << " ";

}

}

std::cout << "] " << int(progress \* 100.0) << " %";

std::cout.flush();

}

int main(int argc, const char\*\* argv) {

if (argc != 6) {

std::cout << "You should specify start size, end size, step size of matrix and output filename and use "

"parallel or not!"

<< std::endl;

return -1;

}

std::ostringstream output;

output << "count,allocation\_time,fill\_time,calculation\_time\n";

size\_t left = std::atoi(argv[1]), right = std::atoi(argv[2]), step = std::atoi(argv[3]);

size\_t block\_count = 0, current = 0;

double progress = 0;

for (size\_t N = left; N <= right; N += step) {

block\_count += N \* N;

}

for (size\_t N = left; N <= right; N += step) {

progress = static\_cast<double>(current) / static\_cast<double>(block\_count);

printProgress(progress);

std::mt19937 random(time(0));

std::uniform\_real\_distribution<float> dist(1, 10);

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

std::vector<std::vector<float>> A(N, std::vector<float>(N));

std::vector<std::vector<float>> B(N, std::vector<float>(N));

std::vector<std::vector<float>> C(N, std::vector<float>(N));

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

auto elapsed = std::chrono::duration\_cast<std::chrono::nanoseconds>(end\_serial - start\_serial).count();

output << N << ',' << elapsed << ',';

start\_serial = std::chrono::high\_resolution\_clock::now();

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

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

A[i][j] = dist(random);

B[i][j] = dist(random);

}

}

end\_serial = std::chrono::high\_resolution\_clock::now();

elapsed = std::chrono::duration\_cast<std::chrono::nanoseconds>(end\_serial - start\_serial).count();

output << elapsed << ',';

bool use\_mp = std::atoi(argv[5]);

start\_serial = std::chrono::high\_resolution\_clock::now();

if (use\_mp) {

multiplyOMP(A, B, C);

} else {

multiply(A, B, C);

}

end\_serial = std::chrono::high\_resolution\_clock::now();

elapsed = std::chrono::duration\_cast<std::chrono::nanoseconds>(end\_serial - start\_serial).count();

output << elapsed << '\n';

current += N \* N;

}

printProgress(progress);

std::cout << std::endl;

std::ofstream output\_file(std::string(argv[4]) + ".csv");

output\_file << output.str();

return 0;

}

Листинг кода 3.

#include "cuda\_runtime.h"

#include <thrust/device\_vector.h>

#include <thrust/functional.h>

#include <thrust/iterator/counting\_iterator.h>

#include <thrust/random.h>

#include <iostream>

struct GenRand {

private:

int seed;

public:

GenRand(int seed)

: seed(seed) {

}

\_\_device\_\_ float operator()(int idx) {

thrust::default\_random\_engine randEng(seed);

thrust::uniform\_int\_distribution<int> uniDist(1, 10);

randEng.discard(idx);

return uniDist(randEng);

}

};

int main(int argc, const char\*\* argv) {

if (argc != 2) {

std::cout << "Enter how much bytes to allocate!" << std::endl;

}

size\_t count = std::atoll(argv[1]);

thrust::device\_vector<std::uint8\_t> garbage(count, 0);

thrust::transform(thrust::make\_counting\_iterator(0ULL), thrust::make\_counting\_iterator(garbage.size()),

garbage.begin(), GenRand(0));

std::cout << "Allocated " << count << " bytes" << std::endl;

std::string input;

std::cin >> input;

return 0;

}