Министерство образования и науки Российской Федерации
федеральное государственное автономное образовательное учреждение
высшего образования
**«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»**

Институт Кибернетики
Направление подготовки 09.04.01 Информатика и вычислительная техника
Кафедра Информационных систем и технологий

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

Тема работы
Применение параллельного программирования технологии CUDA в методах случайной оптимизации

УДК 004.4.032.24:519.85

Студент

Группа	ФИО	Подпись	Дата
8ВМ5Б	Нефедова А.А.		

Руководитель

Должность	ФИО	Ученая степень, звание	Подпись	Дата
Доцент	Рейзлин В.И.	к.ф.-м.н.		

КОНСУЛЬТАНТЫ:

По разделу «Финансовый менеджмент, ресурсоэффективность и ресурсосбережение»

Должность	ФИО	Ученая степень, звание	Подпись	Дата
Доцент	Конотопский В.Ю.	к.э.н.		

По разделу «Социальная ответственность»

Должность	ФИО	Ученая степень, звание	Подпись	Дата
Ассистент	Акулов П.А.			

ДОПУСТИТЬ К ЗАЩИТЕ:

Зав. кафедрой	ФИО	Ученая степень, звание	Подпись	Дата
ИСТ	Мальчуков А.Н.	к.т.н.		

Томск – 2017 г.

ЗАПЛАНИРОВАННЫЕ РЕЗУЛЬТАТЫ ОБУЧЕНИЯ ПО ООП

Код результата	Результат обучения (выпускник должен быть готов)
Профессиональные компетенции	
Р1	Применять глубокие естественнонаучные и математические знания для решения научных и инженерных задач в области информатики и вычислительной техники.
Р2	Применять глубокие специальные знания в области информатики и вычислительной техники для решения междисциплинарных инженерных задач.
Р3	Ставить и решать инновационные задачи инженерного анализа, связанные с созданием аппаратных и программных средств информационных и автоматизированных систем, с использованием аналитических методов и сложных моделей.
Р4	Выполнять инновационные инженерные проекты по разработке аппаратных и программных средств автоматизированных систем различного назначения с использованием современных методов проектирования, систем автоматизированного проектирования, передового опыта разработки конкурентно способных изделий.
Р5	Планировать и проводить теоретические и экспериментальные исследования в области проектирования аппаратных и программных средств автоматизированных систем с использованием новейших достижений науки и техники, передового отечественного и зарубежного опыта. Критически оценивать полученные данные и делать выводы.
Р6	Осуществлять авторское сопровождение процессов проектирования, внедрения и эксплуатации аппаратных и программных средств автоматизированных систем различного назначения.
Универсальные компетенции	
Р7	Использовать глубокие знания по проектному менеджменту для ведения инновационной инженерной деятельности с учетом юридических аспектов защиты интеллектуальной собственности.
Р8	Осуществлять коммуникации в профессиональной среде и в обществе в целом, активно владеть иностранным языком, разрабатывать документацию, презентовать и защищать результаты инновационной инженерной деятельности, в том числе на иностранном языке.
Р9	Эффективно работать индивидуально и в качестве члена и руководителя группы, в том числе междисциплинарной и международной, при решении инновационных инженерных задач.
Р10	Демонстрировать личную ответственность и ответственность за работу возглавляемого коллектива, приверженность и готовность следовать профессиональной этике и нормам ведения инновационной инженерной деятельности. Демонстрировать глубокие знания правовых, социальных, экологических и культурных аспектов инновационной инженерной деятельности.
Р11	Демонстрировать способность к самостоятельному обучению, непрерывному самосовершенствованию в инженерной деятельности, способность к педагогической деятельности.

Министерство образования и науки Российской Федерации

федеральное государственное автономное образовательное учреждение
высшего образования
**«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»**

Институт кибернетики
Направление подготовки 09.04.01 Информатика и вычислительная техника
Кафедра Информационных систем и технологий

УТВЕРЖДАЮ:
Зав. Кафедрой

_____ _____ _____
(Подпись) (Дата) (Ф.И.О.)

ЗАДАНИЕ
на выполнение выпускной квалификационной работы

В форме:

Магистерской диссертации

(бакалаврской работы, дипломного проекта/работы, магистерской диссертации)

Студенту:

Группа	**ФИО**
8ВМ5Б	Нефедова Анастасия Алексеевна

Тема работы:

Проектирование инфраструктуры и реализация распределенного информационно-вычислительного кластера на базе персональных компьютеров.	
Утверждена приказом директора (дата, номер)	

Срок сдачи студентом выполненной работы:	

ТЕХНИЧЕСКОЕ ЗАДАНИЕ:

Исходные данные к работе *(наименование объекта исследования или проектирования; производительность или нагрузка; режим работы (непрерывный, периодический, циклический и т. д.); вид сырья или материал изделия; требования к продукту, изделию или процессу; особые требования к особенностям функционирования (эксплуатации) объекта или изделия в плане безопасности эксплуатации, влияния на окружающую среду, энергозатратам; экономический анализ и т. д.).*	Применение параллельного программирования технологии CUDA в методах случайной оптимизации.

Перечень подлежащих исследованию, проектированию и разработке вопросов *(аналитический обзор по литературным источникам с целью выяснения достижений мировой науки техники в рассматриваемой области; постановка задачи исследования, проектирования, конструирования; содержание процедуры исследования, проектирования, конструирования; обсуждение результатов выполненной работы; наименование дополнительных разделов, подлежащих разработке; заключение по работе).*	1) Введение 2) Обзор литературы 3) Техника общий вычислений на графическом процессоре 4) Архитектура NVidia CUDA 5) Решение задач многомерной оптимизации 6) Постановка компьютерного эксперимента 7) Финансовый менеджмент, ресурсоэффективность и ресурсосбережение 8) Социальная ответственность 9) Заключение
Перечень графического материала *(с точным указанием обязательных чертежей)*	

Консультанты по разделам выпускной квалификационной работы

Раздел	Консультант
Финансовый менеджмент, ресурсоэффективность и ресурсосбережение	Доцент Конотопский В.Ю.
Социальная ответственность	Ассистент Акулов П.А.
Раздел на иностранном языке	Старший преподаватель Кудряшова А.В.

Названия разделов, которые должны быть написаны на русском и иностранном языках:

Дата выдачи задания на выполнение выпускной квалификационной работы по линейному графику	

Задание выдал руководитель:

Должность	ФИО	Ученая степень, звание	Подпись	Дата
Доцент	Рейзлин В.И.	к.ф.-м.н.		

Задание принял к исполнению студент:

Группа	ФИО	Подпись	Дата
8ВМ5Б	Нефедова А.А.		

Реферат

Выпускная квалификационная работа содержит 120 стр., 21 рис., 8 табл., 32 источника, 1 приложение.

Ключевые слова: Graphics processing unit, Compute Unified Device Architecture, многомерная многоэкстремальная оптимизация, алгоритм случайного поиска,

Цель работы: с помощью параллельных вычислений максимально ускорить процесс получения экстремума в задачах многомерной оптимизации и тем самым показать преимущества использования вышеуказанной технологии по сравнению с технологией последовательных вычислений

В процессе исследования проводилось изучение алгоритмов случайного поиска, функций многомерной оптимизации, основные принципы разработки алгоритмов для GPU.

В результате исследования было разработано приложение, реализующее методы многомерной оптимизации. В приложении есть возможность добавления новых функций.

Определения, обозначения, сокращения, нормативные ссылки

В данной работе применены следующие термины:

General-purpose graphics processing units (GPGPU): Техника использования графического процессора видеокарты для общих вычислений, которые обычно проводит центральный процессор

Graphics processing unit (GPU): Графический процессор

Central processing unit (CPU): Центральный процессор

Single Instruction stream Multiple Data stream (SIMD): Метод вычислений, при котором ядра GPU выполняют один и тот же набор инструкций для каждого экземпляра данных.

Compute Unified Device Architecture (CUDA): Программно-аппаратная архитектура, разработанная компанией NVIDIA, позволяющая производить вычисления с использованием графических процессоров NVIDIA, поддерживающих технологию GPGPU (произвольных вычислений на видеокартах).

Многоэкстремальная задача: Нелинейная задача математического программирования, целевая функция которой может иметь как глобальный, так и локальные оптимумы.

Мультимодальная функция: Функция, имеющая более одного экстремума.

Функция Розенброка: Невыпуклая функция, используемая для оценки производительности алгоритмов оптимизации.

Функция Химмельблау: Мультимодальная функция двух переменных, используемая для проверки эффективности алгоритмов оптимизации.

Оглавление

Обзор литературы

За последние десятилетия во всех областях науки значительно выросла потребность в обработке огромного количества данных. Для этого необходимы мощные вычислительные системы. Из-за того, что стало невозможно увеличивать тактовую частоту процессора, появились многоядерные архитектуры, которые потребовали навыков параллельного программирования. В связи с нехваткой высококвалифицированных специалистов стало необходимым создание эффективных высокоуровневых технологий программирования. В результате стало происходить бурное развитие суперкомпьютерных технологий по всему миру.

Пиковая производительность суперкомпьютеров TOP500 [1] растет с большой скоростью. В первой тройке рейтинга TOP500 на ноябрь 2016 года: Sunway TaihuLight с производительностью 93 петафлопс. Эта монструозная система включает почти 11 миллионов вычислительных ядер, а её потребляемая мощность достигает 15,4 МВт. За ней следует Tianhe-2 с производительностью 34 петафлопс. По энергоэффективности она существенно уступает лидеру. При количестве ядер 3,1 млн и в три раза меньшей производительности Tianhe-2 потребляет 17,8 МВт электроэнергии. Третью позицию занял суперкомпьютер Titan из США с производительностью 17,6 петафлопс. Также системы США завоевали четвёртую и пятую строчки рейтинга - Sequoia и Cori. Что касается энергоэффективности, то лидером является система DXG SATURNV, разработанная компанией NVIDIA. Её особенностью являются новые ускорители P100, которые обеспечили 3,3-петафлопсному кластеру энергоэффективность 9,46 гигафлопс на ватт. Второе место занимает система Piz Daint с показателем 7,45 гигафлопс на ватт. Из 86 систем, использующих сопроцессоры в виде видеочипов, 60 оснащены NVIDIA GPU, 21 - Intel Xeon Phi, 1 — AMD FirePro, 1 — PEZY-технологией, 3 — одновременно используют NVIDIA GPU и Xeon Phi. Процессоры Intel

используются в 92,4 % суперкомпьютеров. Количество систем на базе IBM Power составило 22.

Графические ускорители (GPU) стали использоваться для неграфических вычислений, что позволило достичь ускорения работы приложений в десятки, а то и в сотни раз. Для написания программ, которые предназначены для выполнения на графическом процессоре, были разработаны специальные языки программирования: C–CUDA, Fortran–CUDA [2-5], OpenCL [6], OpenACC [7] и другие.

Существует много успешных примеров использования технологии параллельных вычислений для увеличения производительности программных продуктов, и, как следствие, увеличение числа их пользователей. По данным NVIDIA[8], на финансовом рынке компания Numerix анонсировала технологию параллельных вычислений в новом приложении анализа риска контрагентов и достигла ускорения работы в 18 раз. Более 400 финансовых институтов используют в своей деятельности программное обеспечение Numerix. Почти все основные приложения для работы с видео уже используют CUDA-технологию для ускорения вычислений, включая продукты от Elemental Technologies, MotionDSP и LoiLo. Сейчас CUDA ускоряет AMBER – программу для моделирования молекулярной динамики, которая используется более 60 000 исследователей в академической среде и фармацевтическими компаниями по всему миру для сокращения сроков создания лекарственных препаратов.

Показателем широкого применения GPGPU технологии является постоянный рост использования графических ускорителей в компаниях из списка Fortune 500, таких как Schlumberger и Chevron в энергетическом секторе, а также BNP Paribas в секторе банковских услуг.

Основываясь на многолетнем опыте в разработке и использовании технологии параллельных вычислений, компания Simmakers предлагает услуги по внедрению этой технологии для компаний из различных отраслей. Это позволит повысить вычислительную производительность программных

продуктов, что увеличит их конкурентоспособность в современных условиях развития рынка программного обеспечения.

1. Техника общих вычислений на графическом процессоре

1.1 General - purpose graphics processing units.

GPGPU (**англ. *G**eneral-**p**urpose **g**raphics **p**rocessing **u**nits* «GPU общего назначения») - техника использования графического процессора видеокарты для общих вычислений, которые обычно проводит центральный процессор, т.е. это набор аппаратных и программных технологий, позволяющие использовать графические процессоры, для ускорения многих не графических приложений [1]. Graphics Processing Units являются высокоэффективными многоядерными процессорами, способными к сложным вычислениям и очень высокой пропускной способности данных. GPGPU является результатом развития шейдерных программ и специализированных на них языков программирования высокого уровня, таких как, Cg, GLSL и HLSL.

Изначальное предназначение графических процессоров направлено на решение узкого круга задач, заключающегося в обработке графических данных. Исходя из этого, архитектуры центрального и графического процессоров существенно различаются (устройство CPU и GPU смотри рисунок 1.1). Например, основой видеочипов NVIDIA является восьмидесяти ядерный мультипроцессор, располагающий несколькими тысячами регистров.

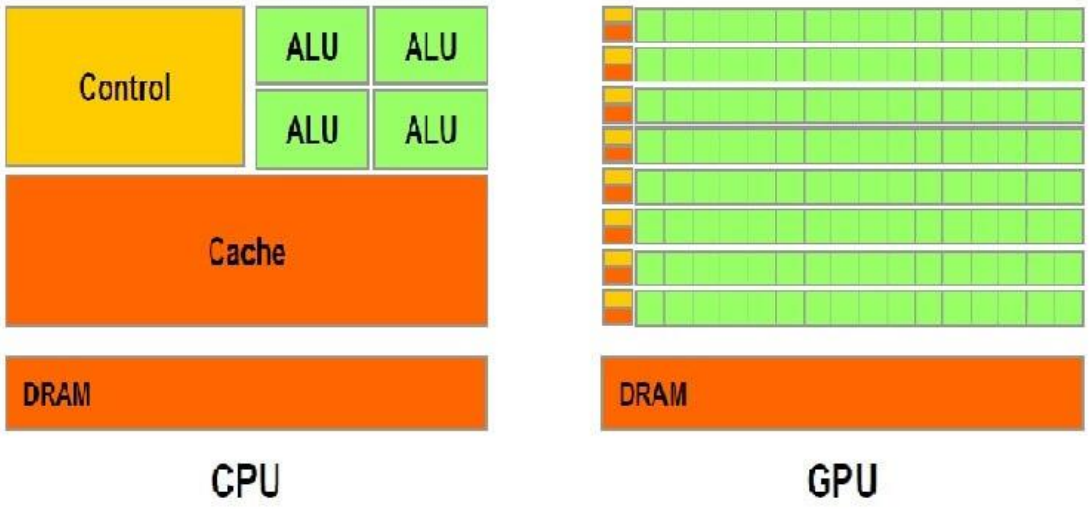

Рисунок 1.1 - Устройство GPU и CPU. Слева – CPU, справа GPU

Графический процессор поддерживает метод вычислений SIMD (Single Instruction stream Multiple Data stream) который означает, что ядра GPU выполняют один и тот же набор инструкций для каждого экземпляра данных. Большинство графических алгоритмов используют именно такой подход, как наиболее эффективное средство для задач графической визуализации. Такой модуль, преобразующий поток входных данных в выходные, называется kernel – ядро [2].

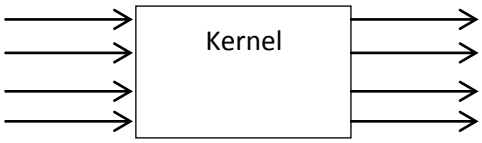

Рисунок 1.2 - Работа SIMD-архитектуры

Для обобщения основных различий между архитектурами центрального и графического процессоров стоит отметить, что основной задачей CPU является последовательное исполнение одного потока инструкций с максимальной производительностью, тогда как конструкция GPU предполагает выполнение наибольшего количества параллельных потоков одновременно.

С целью достижения максимальной производительности центрального процессора разработчики повышают число задач, выполняемых за один такт. Однако, поток выполняемых CPU команд неизменно является последовательным, что исключает всякую возможность при текущей архитектуре увеличить скорость исполнения алгоритмов пропорциональным увеличением количества ядер.

На каждом этапе графического конвейера данные друг от друга не зависят и могут обрабатываться параллельно, благодаря тому, что графический процессор занят обработкой графических примитивов. В отличие от последовательного потока инструкций для центрального

процессора, графический процессор использует исполнительные блоки, которые легко загрузить.

Принцип организации доступа к памяти у центрального и графического процессоров тоже существенно различается. В отличие от CPU, GPU осуществляет доступ последовательно, что означает для каждого момента времени если было обращение к ячейке памяти, далее будут задействованы данные идущие следом. Закономерно, что запись данных осуществляется по тому же принципу. Помимо прочего, колоссальный объем данных характеризует большинство задач, решаемых на графическом процессоре. Проблема задержки доступа к данным на CPU решается посредствам технологии кэширования и предсказания ветвления кода. Решение той же задачи средствами GPU выглядит совершенно иначе - если в ожидании доступа к памяти один из параллельно исполняемых процессов приостановил выполнение, то видеочип переключается на другой процесс, уже имеющий все необходимые для дальнейшего выполнения данные. Пропускная способность памяти видеокарт, кстати, имеет существенно большую пропускную способность, нежели оперативная. Механизм кэширования для уменьшения задержки доступа к памяти используется и в GPU. Однако если в центральном процессоре кэш занимает внушительную долю чипа, то GPU отводит под нужды кэша всего 128-256 килобайт, что в свою очередь предпринимается скорее для повышения пропускной возможности.

Многопоточность в графических процессорах также реализована и на аппаратном уровне. На CPU же ее задействовать не целесообразно, так как каждое переключение между потоками неизбежно ведет к значительным временным задержкам продолжительностью в несколько сотен тактов. Вдобавок, ядро CPU способно исполнять всего несколько одновременных потоков (1-2). GPU, в свою очередь, способен мгновенно переключаться между потоками, а каждому ядру под силу обработка до 1024 потоков.

CPU, будучи универсальным вычислительным устройством, эффективно справляется с целым спектром различных задач, в то время как предназначение графических процессоров гораздо более узконаправленное. В задачах с множественными ветвлениями и переходами графический процессор не столь эффективен как центральный.

Поэтому следует помнить, что технология GPGPU актуальна при обработке больших объемов данных. На ранней стадии развития этой технологии, производительность GPU превосходила в 10 раз CPU.

Производители видеокарт увидели большую перспективу в GPGPU, поэтому стремительно стали развиваться в данном направлении.

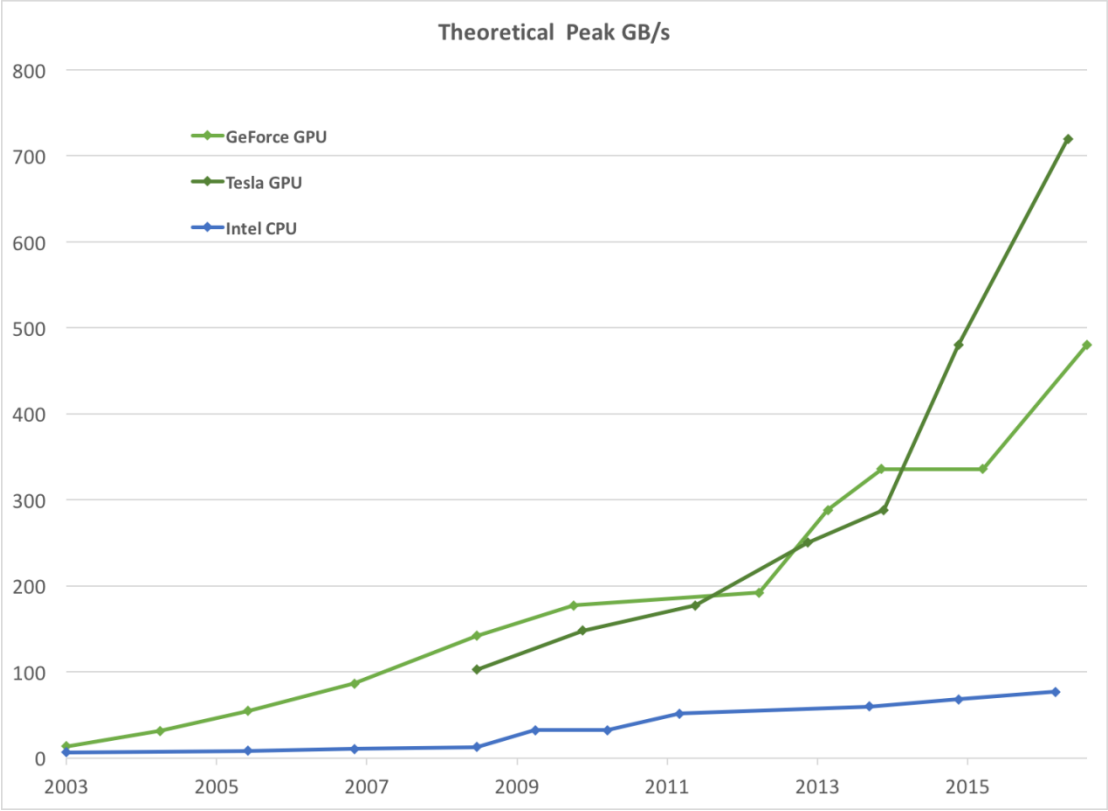

Рисунок 1.3 - Динамика роста производительности CPU и GPU

На сегодняшний день графические ускорители работают в 2 раза быстрее. Например, в марте 2012 г. была презентована видеокарта NVIDIA GeForce GTX 690 производительностью 2488 GFlops. В марте 2011г. была презентована ATIRadeon HD 6990 производительностью 1500 GFlops. Ни один

пользовательский CPU не может сравниться с GPU, учитывая, что производительность процессоров Intel не выходит за 300 GFlops.

Поэтому использование технологии GPGPU как никогда актуально для разработчиков. В связи с этим, производители видеокарт еще в 2003 году начали активно развиваться в области неграфических вычислений. Таким образом, результатом работы компании nVidia, показатели производительности которой наиболее высокие, стала NVIDIA CUDA.

1.2 Программная архитектура NVIDIA CUDA

CUDA (англ. Compute Unified Device Architecture) - программноаппаратная архитектура, позволяющая производить вычисления с использованием графических процессоров NVIDIA [3], поддерживающих технологию GPGPU (произвольных вычислений на видеокартах).

В архитектуру CUDA включен унифицированный шейдерный конвейер, который выполняет вычисления общего назначения и задействует любые арифметически-логические устройства, входящее в микросхему. Целью CUDA является создать GPU, который бесперебойно справляется с вычислениями общего назначения, а не только с традиционными задачами компьютерной графики, поэтому исполняющим устройствам GPU разрешен произвольный доступ к памяти для чтения и записи, а также доступ к программно-управляемому кешу, получившему название разделяемая память.

Чтобы охватить максимальное количество разработчиков, NVIDIA взяла стандартный язык СИ и дополнила его несколькими новыми ключевыми словами, позволяющими задействовать специальные средства, присущие архитектуре CUDA. Через несколько месяцев после выпуска GeForce 8800GTX открыла доступ к компилятору нового языка CUDAC. Он стал первым языком, разработанным компанией по производству GPU с целью упростить программирование GPU для вычислений общего назначения.

Помимо создания языка для программирования GPU, NVIDIA предлагает специализированный драйвер, позволяющий использовать

возможности массивно-параллельных вычислений в архитектуре CUDA. Теперь пользователям нет нужды изучать программные интерфейсы OpenGL или DirectX или представлять свои задачи в виде задач компьютерной графики.

1.3 AMD ATI Stream Technology

Технология AMD STREAM представляет собой ряд аппаратных и программных оптимизаций, предназначенных для высокопроизводительных вычислительных рабочих потоков и ресурсоемких приложений с использованием технологии OpenCL™ — открытого межплатформенного стандарта программирования, который используется для стандартных вычислений. Среди функций можно выделить поддержку памяти с коррекцией ошибок (ECC), быстрые операции с плавающей запятой и непосредственный доступ к памяти, который обеспечивает обмен данными между несколькими графическими процессорами с низкой задержкой. В сочетании с библиотеками, оптимизированными для работы с графическими процессорами, и сторонним промежуточным программным обеспечением технология AMD STREAM позволяет реализовать вычислительную производительность видеокарт AMD FirePro [9].

1.4 Преимущества технологии CUDA

Технология NVIDIA CUDA - это единственная среда разработки на языке программирования C, которая позволяет разработчикам создавать программное обеспечение для решения сложных вычислительных задач за меньшее время, благодаря многоядерной вычислительной мощности графических процессоров. Гибкость, простота (расширенный язык программирования C) и бесплатность (SDK свободно скачивается с developer.nvidia.com) являются основными плюсами CUDA.

Изучив рынок видеокарт, можно сказать, что NVIDIACUDA наиболее распространенная. По данной технологии есть большее количество учебных материалов, чем для других аналогичных.

NVIDIA ведет активную поддержку разработчиков и бесплатные видеоуроки по работе с CUDA. А также на официальном сайте выделен специальный раздел CUDAZONE для общения программистов, обмена материалами, переводов статей и описаний технологии, возможности задать вопросы создателям CUDA.

2 АРХИТЕКТУРА NVIDIA CUDA

2.1 Программмно – аппаратная платформа CUDA

Платформа CUDA - это программно-аппаратная платформа компании nVidia, предназначенная для решения задач общего назначения – GPGPU (General-PurposecomputingonGraphicsProcessingUnits). Первый релиз вышел 15 февраля 2007 года. Основное назначение - дать программисту возможность использовать GPU (в дальнейшем «устройство») в качестве сопроцессора для задач, требующих параллельных вычислений, абстрагируясь при этом от терминологии и не используя библиотеки специфичные для обработки 3Dграфики. Однако, не все так просто, хотя бы потому, что видеокарты изначально ориентированы и много лет развивались как устройства обработки графики. И именно особенности архитектуры GPU вызывают наибольшие сложности при первом знакомстве с платформой.

В состав платформы, помимо самой видеокарты (начиная с GeForce 8митысячной серии) входит, так называемый драйвер CUDA, CUDA Toolkit и CUDA SDK. CUDA Toolkit включает в себя библиотеки, необходимые для работы с платформой, и компилятор nvcc, транслирующий исходный код программ в промежуточный ассемблерный код, а также дополнительные библиотеки. Драйвер CUDA в ранних версиях включал в себя только компилятор, осуществляющий преобразование промежуточного ассемблера, генерируемого компилятором nvcc в микрокод, исполняемый на GPU. Сейчас

драйвер CUDA и драйвер устройства объединены в один пакет. CUDA SDK содержит набор исходных кодов простых программ, иллюстрирующих методы и возможности работы с платформой CUDA.

CUDA поддерживается лишь процессорами видеоускорителей GeForce восьмого поколения и старше (GeForce 8, GeForce 9, GeForce 200), а также Quadro и Tesla, на это стоит обратить особое внимание. Основными характеристиками CUDA является взаимодействие с графическими API OpenGL и DirectX.25, возможность разработки на низком уровне, обеспечение доступа с быстрой разделяемой памяти, поддержка 32- и 64-битных операционных систем, также CUDA использует расширенный вариант языка C.

Рассмотрим логическую архитектуру, для понимания работы с платформой на рисунке 2.1:

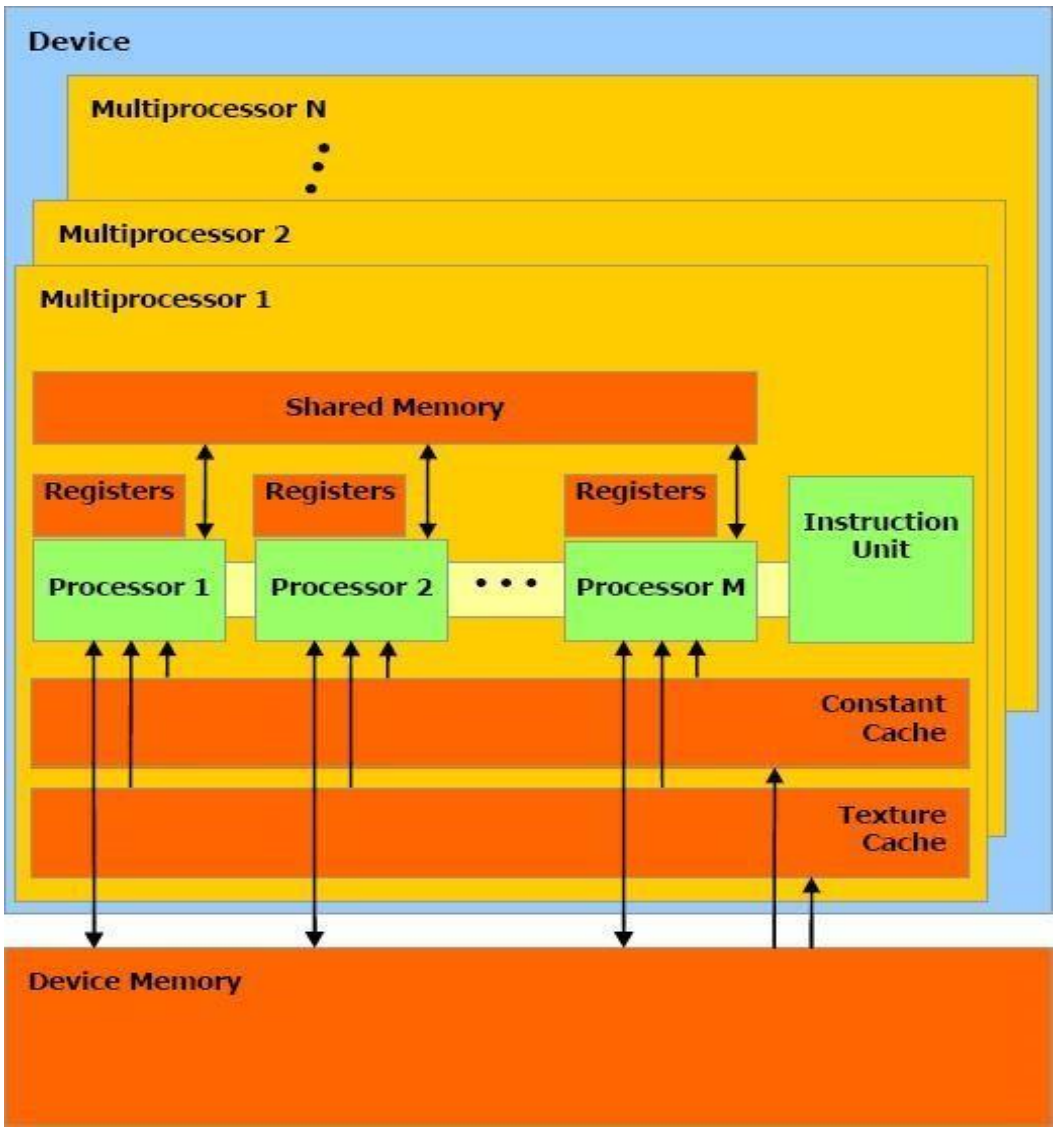

Рисунок 2.1 - Архитектура CUDA.

Рассмотрим данную модель более подробно.

2.2 Модель памяти технологии CUDA.

Свободный доступ к памяти с возможностью побайтовой адресации, является одной из наиболее важных особенностей для разработчиков. Потоки CUDA могут обращаться к данным из нескольких пространств памяти во время их выполнения, как показано на рисунке 2.2. Каждый поток имеет частную локальную память. Каждый блок потока имеет общую память, видимую для всех потоков блока и с тем же временем жизни, что и блок. Все потоки имеют доступ к одной и той же глобальной памяти.

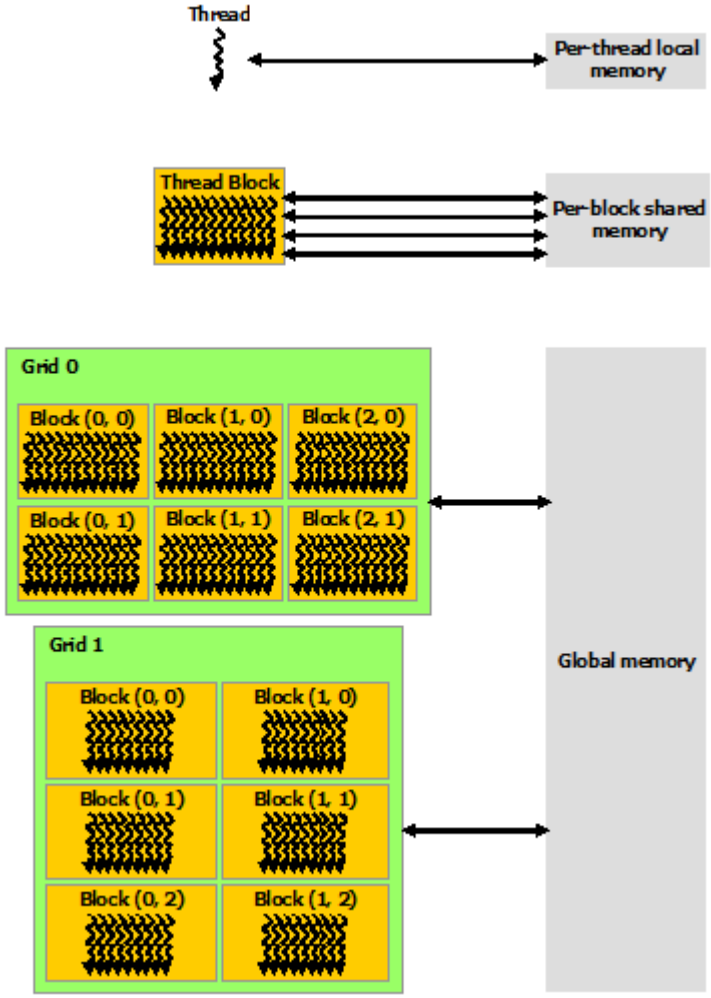

Рисунок 2.2 – Иерархия памяти.

Есть также два дополнительных пространства для чтения, доступных для всех потоков: пространства постоянной и текстурной памяти. Глобальные, постоянные и текстурные пространства памяти оптимизированы для разных применений памяти. Память текстур также предлагает различные режимы адресации, а также фильтрацию данных для некоторых конкретных форматов данных. Например, инструкции, которые обращаются к адресуемой памяти (т. е. к глобальной, локальной, общей, постоянной или текстурной памяти), могут потребоваться повторно выдать несколько раз в зависимости от распределения адресов памяти по потокам в основе. Для глобальной памяти, как правило, чем больше разбросаны адреса, тем более сокращена пропускная способность.

Глобальная память

Глобальная память хранится в памяти устройства, а память устройства - через 32-, 64- или 128-байтовые транзакции памяти. Эти транзакции памяти должны быть естественно выровнены: только 32-, 64- или 128-байтовые сегменты памяти устройства, которые выровнены по своему размеру (то есть, чей первый адрес кратен их размеру), могут быть прочитаны или записаны в память сделки.

Когда warp выполняет инструкцию, которая обращается к глобальной памяти, она объединяет обращения с памятью потоков в warp в одну или несколько транзакций памяти в зависимости от размера слова, доступного каждому потоку, и распределения адресов памяти через потоки. В общем, чем больше транзакций необходимо, тем больше неиспользуемых слов передаются в дополнение к словам, доступным по потокам, соответственно сокращая пропускную способность команд. Например, если 32-байтная транзакция памяти генерируется для 4-байтного доступа каждого потока, пропускная способность делится на 8.

Сколько транзакций необходимо и сколько пропускной способности в конечном итоге влияет, зависит от вычислительной способности устройства. Compute Capability 2.x, Compute Capability 3.x, Compute Capability 5.x и Compute Capability 6.x дают более подробную информацию о том, как обрабатываются глобальные обращения к памяти для различных вычислительных возможностей.

Инструкции по глобальной памяти поддерживают чтение или запись слов размером 1, 2, 4, 8 или 16 байт. Любой доступ (через переменную или указатель) к данным, находящимся в глобальной памяти, компилируется в одну команду глобальной памяти тогда и только тогда, когда размер типа данных равен 1, 2, 4, 8 или 16 байтам, и данные, естественно, (т. е. его адрес кратен этому размеру).

Если этот размер и требование выравнивания не выполняются, доступ компилируется в несколько инструкций с чередующимися шаблонами доступа, которые не позволяют этим инструкциям полностью объединиться.

Поэтому рекомендуется использовать типы, соответствующие этому требованию, для данных, которые находятся в глобальной памяти.

Требование выравнивания автоматически выполняется для встроенных типов char, short, int, long, longlong, float, double, как float2 или float4.

Локальная память.

Доступ к локальной памяти происходит только для некоторых автоматических переменных. Автоматическими переменными, которые компилятор может разместить в локальной памяти, являются:

- Массивы, для которых он не может определить, что они индексируются с постоянными величинами,
- Большие структуры или массивы, которые будут потреблять слишком много пространства для регистрации,
- Любая переменная, если ядро использует больше регистров, чем доступно (это также известно как разлитие регистров).

Проверка кода сборки PTX (полученная путем компиляции с опцией -ptx или-keep) будет определять, была ли переменная помещена в локальную память в течение первых фаз компиляции, поскольку она будет объявлена с использованием .local мнемоники и будет доступна с использованием ld .local и st.local mnemonics. Даже если это не так, последующие этапы компиляции могут по-прежнему решаться иначе, хотя если они находят, что потребляют слишком много места для хранения целевой архитектуры: проверка объекта кубика с помощью cuobjdump скажет, действительно ли это так. Кроме того, компилятор сообщает об общем использовании локальной памяти для каждого ядра (lmem) при компиляции с параметром --ptxas-options = -v. Обратите внимание, что некоторые математические функции имеют пути реализации, которые могут обращаться к локальной памяти.

Локальное пространство памяти находится в памяти устройства, поэтому доступ к локальной памяти имеет такую же высокую задержку и низкую пропускную способность, что и для доступа к глобальной памяти, и

для них одинаковые требования к коалесценции памяти, как описано в Access Memory Access Access. Однако локальная память организована таким образом, что последовательные 32-битные слова получают доступ к последовательному идентификатору потока. Таким образом, доступ к ним полностью объединен, пока все нити в warp получают один и тот же относительный адрес (например, один и тот же индекс в переменном массиве, тот же член в структурной переменной).

На устройствах вычислительной способности 2.x и 3.x локальные обращения к памяти всегда кэшируются в L1 и L2 так же, как и доступ к глобальной памяти.

На устройствах вычислительной возможности 5.x и 6.x локальные обращения к памяти всегда кэшируются в L2 так же, как и доступ к глобальной памяти.

Общая память

Поскольку общая память встроена в чип, она имеет гораздо более высокую пропускную способность и значительно меньшую задержку, чем локальная или глобальная память.

Для достижения высокой пропускной способности разделяемая память делится на модули памяти одинакового размера, называемые банками, к которым можно одновременно обращаться. Таким образом, любой запрос на чтение или запись в память, состоящий из n адресов, которые попадают в n разных банков памяти, может обслуживаться одновременно, что дает общую пропускную способность, которая в n раз превышает пропускную способность одного модуля.

Однако, если два адреса запроса памяти попадают в один и тот же банк памяти, возникает конфликт в банке, и доступ должен быть сериализован. Аппарат разбивает запрос на память с банковскими конфликтами на как можно большее количество отдельных запросов без конфликтов, уменьшая пропускную способность на коэффициент, равный количеству отдельных запросов на память. Если количество отдельных

запросов на память равно n, считается, что запрос начальной памяти вызывает n-way конфликты банка.

Поэтому, чтобы получить максимальную производительность, важно понять, как адреса памяти сопоставляются с банками памяти, чтобы запланировать запросы памяти, чтобы минимизировать конфликты в банках. Это описано в Compute Capability 2.x, Compute Capability 3.x, Compute Capability 5.x и Compute Capability 6.x для устройств вычислительной способности 2.x, 3.x, 5.x и 6.x соответственно.

Постоянная память

Постоянное пространство памяти находится в памяти устройства и кэшируется в постоянном кеше, указанном в Compute Capability 2.x.

Затем запрос разбивается на столько отдельных запросов, что в исходном запросе есть разные адреса памяти, что уменьшает пропускную способность на коэффициент, равный количеству отдельных запросов.

Затем полученные запросы обслуживаются при пропускной способности постоянного кеша в случае попадания в кэш или при пропускной способности памяти устройства в противном случае.

Текстурная память

Области памяти текстуры и поверхности хранятся в памяти устройства и кэшируются в кеше текстуры, поэтому при считывании текстуры или считывании поверхности стоит одна память, считываемая из памяти устройства только при пропуске кеша, в противном случае это просто стоит одного чтения из кэша текстур. Кэш текстуры оптимизирован для 2D пространственной локальности, поэтому потоки одного и того же детектора, которые читают текстурные или поверхностные адреса, которые находятся близко друг к другу в 2D, достигнут наилучшей производительности. Кроме того, он предназначен для потоковой загрузки с постоянной задержкой; Кэш-кеш уменьшает потребность в пропускной способности DRAM, но не обеспечивает задержку.

Чтение памяти устройства через текстуру или извлечение поверхности дает некоторые преимущества, которые могут сделать ее выгодной альтернативой чтению памяти устройства из глобальной или постоянной памяти:

- Если считываемые данные памяти не соответствуют шаблонам доступа, которые должны считывать глобальные или постоянные чтения в памяти, чтобы получить хорошую производительность, может быть достигнута более высокая пропускная способность, при условии, что в выборках текстуры или поверхностных чтениях имеется местность;
- Адресационные вычисления выполняются вне ядра выделенными единицами;
- Упакованные данные могут быть переданы для разделения переменных за одну операцию;
- 8-битные и 16-разрядные целочисленные входные данные могут быть необязательно преобразованы в 32-разрядные значения с плавающей запятой в диапазоне [0.0, 1.0] или [-1.0, 1.0].

2.3 Мультипроцессоры.

Появление многоядерных процессоров и многопроцессорных графических процессоров означает, что основные процессоры теперь являются параллельными системами. Более того, их параллелизм продолжает действовать в соответствии с законом Мура. Задача состоит в том, чтобы разработать прикладное программное обеспечение, которое прозрачно масштабирует свой параллелизм, чтобы использовать все большее число процессорных ядер, так же, как приложения 3D-графики прозрачно масштабируют их параллелизм для многих чистых графических процессоров с самым разным количеством ядер.

Модель параллельного программирования CUDA предназначена для преодоления этой проблемы, сохраняя при этом низкую кривую обучения для

программистов, знакомых со стандартными языками программирования, такими как C.

По сути, это три ключевые абстракции - иерархия групп потоков, разделяемых воспоминаний и барьерной синхронизации, которые просто предоставляются программисту как минимальный набор языковых расширений.

Эти абстракции обеспечивают мелкозернистый параллелизм данных и параллелизм потоков, вложенные в грубый параллелизм данных и параллелизм задач. Они направляют программиста на то, чтобы разделить проблему на грубые подзадачи, которые могут решаться независимо параллельно блоками потоков, а каждая подзадача - в более мелкие части, которые могут совместно решаться совместно всеми потоками внутри блока.

Это разложение сохраняет выразительность языка, позволяя потокам взаимодействовать при решении каждой подзадачи и в то же время обеспечивает автоматическую масштабируемость. Действительно, каждый блок потоков может быть запланирован на любом из доступных многопроцессоров в графическом процессоре в любом порядке одновременно или последовательно, так что скомпилированная программа CUDA может выполняться на любом количестве мультипроцессоров, как показано на рисунке 2.3.

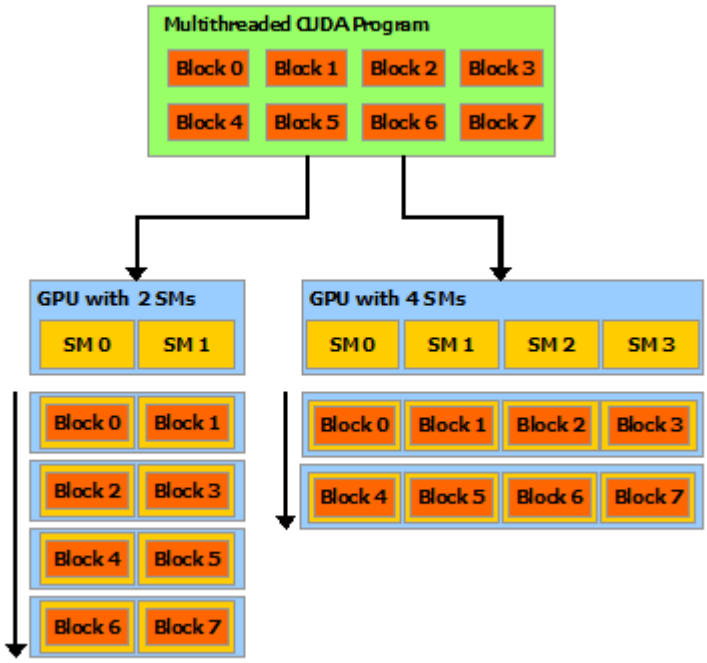

Рисунок 2.3. – Модель мультипроцессора CUDA

Архитектура NVIDIA GPU построена вокруг масштабируемого массива многопоточных потоковых мультипроцессоров (SMs)[4]. Когда программа CUDA на главном процессоре вызывает сетку ядра, блоки сетки перечисляются и распределяются по мультипроцессорам с доступной пропускной способностью. Нити блока потока выполняются одновременно на одном мультипроцессоре, и несколько блоков потоков могут выполняться одновременно на одном многопроцессорном компьютере. По завершении блокировки потоков на освобожденных мультипроцессорах запускаются новые блоки.

Мультипроцессор предназначен для одновременного выполнения сотен потоков. Для управления таким большим количеством потоков он использует уникальную архитектуру, называемую SIMT (Single-Instruction, Multiple-Thread) - одна инструкция и много потоков []. Инструкции конвейерны, чтобы задействовать параллелизм на уровне инструкций в рамках одного потока, а также параллельность параллельных потоков на основе одновременного аппаратного многопоточности, как описано в разделе «Многопоточность аппаратного обеспечения». В отличие от процессорных

ядер, они выдаются в порядке, но нет прогноза ветвления и нет спекулятивного исполнения.

Архитектура SIMT и аппаратная многопоточность описывают характеристики архитектуры потокового многопроцессора, которые являются общими для всех устройств. Compute Capability 2.x, Compute Capability 3.x, Compute Capability 5.x и Compute Capability 6.x обеспечивают особенности для устройств вычислительных возможностей 2.x, 3.x, 5.x и 6.x соответственно.

Мультипроцессор создает, управляет, планирует и выполняет потоки в группах из 32 параллельных потоков, называемых перекосами. Отдельные потоки, составляющие основу, начинаются вместе по одному и тому же адресу программы, но у них есть свой собственный счетчик адресов команд и состояние регистрации, и поэтому они свободно разветвляются и выполняются независимо. Термин warp происходит от качества, первой технологии параллельных потоков. Половина деформации - это либо первая, либо вторая половина деформации. Квадрат - это либо первый, второй, третий, либо четвертый квартал варпа.

Когда мультипроцессору предоставляется один или несколько блоков потоков для выполнения, он разбивает их на перекосы, и каждый warp получает запланированный планировщик warp для выполнения. Способ, которым блок разбивается на перекосы, всегда один и тот же. Raждый warp содержит потоки последовательных, увеличивающих идентификаторы потоков с первым нитьем, содержащим warp. Иерархия потоков описывает, как идентификаторы потоков относятся к индексам потоков в блоке.

Варп выполняет одну общую инструкцию за раз, поэтому полная эффективность реализуется, когда все 32 нити варпа согласуют их путь выполнения. Если потоки warp расходятся через зависящую от данных условную ветвь, warp последовательно выполняет каждый путь ветки, отключая потоки, которые не находятся на этом пути, и когда все пути завершены, потоки сходятся к одному и тому же пути выполнения. Расхождение дивергенции происходит только в пределах основы. Различные

деформации выполняются независимо независимо от того, выполняются ли они с помощью общих или непересекающихся кодов.

Архитектура SIMT похожа на организацию векторов SIMD (Single Instruction, Multiple Data), поскольку одна команда управляет несколькими элементами обработки. Ключевым отличием является то, что организации векторов SIMD предоставляют SIMD-ширину программному обеспечению, тогда как инструкции SIMT определяют поведение выполнения и ветвления одного потока [9]. В отличие от SIMD-векторных машин, SIMT позволяет программистам писать параллельный код уровня нити для независимых, скалярных потоков, а также параллельный параллельный код для координированных потоков. В целях правильности программист может по существу игнорировать поведение SIMT, однако существенные улучшения производительности могут быть реализованы, если учесть, что код редко требует, чтобы потоки в деформации расходились. На практике это аналогично роли строк кеша в традиционном коде: размер линии кэша можно безопасно игнорировать при разработке для правильности, но должен учитываться в структуре кода при проектировании для максимальной производительности. С другой стороны, векторные архитектуры требуют, чтобы программное обеспечение объединяло нагрузки в векторы и управляло расхождением вручную.

Нити основы, которые находятся на текущем пути выполнения этого варпа, называются активными потоками, тогда как потоки, не относящиеся к текущему пути, неактивны (отключены). Темы могут быть неактивными, поскольку они вышли раньше других потоков их деформации или потому, что они находятся на другом пути ветвления, чем путь ветвления, выполняемый в данный момент warp, или потому, что они являются последними потоками блока, число потоков которого не кратно размеру основы.

Контекст выполнения (счетчики программ, регистры и т. д.) для каждой основы, обработанной мультипроцессором, поддерживается на чипе в течение всего срока службы основы. Поэтому переход от одного контекста

выполнения к другому не требует затрат, и в каждый момент выполнения каждой команды планировщик warp выбирает warp, у которого есть потоки, готовые выполнить свою следующую команду (активные потоки warp) и выдает инструкцию этим потокам, в частности, каждый мультипроцессор имеет набор 32-разрядных регистров, которые разделены между перекосами, и параллельный кеш данных или разделяемая память, которая разделена между блоками потоков.

Количество блоков и перекосов, которые могут находиться и обрабатываться вместе на мультипроцессоре для заданного ядра, зависит от количества регистров и общей памяти, используемых ядром, и количества регистров и общей памяти, доступных на многопроцессорном компьютере.

2.4 Модель программирования технологии CUDA.

Модель исполнения CUDA основана на примитивах потоков, блоков потоков и сеток с функциями ядра, определяющими программу, выполняемую отдельными потоками в поточном блоке и сетке. Когда вызывается функция ядра, свойства сетки описываются конфигурацией выполнения, которая имеет специальный синтаксис в CUDA. Поддержка динамического параллелизма в CUDA расширяет возможности настройки, запуска и синхронизации на новых сетях для потоков, работающих на устройстве.

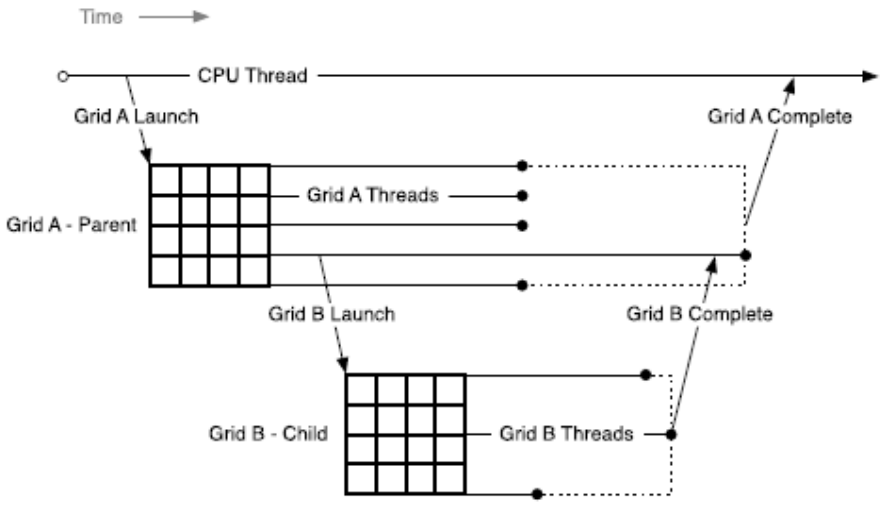

Рисунок 2.4. – Модель программирования CUDA

Поток устройства, который настраивает и запускает новую сетку, принадлежит родительской сетке, а сетка, созданная вызовом, представляет собой дочернюю сетку.

Вызов и завершение дочерних сеток правильно сложен, что означает, что родительская сетка не считается завершенной до тех пор, пока все дочерние сетки, созданные ее потоками, не будут завершены. Даже если вызывающие потоки явно не синхронизируются на запущенных дочерних сетках, среда выполнения гарантирует неявную синхронизацию между родительским и дочерним.

На хосте и устройстве среда выполнения CUDA предлагает API для запуска ядер, для ожидания завершения работы и для отслеживания зависимостей между запусками через потоки и события. В хост-системе состояние запусков и примитивы CUDA, ссылающиеся на потоки и события, разделяются всеми потоками внутри процесса; Однако процессы выполняются независимо и не могут совместно использовать объекты CUDA.

На устройстве существует аналогичная иерархия: запущенные ядра и объекты CUDA видны для всех потоков в блоке потоков, но независимы между блоками потоков. Это означает, например, что поток может быть создан одним потоком и использоваться любым другим потоком в одном блоке потока, но не может быть разделен нитями в любом другом блоке потока.

Операции выполнения CUDA из любого потока, включая запуск ядра, видны через блок потоков. Это означает, что вызывающий поток в родительской сетке может выполнять синхронизацию на сетках, запущенных этим потоком, другими потоками в поточном блоке или потоками, созданными в одном блоке потоков. Выполнение блока потока не считается завершенным до тех пор, пока все запуски всех потоков в блоке не будут завершены. Если все потоки в блочном завершении до запуска всех дочерних запусков завершены, операция синхронизации будет автоматически запущена.

Потоки и события CUDA позволяют управлять зависимостями между запуском сетки: сетки, запущенные в один поток, выполняются в порядке, а

события могут использоваться для создания зависимостей между потоками. Потоки и события, созданные на устройстве, служат этой же цели.

Потоки и события, созданные в сетке, существуют в области потока потока, но имеют неопределенное поведение при использовании вне блока потока, где они были созданы. Как описано выше, вся работа, запущенная блоком потока, неявно синхронизируется при выходе из блока; Работа, запущенная в потоки, включена в это, причем все зависимости решаются соответствующим образом. Поведение операций над потоком, которое было изменено вне области потока, не определено.

Потоки и события, созданные на хосте, имеют неопределенное поведение при использовании в любом ядре, так же как потоки и события, созданные родительской сеткой, имеют неопределенное поведение, если они используются в дочерней сетке.

Порядок запуска ядра из среды выполнения устройства следует за семантикой упорядочения потока CUDA. В блоке потока все ядра запускаются в один поток, исполняются в порядке. С несколькими потоками в одном потоковом блоке, запущенном в один и тот же поток, упорядочение внутри потока зависит от планирования потока внутри блока, которое может управляться с помощью примитивов синхронизации, таких как __syncthreads ().

Поскольку потоки разделяются всеми потоками в поточном блоке, неявный поток NULL также используется совместно. Если несколько потоков в поточном блоке запускаются в неявный поток, то эти запуска будут выполняться в порядке. Если требуется параллелизм, следует использовать явные именованные потоки.

Динамический параллелизм позволяет упростить параллелизм внутри программы.Однако время выполнения устройства не вносит никаких новых гарантий параллелизма в модель исполнения CUDA. Нет гарантии одновременного выполнения между любым количеством блоков потоков на устройстве.

Отсутствие гарантии параллелизма распространяется на блоки родительских потоков и их дочерние сетки. Когда блок родительского потока запускает дочернюю сетку, дочернему устройству не гарантируется начало выполнения, пока блок родительского потока не достигнет явной точки синхронизации (например, cudaDeviceSynchronize ()).

Хотя параллелизм часто может быть легко достигнут, он может варьироваться в зависимости от deviceconfiguration, рабочей нагрузки приложения и планирования времени выполнения. Поэтому небезопасно зависеть от любого параллелизма между различными блоками потоков.

2.4.1 Модель программирования CPU на CUDA.

Поскольку у GPU нет доступа к ОЗУ, программист должен заранее заботиться о том, чтобы все ресурсы, необходимые для запуска ядра приложения, находились в памяти видеокарты. Для этих целей из CUDA SDK используются три основные функции: cudaMalloc, cudaMemcpy и cudaFree. Эти функции имеют ту же цель, что и стандартные malloc, memcpy и бесплатно, но, конечно, все операции выполняются в видеопамяти.

Процесс настройки сетки и блоков заключается в настройке размера сетки и блоков. Основная задача программиста на этом этапе - найти оптимальный баланс между размером и количеством блоков. Увеличивая количество потоков в блоке, можно уменьшить количество вызовов в глобальной памяти за счет увеличения интенсивности обмена данными между потоками через быструю разделяемую память. С другой стороны, количество регистров, выделенных блоку, фиксировано, и, если количество потоков намного больше, то увеличится время исполнения ядра, потому что GPU начнет размещать данные в локальной памяти. При вызове ядра передаются все размерности сетки и блока, определенные ранее.

Ядро вызывается как регулярная функция на языке C. Единственное существенное различие заключается в том, что при вызове ядра нужно перенести ранее определенные размеры сетки и блока.

После выполнения ядра нужно скопировать результаты программы обратно в память с помощью функции cudaMemcpy, указав направление обратной копии (от GPU до CPU).

2.4.2 Модель программирования GPU на CUDA.

Квалификаторы типа функции определяют, будет ли выполняться функция на хосте или на устройстве и может ли он быть вызван с хоста или с устройства.

__device__ объявляет функцию, которая будет выполняться на устройстве и вызываться только с устройства.

__global__ объявляет функцию как ядро. Такая функция выполняется на устройстве и вызывается с хоста. Вызывается с устройства для устройств с вычислительной способностью 3,2 или выше. Вызов функции __global__ является асинхронным, то есть он возвращается до того, как устройство завершит выполнение.

__host__ объявляет функцию, которая выполняется на хосте и вызывается только с хоста.

__global__ и __host__ не могут использоваться вместе. Однако квалификаторы __device__ и __host__ могут использоваться вместе, и в этом случае функция компилируется как для хоста, так и для устройства.

__noinline__ и __forceinline__

Спецификатор функции __noinline__ может использоваться как подсказка для компилятора, чтобы не включать функцию, если это возможно. Тело функции все равно должно находиться в том же файле, где он вызывается.

Обозначение функции __forceinline__ можно использовать, чтобы заставить компилятор встроить функцию.

Определители типа переменной определяют местоположение памяти на устройстве переменной.

Автоматическая переменная, объявленная в коде устройства без каких-либо спецификаторов __device__, __shared__ и __constant__, обычно находится в регистре. Однако в некоторых случаях компилятор может захотеть поместить его в локальную память, что может иметь неблагоприятные последствия для производительности. Рассмотрим набор спецификаторов, определяющих тип памяти для размещения переменных:

__device__ объявляет переменную, которая находится на устройстве. Не более одного из квалификаторов другого типа могут использоваться вместе с __device__ для дальнейшего определения того, какое пространство памяти принадлежит переменной. Если ни один из них не присутствует, то переменная сохраняется в глобальной памяти. Доступен из всех потоков в сетке и от хоста через библиотеку времени выполнения (cudaGetSymbolAddress () / cudaGetSymbolSize () / cudaMemcpyToSymbol () / cudaMemcpyFromSymbol ()).

__constant__ объявляет переменную, которая остается в постоянном пространстве памяти. Доступна из всех потоков в сетке и от хоста через библиотеку времени выполнения (cudaGetSymbolAddress () / cudaGetSymbolSize () / cudaMemcpyToSymbol () / cudaMemcpyFromSymbol ()).

__shared__объявляет переменную, которая остается в области общей памяти блока. Доступна для всех потоков внутри блока, при объявлении переменной в общей памяти в качестве внешнего массива, такого как Extern __shared__ float shared [];

Размер массива определяется во время запуска. Все переменные, объявленные таким образом, начинаются с одного и того же адреса в памяти, так что макет переменных в массиве должен явно управляться с помощью смещений. Например, если требуется эквивалент

short array0[128];

float array1[64];

```
int   array2[256];
```

В динамически распределенной общей памяти можно объявить и инициализировать массивы следующим образом:

```
Extern __shared__ float array [];
__device__ void func () // Функция __device__ или __global__
{
   Short * array0 = (короткий *) массив;
   Float * array1 = (float *) & array0 [128];
   Int * array2 = (int *) & array1 [64];
}
```

Указатели должны быть привязаны к типу, на который они указывают.

__managed__объявляет переменную, которая может ссылаться как на устройство, так и на хост-код, например, его адрес можно взять или его можно прочитать или записать непосредственно с устройства или функции хоста.

Nvcc поддерживает ограниченные указатели с помощью ключевого слова __restrict__. Ограниченные указатели были введены на C99, чтобы облегчить проблему сглаживания, которая существует на языке C, и которая запрещает все виды оптимизации от переупорядочения кода до общего исключения суб-экспрессии.

В языке C указатели a, b и c могут быть сглажены, поэтому любая запись через c может изменять элементы a или b. Это означает, что для обеспечения функциональной корректности компилятор не может загружать [0] и b [0] в регистры, умножать их и сохранять результат как c [0], так и c [1], поскольку результаты будут отличаться от результатов Абстрактная модель исполнения, если, скажем, [0] - это действительно то же место, что и c [0]. Поэтому компилятор не может воспользоваться общим подвыражением. Аналогично, компилятор не может просто переупорядочить вычисление c [4] в непосредственной близости от вычисления c [0] и c [1], поскольку предыдущая запись в c [3] может изменить входы на вычисление c [4].

3 РЕШЕНИЕ ЗАДАЧ МНОГОМЕРНОЙ ОПТИМИЗАЦИИ

3.1 Функции многомерной оптимизации

Для тестирования алгоритмов многомерной оптимизации, основанных как на методах эволюционного моделирования, так и на любых других следует использовать специальные тестовые функции. Ниже приведён перечень некоторых из них, указаны рекомендуемые интервалы изменения переменных и глобальные оптимальные значения.

3.1.1 Функция Розенброка

Функция Розенброка представляет собой невыпуклую функцию, используемую для оценки производительности алгоритмов оптимизации. Минимум в точке (1, 1) [1]

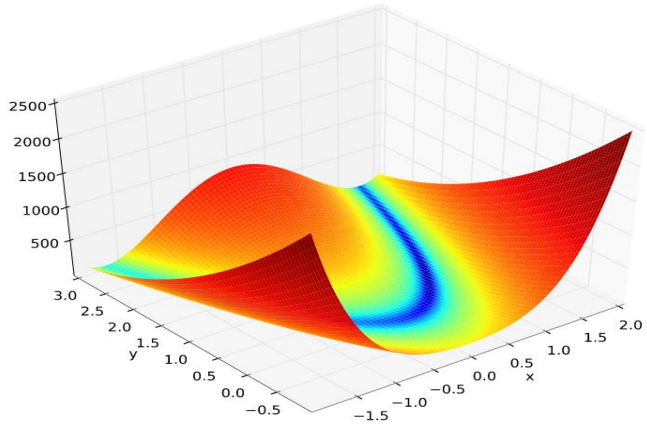

Рисунок 3.1. – График функции Розенброка

Функцию Розенброка можно определить следующим образом:

$$f(x,y) = (1-x^2) + 100(y - x^2)^2. \quad (1)$$

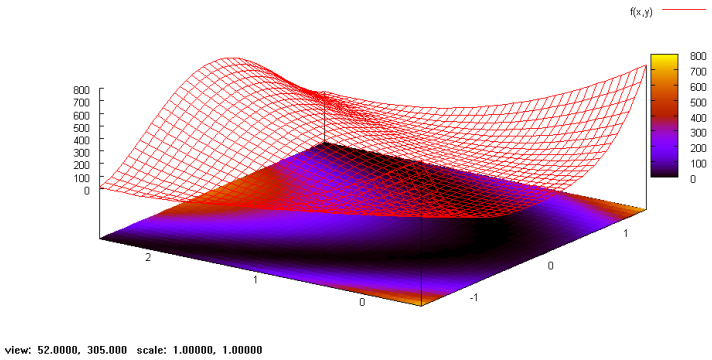

Рисунок 3.2. - Значение функции в окрестности точки (0,0).

Рекомендуемый интервал поиска оптимального решения по каждой переменной (–5; 5). Функция имеет глобальный минимум, в точке (x,y) = (1,1), где f(x,y) = 0.

3.1.2 Функция Химмельблау

Функция Химмельблау преставляет собой мультимодальную функцию двух переменных, которая используется для проверки эффективности алгоритмов оптимизации, определяется формулой:

$$f(x, y) = (x^2+y-11)^2 + (x+y^2-7)^2. \quad (2)$$

Имеет локальный максимум в x = -0.270845, y = -0.923039 со значением f(x,y) = 181.617 и четыре равнозначных локальных минимума:

$$\begin{gathered} f(3.0, 2.0) = 0.0, \\ f(-2.805118, 3.131312) = 0.0, \\ f(-3.779310, -3.283186) = 0.0, \\ f(3.584428, -1.848126) = 0.0. \end{gathered} \quad (3)$$

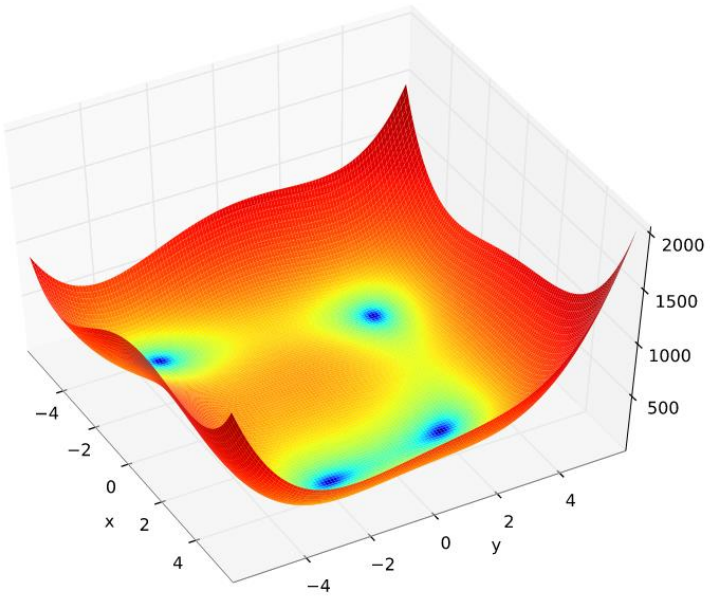

Рисунок 3.3. - График функции Химмельблау для двух переменных.

3.1.3 Функция Растригина

Функция Растригина представляет собой невыпуклую функцию, используемую для тестирования эффективности алгоритмов оптимизации. Функцию можно назвать типичным примером нелинейной мультимодальной функции [10].

Определение функции:

$$f(x) = An + \sum_{i=1}^{n}[x_i^2 - A\cos(2\pi x_i)], \quad (4)$$

где A=10 и $x_i \in$ [- 5,12; 5,12].

Глобальный минимум в точке x=0, где f (x)=0.

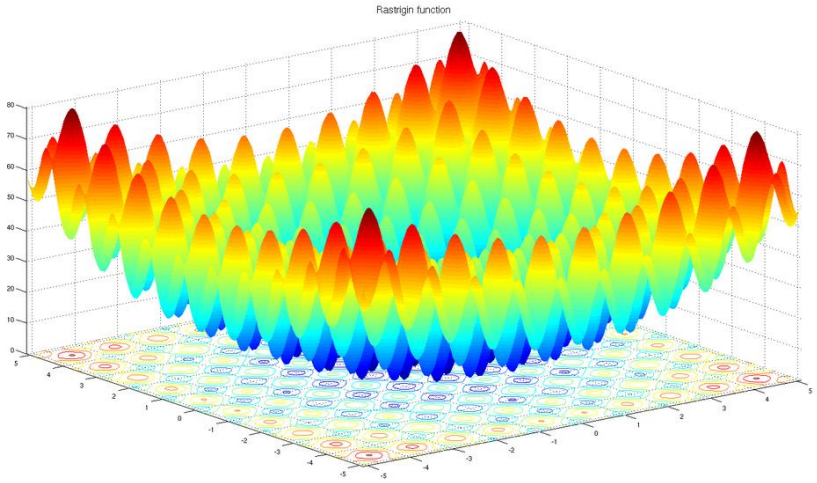

Рисунок 3.4. - График функции Растригина для двух переменных

Для исследования использованы функции Розенброка и Растригина. Выбор данных функций обусловлен возможностью обобщения для больших размерностей, а также наличием множества локальных минимумов что делает задачу нахождения глобального минимума – относительно трудной. Еще одним фактом является то что выбранные функции являются наиболее распространенными для тестирования различных алгоритмов оптимизации.

3.2 Алгоритмы случайного поиска

Различают направленный и ненаправленный случайный поиск. В случае направленного случайного поиска, испытания между собой связаны. Методы, приводят только к локальным экстремумам. Сходимость таких методов высокая. Результаты используются для формирования последующих испытаний.

При ненаправленном поиске, результаты не зависят от предыдущих. Сходимость такого поиска очень мала. Методы имеют преимущество, такое как решение многоэкстремальных задач. Для более интересного результата исследуем поставленную задачу на направленном и ненаправленном случайном поиске. Примером ненаправленного поиска является простой случайный поиск.

3.2.1 Простой случайный поиск

Предполагается, что требуемый минимум лежит в некотором n-мерном параллелепипеде. В этом параллелепипеде согласно единому закону N случайных выборок выбираются случайным образом и в них рассчитывается целевая функция. В качестве решения проблемы берется точка, в которой функция имеет минимальное значение. Вероятность мала, что, хотя бы одна точка попадет в небольшую окрестность локального минимума. Действительно, пусть N = 106, а диаметр бассейна составляет 10% вблизи минимума. Тогда объем этой депрессии составляет 0,1 н. Части объема n-

мерного параллелепипеда. Даже с числом переменных n> 6 практически нет точки в бассейне.

Поэтому берут небольшое число точек N и каждую точку рассматривают как нулевое приближение. Из каждой точки совершают спуск, быстро попадая в ближайший овраг или котловину; когда шаги спуска быстро укорачиваются, его прекращают, не добиваясь высокой точности. Этого уже достаточно, чтобы судить о величине функции в ближайшем локальном минимуме с удовлетворительной точностью. Сравнивая окончательные значения функции на всех спусках между собой, можно изучить расположение локальных минимумов и сопоставить их величины. После этого можно отобрать нужные по смыслу задачи минимумы и провести в них дополнительные спуски для получения координат точек минимума с более высокой точностью [16].

Данную область нужно вписать в n-мерный гиперпараллелепипед и оставить только те точки, что попадают в допустимую область.

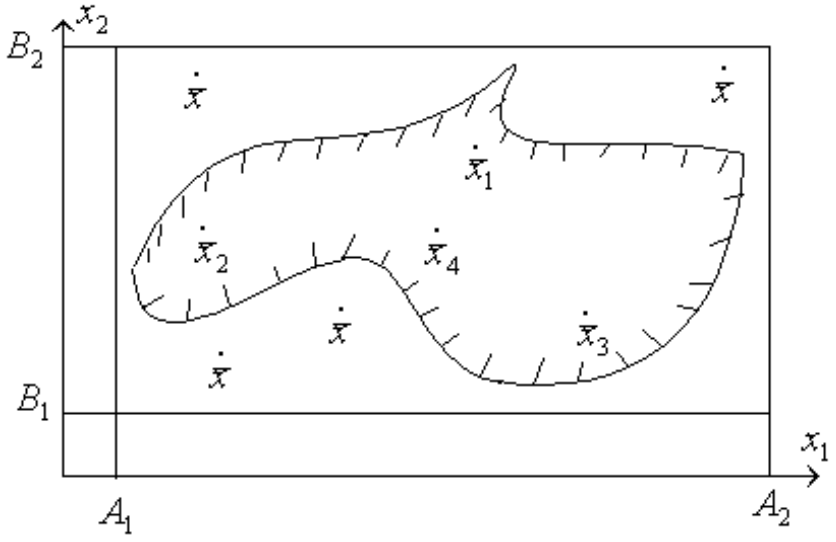

Рисунок 3.5 - Построение включающего гиперпараллелепипеда (А, В - границы параллелепипеда)

3.2.2. Алгоритм наилучшей пробы с направляющим гиперквадратом

Алгоритм наилучшей пробы с направляющим гиперквадратом относится к ненаправленному поиску. Пусть есть допустимаю область, в которой строится гиперквадрат. Значения функции вычисляются в случайно разбросанных точках $x_1,\ldots,x_m$, далее выбирается наилучшая среди них. Строится новый гиперквадрат, на этой точке. Точка, в которой достигается минимум функции на k-м этапе, берется в качестве центра нового гиперквадрата на (k+1)-м этапе. Координаты вершин гиперквадрата на (k+1)-м этапе определяются соотношениями [16]

$$a_i^{k+1} = x_i^{k+1} - \frac{b_i^k - a_i^k}{2}, \quad b_i^{k+1} = x_i^{k+1} + \frac{b_i^k - a_i^k}{2}, \tag{5}$$

Где x^k– наилучшая точка в гиперквадрате на k-м этапе.

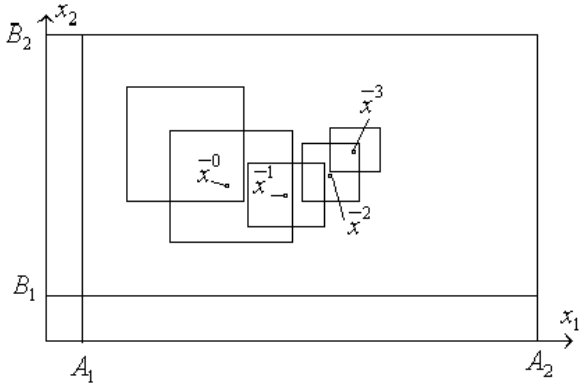

Рисунок 3.6 - К алгоритму с направляющим гиперквадратом

В новом гиперквадрате выполняем ту же последовательность действий, случайным образом разбрасывая m точек, и т.д.

Таким образом на 1-м этапе координаты случайных точек удовлетворяют неравенствам $a_i^1 \le x_i \le b_i^1$, i=1,…,n и

$x^1 = \arg\min_{j=1,m}\{f(x_j)\}$ - точка с минимальным значением целевой функции.

В алгоритме с *обучением* стороны гиперквадрата могут регулироваться в соответствии с изменением параметра α по некоторому правилу. В этом случае координаты вершин гиперквадрата на (k+1)-м этапе будут определяться соотношениями [16]

$$a_i^{k+1} = x_i^{k+1} - \frac{b_i^k - a_i^k}{2\alpha}, \quad b_i^{k+1} = x_i^{k+1} + \frac{b_i^k - a_i^k}{2\alpha}. \qquad (6)$$

Можно использовать также алгоритмы случайного поиска с направляющим гиперконусом и гиперконусом, для решения подобных задач.

4 РАЗРАБОТКА ПРИЛОЖЕНИЯ

4.1 Руководство по установке CUDA для Microsoft Windows

CUDA - это параллельная вычислительная платформа и модель программирования, изобретенная NVIDIA. Это позволяет значительно увеличить вычислительную производительность за счет использования мощности графического процессора (GPU).

CUDA была разработана с учетом нескольких целей проектирования:

Предоставьте небольшой набор расширений для стандартных языков программирования, таких как C, которые обеспечивают прямую реализацию параллельных алгоритмов. С CUDA C / C + + программисты могут сосредоточиться на задаче распараллеливания алгоритмов, а не тратить время на их реализацию.

Поддержка гетерогенных вычислений, где приложения используют как процессор, так и графический процессор. Серийные части приложений запускаются на CPU, а параллельные порты выгружаются на GPU. Таким образом, CUDA можно постепенно применять к существующим приложениям. Процессор и графический процессор рассматриваются как отдельные устройства, у которых есть свои собственные пространства памяти. Эта конфигурация также позволяет одновременно вычислять на CPU и GPU без конкуренции за ресурсы памяти.

Графические процессоры с поддержкой CUDA имеют сотни ядер, которые могут совместно выполнять тысячи вычислительных потоков. Эти ядра имеют общие ресурсы, включая файл регистров и общую память. Встроенная совместно используемая память позволяет параллельным задачам, выполняемым на этих ядрах, обмениваться данными, не отправляя их по шине системной памяти.

Это руководство покажет вам, как установить и проверить правильность работы инструментов разработки CUDA.

Системные требования

Чтобы использовать CUDA в вашей системе, вам потребуется следующее:

- Графический процессор с поддержкой CUDA
- Поддерживаемая версия Microsoft Windows
- Поддерживаемая версия Microsoft Visual Studio
- NVIDIA CUDA Toolkit (доступен по адресу http://developer.nvidia.com/cuda-downloads)

В следующих двух таблицах перечислены текущие поддерживаемые операционные системы и компиляторы Windows.

Таблица 4.1 Поддержка операционной системы Windows в CUDA 8.0

Operating System	**Native x86_64**	**Cross (x86_32 on x86_64)**
Windows 10	YES	YES
Windows 8.1	YES	YES
Windows 7	YES	YES
Windows Server 2016	YES	NO
Windows Server 2012 R2	YES	NO
Windows Server 2008 R2 **DEPRECATED**	YES	YES

Таблица 4.2 Поддержка компилятора Windows в CUDA 8.0

Compiler	**IDE**	**Native x86_64**	**Cross (x86_32 on x86_64)**
Visual C++ 14.0	Visual Studio 2015	YES	NO
	Visual Studio Community 2015	YES	NO
Visual C++ 12.0	Visual Studio 2013	YES	YES
Visual C++ 11.0	Visual Studio 2012	YES	YES
Visual C++ 10.0 **DEPRECATED**	Visual Studio 2010	YES	YES

Поддержка x86_32 ограничена.

Родная разработка с использованием CUDA Toolkit на x86_32 не поддерживается. Развертывание и выполнение приложений CUDA на x86_32

по-прежнему поддерживается, но ограничено использованием графических процессоров GeForce. Чтобы создать 32-битные приложения CUDA, используйте возможности кросс-разработки CUDA Toolkit на x86_64.

Поддержка разработки и запуска 32-разрядных приложений x86 на x86_64 Windows ограничена использованием:

- Графические процессоры GeForce
- Драйвер CUDA
- Время выполнения CUDA (cudart)
- CUDA Math Library (math.h)
- Компилятор CUDA C ++ (nvcc)
- Инструменты разработки CUDA

Установка средств разработки CUDA

Настройка инструментов разработки CUDA в системе под управлением соответствующей версии Windows состоит из нескольких простых шагов:

- Убедитесь, что в системе установлен графический процессор с поддержкой CUDA.
- Загрузите NVIDIA CUDA Toolkit.
- Установите NVIDIA CUDA Toolkit.
- Проверьте, что установленное программное обеспечение работает правильно и взаимодействует с оборудованием.

Вы можете убедиться, что у вас есть графический процессор с поддержкой CUDA через раздел «Дисплеи» в Диспетчере устройств Windows. Здесь вы найдете имя и модель вашего видеокарты. Если у вас есть карта NVIDIA, указанная в http://developer.nvidia.com/cuda-gpus, этот графический процессор является CUDA-совместимым. Примечания к выпуску инструментария CUDA также содержат список поддерживаемых продуктов.

Диспетчер устройств Windows можно открыть с помощью следующих действий:

- Откройте окно запуска из меню «Пуск»

- Введите: control /name Microsoft.DeviceManager

Инструмент NVIDIA CUDA Toolkit доступен по адресу http://developer.nvidia.com/cuda-downloads. Выберите платформу, которую вы используете, и один из следующих форматов установщика:

Сетевой установщик: минимальный установщик, который затем загружает пакеты, необходимые для установки. Загружаются только пакеты, выбранные на этапе выбора установщика. Этот установщик полезен для пользователей, которые хотят минимизировать время загрузки.

Полный установщик: установщик, который содержит все компоненты CUDA Toolkit и не требует дальнейшей загрузки. Этот установщик полезен для систем, которым не хватает доступа к сети и для развертывания на предприятии.

CUDA Toolkit устанавливает драйвер CUDA и инструменты, необходимые для создания, сборки и запуска приложения CUDA, а также библиотек, файлов заголовков, исходного кода CUDA и других ресурсов.

Загрузить подтверждение

Загрузка может быть проверена путем сравнения контрольной суммы MD5, опубликованной по адресу http://developer.nvidia.com/cuda-downloads/checksums, с информацией о загруженном файле. Если какая-либо из контрольных сумм отличается, загруженный файл поврежден и нуждается в повторной загрузке.

Чтобы вычислить контрольную сумму MD5 загруженного файла, следуйте инструкциям на странице http://support.microsoft.com/kb/889768.

Установка программного обеспечения CUDA

Перед установкой инструментария вы должны прочитать примечания к выпуску, так как они содержат подробную информацию об установке и функциональности программного обеспечения.

Примечание. Для работы CUDA необходимо установить драйвер и инструментарий. Если вы не установили автономный драйвер, установите драйвер из набора инструментов NVIDIA CUDA.

Примечание. Установка может завершиться неудачей, если Windows Update начнется после начала установки. Дождитесь завершения обновления Windows Update и повторите попытку установки.

Графическая установка

Установите программное обеспечение CUDA, выполнив установку CUDA и следуя инструкциям на экране.

Например, для установки только компонентов компилятора и драйвера:

<PackageName>.exe -s compiler_8.0 Display.Driver

Извлечение и проверка файлов вручную

Иногда бывает целесообразно извлекать или инспектировать устанавливаемые файлы напрямую, например, при развертывании на предприятии или просматривать файлы перед установкой. Полный пакет установки можно извлечь, используя инструмент декомпрессии, который поддерживает метод сжатия LZMA, такой как 7-zip или WinZip.

После извлечения файлы CUDA Toolkit будут находиться в папке CUDAToolkit и аналогично для CUDA Samples и CUDA Visual Studio Integration. Внутри каждого каталога есть файл .dll и .nvi, который можно игнорировать, поскольку они не являются частью установочных файлов.

Примечание. Доступ к файлам таким образом не устанавливает никаких параметров среды, таких как переменные или интеграция с Visual Studio. Это предназначено для развертывания на уровне предприятия.

Используйте подходящую модель драйвера

В Windows 7 и более поздних версиях операционная система предоставляет две модели драйверов, под которыми может работать драйвер NVIDIA:

- Модель драйвера WDDM используется для устройств отображения.
- Режим Tesla Compute Cluster (TCC) Драйвера NVIDIA доступен для устройств без дисплея, таких как графические процессоры NVIDIA Tesla и графические процессоры GeForce GTX Titan; Он использует модель драйвера Windows WDM.

Режим драйвера TCC предоставляет ряд преимуществ для приложений CUDA на графических процессорах, поддерживающих этот режим. Например:

- TCC устраняет таймауты, которые могут возникать при работе под WDDM из-за механизма обнаружения и восстановления времени ожидания для устройств отображения.
- TCC позволяет использовать CUDA с Windows Remote Desktop, что невозможно для WDDM-устройств.
- TCC позволяет использовать CUDA из процессов, работающих под управлением Windows, что невозможно для WDDM-устройств.
- TCC уменьшает задержку запуска ядра CUDA.
- TCC включен по умолчанию на самых последних графических процессорах NVIDIA Tesla. Чтобы проверить, какой режим драйвера используется и / или переключать режимы драйвера, используйте инструмент nvidia-smi, который входит в комплект установки драйвера NVIDIA (подробнее см. Nvidia-smi -h).

Примечание. Имейте в виду, что, когда режим TCC включен для конкретного графического процессора, этот графический процессор нельзя использовать в качестве устройства отображения.

Примечание. Графические процессоры NVIDIA GeForce (за исключением графических процессоров GeForce GTX Titan) не поддерживают режим TCC.

Проверить установку

Прежде чем продолжить, важно убедиться, что набор инструментов CUDA может правильно найти и установить связь с оборудованием, поддерживающим CUDA. Для этого вам нужно скомпилировать и запустить некоторые из включенных примерных программ.

Выполнение скомпилированных примеров

Версия CUDA Toolkit можно проверить, запустив nvcc -V в окне командной строки. Вы можете отобразить окно командной строки, перейдя к:

Start > All Programs > Accessories > Command Prompt

Образцы CUDA включают примеры программ в исходной и скомпилированной форме. Чтобы проверить правильную конфигурацию аппаратного и программного обеспечения, настоятельно рекомендуется запустить программу deviceQuery, расположенную в

C:\ProgramData\NVIDIACorporation\CUDASamples\v8.0\bin\win64\Release

Предполагается, что вы использовали структуру каталогов установки по умолчанию. Если CUDA установлен и настроен правильно, выход должен выглядеть так, как показано на рисунке 4.1.

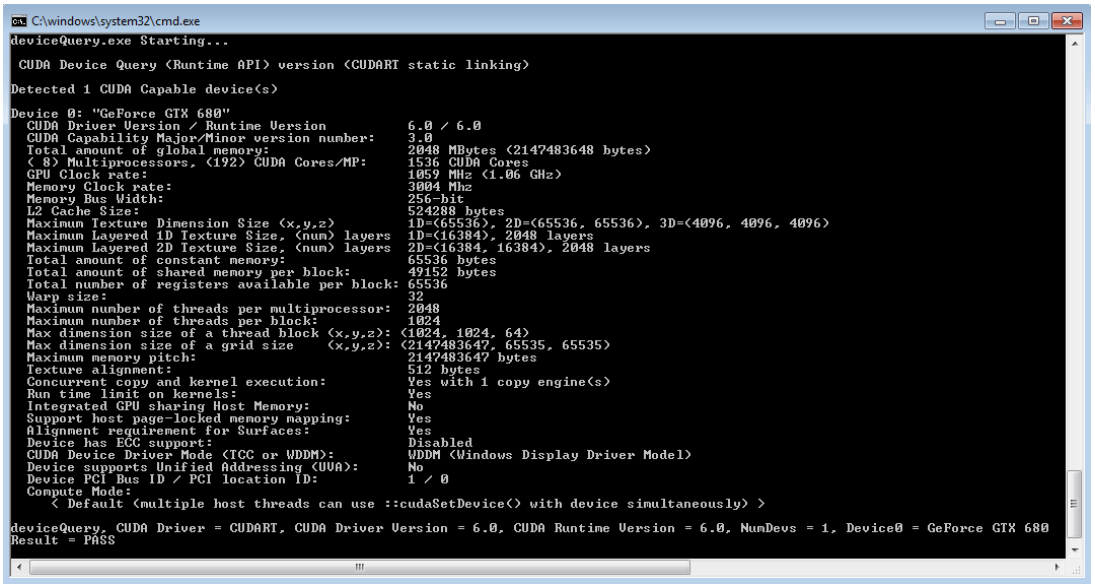

Рисунок 4.1 - Правильные результаты из примера DeviceQuery CUDA

Правильный внешний вид и выходные линии могут отличаться в вашей системе. Важными результатами являются обнаружение устройства, что устройство соответствует тому, что установлено в вашей системе, и что тест прошел.

Если установлено устройство с поддержкой CUDA и драйвер CUDA, но DeviceQuery сообщает, что нет устройств с поддержкой CUDA, убедитесь, что драйвер и драйвер установлены правильно.

Запуск программы bandwidthTest, расположенной в том же каталоге, что и DeviceQuery выше, гарантирует правильную связь системы и устройства с поддержкой CUDA. Результат должен выглядеть как на рисунке 4.2.

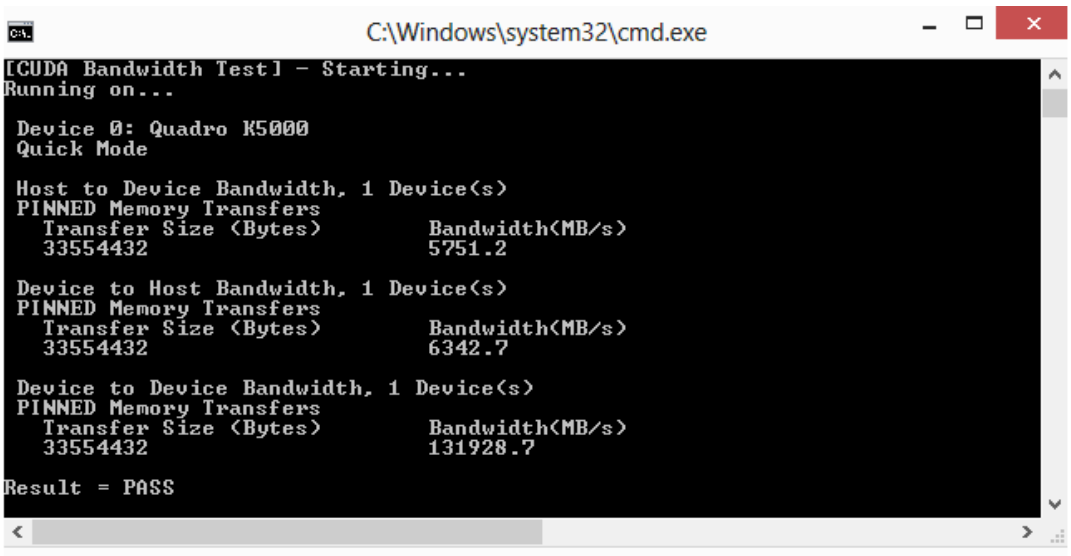

Рисунок 4.2 - Правильные результаты с использованием теста bandwidthTest CUDA

Имя устройства (вторая строка) и номера полосы пропускания варьируются от системы к системе. Важными пунктами являются вторая строка, которая подтверждает, что устройство CUDA найдено, и вторая строка, которая подтверждает, что все необходимые тесты прошли.

Если тесты не пройдут, убедитесь, что в вашей системе имеется графический процессор NVIDIA с поддержкой CUDA и убедитесь, что он правильно установлен.

Чтобы увидеть графическое представление того, что может сделать CUDA, запустите образец Particles, исполняемый в

C:\ProgramData\NVIDIA Corporation\CUDA Samples\v8.0\bin\win64\Release

Компиляция программ CUDA

Файлы проектов в CUDA Samples были разработаны для создания простых однострочных программ, включающих весь исходный код. Для создания проектов Windows (для режима выпуска или отладки) используйте предоставленные файлы решений * .sln для Microsoft Visual Studio 2010, 2012 или 2013. Вы можете использовать файлы решений, расположенные в каждом из каталогов примеров в

C:\ProgramData\NVIDIACorporation\CUDASamples\v8.0\<category>\<sample_name>

Или глобальные файлы решений. Samples * .sln, расположенные в C:\ProgramData\NVIDIA Corporation\CUDA Samples\v8.0

Образцы CUDA организованы в соответствии с <category>. Каждый образец организован в одну из следующих папок: (0_Simple, 1_Utilities, 2_Graphics, 3_Imaging, 4_Finance, 5_Simulations, 6_Advanced, 7_CUDALibraries).

Компиляция примеров проектов

Проект bandwidthTest - хороший примерный проект для создания и запуска. Он расположен в каталоге NVIDIA Corporation \ CUDA Samples \ v8.0 \ 1_Utilities \ bandwidthTest.

Если вы решили использовать место установки по умолчанию, вывод будет помещен в CUDA Samples \ v8.0 \ bin \ win64 \ Release. Создайте программу, используя соответствующий файл решения и запустите исполняемый файл. Если все работает правильно, выход должен быть похож на рисунок 4.2.

Создание настроек для новых проектов

При создании нового приложения CUDA файл проекта Visual Studio должен быть настроен для включения настроек сборки CUDA. Для этого нажмите «Файл»> «Создать | Project ... NVIDIA-> CUDA->, затем выберите шаблон для вашей версии CUDA Toolkit. Например, выбор шаблона «CUDA 8.0 Runtime» позволит настроить ваш проект для использования с CUDA 8.0 Toolkit. Новый проект представляет собой проект C ++ (.vcxproj), который предварительно сконфигурирован для использования настроек сборки NVIDIA. Все стандартные возможности проектов Visual Studio C ++ будут доступны.

Чтобы указать настраиваемое местоположение инструмента CUDA Toolkit, в разделе CUDA C / C ++ выберите «Общий» и настройте поле Custom Dir Toolkit CUDA по желанию. Обратите внимание, что выбранный набор инструментов должен соответствовать версии настроек сборки.

Создание настроек для существующих проектов

При добавлении ускорения CUDA к существующим приложениям соответствующие файлы проекта Visual Studio должны быть обновлены для включения настроек сборки CUDA. Это можно сделать одним из следующих двух способов:

Откройте проект Visual Studio, щелкните правой кнопкой мыши имя проекта и выберите «Построить настройки» ... затем выберите версию инструментария CUDA Toolkit, на которую вы хотите настроить таргетинг.

Кроме того, вы можете настроить свой проект всегда на сборку с самой последней установленной версией CUDA Toolkit. Сначала добавьте настройку сборки CUDA в свой проект, как указано выше. Затем щелкните правой кнопкой мыши имя проекта и выберите «Свойства». В разделе CUDA C / C ++ выберите «Общий» и установите для параметра CUDA Toolkit Custom Dir значение $ (CUDA_PATH). Обратите внимание, что переменная среды $ (CUDA_PATH) устанавливается установщиком.

В то время как Вариант 2 позволит вашему проекту автоматически использовать любую новую версию CUDA Toolkit, которую вы можете установить в будущем, выбор версии инструментария явно, как и в Варианте 1, часто лучше на практике, потому что, если в настройке сборки добавлены новые параметры конфигурации CUDA

Если вы используете переменную среды $ (CUDA_PATH) для целевой версии CUDA Toolkit для построения, и вы выполняете установку или удаление любой версии CUDA Toolkit, вы должны проверить, что переменная среды $ (CUDA_PATH) указывает на Правильный каталог установки CUDA Toolkit для ваших целей. Вы можете получить доступ к переменной окружения $ (CUDA_PATH) с помощью следующих шагов:

- Откройте окно запуска из меню «Пуск»
- Далее: control sysdm.cpl
- Выберите вкладку «Дополнительно» в верхней части окна.
- Нажмите «Переменные среды» в нижней части окна.

Файлы, содержащие код CUDA, должны быть помечены как CUDA C / C ++. Это можно сделать, добавив файл, щелкнув правой кнопкой мыши проект, к которому вы хотите добавить файл, выбрав «Добавить \ Новый элемент», выбрав «NVIDIA CUDA 8.0 \ Code \ CUDA C / C ++», а затем выбрав файл, который хотите добавить.

Технические характеристики ПК, на котором проводилось исследование.

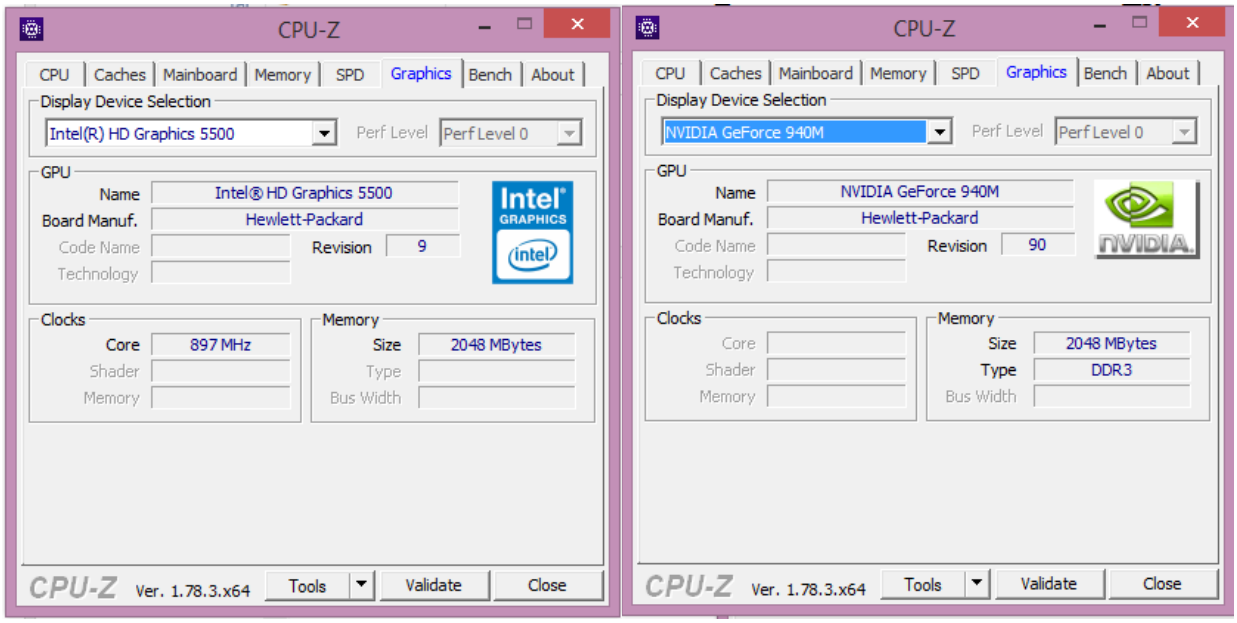

Рисунок 5 – Характеристика GPU

NVIDIA GeForce 940M – среднеуровневая видеокарта с поддержкой DirectX 11, которая была представлена в марте 2015 года. В основе она имеет архитектуру Maxwell, получила 384 шейдерных блоков, изготовлена по нормам 28-нм техпроцесса и работает с 2 ГБ видеопамяти DDR3.

Энергопотебление видеокарты составляет около 30 Вт, что обеспечивает ее использование в 13-14-дюймовых лэптопах и выше.

Производитель:	NVIDIA

Серия:	GeForce 900M
Архитектура:	Maxwell
Потоки:	384-unified
Частота ядра:	1072-1176 (Boost) МГц
Частота памяти:	2000 МГц
Разрядность шины памяти:	64 Бит
Тип памяти:	DDR3
Максимум памяти:	4096 МБ
Общая память:	нет
DirectX:	DirectX 11, Shader 5.0
Технология:	28 нм
Дополнительно:	GPU Boost 2.0, Optimus, PhysX, CUDA, GeForce Experience
Размер ноутбука:	средний
Дата выхода:	12.03.2015

4.2 Структура программы

Основной код программы содержится в файле search.cu и представлен в виде набора исполняемых на устройстве (графическом ускорителе) функций.

1. Код ядра инициализации генератора случайных чисел для каждого экземпляра

```
__global__ void init(unsigned int seed, curandState_t* states)
```

```
{
              curand_init(seed
              blockIdx.x,
              0,&states[blockIdx.x]);
}
```

2. Код ядра равномерного распределения точек внутри текущих границ

```
__device__    void distributionC(square* borders, coord* point, curandState_t* states)
{
       //curandState_t state;
       //curand_init(0, 0, 0, &state);
       int inst = threadIdx.x + blockIdx.x * blockDim.x;
       point[inst]->x1 = borders->X1.x1 + (borders->X1.x2 - borders->X1.x1) *
curand(&states[blockIdx.x]);
       point[inst]->x2 = borders->X2.x1 + (borders->X2.x2 - borders->X2.x1) *
curand(&states[blockIdx.x]);

}
```

3. Ядро вычисления алгоритма случайного поиска

```
__device__ void Ordinary_Search(square current_square, float * F, int func_type, coord crds)
{
       InitSyncWholeDevice(threadIdx.x + blockIdx.x*blockDim.x);

       /* I */

       SyncWholeDevice();

       int inst = threadIdx.x + blockIdx.x * blockDim.x;
       F[inst] = find_ordinary(func_type, crds);

       /* II */

   }
```

4. Ядро вычисления алгоритма с направляющим гиперквадратом

```
__device__ void Ordinary_Search(square current_square, float * F, int func_type, coord crds)
{
       InitSyncWholeDevice(threadIdx.x + blockIdx.x*blockDim.x);

       /* I */

       SyncWholeDevice();

       int inst = threadIdx.x + blockIdx.x * blockDim.x;
       F[inst] = find_hypersquare(func_type, crds);

       /* II */

   }
```

5. Функция инициализации синхронизации исполнения ядер на всем устройстве

```
__device__ void InitSyncWholeDevice(const int index)
{
       if (index == 0) count = 0;
       if (threadIdx.x == 0) while (count != 0);

       __syncthreads();
```

```
}
```

6. Функция синхронизации исполнения ядер на всем устройстве

```
__device__ void SyncWholeDevice()
{
	unsigned int oldc;
	__threadfence();
	{
		oldc = atomicInc((unsigned int*)&count, gridDim.x - 1);
		__threadfence();
		if (oldc != (gridDim.x - 1))
			while (count != 0);
	}
	__syncthreads();
    }
```

В файле rand_search.h содержится код функций, исполняемых на цпу

1. Функция случайного поиска

```
void compute_ordinary(square current_square, int func_type, int iterations)
{

	float step;
	const int instances = 1500;
	float F[instances] = {0};
	float F_min;
	float prev_F;
	int it;
	float eps = 0.05;
	int instance;

	coord point;
	coord init_crds;
	coord current_crds;
	coord prev_crds;

	size_t points_num;


	clock_t time;
	time = clock();
	init_crds = distribution(current_square);
	F[0] = function(func_type, init_crds);
	printf("  %.0d\t", it);
	printf("%.0f (%f, %f)\t \n", F[0], current_crds.x1, current_crds.x2);
	F_min = F[0];
	prev_F = F[0];

	current_crds = init_crds;
	it = 0;

	while (it<= iterations && std::abs(0.0 - F_min) >= eps)
	{
		prev_F = F[instiance];
		prev_crds = current_crds;

		for (size_t i = 0; i < instances; i++)
		{
			current_crds = distribution(current_square);

			F[i] = function(func_type, current_crds);
			if (F_min > F[i])
			{
				F_min = F[i];
			}
		}
		printf(" %.0d, %f (%f, %f)\r",it, F_min, current_crds.x1, current_crds.x2);
```

```
	//	log_("log_file.txt", F_min,it, current_crds);

		it++;
	}
	time = clock() - time;
	float temp = (float)time / CLOCKS_PER_SEC / 60;
	printf("\n %f \r", time);
	//log_time("log_file.txt", temp);
	//cout << "total time in cpu mode\t" << (double)time / CLOCKS_PER_SEC / 60 << "sec" << endl;
}
```

2. Функция с направляющим гиперквадратом

```
void compute_hypersquare(square current_square, int func_type, int iterations)
{
	float init_X1;
	float init_X2;
	float step;
	const int instances = 1500;
	float F[instances] = { 0 };
	float F_min;
	float prev_F;
	int it;
	float eps = 0.05;
	int instance;
	float alpha = 1;

	coord point;
	coord init_crds;
	coord current_crds;
	coord prev_crds;
	square new_square;

	size_t points_num;

	init_crds = distribution(current_square);
	F[0] = function(func_type, init_crds);
	F_min = F[0];
	prev_F = F[0];

	clock_t time;
	time = clock();
	current_crds = init_crds;
	it = 0;

	new_square = current_square;

	while (it <= iterations && std::abs(0.0 - F_min) >= eps)
	{
		prev_F = F[i];
		prev_crds = current_crds;
		current_square = new_square;

		for (size_t i = 0; i < instances; i++)
		{
			current_crds = distribution(current_square);

			F[i] = function(func_type, current_crds);
			if (F_min > F[i])
			{
				F_min = F[i];

			}
		}
		printf(" %.0d, %f (%f, %f) \r", it, F_min, current_crds.x1, current_crds.x2);

		//("log_file.txt", F_min, it, current_crds);

		get_new_hypersquare(current_square, new_square, current_crds, alpha);
```

```
        it++;
}
time = clock() - time;
float temp = (float)time / CLOCKS_PER_SEC / 60;
printf("\n %f \r", time);

//log_time("log_file.txt", temp);

}
```

3. Пример реализации функций для нахождения минимума (с размерностью 2)

```
inline float function_rosenbrock(coord crds)
{
        // min at     f(0 , 0) = 0
        float a = 1;
        float b = 100;
        float res = (a - crds.x1) * (a - crds.x1) + b*(crds.x2 - crds.x1 *
crds.x1)*(crds.x2 - crds.x1 * crds.x1);
        return res;
}

inline float function_himm(coord crds)
{
        // min at   f(3 , 2) = 0
        //          f(-2.805118 ,  3.131312) = 0
        //          f(-3.779310 , -3.283186) = 0
        //          f( 3.584428 , -1.848126) = 0
        float res = (crds.x1 * crds.x1 + crds.x2 - 11) * (crds.x1 * crds.x1 + crds.x2 -
11) + (crds.x1 + crds.x2 * crds.x2 - 7) * (crds.x1 + crds.x2 * crds.x2 - 7);
        return res;
}

inline float function_rastrig(coord crds)
{
        // min at    [0, 0]
        float res = 20 + crds.x1 * crds.x1 + crds.x2 * crds.x2 - 10 * (std::cosf( 2 *
3.1415926535 * crds.x1)  + std::cosf(2 * 3.1415926535 * crds.x2));
        return res;
}
```

Основные этапы CUDA-программы

1. Хост выделяет нужное количество памяти на устройстве.

```
cudaMalloc();
```

1. Хост копирует данные из своей памяти в память устройства.

```
cudaMemcpy();
```

2. Хост стартует выполнение определенных ядер на устройстве.
   Параметры запуска ядер

```
dim3 blockSize;
dim3 gridSize;
int threadNum;
```

   соответственно `blockSize` – задаваемый размер блока

`gridSize` – размер сетки

```
gridSize = dim3(instances, 1, 1);
```

`instances` – количество запускаемых экземпляров вычислений функции

`blockSize = dim3(threadNum, 1, 1);` - задание размерности блока

`threadNum = 1024;` - максимально возможное количество тредов на устройстве

3. Устройство выполняет ядра.

```
__device__ void Ordinary_Search<<<gridSize, blockSize>>>( current_square, F, func_type, crds);

__device__ void Hypersquare_Search<<<gridSize, blockSize>>>( current_square, F, func_type, crds);
```

4. Хост копирует результаты из памяти устройства в свою память.

Для наибольшей эффективности использования GPU нужно чтобы соотношение времени, потраченного на работу ядер, к времени, потраченному на выделение памяти и перемещение данных, было как можно больше.

5 РЕЗУЛЬТАТ РАЗРАБОТКИ ПРИЛОЖЕНИЯ

Для проведения численных экспериментов была разработана программа, в которой реализованы описанные ранее алгоритмы оптимизации - Случайный поиск и алгоритм наилучшей пробы с направляющим гиперквадратом. В реализации предусмотренно исполенние программного кода на устройстве с поддержкой CUDA технологии, CPU, и CPU с использованием технологии OpenMP.

Рабочий вид приложения представлен на рисунке 5.1

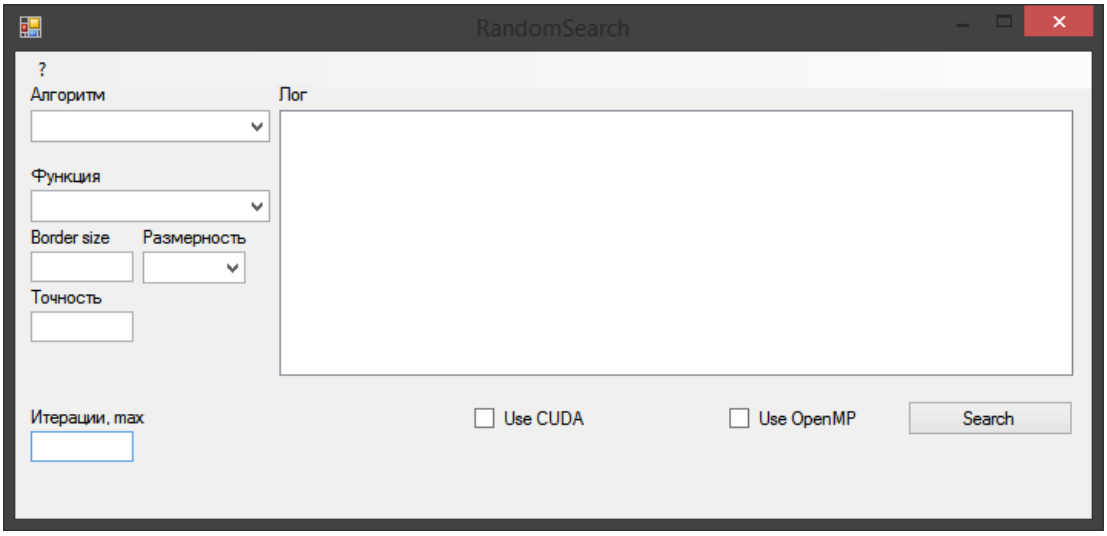

Рисунок 5.1 – Интерфейс программы

Для работы с программой нужно из выпадающего списка выбрать нужный алгоритм и функцию, также нужно задать начальные переменные, точность, количество итераций и границы. После этого выбираем нужное исполнение программного кода с поддержкой CUDA-технологии и нажимаем кнопку Search.

Рассмотрим работу программы с использованием функции Розенброка. Выберем алгоритм простого случайного. Далее зададим точность и число итераций. Задаем технологию OpenMP. После этого нажимаем на кнопку Search и получаем результат.

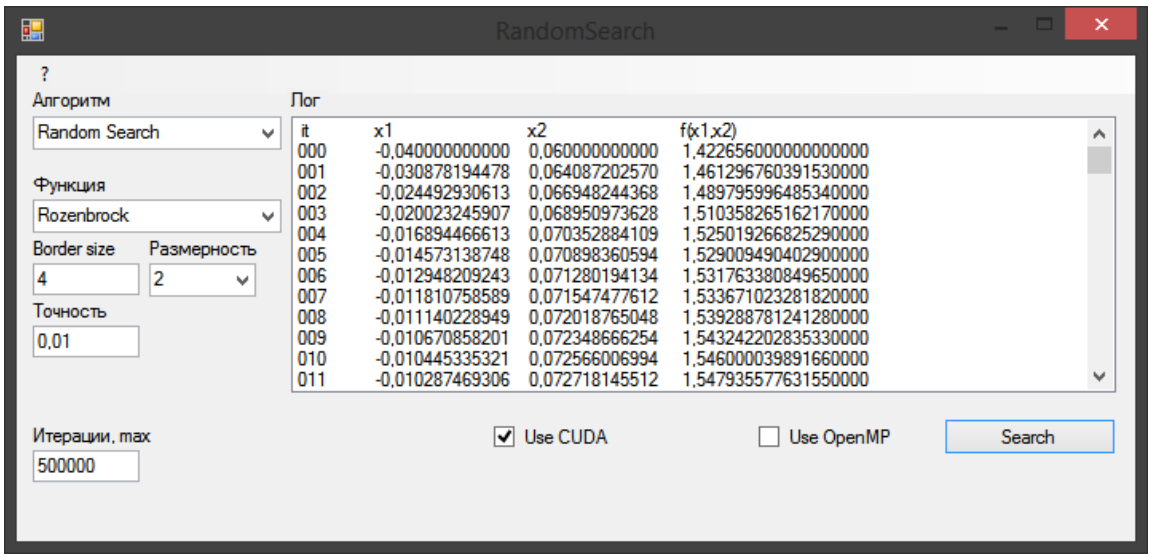

Рисунок 5.3 - Результат работы приложения для простого случайгного поиска на функции Розенброка.

В результате получили время нахождения экстремума для функции Розенброка по методу простого случайного поиска, при заданных параметрах: сторонв квадрата =4, в двумерном пространстве с точностью 0,01. Также задали количество итераций 500000.

Результаты численных экспериментов на функции Растригина

Параметры:

Точность eps = 0.01

Ограничение по количеству итераций 500 000

Таблица 5.1. Размерность = 2

Алгоритм	CPU, время сек	CPU, OpenMP, время сек	GPU, CUDA, время сек
Случайный поиск	43.003	38.540	12.090
Наилучшей пробы с направляющим гиперквадратом	24.112	29.010	11.370

Таблица 5.2. Размерность = 3

Алгоритм	CPU, время сек	CPU, OpenMP, время сек	GPU, CUDA, время сек
Случайный поиск	57.124	49.000	14.010
Наилучшей пробы с направляющим гиперквадратом	34.178	32.149	13.730

Таблица 5.3. Размерность = 4

Алгоритм	CPU, время сек	CPU, OpenMP, время сек	GPU, CUDA, время сек
Случайный поиск	84.0	62.000	30.007
Наилучшей пробы с направляющим гиперквадратом	52.0	49.001	24.031

Эксперимент на функции Растригина показал, что при увеличении размерности производительность падает, результате численных экспериментов проводившихся на различном наборе функций с размерностями 2,3,4 можно сделать вывод об эффективности использования GPU для задач оптимизации. Однако стоит отметить что наблюдать прирост производительности лишь в случае, когда соотношение времени, потраченного на выделение и перемещение/работу с данными ко времени исполнения ядер, было как можно меньше. Для более ясного понимания представим результаты в графическом виде. На графике 5.4 представлен результат на функции Растригина.

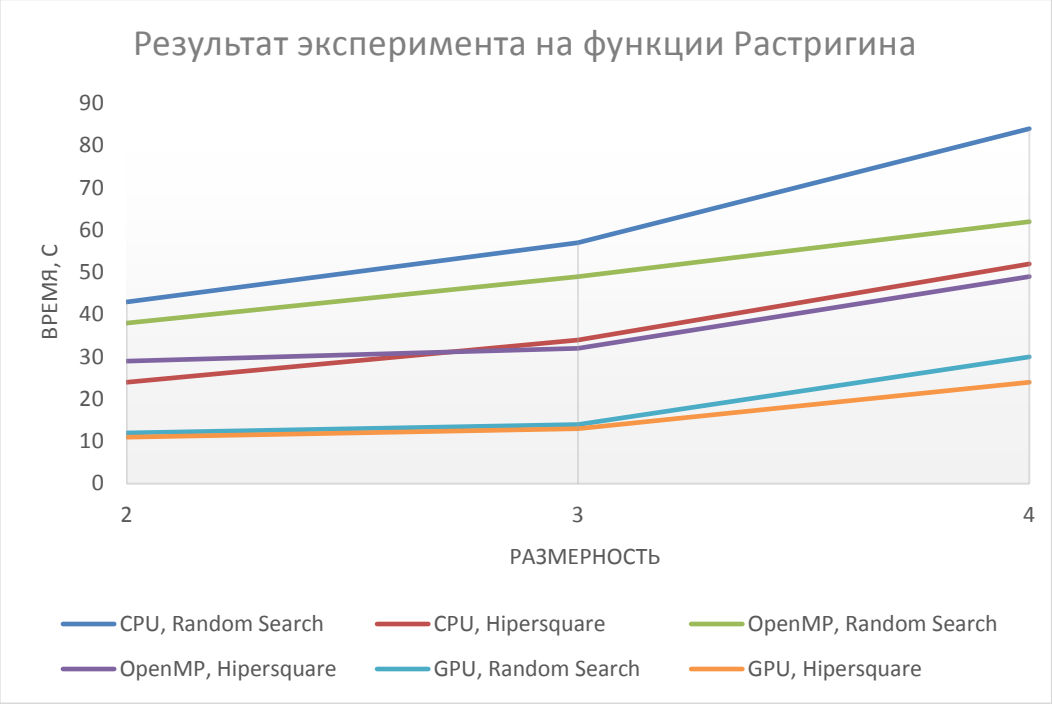

Рисунок 5.4 – Результат эксперимента на функции Растригина

Из графика видно, что при повышении размерности, производительность всех алгоритмов падает.

Также сравним на графике 5.5. производительность исполняемой реализации на графическом и центральном процессах.

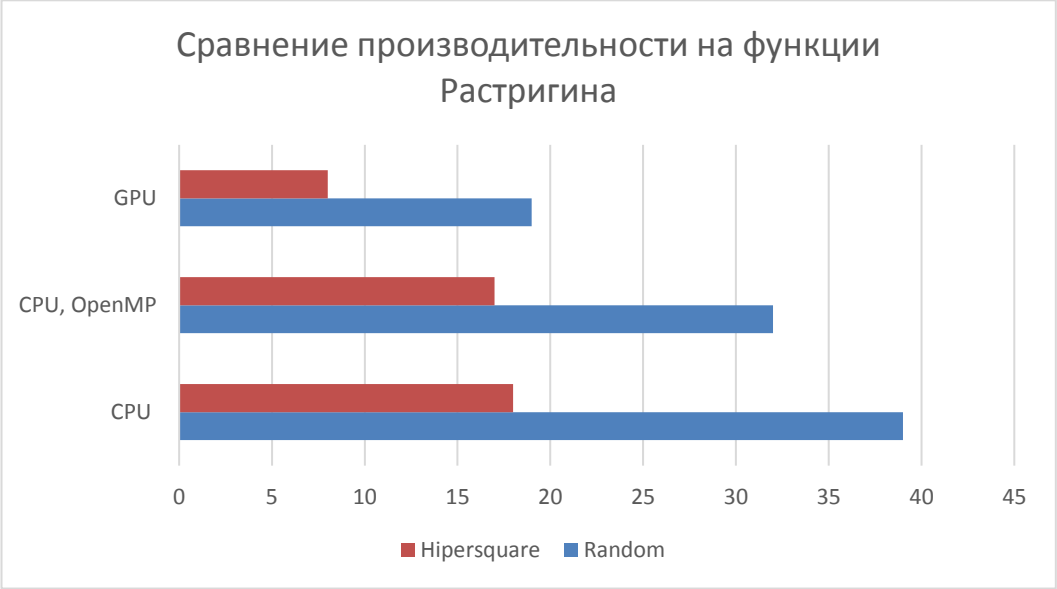

Рисунок 5.5. – Сравнение производительности на функции Растригина

Как видно из графика реализация в виде исполняемого на графическом процессоре кода позволяет получить прирост в производительности – до 6 раз.

Результаты численных экспериментов на функции Розенброка

Параметры:

Точность eps = 0.005

Ограничение по количеству итераций 500 000

Таблица 5.4. Размерность = 2

Алгоритм	CPU, время сек	CPU, OpenMP, время сек	GPU, CUDA, время сек
Случайный поиск	39.043	32.432	10.026
Наилучшей пробы с направляющим гиперквадратом	18.003	17.026	8.211

Таблица 5.5. Размерность = 3

Алгоритм	CPU, время сек	CPU, OpenMP, время сек	GPU, CUDA, время сек
Случайный поиск	62.315	55.003	12.006
Наилучшей пробы с направляющим гиперквадратом	38.702	31.210	10.214

Таблица 5.6. Размерность = 4

Алгоритм	CPU, время сек	CPU, OpenMP, время сек	GPU, CUDA, время сек
Случайный поиск	69.140	55.049	19.040
Наилучшей пробы с	49.098	42.306	11.148

направляющим гиперквадратом			

Эксперимент на функции Розенброка показал, что при увеличении размерности производительность падает, результате численных экспериментов проводившихся на различном наборе функций с размерностями 2,3,4 можно сделать вывод об эффективности использования GPU для задач оптимизации. Однако стоит отметить что наблюдать прирост производительности лишь в случае, когда соотношение времени, потраченного на выделение и перемещение/работу с данными ко времени исполнения ядер, было как можно меньше. Для более ясного понимания представим результаты в графическом виде. На графике 5.6 представлен результат на функции Розенброка.

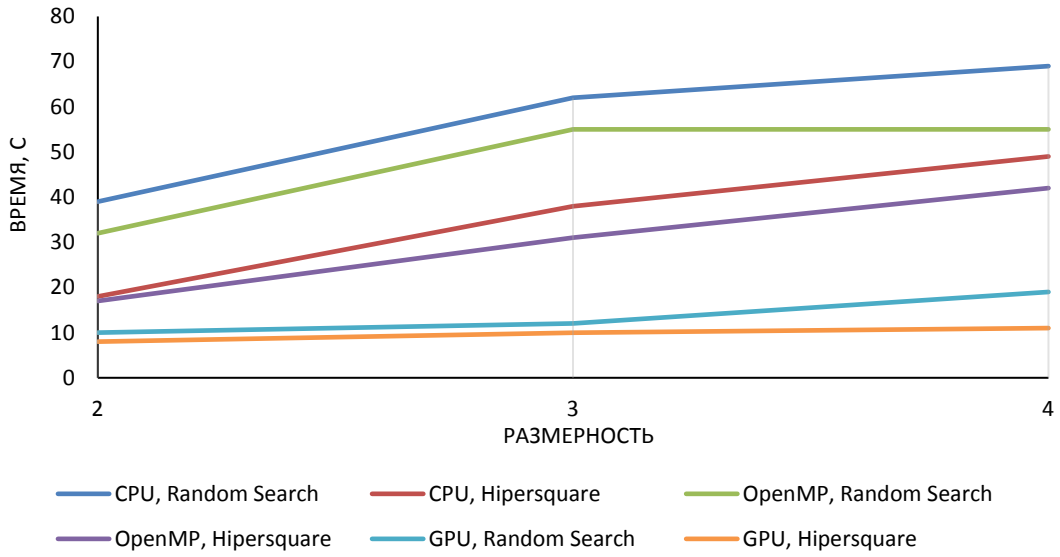

Рисунок 5.6 - Результат эксперимента на функции Розенброка

Из графика видно, что при повышении размерности, производительность всех алгоритмов падает.

Также сравним на графике 5.7. производительность исполняемой реализации на графическом и центральном процессах.

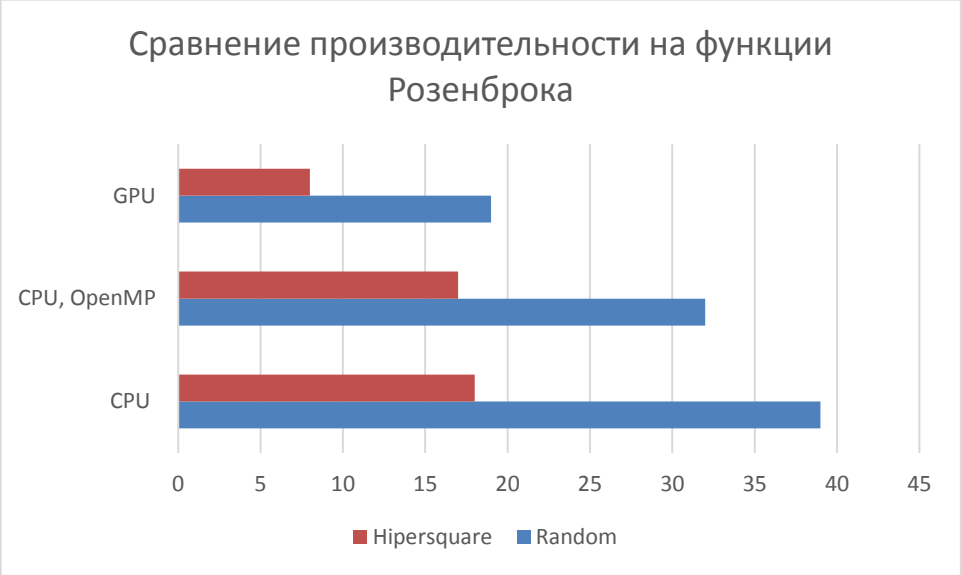

Рисунок 5.7 – Сравнение производительности на функции Розенброка

Как видно из графика реализация в виде исполняемого на графическом процессоре кода позволяет получить прирост в производительности, также ка и на функции Растригина– до 6 раз.

6 ФИНАНСОВЫЙ МЕНЕДЖМЕНТ, РЕСУРСОЭФФЕКТИВНОСТЬ И РЕСУРСОСБЕРЕЖЕНИЕ

Целью данного раздела является анализ перспективности проведения научно-исследовательской работы и технико-экономическое обоснование разработки системы, а также определение организационных и экономических условий её эффективного функционирования.

Научно-исследовательская работа (НИР) направлена на применение параллельного программирования технологии CUDA в методах случайной оптимизации.

Главная задача разрабатываемого приложения - минимизировать обмен данными между памятью GPU и оперативной памятью, также произвести перенос основных вычислений на GPU и их распараллеливание.

Вычисления, необходимые для расчета схемы довольно громоздки и требуют больших вычислительных мощностей. Как правило, для данной задачи требуется расчет для большого объема входных данных, что занимает довольно много времени. Для ускорения работы алгоритма принято решение выполнять алгоритм в несколько потоков. Сделать это удалось следующим образом : весь массив данных хранится в глобальной памяти, к которой имеют доступ все потоки, затем каждый поток рассчитывает значение потока через правую границу ячейки. Какие ячейки в каком потоке обрабатываются решается в зависимости от количества потоков.

6.1 Организация и планирование работ

При организации работы, в том числе и научно-исследовательской необходимо планировать занятость каждого участника и определить сроки выполнения отдельных работ.

На данном этапе должен быть определен полный перечень работ, их исполнители и продолжительность. В результате планирования должен быть составлен график (линейный или сетевой) выполнения НИР. Линейный график является наиболее удобным и компактным способом представления

данных. Для его построения составляется перечень работ, и назначаются их исполнители с указание продолжительности. Полученный график выполнения НИР приведен в таблице 6.1.

Таблица 6.1 – Перечень работ и продолжительность их выполнения

Этапы работы	**Исполнители**	**Загрузка исполнителей**
1. Постановка целей и задач, получение исходных данных	НР	НР – 100%
2. Составление и утверждение ТЗ	НР, С	НР –90% С – 10%
3. Подбор и изучение материалов по тематике	НР, С	НР – 20% С – 80%
4. Разработка календарного плана	НР, С	НР – 70% С – 30%
5. Обсуждение литературы	НР, С	НР – 40% С – 60%
6. Выбор структурных элементов метода	НР, С	НР – 50% С – 50%
7. Реализация метода	НР, С	С – 100%
8. Тестирование и доработка метода	НР, С	НР – 20% С – 80%
9. Анализ результатов	НР, С	НР – 30% С – 70%
10. Оформление пояснительной записки	С	С – 100%
11. Проверка работы	НР, С	НР – 60% С – 40%

Примечание к таблице 6.1:

НР — научный руководитель;

С — студент.

6.1.1 Продолжительность этапов работ

Расчет продолжительности этапов работ может выполняться двумя методами: технико-экономическим, опытно-статистическим.

В данном случае используется опытно-статистический метод, который может быть реализован двумя способами: аналоговый и вероятностный.

Для расчета ожидаемого значения продолжительности работ $t_{ож}$ применяется две оценки: t_{min} и t_{max} (метод двух оценок).

$$t_{ожс} = \frac{3 \cdot t_{min} + 2 \cdot t_{max}}{5}, \quad (6.1)$$

где t_{min} – минимальная трудоемкость работ, чел/дн.;

t_{max} – максимальная трудоемкость работ, чел/дн.

Для выполнения перечисленных в таблице 5.1 работ требуются специалисты: студент, научный руководитель.

Для построения линейного графика необходимо рассчитать длительность этапов в рабочих днях, а затем перевести в календарные дни. Расчет продолжительности выполнения каждого этапа в рабочих днях выполняется по формуле:

$$T_{РД} = \frac{t_{ож}}{K_{ВН}} \cdot K_{Д}, \quad (6.2)$$

где $t_{ож}$ – трудоемкость работы, чел/дн.;

$K_{ВН}$ – коэффициент выполнения работ ($K_{ВН}$ = 1);

$K_{Д}$ – коэффициент, учитывающий дополнительное время на компенсации и согласование работ ($K_{Д}$ = 1,2).

$$T_{К} = \frac{T_{КАЛ}}{T_{КАЛ} - T_{ВД} - T_{ПД}}, \quad (6.3)$$

где $T_{КАЛ}$ – календарные дни ($T_{КАЛ}$ = 366);

$T_{ВД}$ – выходные дни ($T_{ВД}$ = 52);

$T_{ПД}$ – праздничные дни ($T_{ПД}$ = 12).

$$T_{К} = \frac{365}{365 - 52 - 10} = 1{,}205 \quad (6.5)$$

В таблице 6.2 приведена длительность этапов работ и число исполнителей, занятых на каждом этапе.

Таблица 6.2 – Трудозатраты на выполнение проекта

Этап	**Исполнители**	**Продолжительность работ, дни**			**Длительность работ, чел/дн.**			
					$T_{РД}$		$T_{КД}$	
		t_{min}	t_{max}	$t_{ож}$	**НР**	**С**	**НР**	**С**
1. Постановка целей и задач, получение исходных данных	НР	3	5	3.8	4.56	0	5.52	0
2. Составление и утверждение ТЗ	НР, С	2	4	2.8	3.02	0.33	3.66	0.40
3. Подбор и изучение материалов по тематике	НР, С	20	25	22	5.28	21.12	6.39	25.5
4. Разработка календарного плана	НР, С	1	2	1.4	1.17	0.50	1.42	0.6
5. Обсуждение литературы	НР, С	3	6	4.2	2.02	3.02	2.44	3.66
6. Выбор структурных элементов метода	НР, С	5	7	5.8	3.48	3.48	4.21	4.21
7. Реализация метода	НР, С	12	25	17.2	0	20.64	0	24.97
8. Тестирование и доработка метода	НР, С	10	20	14	3.36	13.44	4.06	16.26
9. Анализ результатов	С	4	8	5.6	2.02	4.70	2.44	5.69
10. Оформление пояснительной записки	С	15	25	19	0	22.8	0	27.59
11. Проверка работы	НР, С	4	8	5.6	4.03	2.68	4.88	3.25
Итого:				101.4	28.94	92.73	35.02	112.21

6.1.2 Расчет нарастания технической готовности

Величина нарастания технической готовности работы показывает, на сколько процентов выполнена работа на каждом этапе. Данная величина вычисляется по формуле:

$$\text{Н}_i = \frac{t_{\text{Н}i}}{t_{\text{О}}} \cdot 100\%, \quad (6.5)$$

где $t_{\text{Нi}}$ – нарастающая трудоемкость с момента начала работы i-го этапа;

$t_{\text{О}}$ – общая трудоемкость.

Общая трудоемкость вычисляется по формуле:

$$t_{\text{О}} = \sum_{i=1}^{n} t_{\text{ОЖ}i}, \quad (6.6)$$

где $t_{\text{ОЖi}}$ – ожидаемая продолжительность i-го этапа.

Удельный вес каждого этапа $У_i$ определяется по формуле:

$$\text{У}_i = \frac{t_{\text{ОЖ}i}}{t_{\text{О}}} \cdot 100\% \quad (6.7)$$

Результаты вычислений H_i и $У_i$ отражены в таблице 5.3.

Таблица 6.3 — Нарастание технической готовности работы и удельный вес каждого этапа

Этап	$t_{ож}$	$t_{Нi}$	**Н$_i$, %**	**У$_i$, %**
1. Постановка целей и задач, получение исходных данных	3.8	3.8	3.75	3.75
2. Составление и утверждение ТЗ	2.8	6.6	6.51	2.76
3. Подбор и изучение материалов по тематике	22	28.6	28.20	21.69
4. Разработка календарного плана	1.4	30	29.58	1.38

5. Обсуждение литературы	4.2	34.2	33.73	4.14
6. Выбор структурных элементов метода	5.8	40	39.45	5.72
7. Реализация метода	17.2	57.2	56.41	16.96
8. Тестирование и доработка метода	14	71.2	70.22	13.8
9. Анализ результатов	5.6	76.8	75.74	5.52
10. Оформление пояснительной записки	19	95.8	94.48	18.74
11. Проверка работы	5.6	101.4	100	5.52
	101.4			

Таблица 6.4 иллюстрирует получившийся линейный график работ на основе рассчитанного для студента и научного руководителя времени $T_{КД}$.

Таблица 5.4 — Линейный график работ

Этап	Ткд, НР	Ткд, С	Дата начала	Дата окончания	Январь				Февраль				Март				Апрель				Май			
					1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
1. Постановка целей и задач, получение исходных данных	5.52	0	01.01.2017	06.01.2017																				
2. Составление и утверждение ТЗ	3.66	0.40	06.01.2017	10.01.2017																				
3. Подбор и изучение материалов по тематике	6.39	25.5	10.01.2017	12.02.2017																				
4. Разработка календарного плана	1.42	0.6	12.02.2017	14.02.2017																				
5. Обсуждение литературы	2.44	3.66	14.02.2017	20.02.2017																				
6. Выбор структурных элементов метода	4.21	4.21	20.02.2017	1.03.2017																				
7. Реализация метода	0	24.97	1.03.2017	25.03.2017																				
8. Тестирование и доработка метода	4.06	16.26	25.03.2017	14.04.2017																				
9. Анализ результатов	2.44	5.69	14.04.2017	22.04.2017																				
10. Оформление пояснительной записки	0	27.59	22.04.2017	20.05.2017																				
11. Проверка работы	4.88	3.25	20.05.2017	28.05.2017																				

6.2 Расчет стоимости

В состав затрат на научно-исследовательскую работу включается стоимость всех расходов, необходимых для ее выполнения. Расчет сметной стоимости на выполнение НИР производится по следующим статьям затрат:

— материалы и покупные изделия;
— заработная плата;
— отчисления в социальные фонды;
— расходы на электроэнергию;
— амортизационные отчисления;
— работы, выполняемые сторонними организациями;
— прочие расходы.

6.2.1 Расчет затрат на материалы

К данной статье расходов относится стоимость материалов, покупных изделий, полуфабрикатов и других материальных ценностей, расходуемых непосредственно в процессе выполнения работ.

Данная работа выполнялась преимущественно на компьютерном оборудовании и не потребовала существенных затрат на материалы. Мелкие расходы (канцелярия и др.) могут быть отнесены к статье прочих расходов (см. п. 5.2.6). Следовательно, можно принять $С_М = 0$.

6.2.2 Расчет заработной платы

Данная статья расходов включает заработную плату научного руководителя и студента, а также премии, входящие в фонд заработной платы. Расчет основной заработной платы выполняется на основе трудоемкости выполнения каждого этапа и величины месячного оклада исполнителя.

Среднедневная заработная плата рассчитывается по формуле:

$$ЗПдн\text{-}т = МО/24{,}83 \quad (6.8)$$

Расчеты затрат на заработную плату приведены в таблице 5.5. При расчете учитывалось, что в году 301 рабочий день и, следовательно, в месяце

25,08 рабочих дня. Затраты времени на выполнение работы по каждому исполнителю брались из таблицы 5.2. Также был принят во внимание коэффициент, учитывающий коэффициент по премиям $K_{ПР}$ = 1,1, коэффициент дополнительной заработной платы (учитывающий оплату отпуска, простой и пр.) $К_{ДОП}$ = 1,188 и районный коэффициент $K_{РК}$ = 1,3.

Таблица 6.5 — Затраты на основную заработную плату

Исполнитель	**Оклад, руб./мес**	**Среднедневная ставка, руб./день**	**Затраты времени, дни**	**Коэффициент**	**Фонд з/платы, руб.**
НР	30000	1196,17	29	1,57	54461,62
С	10000	398,72	93	1,57	58217,11
Итого:					**112678,72**

Таким образом, затраты на заработную плату составили СЗП = 112678,72 руб.

6.2.3 Расчет отчислений от заработной платы

Затраты по этой статье составляют отчисления по единому социальному налогу (ЕСН).

Отчисления по заработной плате определяются по следующей формуле:

$$Ссоц. = Сзп*0,27 \quad (6.9)$$

где $K_{СОЦ}$ – коэффициент, учитывающий размер отчислений из заработной платы. Данный коэффициент составляет 30% от затрат на заработную плату и включает в себя:

— отчисления в пенсионный фонд;

— на социальное страхование;

— на медицинское страхование.

Итак, отчисления из заработной платы составили:

$$С_{СОЦ} = 0{,}27 \cdot 112\,678{,}72 = 33\,803{,}62 \text{ руб.}$$

6.2.4 Расчет затрат на электроэнергию

Данный вид расходов включает в себя затраты на электроэнергию при работе оборудования, а также затраты на электроэнергию, потраченную на освещение. Затраты на электроэнергию при работе оборудования для технологических целей рассчитываются по формуле:

$$Э_{ОБ} = P_{ОБ} \cdot Ц_{Э} \cdot t_{ОБ}, \quad (6.10)$$

где $Э_{ОБ}$ – затраты на электроэнергию, потребляемую оборудованием, руб.;

$P_{ОБ}$ – мощность, потребляемая оборудованием, кВт;

$Ц_{Э}$ – тарифная цена за 1 кВт·час,

Для ТПУ $Ц_{Э}$ = 5,789 руб/кВт·час (с НДС);

$t_{ОБ}$ – время работы оборудования, час.

Время работы оборудования вычисляется на основе итоговых данных таблицы 5.7 для инженера (ТРД) из расчета, что продолжительность рабочего дня равна 8 часов.

$$tоб = ТРД * Kt, \quad (6.11)$$

где Kt £ 1– коэффициент использования оборудования по времени, равный отношению времени его работы в процессе выполнения проекта к ТРД, определяется исполнителем самостоятельно. В ряде случаев возможно определение tоб путем прямого учета, особенно при ограниченном использовании соответствующего оборудования.

Мощность, потребляемая оборудованием, определяется по формуле:

$$РОБ = Рном. * КС \quad (6.12)$$

где Рном. – номинальная мощность оборудования, кВт;

КС £ 1 – коэффициент загрузки, зависящий от средней степени использования номинальной мощности. Для технологического оборудования малой мощности КС = 1.

Затраты на электроэнергию для технологических целей приведены в таблице 6.7.

Таблица 6.1 – Затраты на электроэнергию для технологических целей

Наименование оборудования	**Время работы оборудования $t_{ОБ}$, час**	**Потребляемая мощность $P_{ОБ}$, кВт**	**Затраты $Э_{ОБ}$, руб.**
Ноутбук	742	0,1	192,92
Струйный принтер	30	0,1	7,8
Итого:			**200,72**

6.2.5 Расчет амортизационных расходов

В статье «Амортизационные отчисления» от используемого оборудования рассчитывается амортизация за время выполнения работы для оборудования, которое имеется в наличии.

Амортизационные отчисления рассчитываются на время использования ЭВМ по формуле:

$$С_{АМ} = \frac{Н_А * Ц_{ОБ} * t_{рф} * n}{F_Д}, \quad (6.13)$$

где $Н_А$ – годовая норма амортизации, $Н_А$ = 25%;

$Ц_{ОБ}$ – цена оборудования, $Ц_{ОБ}$ = 27 000 руб.;

$F_Д$ – действительный годовой фонд рабочего времени, $F_Д$ = 298*8 = 2384 часов;

$t_{ВТ}$ – время работы вычислительной техники во время проведения НИР, $t_{ВТ} = 101,4 \cdot 8 = 811,2$ часа;

n – число задействованных ПЭВМ, n = 1.

Итак, затраты на амортизационные отчисления составили:

6.2.6 Расчет накладных расходов

В статье «Накладные расходы» отражены расходы на разработку проекта, которые не учтены в предыдущих статьях.

В том случае, когда основную статью расходов составляет заработная плата, накладные расходы принимают равными 50 % от единовременных затрат на выполнение технического продукта и вычисляют по формуле:

$$C_{ПРОЧ} = (112\ 678{,}72 + 33\ 803{,}62) \cdot 0{,}5 = 73\ 241{,}17 \text{ руб.} \quad (6.14)$$

6.2.7 Расчет общей себестоимости

Проведя расчет сметы затрат на разработку, можно определить общую стоимость разработки проекта «Разработка и оптимизация алгоритма предобработки изображений для оптического распознавания символов».

Таблица 6.8 — Смета затрат на разработку проекта

Статья затрат	Условное обозначение	Сумма, руб.
1 Материалы и покупные изделия	$C_{МАТ}$	0
2 Основная заработная плата	$C_{ОСН}$	112 678,72
3 Отчисления в социальные фонды	$C_{СОЦ}$	33 803,62
4 Расходы на электроэнергию	$C_{Э}$	953,1
5 Амортизационные отчисления	$C_{АМ}$	2 266,39
6 Работы, выполняемые сторонними организациями	$C_{СТОР}$	0
7 Прочие расходы	$C_{ПРОЧ}$	73 241,17
Итого:		**222 943,1**

Таким образом, расходы на разработку составили C = 222 943,1 руб.

6.3 Оценка эффективности проекта

Выполнение научно-исследовательских работ оценивается уровнями достижения экономического, научного, научно-технического и социального эффектов.

Научный эффект характеризует получение новых научных знаний и отображает прирост информации, предназначенной для внутринаучного потребления. Научно-технический эффект характеризует возможность использования результатов в других проектах и обеспечивает получение информации, необходимой для создания новой техники. Экономический эффект характеризуется выраженной в стоимостных показателях экономией живого общественного труда. Социальный эффект проявляется в улучшении условий труда.

Для итоговой оценки результатов проекта в зависимости от поставленных целей в качестве критерия эффективности принимается один из видов эффекта, а остальные используются в качестве дополнительных характеристик.

Результаты данной научно-исследовательской работы применяются в программном продукте, разработанного для закрытого внутреннего пользования, потому оценить экономический эффект в количественных показателях не представляется возможным. Поэтому в качестве критерия эффективности проекта оценим научно-технический уровень НИР.

6.3.1 Оценка научно-технического уровня НИР

Научно-технический уровень характеризует, в какой мере выполнены работы и обеспечивается научно-технический прогресс в данной области. Для оценки научной ценности, технической значимости и эффективности, планируемых и выполняемых НИР, используется метод бальных оценок. Бальная оценка заключается в том, что каждому фактору по принятой шкале присваивается определенное количество баллов. Обобщенную оценку проводят по сумме баллов по всем показателям или рассчитывают по формуле. На этой основе делается вывод о целесообразности НИР.

Сущность метода заключается в том, что на основе оценок признаков работы определяется коэффициент ее научно-технического уровня по формуле:

$$K_{\text{НТУ}} = \sum_{i=1}^{3} R_i \cdot n_i, \qquad (6.15)$$

где $K_{\text{НТУ}}$ – коэффициент научно-технического уровня;

R_i – весовой коэффициент i-го признака научно-технического эффекта;

n_i – количественная оценка i-го признака научно-технического эффекта, в баллах.

Таблица 6.10 — Весовые коэффициенты признаков НТУ

Признак научно-технического эффекта НИР	**Характеристика признака НИОКР**	R_i
1Уровень новизны	Систематизируются и обобщаются сведения, определяются пути дальнейших исследований	0,4
2 Теоретический уровень	Разработка способа (алгоритм, программа мероприятий, устройство, вещество и т.п.)	0,1
3 Возможность реализации	Время реализации в течение первых лет	0,5

Таблица 6.11 — Баллы для оценки уровня новизны

Уровень новизны	**Характеристика уровня новизны**	**Баллы**
Принципиально новая	Новое направление в науке и технике, новые факты и закономерности, новая теория, вещество, способ	8 – 10
Новая	По-новому объясняются те же факты, закономерности, новые понятия дополняют ранее полученные результаты	5 – 7
Относительно новая	Систематизируются, обобщаются имеющиеся сведения, новые связи между известными факторами	2 – 4
Не обладает новизной	Результат, который ранее был известен	0

Таблица 6.12 — Баллы значимости теоретических уровней

Теоретический уровень полученных результатов	**Баллы**
1 Установка закона, разработка новой теории	10
2 Глубокая разработка проблемы, многоспектральный анализ, взаимодействия между факторами с наличием объяснений	8
3 Разработка способа (алгоритм, программа и т. д.)	6
4 Элементарный анализ связей между фактами (наличие гипотезы, объяснения версии, практических рекомендаций)	2
5 Описание отдельных элементарных факторов, изложение наблюдений, опыта, результатов измерений	0,5

Таблица 6.13 — Возможность реализации научных, теоретических результатов по времени и масштабам

Время реализации	**Баллы**
В течение первых лет	10
От 5 до 10 лет	4
Свыше 10 лет	2

Обоснование оценки признаков НИР приводится в таблице 6.14.

Таблица 6.14 — Сводная таблица оценки научно-технического уровня НИР

Фактор НТУ	**Значимость**	**Уровень фактора**	**Выбранный балл**	**Обоснование выбранного балла**
Уровень новизны	0,4	Относительно новая	4	Создание оригинального способа на основе наиболее перспективных новых алгоритмов
Теоретический уровень	0,1	Разработка алгоритма	7	Предложен новый алгоритм предобработки зашумленных изображений
Возможность реализации	0,5	В течение первых лет	10	Возможность быстрого внедрения в существующие системы

Исходя из оценки признаков, показатель научно-технического уровня для данного проекта составил:

$$K_{НТУ} = 0,4 \cdot 4 + 0,1 \cdot 7 + 0,5 \cdot 10 = 1,6 + 0,7 + 5 = 7,3$$

Таблица 6.15 — Оценка уровня научно-технического эффекта

Уровень НТЭ	**Показатель НТЭ**
Низкий	1-4
Средний	4-7
Высокий	8-10

Исходя из данных в таблице 6.15, проект «Применение параллельного программирования технологии CUDA в методах случайной оптимизации» имеет средний уровень научно-технического эффекта.

7 СОЦИАЛЬНАЯ ОТВЕТСТВЕННОСТЬ

В соответствии со стандартом ICCSR26000: 2011, социальная ответственность – это ответственность организации за воздействие ее деятельности и решений на общество и окружающую среду через прозрачное и этичное поведение, которое:

- способствует устойчивому развитию, включая благосостояние и здоровье общества;
- учитывает ожидания заинтересованных сторон;
- соответствует применяемому законодательству;
- согласуется с международными нормами поведения;
- интегрировано в деятельность всей организации и применяется в ее взаимоотношениях.

Деятельность включает в себя продукты, услуги и процессы. Взаимоотношения относятся к деятельности организации в рамках ее сферы влияния.

Общие сведения об объекте.

Объектом диссертации являются работы, направленные на применение параллельного программирования технологии CUDA в методах случайной оптимизации:

- анализ научных статей, посвященных новым методам и способам случайной оптимизации;
- формирование задач;
- разработка алгоритмов;
- программная реализация;
- тестирование.

Описанные выше работы проводятся в помещении, далее кабинет, на кафедре Информационных Систем и Технологий Института Кибернетики. Кабинет оснащен компьютерной техникой:

- персональные компьютеры (ПК) – под ПК будем понимать

совокупность из монитора, системного блока, клавиатуры, мыши и проводов для подключения описанных выше устройств;

- принтеры;
- проекторы;
- столы и стулья;
- телефоны;
- распределительный щиток;
- огнетушители.

На рисунке 7.1 представлена схема помещения. Данный кабинет относится к классу помещений без повышенной опасности, так как отсутствуют условия, создающие повышенную или особо повышенную опасность.

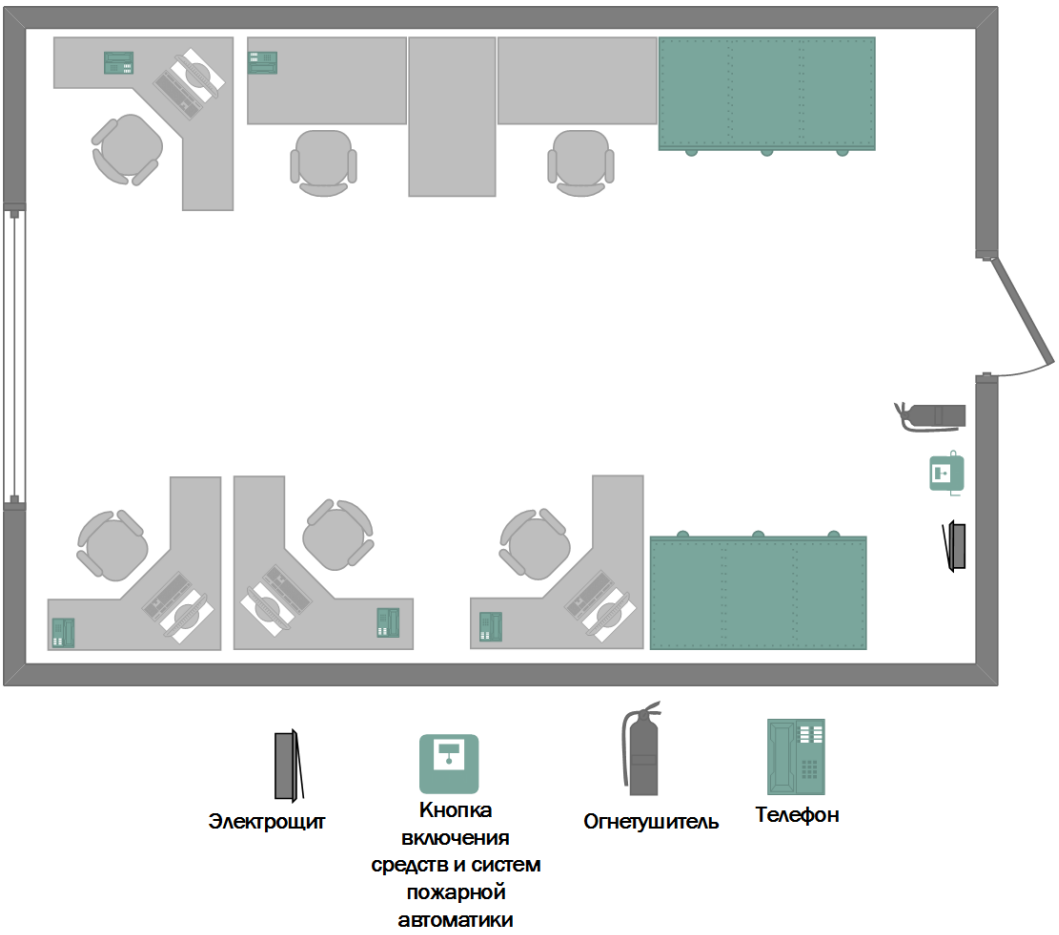

Рисунок 7.1 – План помещения

Также кабинетное помещение оснащено противопожарной сигнализацией и датчиками дыма.

Согласно требованиям, СанПиН 2.2.2/2.4.1340–03, расстояние между

рабочими столами с видеомониторами, равно 2 м, а расстояние между боковыми поверхностями видеомониторов примерно 1,2 м. Площадь на одно рабочее место пользователей ПК с монитором на базе плоских дискретных экранов (жидкокристаллические, плазменные) - 4,5 м^2 [22].

7.1 Анализ выявленных вредных факторов проектируемой производственной среды

Для обеспечения производственной безопасности необходимо проанализировать воздействия на человека вредных и опасных производственных факторов, которые могут возникать при разработке или эксплуатации проекта.

В рамках работы были выявлены следующие вредные факторы [21]:

- повышенная температура поверхностей персональных компьютеров (ПК);
- низкая или высокая температура воздуха на рабочем месте;
- пониженная или повышенная влажность воздуха;
- критичные уровни отрицательных и положительных аэроионов;
- действие статического электричества;
- электромагнитные излучения;
- нехватка естественного света;
- нехватка искусственного освещения рабочей зоны;
- зрительное напряжение.

Каждый из перечисленных факторов негативно влияет на организм человека и его здоровье. Рассмотрим действия перечисленных выше факторов подробней.

7.1.1 Микроклимат

Микроклимат производственных (рабочих) помещений – климат внутренней среды этих помещений, который определяется действующими на организм человека сочетаниями:

- температуры воздуха;

- температуры поверхностей;
- влажности;
- скорости движения воздуха;
- интенсивности теплового излучения от нагретых поверхностей.

Влажность воздуха – содержание в воздухе водяного пара.

Абсолютная влажность W – масса водяного пара в 1 $м^3$ воздуха.

Максимальная влажность F – масса водяного пара, который может насытить 1 $м^3$ воздуха при данной температуре.

Относительная влажность R – это отношение абсолютной влажности к максимальной.

Указанные параметры – каждый в отдельности и в совокупности – оказывают значительное влияние на работоспособность человека, его самочувствие и здоровье. При определенных их значениях человек испытывает состояние теплового комфорта, что способствует повышению производительности труда, предупреждению простудных заболеваний. И, наоборот, неблагоприятные значения микроклиматических показателей могут стать причиной снижения производственных показателей в работе, привести к таким заболеваниям работающих как различные формы простуды, радикулит, хронический бронхит, тонзиллит и др. Мероприятия по доведению микроклиматических показателей до нормативных значений включаются в комплексные планы предприятий по охране труда. Для создания благоприятных условий работы, соответствующих физиологическим потребностям человеческого организма, санитарные нормы устанавливают оптимальные и допустимые метеорологические условия в рабочей зоне помещения (таблица 7.1 – 7.2) [22]. Для программиста или оператора ПЭВМ категория работ является лёгкой (1а), т.к. работа проводится сидя, без систематических физических нагрузок. Оптимальные параметры микроклимата в офисных помещениях приведены в таблице 6.1.

Таблица 7.1. Оптимальные величины показателей микроклимата на рабочих местах производственных помещений (СанПиН 2.2.4.548-96)

Период года	Температура воздуха, C^0	Температура поверхностей, C^0	Относительная влажность воздуха, %	Скорость движения воздуха, м/с
Холодный	21 - 23	20 - 24	60-40	0,1
Теплый	23-25	22-26	60-40	0,1

Таблица 7.2. Допустимые величины показателей микроклимата на рабочих местах производственных помещений (СанПиН 2.2.4.548-96)

Период года	Температура воздуха, °С		Температура поверхностей, °С	Относительная влажность воздуха, %	Скорость движения воздуха, м/с	
	диапазон ниже оптимальных величин	диапазон выше оптимальных величин			для диапазона температур воздуха ниже оптимальных величин, не более	для диапазона температур воздуха выше оптимальных величин, не более**
Холодный	19,0 - 20,9	23,1 - 24,0	18,0 - 25,0	15 - 75	0,1	0,2
Теплый	20,0 - 21,9	24,1 - 28,0	19,0 - 29,0	15 - 75	0,1	0,3

К мероприятиям по оздоровлению воздушной среды в производственном помещении относятся правильная организация вентиляции и кондиционирования воздуха, отопление помещений. Вентиляция может осуществляться естественным и механическим путём. В зимнее время в помещении необходимо предусмотреть систему отопления.

7.1.2 Повышенный уровень электромагнитных излучений

Уровень электромагнитных излучений на рабочем месте оператора ПЭВМ является вредным фактором производственной среды, величины параметров которого определяются СанПиН 2.2.2/2.4.1340-03. Основными источниками электромагнитных излучений в помещениях для работы операторов ПЭВМ являются дисплеи компьютеров и мобильных устройств, сеть электропроводки, системный блок, устройства бесперебойного питания, блоки питания.

Излучения, применительно к дисплеям современных ПЭВМ, можно разделить на следующие классы:

- Переменные электрические поля (5 Гц – 400 кГц);
- Переменные магнитные поля (5 Гц – 400 кГц).

Воздействие данных излучений на организм человека носит необратимый характер и зависит от напряженности полей, потока энергии, частоты колебаний, размера облучаемого тела. При воздействии полей, имеющих напряженность выше предельно допустимого уровня, развиваются нарушения нервной системы, кровеносной сердечно-сосудистой системы, органов пищеварения и половой системы [22].

В таблице 7.3 приведены допустимые уровни параметров электромагнитных полей

Таблица 7.3 – Временные допустимые уровни электромагнитных полей, создаваемых ПЭВМ на рабочих местах [22]

Наименование параметров		Допустимые значения
Напряженность электрического поля	в диапазоне частот 5 Гц - 2 кГц	25 В/м
	в диапазоне частот 2 кГц - 400 кГц	2,5 В/м
Плотность магнитного потока	в диапазоне частот 5 Гц - 2 кГц	250 нТл
	в диапазоне частот 2 кГц - 400 кГц	25 нТл
Напряженность электростатического поля		15 кВ/м

7.1.3 Недостаточная освещенность рабочей зоны

Недостаточная освещенность рабочей зоны является вредным производственным фактором, возникающим при работе с ПЭВМ, уровни которого регламентируются СП 52.13330.2011. Освещение влияет на настроение и самочувствие, определяет эффективность труда. Рациональное освещение помещений и рабочих мест – одно из важнейших условий создания благоприятных и безопасных условий труда. Около 80 % из общего объема информации человек получает через зрительный аппарат. Качество получаемой информации во многом зависит от освещения: неудовлетворительное в количественном или качественном отношении освещение не только утомляет зрение, но и вызывает утомление организма в целом. Нерационально организованное освещение может, кроме того явиться причиной травматизма: плохо освещенные опасные зоны, слепящие источники света и блики от них, резкие тени и пульсации освещенности ухудшают видимость и могут вызвать неадекватное восприятие наблюдаемого объекта. Поэтому рациональное освещение помещений и рабочих мест – одно из важнейших условий для создания благоприятных и безопасных условий труда.

Искусственное освещение предусматривается в помещениях, в которых испытывается недостаток естественного света, а также для освещения помещения в те часы суток, когда естественная освещенность отсутствует. По принципу организации искусственное освещение можно разделить на два вида: общее и комбинированное.

Общее освещение предназначено для освещения всего помещения, оно может быть равномерным или локализованным. Общее равномерное освещение создает условия для выполнения работ в любом месте освещаемого пространства. При общем локализованном освещении светильники размещают в соответствии с расположением оборудования, что позволяет создавать повышенную освещенность на рабочих местах [22].

Комбинированное освещение состоит из общего и местного. Его целесообразно устраивать при работах высокой точности, а также при

необходимости создания в процессе работы определенной направленности светового потока: Местное освещение предназначено для освещения только рабочих поверхностей и не создает необходимой освещенности даже на прилегающих к ним участкам. Оно может быть стационарным и переносным [26].

Правильно спроектированное освещение производственных помещений оказывает положительное воздействие на сотрудников, способствует повышению эффективности и безопасности труда, снижает утомление и травматизм, сохраняет высокую работоспособность.

Основной задачей светотехнических расчётов для искусственного освещения является определение требуемой мощности электрической осветительной установки для создания заданной освещённости [22].

1) Выбор системы освещения и источников света.

Для обеспечения требуемого уровня освещения в офисном помещении используется лампы дневного освещения, равномерно распределенные по всему потолку офиса. Для освещения помещения выбраны наиболее широко применяемые лампы типа ЛБ.

2) Выбор светильников и их размещение.

Для освещения офисного помещения используются встраиваемые светильники размером 600 х 600 мм. (с 4мя лампами).

Для создания благоприятных зрительных условий на рабочем месте, для борьбы со слепящим действием источников света введены требования ограничения наименьшей высоты светильников над полом. Размещение светильников в помещении определяется следующими размерами:

- H – высота помещения = 3,54 м;
- h_c – расстояние светильников от перекрытия (свес) = 0,5 м;
- $h_n = H – h_c$ – высота светильника над полом, высота подвеса = 3 м;
- h_p – высота рабочей поверхности над полом = 0,8 м;
- $h = h_n – h_p$ – расчётная высота, высота светильника над рабочей поверхностью = 2,2 м.

Расстояние между светильниками L определяется как

$$L = \lfloor \oplus h = 1{,}1 \oplus 2{,}2 = 2{,}42 \text{ м}. \quad (7.1)$$

Расстояние от крайних светильников или рядов до стены:

$$I = L/3 = 2{,}42/3 = 0{,}81 \text{ м}. \quad (7.2)$$

Используя один из наилучших способов размещения светильников по сторонам квадрата и оптимальное расстояние светильников от стены, разместим светильники в три ряда, по три в каждом ряду. При этом расстояние между светильниками, учитывая размеры помещения, возьмем равным 0,92 м. Примерный план расположения светильников в помещении изображен на рисунке 6.1. Учитывая, что в каждом светильнике установлено четыре лампы, общее число ламп в помещении n = 36.

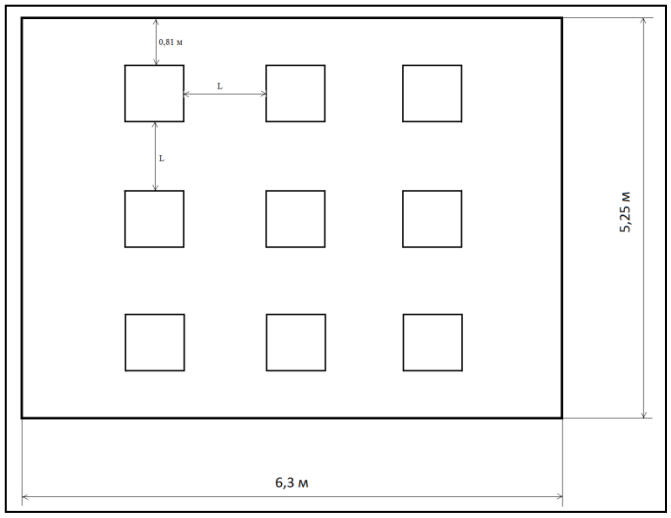

Рисунок 7.1 – План расположения светильников в помещении

3) Выбор нормируемой освещенности.

Согласно [6] нормируемая минимальная освещённость в соответствии с IV разрядом зрительной работы, большой контрастностью и светлым фоном равна 400 лк, в том числе от общего освещения 200 лк.

4) Расчет общего равномерного освещения.

Расчёт общего равномерного искусственного освещения горизонтальной рабочей поверхности выполняется методом коэффициента светового потока, учитывающим световой поток, отражённый от потолка и стен.

Световой поток лампы накаливания или группы люминесцентных ламп

светильника определяется по формуле:

$$\Phi = \frac{E \cdot S \cdot k_з \cdot z}{n \cdot \eta} \qquad (7.3)$$

где E – нормируемая минимальная освещённость по, лк;

S – площадь освещаемого помещения, м^2;

$K_з$ – коэффициент запаса, учитывающий загрязнение светильника (источника света, светотехнической арматуры, стен и пр., т.е. отражающих поверхностей), (наличие в атмосфере цеха дыма), пыли;

Z – коэффициент неравномерности освещения, для люминесцентных ламп при расчётах берётся равным 1,1;

n – число светильников;

| - коэффициент использования светового потока.

Индекс помещения определяется по формуле:

$$i = \frac{S}{h \cdot (a + b)} \qquad (7.4)$$

Индекс помещения равен 1,3.

Коэффициенты отражения имеют следующие значения:

$\rangle_c$ = 10% (стены оклеены светлыми обоями);

$\rangle_n$ = 50% (потолок побеленный);

Значения коэффициента использования светового потока | для данного примера |=0,41. Для помещения с малым выделением пыли $K_з$ = 1,5. Таким образом, Ф = 2951 Лм.

Вычисленному световому потоку соответствует лампа типа ЛБ мощностью 40 Вт с потоком Ф = 3200 Лм. Делаем проверку выполнения условия:

$$-10\% \le \frac{\Phi_{л.станд} - \Phi_{л.расч}}{\Phi_{л.станд}} \cdot 100\% \le +20\% \qquad (7.5)$$

Получаем −10% ≤ 7,8% ≤ +20%.

Необходимый поток света не выходит за пределы диапазона, поэтому

корректировать число светильников, либо высоту подвеса светильников нет необходимости.

7.1.4 Монотонный режим работы

При работе с ПЭВМ основным фактором, влияющим на нервную систему программиста или пользователя, является огромное количество информации, которое он должен воспринимать. Это является сложной задачей, которая очень сильно влияет на сознание и психофизическое состояние из-за монотонности работы. Поэтому меры, позволяющие снизить воздействие этого вредного производственного фактора, которые регулируются СанПиН 2.2.2/2.4.1340-03, являются важными в работе оператора ПЭВМ. Они позволяют увеличить производительность труда и предотвратить появление профессиональных болезней.

Организация работы с ПЭВМ осуществляется в зависимости от вида и категории трудовой деятельности. Виды трудовой деятельности разделяются на 3 группы: группа А – работа по считыванию информации с экрана с предварительным запросом; группа Б – работа по вводу информации; группа В – творческая работа в режиме диалога с ПЭВМ. Работа программиста-разработчика рассматриваемой в данной работе системы относится к группам А и Б, в то время, как деятельность врача-специалиста, который будет использовать систему в профессиональной деятельности, относится к группе В. Категории туровой деятельности различаются по степени тяжести выполняемых работ. Для снижения воздействия рассматриваемого вредного фактора предусмотрены регламентированные перерывы для каждой группы работ – таблица 7.4.

Таблица 7.4 – Суммарное время регламентированных перерывов в зависимости от продолжительности работы, вида категории трудовой деятельности с ПЭВМ [22]

Категория работы с ПЭВМ	Уровень нагрузки за рабочую смену при видах работ с ПЭВМ	Суммарное время регламентированных перерывов, мин.

	группа А, количество знаков	группа Б, количество знаков	группа В, ч	при 8-часовой смене	при 12-часовой смене
I	до 20 000	до 15 000	до 2	50	80
II	до 40 000	до 30 000	до 4	70	110
III	до 60 000	до 40 000	до 6	90	140

7.2 Анализ выявленных опасных факторов проектируемой производственной среды

7.2.1 Электробезопасность

Электробезопасность является опасным фактором и обычно она связана со следующими источниками:

- поражение электрическим током;
- статическое электричество;
- молниезащита.

Мероприятия защиты при электробезопасности следующие:

- отключать электрооборудование при его ремонте;
- периодически снимать электростатическое напряжение, касаясь пальцами рук, заземленных поверхностей;
- для безопасности во время гроз необходимо удостовериться о наличие молниеотвода, и того факта что все розетки в кабинетном помещении заземлены.

Для оператора ПЭВМ при работе с электрическим оборудованием обязательны следующие меры предосторожности:

- Перед началом работы нужно убедиться, что выключатели и розетка закреплены и не имеют оголённых токоведущих частей;
- При обнаружении неисправности оборудования и приборов необходимо, не делая никаких самостоятельных исправлений, сообщить человеку, ответственному за оборудование [22,23].

7.2.2 Пожаробезопасность

Пожаро-взрывобезопасность характеризуется следующими причинами:

- возгорание на рабочем месте в связи с коротким замыканием;
- возгорание на рабочем месте в связи с неправильным обращением с огнем.

Помещение оснащено средствами пожаротушения в соответствии с нормами. На 100 м^2 пола имеется:

- пенный огнетушитель ОП-10 – 1 шт.;
- углекислотный огнетушитель ОУ-5 – 1 шт.;
- ящик с песком на 0,5 м^3 – 1 шт.;
- железные лопаты – 2 шт.

При невозможности самостоятельно потушить пожар необходимо вызвать пожарную команду, после чего поставить в известность о случившемся инженера по техники безопасности [22,24].

Кабинет постоянно содержится в чистоте, каждый будний день моется пол, выбрасывается мусор и протирается пыль. Кабинет обеспечен средствами пожаротушения и сигнализацией о наличие продуктов горения в кабинетном помещении. Компьютерное оборудование для работы в кабинете исправно. Пожарные гидранты, пожарный водопровод и средства пожаротушения исправны и находятся на своих штатных местах в состоянии готовности к работе [24,25].

Кабинетное помещение относится к категории В, согласно СП 12.13130.2009 [26].

В зимнее время гидранты утеплены, пожарный водопровод заизолирован и утеплен, и не разморожен.

В кабинете приказом назначается лицо, отвечающее за соблюдение правил пожарной безопасности, за исправное состояние пожарного инвентаря и за применение первичных способов пожаротушения.

Краны противопожарного водопровода оборудованы брезентовыми шлангами с брандспойтами. Соединительные головки кранов и шлангов

должны иметь резиновые прокладки. Скрученные прорезиненные шланги и брандспойты хранятся в опломбированных шкафчиках, размещенных вблизи кранов.

Ящики и щиты, где хранится противопожарный инвентарь, ручки лопат и пожарных топоров, окрашены в красный цвет, а металлические части периодически смазываются и очищаются для предотвращения коррозии.

7.3 Охрана окружающей среды

В данном разделе рассматривается воздействие на окружающую среду деятельности по разработке проекта, а также самого продукта в результате его реализации на производстве.

Разработка программного обеспечения и работа за ПЭВМ не являются экологически опасными работами, потому объект, на котором производилась разработка продукта, а также объекты, на которых будет производиться его использование операторами ПЭВМ относятся к предприятиям пятого класса, размер селитебной зоны для которых равен 50 м [27].

Непосредственно программный продукт, разработанный в ходе выполнения магистерской диссертации, не наносит вреда окружающей среде ни на стадиях его разработки, ни на стадиях эксплуатации. Однако, средства, необходимые для его разработки и эксплуатации могут наносить вред окружающей среде.

Современные ПЭВМ производят практически без использования вредных веществ, опасных для человека и окружающей среды. Исключением являются аккумуляторные батареи компьютеров и мобильных устройств. В аккумуляторах содержатся тяжелые металлы, кислоты и щелочи, которые могут наносить ущерб окружающей среде, попадая в гидросферу и литосферу, если они были неправильно утилизированы. Для утилизации аккумуляторов необходимо обращаться в специальные организации, специализировано занимающиеся приемом, утилизацией и переработкой аккумуляторных батарей [26].

Люминесцентные лампы, применяющиеся для искусственного освещения рабочих мест, также требуют особой утилизации, т.к. в них присутствует от 10 до 70 мг ртути, которая относится к чрезвычайно-опасным химическим веществам и может стать причиной отравления живых существ, а также загрязнения атмосферы, гидросферы и литосферы. Сроки службы таких ламп составляют около 5-ти лет, после чего их необходимо сдавать на переработку в специальных пунктах приема. Юридические лица обязаны сдавать лампы на переработку и вести паспорт для данного вида отходов [27,29].

7.4 Защита в чрезвычайных ситуациях

На таком объекте как кабинетное помещение могут возникнуть такие чрезвычайные ситуации (ЧС) как:

- техногенные;
- экологические;
- природные.

Рассмотрим наиболее типичную ЧС, такую как пожар в кабинетном помещении. Эта ЧС может произойти в случае замыкания электропроводки оборудования, обрыву проводов, не соблюдению мер пожаробезопасности в кабинете и т.д.

Для того что бы избежать возникновения пожара необходимо проводить следующие профилактические работы, направленные на устранение возможных источников возникновения пожара:

- периодическая проверка проводки;
- проведение инструктажа кабинетных работников о пожаробезопасности.

Для того что бы увеличить устойчивость кабинетного помещения к ЧС необходимо устанавливать системы противопожарной сигнализации, реагирующие на дым и другие продукты горения, установка огнетушителей, обеспечить кабинет и проинструктировать рабочих о плане эвакуации из

кабинета, а также назначить ответственных за эти мероприятия. Периодически проводить ложные тревоги, для проверки готовности кабинета к ЧС. В ходе осмотра кабинетного помещения были выявлены системы, сигнализирующие о наличие пожара или задымленности помещения, наличие огнетушителей и средств тушения пожара (ведра, лопаты и песок, находящиеся в специально оборудованном шкафу, окрашенному в красный цвет). Также, ответственные за пожарную безопасность и охрану труда, периодически проводятся инструктажи и учебные тревоги.

В случае возникновения ЧС как пожар, необходимо предпринять меры по эвакуации персонала из кабинетного помещения в соответствии с планом эвакуации (рисунок 6.2). При отсутствии прямых угроз здоровью и жизни произвести попытку тушения возникшего возгорания огнетушителем. В случае потери контроля над пожаром, необходимо эвакуироваться вслед за сотрудниками по плану эвакуации и ждать приезда специалистов, пожарников. При возникновении пожара должна сработать система пожаротушения, издав предупредительные сигналы, и передав на пункт пожарной станции сигнал о ЧС, в случае если система не сработала, по каким-либо причинам, необходимо самостоятельно произвести вызов пожарной службы по телефону 101, сообщить место возникновения ЧС и ожидать приезда специалистов.

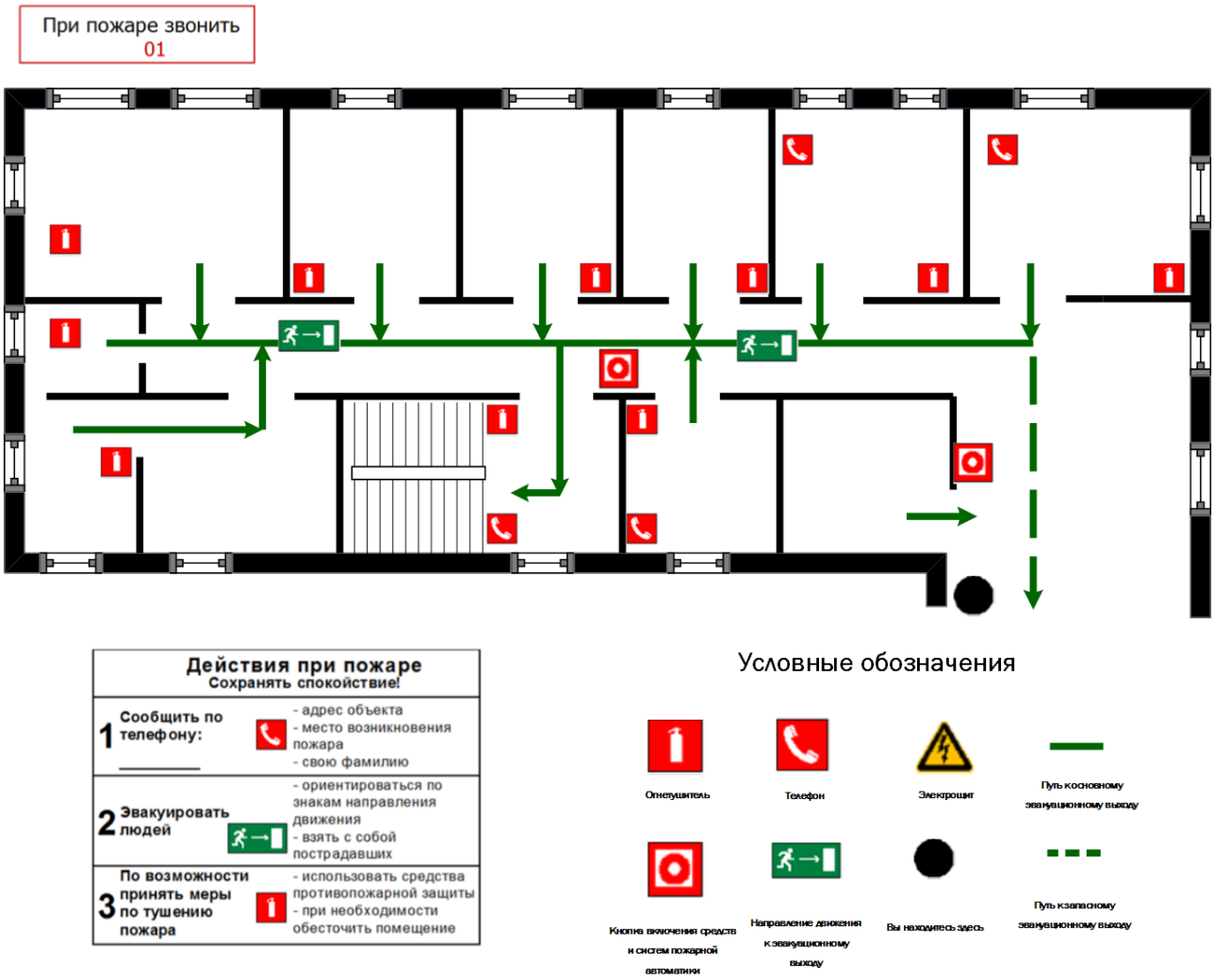

Рисунок 7.2 – План эвакуации

7.5 Правовые и организационные вопросы обеспечения безопасности

7.5.1 Правовые нормы трудового законодательства для рабочей зоны оператора ПЭВМ

Регулирование отношений между работником и работодателем, касающихся оплаты труда, трудового распорядка, особенности регулирования труда женщин, детей, людей с ограниченными способностями и проч., осуществляется законодательством РФ, а именно трудовым кодексом РФ.

Продолжительность рабочего дня не должна быть меньше указанного времени в договоре, но не больше 40 часов в неделю. Для работников до 16 лет – не более 24 часов в неделю, от 16 до 18 лет и инвалидов I и II группы – не более 35 часов.

Возможно установление неполного рабочего дня для беременной женщины; одного из родителей (опекуна, попечителя), имеющего ребенка в возрасте до четырнадцати лет (ребенка-инвалида в возрасте до восемнадцати лет). Оплата труда при этом производится пропорционально отработанному времени, без ограничений оплачиваемого отпуска, исчисления трудового стажа и других прав.

При работе в ночное время продолжительность рабочей смены сокращается на один час. К работе в ночную смену не допускаются беременные женщины; работники, не достигшие возраста 18 лет; женщины, имеющие детей в возрасте до трех лет, инвалиды, работники, имеющие детей-инвалидов, а также работники, осуществляющие уход за больными членами их семей в соответствии с медицинским заключением, матери и отцы-одиночки детей до пяти лет.

Организация обязана предоставлять ежегодный отпуск продолжительностью 28 календарных дней. Дополнительные отпуска предоставляются работникам, занятым на работах с вредными или опасными условиями труда, работникам имеющими особый характер работы, работникам с ненормированным рабочим днем и работающим в условиях Крайнего Севера и приравненных к нему местностях.

В течение рабочего дня работнику должен быть предоставлен перерыв для отдыха и питания продолжительностью не более двух часов и не менее 30 минут, который в рабочее время не включается. Всем работникам предоставляются выходные дни, работа в выходные дни осуществляется только с письменного согласия работника.

Организация-работодатель выплачивает заработную плату работникам. Возможно удержание заработной платы только в случаях, установленных ТК РФ ст. 137. В случае задержки заработной платы более чем на 15 дней, работник имеет право приостановить работу, письменно уведомив работодателя.

Законодательством РФ запрещена дискриминация по любым признакам и принудительный труд [32].

7.5.2 Организационные мероприятия при компоновке рабочей зоны

К мероприятиям, относящимся к компоновке рабочей зоны относятся работы по организации рабочего места пользователя, позволяющие наилучшим образом организовать деятельность работника, делая его работу максимально удобной и безопасной.

Основным направлением использования разработанной программной системы является применение параллельного программирования технологии CUDA в методах случайной оптимизации.

Требования к помещениям для работы с ПЭВМ регламентируются в соответствии с СанПиН 2.2.2/2.4.1340-03. В документе указаны нормы помещениям для работы с ПЭВМ, норма площади рабочего места с персональным компьютером составляет 4,5 м2.

Разработанный программный продукт не влияет на организацию рабочей зоны, однако работа с ним позволит реорганизовать работу специалистов, что в свою очередь повлияет на организацию рабочей зоны. Это может быть охарактеризовано с помощью следующих факторов:

- Сокращение времени обработки данных внушительных размерностей;
- Удобный интерфейс для работы с данными;
- Максимальнаю эффективность в задачах, не требующих интенсивного обращения к памяти.

Все перечисленные факторы повышают, облегчают работу и положительно сказываются на производительности труда.

ЗАКЛЮЧЕНИЕ

Целью выпускной квалификационной работы является: с помощью параллельных вычислений максимально ускорить процесс получения экстремума в задачах многомерной оптимизации и тем самым показать преимущества использования вышеуказанной технологии по сравнению с технологией последовательных вычислений.

Для достижения цели были выполнены следующие этапы:

1. Разработано приложение для нахождения минимума функций Розенброка и Растригина по алгоритму простого слчайного поиска и алгоритму наилучшей пробы с направляющим гиперквадратом.
2. Произведен сравнительный анализ технологий последовательных и параллельных вычислений.

По материалу исследования опубликована статья в научный журнал «Известия ТПУ»

При тестировании программного программных средств не было выявлено проблем с внешним видом, отображением его элементов, а также работы в целом.

Список публикаций

1. **Reizlin (Reyzlin), Valery Izrailevich.** Effective Method for Constructing Pseudo-Random Vectors Uniformly Distributed in Cone [Electronic resource] / V. I. Reizlin (Reyzlin), A. A. Nefedova // Key Engineering Materials : Scientific Journal. — 2016. — Vol. 685 : High Technology: Research and Applications 2015 (HTRA 2015). — [P. 852-856]. — Title screen. — Доступ по договору с организацией-держателем ресурса.
   Режим доступа: http://dx.doi.org/10.4028/www.scientific.net/KEM.685.852

СПИСОК ЛИТЕРАТУРЫ

1. Рейтинг суперкомпьютеров TOP500. – URL: http://www.top500.org/ (дата обращения: 17.06.2017)
2. Архитектура Nvidia Kepler. – URL: http://www.nvidia.com/content/PDF/kepler/NVIDIAKepler-GK110-Architecture-Whitepaper.pdf/ (дата обращения: 17.06.2017)
3. Что такое CUDA? – URL: www.nvidia.ru/object/what_is_cuda_new_ru.html/(дата обращения: 17.06.2017)
4. Знакомство с NVIDIA CUDA. – URL: www.render.ru/books/show_book.php?book_id=840/(дата обращения: 17.06.2017)
5. Информация о CUDA. – URL: ru.wikipedia.org/wiki/CUDA/(дата обращения: 17.06.2017)
6. Боресков А.В. , Харламов А.А. Основы работы с технологией CUDA, Изд. ДМК Пресс, Москва, 2010г, 234стр.
7. Стандарт OpenCL. – URL: http://www.khronos.org/opencl/
8. Параллельные вычисления с CUDA. – URL: http://www.nvidia.ru/object/cuda-parallel-computing-ru.html/ (дата обращения: 17.06.2017)
9. Технология AMD STREAM. – URL: http://www.amd.com/ru-ru/innovations/software-technologies/firepro-graphics/stream (дата обращения: 17.06.2017)
10. Растригин Л.А. Стохастические методы поиска. – М.: Наука, 1968. – 376 с.
11. Растригин Л.А. Адаптация сложных систем. Рига, Зинатне, 1981, 376 с.
12. Давыдов И.Е. Применение метода случайного поиска в задаче модального формирования динамических свойств системы «ракета-носитель – автомат стабилизации» // Вестник самарского государственного аэрокосмического университета. – 2007. – № 4. – С. 31–35.

13. Теория и применение случайного поиска / под ред. Л.А. Растригина. – Рига: Зинатне, 1969. – 305 с.
14. Точки, равномерно распределенные в гиперконусе / В.И. Рейзлин, В.А. Молодых, В.А. Орлов // Математическое и программное обеспечение САПР: Сб. научно-технических работ. – Томск: изд. Томского политехнического университета, 1997. – С. 126–128.
15. Ермаков С.М., Михайлов Г.А. Курс статистического моделирования. – М.: Наука, 1976. – 320 с.
16. Численные методы оптимизации: учебное пособие / В.И. Рейзлин; Национальный исследовательский Томский политехнический университет. – Томск: Изд-во Национального исследовательского Томского политехнического университета, 2013 – 105 с.
17. Установка на Windows - URL: http://edu.chpc.ru/cuda/mainse5.html/ (дата обращения: 14.06.2017)
18. СанПиН 2.2.2/2.4.1340–03. Гигиенические требования к персональным электронно-вычислительным машинам и организации работы.
19. ГОСТ 12.0.003-74. ССБТ. Опасные и вредные производственные факторы. Классификация.
20. ГОСТ 12.0.003-74 ССБТ. Опасные и вредные производственные факторы. Классификация. – М.: Информационно-издательский центр Минздрава России, 1974.
21. СанПиН 2.2.2/2.4.1340-03 Гигиенические требования к персональным электронно-вычислительным машинам и организации работы: с изменениями от 25 апреля 2007 г. – М.: Информационно-издательский центр Минздрава России, 2003.
22. Правила пожарной безопасности в Российской Федерации. – М.: Министерство Российской Федерации по делам гражданской обороны, чрезвычайным ситуациям и ликвидации последствий стихийных бедствий, 2003.
23. Технический регламент о требованиях пожарной безопасности: Федеральный закон от 22 июля 2008 года N 123-ФЗ.
24. СНиП 21-01-97. Пожарная безопасность зданий и сооружений.

25. СП 12.13130.2009. Определение категорий помещений, зданий и наружных установок по взрывопожарной и пожарной опасности.
26. СанПиН 2.2.1/2.1.1.1200-03. Санитарно-эпидемиологические правила и нормативы. Санитарно-защитные зоны и санитарная классификация предприятий, сооружений и других объектов // Библиотека гостов и нормативов. 2017. URL: http://ohranatruda.ru/ot_biblio/normativ/data_normativ/11/11774/ (дата обращения: 24.04.2017).
27. СанПиН 2.1.7.1322-03. Санитарно-эпидемиологические правила и нормативы. Гигиенические требования к размещению и обезвреживанию отходов производства и потребления. 2.1.7. Почва, очистка населённых мест, бытовые и промышленные отходы, санитарная охрана почвы // Библиотека гостов и нормативов. 2016. URL: http://ohranatruda.ru/ot_biblio/normativ/data_normativ/11/11774/ (дата обращения: 02.05.2017).
28. Постановление Правительства РФ от 03.09.2010 N 681 (ред. от 01.10.2013) "Об утверждении Правил обращения с отходами производства и потребления в части осветительных устройств, электрических ламп, ненадлежащие сбор, накопление, использование, обезвреживание, транспортирование и размещение которых может повлечь причинение вреда жизни, здоровью граждан, вреда животным, растениям и окружающей среде // Консультант Плюс. 2015. URL: http://www.consultant.ru/document/cons_doc_LAW_104420/e1b31c36ed1083efeb6cd9c63ed12f99e2ca77ed/#dst100007 (дата обращения: 02.05.2017).
29. Энергосбережение в компьютерном мире // HWP. 2008. URL: http://www.hwp.ru/articles/Energosberezhenie_v_kompyuternom_mire_CHast_1___osnovnie_tendentsii/?SHOWALL_1=1 (дата обращения: 24.04.2017).
30. НПБ 105-03 Определение категорий помещений, зданий и наружных установок по взрывопожарной и пожарной опасности // Электронный

фонд правовой и нормативно-технической документации. 2016. URL: http://docs.cntd.ru/document/1200032102 (дата обращения: 24.04.2017).

31. ППБ 01–03. Правила пожарной безопасности в Российской Федерации. – М.: Министерство Российской Федерации по делам гражданской обороны, чрезвычайным ситуациям и ликвидации последствий стихийных бедствий, 2003.
32. Трудовой кодекс Российской Федерации" от 30.12.2001 N 197-ФЗ (ред. от 30.12.2015) // Консультант Плюс. 2015. URL: http://www.consultant.ru/document/cons_doc_law_34683/?utm_campaign=law_doc&utm_source=google.adwords&utm_medium=cpc&=utm_content=Labor%20Code&gclid=CjwKEAjwgPe4BRCB66GG8PO69QkSJAC4EhHhU-5yAFZCJfmzkTLNGnrpgHHAYFPhhPzRo-sZGWmqnBoCPynw_wcB (дата обращения: 25.04.2017).

ARCHITECTURE NVIDIA CUDA

Hardware and software platform CUDA

The CUDA platform is nVidia's hardware and software platform for general-purpose tasks - GPGPU (General-PurposecomputingonGraphicsProcessingUnits). The first release was released on February 15, 2007. The main purpose is to give the programmer the opportunity to use the GPU (hereinafter "device") as a coprocessor for tasks that require parallel computations, while abstracting from terminology and not using libraries specific for processing 3D graphics. However, not everything is so simple, if only because the video cards were initially oriented and for many years developed as graphics processing devices. And it is the peculiarities of GPU architecture that cause the greatest difficulties when first getting to know the platform.

In addition to the video card itself (starting with the GeForce 8000th series), the so-called CUDA driver, CUDA TOLKIT and CUDA SDK are included in the platform. CUDA TOLIKIT includes the libraries needed to work with the platform, and the nvcc compiler, which translates the source code of the programs into intermediate assembler code, as well as additional libraries. The CUDA driver in earlier versions included only the compiler that converts the intermediate assembler generated by the nvcc compiler into a microcode executable on the GPU. Now the driver CUDA and the device driver are combined into one package. CUDA SDK contains a set of source codes for simple programs that illustrate the methods and capabilities of working with the CUDA platform.

CUDA is supported only by GeForce 8G and GeForce 8, GeForce 9, GeForce 200 video accelerators, as well as Quadro and Tesla, this is worthy of special attention. The main characteristics of CUDA are interaction with the OpenGL and DirectX.25 graphics APIs, the ability to develop at a low level, providing access with fast shared memory, support for 32- and 64-bit operating systems, and also CUDA uses the extended version of the C language.

Consider the logical architecture for understanding the work with the platform in Figure 2.1:

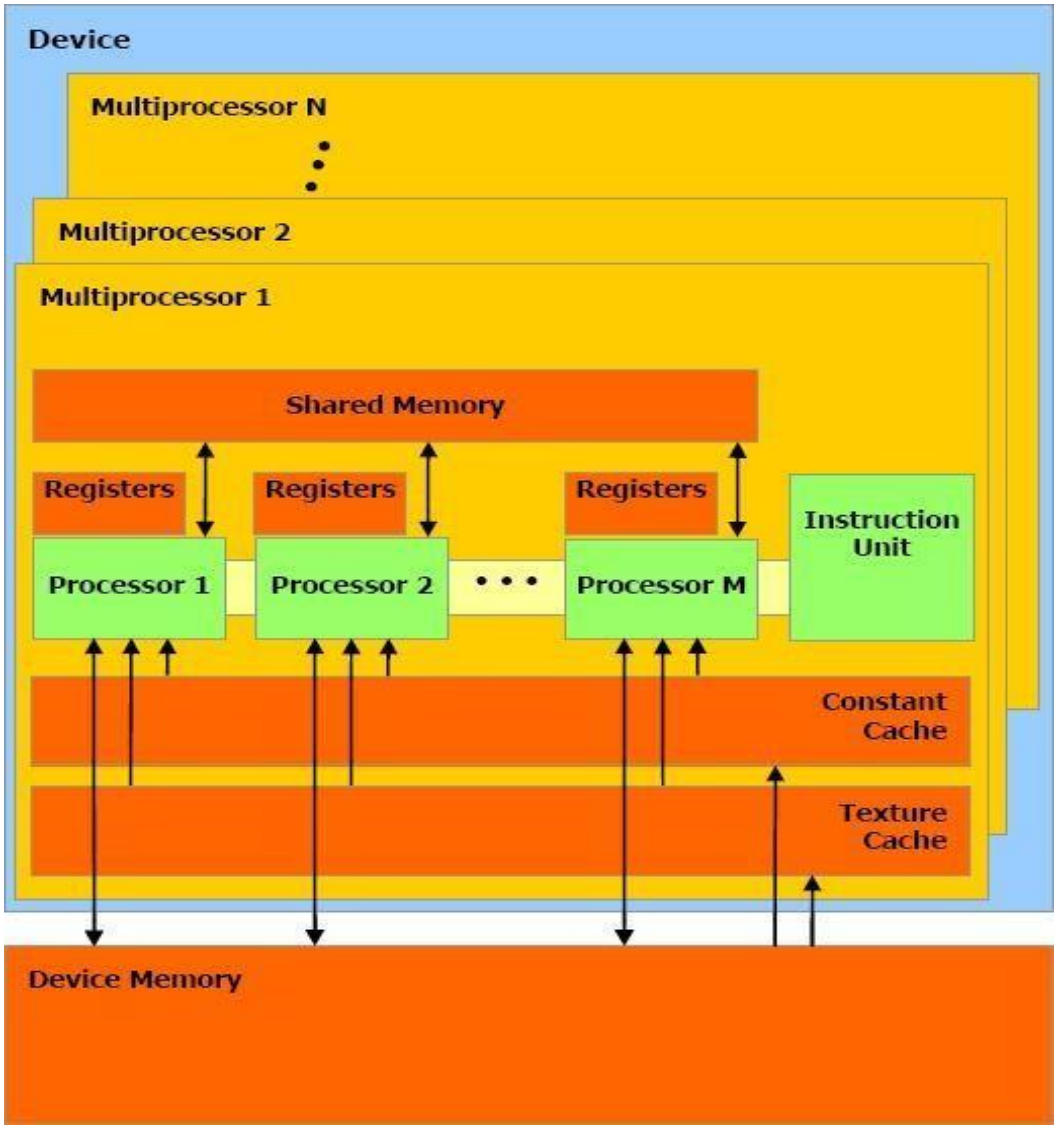

Figure 2.1 - CUDA architecture.

Let's consider this model in more detail.

The memory model of CUDA technology.

Free access to memory with the option of byte addressing is one of the most important features for developers. CUDA flows can access data from several memory spaces during execution, as shown in Figure 2.2. Each thread has a private local memory. Each block of the stream has a common memory, which is visible for all block flows and with the same lifetime as the block. All threads have access to the same global memory.

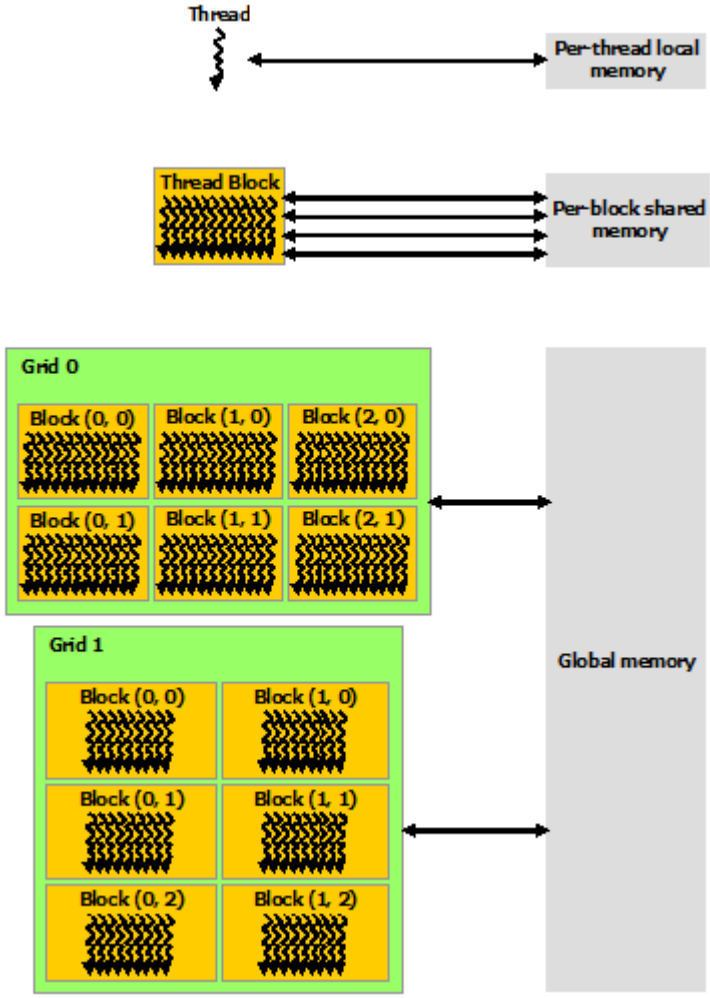

Figure 2.2 - Memory hierarchy.

There are also two additional readable spaces available for all threads: a space of constant and texture memory. Global, persistent and textured memory spaces are optimized for different memory applications. The texture memory also offers different addressing modes, as well as data filtering for some specific data formats. For example, instructions that access addressable memory (i.e., global, local, shared, permanent, or texture memory) may need to be re-issued several times, depending on the allocation of memory addresses to threads at the core. For global memory, as a rule, the more scattered addresses, the more bandwidth is reduced.

Global Memory

The global memory is stored in the device memory, and the device memory is through 32-, 64- or 128-byte memory transactions. These memory transactions must be naturally aligned: only 32-, 64- or 128-byte memory segments of the device that

are aligned in size (that is, whose first address is a multiple of their size) can be read or written to the transaction memory.

When warp executes an instruction that accesses global memory, it combines the treatment of thread memory in warp into one or more memory transactions, depending on the word size available to each thread, and allocating memory addresses through threads. In general, the more transactions necessary, the more unused words are transmitted in addition to words available on streams, thus reducing the bandwidth of teams. For example, if a 32-byte memory transaction is generated for 4-byte access for each stream, the bandwidth is divided by 8.

How many transactions are needed and how much bandwidth ultimately affects depends on the computing power of the device. Compute Capability 2.x, Compute Capability 3.x, Compute Capability 5.x and Compute Capability 6.x provide more detailed information about how global memory accesses are handled for various computational capabilities.

Global memory instructions support reading or writing words that are 1, 2, 4, 8, or 16 bytes in size. Any access (via a variable or pointer) to data in global memory is compiled into one global memory command if and only if the data type is 1, 2, 4, 8, or 16 bytes in size, and the data, of course, Ie its address is a multiple of this size.

If this size and alignment requirement is not met, access is compiled into several instructions with alternating access patterns that do not allow these instructions to fully merge. Therefore, it is recommended that you use the types corresponding to this requirement for data that is in global memory.

The alignment requirement is automatically performed for the built-in types char, short, int, long, longlong, float, double, as float2 or float4.

Local memory.

Access to local memory occurs only for some automatic variables. The automatic variables that the compiler can allocate in local memory are:

• Arrays for which it can not determine that they are indexed with constant values,

• Large structures or arrays that will consume too much space for registration,

• Any variable if the kernel uses more registers than is available (this is also known as register spacing).

Checking the build code of PTX (obtained by compiling with the option -ptx or-keep) will determine whether the variable was placed in local memory during the first phases of compilation, since it will be declared using .local mnemonics and will be accessible using ld .local and St.local mnemonics. Even if this is not the case, subsequent compilation steps can still be handled differently, although if they find they consume too much space to store the target architecture: checking the cube object using cuobjdump will tell whether this is really true. In addition, the compiler reports the common use of local memory for each kernel (lmem) when compiled with the --ptxas-options = -v option. Note that some mathematical functions have implementation paths that can access local memory.

The local memory space is in the device's memory, so accessing the local memory has the same high latency and low bandwidth as for accessing the global memory, and they have the same requirements for memory coalescence, as described in Access Memory Access Access. However, local memory is organized in such a way that successive 32-bit words gain access to the serial stream identifier. Thus, access to them is fully integrated, until all threads in warp get the same relative address (for example, the same index in the variable array, the same term in the structural variable).

On devices of computational ability 2.x and 3.x, local memory accesses are always cached in L1 and L2 as well as access to global memory.

On devices with the computational capabilities of 5.x and 6.x, local memory accesses are always cached in L2 as well as access to global memory.

Common memory

Because the shared memory is built into the chip, it has a much higher throughput and significantly less latency than the local or global memory.

To achieve high throughput, shared memory is divided into memory modules of the same size, called banks, which can be accessed at the same time. Thus, any request to read or write to memory consisting of n addresses that fall into n different

memory banks can be serviced simultaneously, which gives a total throughput that is n times the bandwidth of one module.

However, if two memory request addresses fall into the same memory bank, a conflict arises in the bank, and access must be serialized. The device splits the memory request with bank conflicts into as many separate requests as possible without conflicts, reducing the bandwidth by a factor equal to the number of individual memory requests. If the number of individual memory requests is n, it is assumed that the initial memory query causes n-way bank conflicts.

Therefore, to get maximum performance, it is important to understand how memory addresses are mapped to memory banks to schedule memory requests in order to minimize conflicts in banks. This is described in Compute Capability 2.x, Compute Capability 3.x, Compute Capability 5.x and Compute Capability 6.x for computing power devices 2.x, 3.x, 5.x and 6.x respectively.

Permanent memory

The persistent storage space is in the device's memory and cached in the permanent cache specified in Compute Capability 2.x.

Then the request is divided into as many separate requests, that the original query has different memory addresses, which reduces the bandwidth by a factor equal to the number of individual requests.

Then the received requests are serviced at the constant cache bandwidth in case of cache hit or when the device memory bandwidth is otherwise.

Texture memory

The memory areas of the texture and surfaces are stored in the device's memory and cached in the texture cache, so when reading the texture or reading the surface there is one memory read from the device's memory only when the cache misses, otherwise it just costs one reading from the texture cache. The texture cache is optimized for 2D spatial locality, so threads of the same detector that read texture or surface addresses that are close to each other in 2D will achieve the best performance. In addition, it is designed for streaming with a constant delay; The cache cache reduces the need for DRAM bandwidth, but does not provide a delay.

Reading a device's memory through a texture or extracting a surface provides some advantages that can make it an advantageous alternative to reading device memory from global or permanent memory:

• If the read memory data does not match the access patterns that should read the global or permanent reads in memory in order to obtain good performance, a higher throughput can be achieved, provided that there is terrain in the texture samples or surface readings;

• Addressing computations are performed outside the kernel in selected units;

• Packed data can be transferred to separate variables in one operation;

• 8-bit and 16-bit integer input data can optionally be converted to 32-bit floating point values in the range [0.0, 1.0] or [-1.0, 1.0].

Multiprocessors.

The emergence of multi-core processors and multiprocessor graphics processors means that the main processors are now parallel systems. Moreover, their parallelism continues to act in accordance with Moore's law. The challenge is to develop application software that transparently scales its parallelism to use an increasing number of processor cores, just as 3D graphics applications transparently scale their parallelism for many pure graphics processors with a wide variety of cores.

The parallel programming model CUDA is designed to overcome this problem, while maintaining a low learning curve for programmers familiar with standard programming languages such as C.

In fact, these are three key abstractions - the hierarchy of thread groups, shared memories and barrier synchronization, which are simply provided to the programmer as a minimal set of language extensions.

These abstractions provide a fine-grained data parallelism and stream parallelism, embedded in the crude parallelism of data and the parallelism of tasks. They direct the programmer to divide the problem into coarse subtasks that can be solved independently in parallel by the flow blocks, and each subtask into smaller parts that can be jointly solved by all the flows within the block.

This decomposition preserves the expressiveness of the language, allowing threads to interact with each subtask and at the same time provides automatic scalability. Indeed, each block of threads can be scheduled on any of the available multiprocessors in the GPU in any order simultaneously or sequentially, so that the compiled CUDA program can run on any number of multiprocessors, as shown in Figure 2.3.

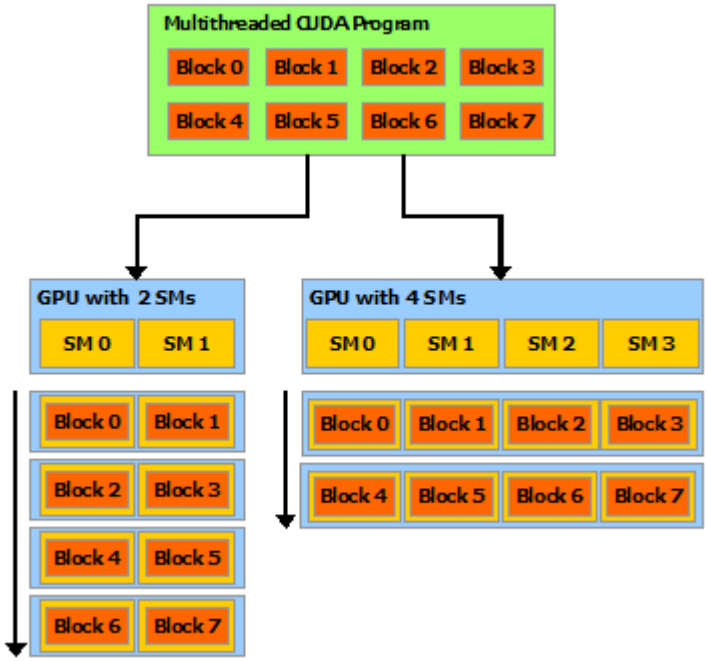

Figure 2.3. - CUDA multiprocessor model

The architecture of NVIDIA GPU is built around a scalable array of multi-threaded multiprocessors (SMs) [4]. When the CUDA program on the main processor calls the kernel grid, the grid blocks are enumerated and distributed across multiprocessors with available bandwidth. Threads of the stream block are executed simultaneously on one multiprocessor, and several stream blocks can be executed simultaneously on one multiprocessor computer. After the threads are terminated, new blocks are started on the released multiprocessors.

The multiprocessor is designed to perform hundreds of threads simultaneously. To manage such a large number of threads, it uses a unique architecture called SIMT (Single-Instruction, Multiple-Thread) - one instruction and many threads [4]. Pipeline instructions to enable parallelism at the instruction level within a single thread, as well as the parallelism of parallel threads based on

simultaneous hardware threading, as described in the section "Multithreading of hardware". Unlike processor cores, they are issued in the order, but there is no branch forecast and there is no speculative execution.

The SIMT architecture and hardware multithreading describe the characteristics of the streaming multiprocessor architecture that are common to all devices. Compute Capability 2.x, Compute Capability 3.x, Compute Capability 5.x and Compute Capability 6.x provide features for 2.x, 3.x, 5.x and 6.x computing devices, respectively.

The multiprocessor creates, manages, schedules and executes flows in groups of 32 parallel threads, called skew. The individual threads that make up the base, start together at the same program address, but they have their own command address counter and the registration state, and therefore they are freely branched and executed independently. The term warp comes from quality, the first technology of parallel flows. Half of the deformation is either the first or second half of the deformation. A square is either the first, second, third, or fourth quarter of the warp.

When the multiprocessor is provided with one or more stream blocks to execute, it splits them into skews, and each warp gets the planned warp scheduler for execution. The way the block is divided into skews is always the same. Each warp contains streams of consecutive threads that increase thread IDs with the first thread containing warp. The thread hierarchy describes how thread identifiers refer to thread indexes in a block.

Warp performs one general instruction at a time, so full efficiency is realized when all 32 warp threads match their execution path. If warp flows diverge through a conditional branch dependent on the data, warp sequentially executes each branch path, disabling threads that are not in that path, and when all paths are completed, threads converge to the same execution path. Divergence divergence occurs only within the framework. Different deformations are performed independently, regardless of whether they are performed using common or non-overlapping codes.

The architecture of SIMT is similar to the organization of SIMD vectors (Single Instruction, Multiple Data), since one command controls several processing

elements. The key difference is that the organization of the SIMD vectors provides SIMD-width to the software, while the SIMT instructions define the behavior of execution and branching of one thread [9]. Unlike SIMD-vector machines, SIMT allows programmers to write parallel thread-level code for independent, scalar streams, as well as parallel parallel code for coordinated threads. For the sake of correctness, a programmer can essentially ignore the behavior of the SIMT, but significant performance improvements can be realized if one considers that the code rarely requires that the flows in the deformation diverge. In practice, this is similar to the role of cache lines in traditional code: the size of the cache line can be safely ignored in development for correctness, but should be taken into account in the code structure when designing for maximum performance. On the other hand, vector architectures require that the software aggregates loads into vectors and manages the discrepancy manually.

The warp threads that are on the current path to this warp are called active threads, while threads that are not related to the current path are inactive (disabled). Themes can be inactive, because they come before other threads of their deformation or because they are on a different branch path than the branching path currently executed by warp, or because they are the last threads of a block whose number of threads is not a multiple of the size Basis.

The execution context (program counters, registers, etc.) for each base processed by the multiprocessor is supported on the chip for the entire.

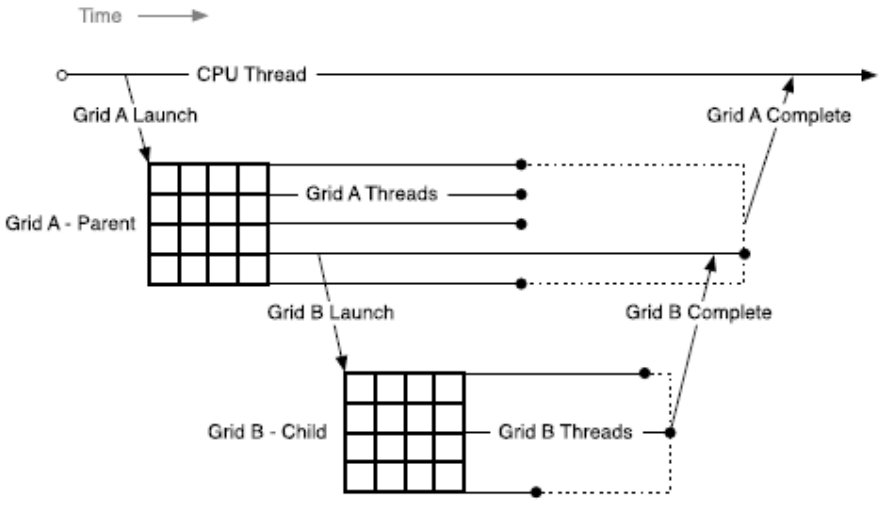

Figure 2.4. - CUDA programming model

The device thread that configures and starts a new grid belongs to the parent grid, and the grid created by the call is a child grid.

The call and completion of the child grids is correctly built, which means that the parent grid is not considered complete until all the child meshes created by its threads are completed. Even if the calling threads are not explicitly synchronized on the running child grids, the runtime guarantees implicit synchronization between the parent and the child.

On the host and device, the CUDA runtime provides an API for launching kernels to wait for shutdown and to monitor the dependencies between startups through threads and events. In the host system, the state of the startup and the CUDA primitives that refer to threads and events are shared by all threads within the process; However, the processes are performed independently and can not share CUDA objects.

The device has a similar hierarchy: running kernels and CUDA objects are visible for all threads in the stream block, but are independent between the thread blocks. This means, for example, that a thread can be created by one thread and used by any other thread in one thread block, but can not be separated by threads in any other flow block.

CUDA execution operations from any thread, including the launch of the kernel, are visible through the thread block. This means that the calling thread in the parent grid can synchronize on the grids launched by this thread, other threads in the flow block, or threads created in one thread block. The execution of the flow block is not considered complete until all the starts of all threads in the block are completed. If all threads in the block completion before the start of all child starts are completed, the synchronization operation will be automatically started.

CUDA streams and events allow you to control dependencies between grid startup: grids running in the same thread are executed in order, and events can be used to create dependencies between threads. Streams and events created on the device serve the same purpose.

Threads and events created in the grid exist in the flow stream area, but have an undefined behavior when used outside the thread block where they were created. As described above, all work started by the flow unit is implicitly synchronized when exiting the block; Work running in threads is included in this, and all dependencies are dealt with appropriately. The behavior of operations on a thread that has been changed outside the flow area is undefined.

Threads and events created on the host have an undefined behavior when used in any kernel, as well as threads and events created by the parent grid, have undefined behavior if they are used in the child grid.

The order of starting the kernel from the execution environment of the device follows the semantics of ordering the CUDA stream. In a thread block, all the kernels are started in a single thread, executed in the order. With multiple threads in the same flow block running in the same thread, the ordering within the flow depends on the scheduling of the flow within the block, which can be managed using synchronization primitives, such as __syncthreads ().

Because threads are shared by all threads in a flow block, an implicit NULL stream is also shared. If several threads in the stream block are launched into an implicit stream, then these starts will be executed in the order. If concurrency is required, you should use explicit named threads.

Dynamic parallelism allows you to simplify the parallelism within the program. However, the execution time of the device does not introduce any new parallelism guarantees into the CUDA execution model. There is no guarantee of simultaneous execution between any number of stream blocks on the device.

The lack of parallelism guarantees extends to blocks of parent threads and their child mesh. When the parent thread block starts the child grid, the child device is not guaranteed to start execution until the parent thread block reaches the explicit synchronization point (for example, cudaDeviceSynchronize ()).

Although concurrency can often be easily achieved, it can vary depending on the deviceconfiguration, application workload, and scheduling runtime. Therefore, it is not safe to depend on any parallelism between different blocks of streams.

The programming model of the CPU on CUDA.

Since the GPU does not have access to RAM, the programmer must take care in advance that all the resources necessary to run the application kernel are in the memory of the video card. For these purposes, three main functions are used from the CUDA SDK: cudaMalloc, cudaMemcpy and cudaFree. These functions have the same purpose as standard malloc, memcpy and for free, but of course, all operations are performed in video memory.

The process of adjusting the grid and blocks is to adjust the size of the grid and blocks. The main task of the programmer at this stage is to find the optimal balance between the size and the number of blocks. By increasing the number of streams in the block, you can reduce the number of calls in the global memory by increasing the intensity of data exchange between the streams via fast shared memory. On the other hand, the number of registers allocated to the block is fixed, and if the number of threads is much larger, the execution time of the kernel will increase, because the GPU will begin to store data in the local memory. When the kernel is called, all the dimensions of the grid and the block defined earlier are transferred.

The kernel is called as a regular function in C. The only significant difference is that when you call the kernel, you must transfer the previously defined grid and block sizes.

After executing the kernel, you must copy the results of the program back into memory using the cudaMemcpy function, specifying the direction of the reverse copy (from the GPU to the CPU).

The programming model of GPU on CUDA.

The function type qualifiers determine whether the function will be executed on the host or on the device and whether it can be called from the host or device.

__device__ declares a function that will be executed on the device and called only from the device.

__global__ declares the function as the core. This function is performed on the device and is called from the host. Called from a device for devices with a

computational capacity of 3.2 or higher. The call to the __global__ function is asynchronous, that is, it is returned before the device completes execution.

__host__ declares a function that is executed on the host and is called only from the host.

__global__ and __host__ can not be used together. However, the __ device__ and __host__ qualifiers can be used together, in which case the function is compiled for both the host and the device.

__noinline__ and __forceinline__

The function specifier __noinline__ can be used as a hint for the compiler, so as not to include the function, if possible. The function body must still be in the same file where it is called.

The __forceinline__ function symbol can be used to force the compiler to embed a function.

Variables of the variable type determine the location of the memory on the variable device.

The automatic variable declared in the device code without any __device__, __shared__ and __constant__ specifiers is usually in the register. However, in some cases, the compiler may want to put it in local memory, which can have adverse effects on performance. Consider a set of specifiers that define the type of memory for placing variables:

__device__ declares a variable that is on the device. No more than one of the qualifiers of the other type can be used together with __device__ to further determine which memory space belongs to the variable. If none of them is present, then the variable is stored in the global memory. Available from all threads in the grid and from the host through the runtime library (cudaGetSymbolAddress () / cudaGetSymbolSize () / cudaMemcpyToSymbol () / cudaMemcpyFromSymbol ()).

__constant__ declares a variable that stays in a constant memory space. It is available from all threads in the grid and from the host through the runtime library (cudaGetSymbolAddress () / cudaGetSymbolSize () / cudaMemcpyToSymbol () / cudaMemcpyFromSymbol ()).

__shared__ declares a variable that remains in the shared memory area of the block. Available for all threads within the block, when declaring a variable in shared memory as an external array, such as Extern __shared__ float shared [];

The size of the array is determined at startup time. All variables declared in this way start with the same address in memory, so that the layout of the variables in the array must be explicitly controlled by the offsets. For example, if an equivalent is required

Short array0 [128];

Float array1 [64];

Int array2 [256];

In dynamically allocated shared memory, you can declare and initialize arrays as follows:

```
Extern __shared__ float array [];
__device__ void func () // Function __device__ or __global__
{
   Short * array0 = (short *) array;
   Float * array1 = (float *) & array0 [128];
   Int * array2 = (int *) & array1 [64];
}
```

Pointers should be tied to the type they point to.

__managed__ declares a variable that can refer to both the device and the host code, for example, its address can be taken or it can be read or written directly from the device or host function.

Nvcc supports limited pointers using the __restrict__ keyword. Limited pointers were introduced in C99 to alleviate the smoothing problem that exists in the C language, and which prohibits all kinds of optimization from reordering the code to a general subexpression exception.

In C, pointers a, b and c can be smoothed, so any record through c can change the elements a or b. This means that the compiler can not load [0] and b [0] into registers, multiply them, and save the result with both [0] and [1] to ensure functional

correctness, since the results will differ from the results. Abstract execution model , If, say, [0] - this is really the same place as c [0]. Therefore, the compiler can not use the general subexpression. Similarly, the compiler can not simply rearrange the computation c [4] in the immediate vicinity of the calculation of c [0] and c [1], since the previous record in c [3] can change the inputs to compute c [4].