МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение
высшего образования
«Тольяттинский государственный университет»

Институт **математики, физики и информационных технологий**
Кафедра **«Прикладная математика и информатика»**

01.03.02 ПРИКЛАДНАЯ МАТЕМАТИКА И ИНФОРМАТИКА

СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ И КОМПЬЮТЕРНЫЕ ТЕХНОЛОГИИ

БАКАЛАВРСКАЯ РАБОТА

на тему «**Исследование технологий реализации алгоритмов линейной алгебры с использованием технологии CUDA**»

Студент А.С.Коновальчук _____

Руководитель Т.Г. Султанов _____

Консультант по аннотации Н.В. Ященко _____

Допустить к защите

Заведующий кафедрой к.т.н., доцент А.В. Очеповский _____

«_____» _____ 2017 г.

Тольятти 2017

АННОТАЦИЯ

Бакалаврскую работу выполнила студентка Тольяттинского государственного университета, Коновальчук Анастасия Сергеевна. Название работы: «Исследование технологий реализации алгоритмов линейной алгебры с использованием технологии CUDA».Актуальность работы заключается в том, что в наши дни компьютерное прогнозирование стало необходимой составляющей решения трудоемких инженерных задач. Компьютерное прогнозирование непосредственно связано с системами линейных алгебраических уравнений, являющиеся одним из главных объектов линейной алгебры. Применяясь во всех разделах аналитической геометрии, матричное умножение, помогает производить сложные и громоздкие вычисления с минимальной затратой времени.

Объект исследования: вычислительный процесс умножения матриц на аппаратно-программной платформе CUDA.

Предмет исследования: параллельные алгоритмы и программная реализация матричного умножения.

Цель: проанализировать эффективность разработанных алгоритмов умножения матриц с использованием программно-аппаратной архитектуры параллельных вычислений NVIDIA CUDA.

Первая глава работы посвящена анализу матричного умножения и численных способов решения под ЭВМ с распараллеливанием. А также анализу технологии CUDA.

Вторая глава описывает особенности матричного умножения под технологию CUDA, реализацию алгоритмов решения и их тестирование.

Третья глава посвящена процессу анализа эффективности реализованных алгоритмов матричного умножения и их улучшению.

Бакалаврская работа выполнена на 53 страницах, состоит из введения, трёх глав, заключения, списка литературы, состоящего из 24 литературных источников, 28 рисунков и 2 таблиц.

ABSTRACT

The title of graduation work is “Implementation Technology Research of the Linear Algebra Algorithms on a Supercomputer Using NVIDIA CUDA Technology”.

The relevance of the work lies in the fact that computer modeling has become an integral part of solving complex engineering problems. Computer simulation is tightly connected to operations over matrices including their multiplication which is one of the main mathematical operations of the linear algebra. Matrix multiplication is used in all sections of analytical geometry; they help to make massive and complex calculations with the minimum time expense.

The object of this work is a parallel approach for matrix multiplication in the CUDA environment.

The subject of this work is demonstrating the way code for matrix multiplication is run in the massively parallel environment presented by today’s graphics cards, and its efficiency.

The goal of this coursework is implementation of the parallel algorithms for matrix multiplication in order to demonstrate how code is run in CUDA environments.

The graduation work includes three chapters.

The first chapter introduces some theoretical details like NVidia CUDA’s expected future development, GPU architecture and some information about matrix algebra in the calculations on the Nvidia Cuda.

The second chapter includes the representation of the matrices and their multiplication. There is also a testing of the sequential program of matrix multiplication on PC.

The third chapter is devoted to the process of carrying out the efficiency experiments and analyzing them for the future improvement.

The graduation thesis is 53 pages, which consist of an introduction, three chapters, conclusion, references from 24 literature sources, 28 figures and 2 tables.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Все вычислительные системы, от мобильных до суперкомпьютеров, становятся гетерогенными, огромное количество параллельных компьютеров используется для повышения энергоэффективности и вычислительной пропускной способности.

Ускоренные вычисления с GPU- это использование графического процессора (GPU) вместе с процессором CPU для ускорения научных, аналитических, инженерных, потребительских и корпоративных приложений. Впервые они были открыты компанией NVIDIA® в 2007 году, ускорители GPU стали энергоэффективными центрами обработки данных в государственных лабораториях, университетах, предприятиях, а также широко используются на малых и средних предприятиях по всему миру. Графические процессоры ускоряют приложения на платформах, начиная от автомобилей, мобильных телефонов и планшетов, до беспилотных летательных аппаратов и роботов.

Существует много преимуществ использования суперкомпьютеров. Во-первых, использование графических ускорителей NVIDIA позволяет значительно увеличить скорость вычислений, она становится в тысячу раз быстрее обычного персонального компьютера, что очень полезно, например, при прогнозировании погоды. Потребовались бы десять дней для ПК, чтобы спрогнозировать прогноз погоды всего на одни сутки, что нельзя сказать о суперкомпьютере, который значительно отливается своей мощностью, а следовательно и скоростью выполнения сложных операций. Второе преимущество суперкомпьютера заключается в то, что он обладает самым передовым технологическим уровнем, например, арсенид Галлума (GaAS).Третий аспект, касающийся этих удивительных машин, которые помогают людям во всем мире, заключается во множестве конкретных архитектурных решений, направленных на повышение быстродействия выполняемых задач.

Суперкомпьютеры дали возможность решению задач огромного размера. Стало возможным приступить к реализации реалистических математических моделей сложных физических явлений и технических устройств, которые ранее невозможно было решать на маломощных последовательных компьютерах[6].

В течение десятилетий суперкомпьютеры стремились к повышению производительности. Но, кроме умопомрачительного повышения быстродействия одного микропроцессора, производители суперкомпьютеров зачастую увеличивали число процессоров в целом. Их труды не прошли даром, потому что в наше время суперкомпьютеры насчитывают десятки и сотни тысяч согласованно работающих процессорных ядер[1].

Производительность в разы увеличилась благодаря распараллеливанию алгоритмов под многоядерные цифровые преобразователи (ЦП), но, к сожалению, увеличивать скорость на традиционных архитектура становится с каждым годом сложнее.

В настоящее время, применение дополнительных ускорителей, которые принимают на себя часть вычислительной нагрузки, является актуальным для высокопроизводительной обработки данных

Основной отличительной чертой современных GPU является включение в архитектуру CUDA (Compute Unified Device Architecture) унифицированного шейдерного конвейера, который в свою очередь позволяет программе задействовать разнообразные фрифмитически-логические устройства (АЛУ), входящее в микросхему [1].

Данное добавление средств было произведено с целью расширения возможности решения спектра задач, начиная от традиционных задач компьютерной графики и заканчивая вычислениями общего назначения.

Производительность вычислений на GPU может в разы увеличиться за счет количества ядер и других архитектурных особенностей. Таким образом, мы приходим к тому, что решение задач матричного умножения на программно-аппаратной архитектуре параллельных вычислений NVidia CUDA является **актуальной.**

Практическая новизна работы заключается в сравнительном анализе программных технологий перемножения матриц на графическом процессоре с учетом архитектурных особенностей плат NVidia Tesla K10.

Объектом бакалаврской работы является вычислительный процесс умножения матриц на аппаратно-программной
платформе CUDA.

Предмет бакалаврской работы – параллельные алгоритмы и программная реализация процесса умножения матриц.

Цель работы: анализ эффективности разработанных алгоритмов матричного умножения с использованием программно-аппаратной архитектуры параллельных вычислений NVidia CUDA.

Задачами работы являются:

1. Изучить методы параллельных алгоритмов матричного умножения.
2. Спроектировать параллельные алгоритмы матричного умножения.
3. Реализовать параллельные алгоритмы умножения матриц на языке CUDA C/C++.
4. Протестировать и исследовать эффективность разработанных алгоритмов матричного умножения.

Бакалаврская работа состоит из трех глав.

Первая глава работы посвящена анализу матричного умножения и численных способов решения под ЭВМ с распараллеливанием. А также анализу суперкомпьютера ТГУ, технологии CUDA и архитектуры NVidia Kepler.

Вторая глава описывает особенности умножения матриц под технологию CUDA, реализацию алгоритмов решения и их тестирование.

Третья глава посвящена процессу анализа эффективности реализованных алгоритмов матричного умножения и их улучшению.

В заключении подводится общая оценка данных реализаций алгоритма, анализ возникших при разработке трудностей, а также перспективы применения алгоритма в различных ситуациях.

Глава 1 УМНОЖЕНИЕ МАТРИЦ БОЛЬШИХ РАЗМЕРНОСТЕЙ НА ЭЛЕКТРОННО-ВЫЧИСЛИТЕЛЬНОЙ МАШИНЕ

1.1 Применение матричного умножения в современном мире

В наше время компьютерное моделирование играется огромную роль в решении сложных инженерных задач. Именно оно дает возможность визуализации задачи и ее представлении в графической форме, более того автоматически переводя данное описание на язык компьютера.

В связи с усложнением объектов исследования и ростом возможностей электронно-вычислительных машин компьютерное моделирование перестало ограничиваться проектированием технических изделий, а, напротив, позволяет в разных областях деятельности человека.

Компьютерное моделирование является неотъемлемой частью решения сложных вычислительных задач линейной алгебры, одной из которых является матричное умножение больших размерностей. Матрицы - это не что иное, как прямоугольное расположение чисел, выражений, символов, которые расположены в столбцах и строках.

Номера, присутствующие в матрице, называются объектами или элементами.

Говорят, что матрица имеет «m» строк и «n» столбцов.

Матрицы находят много применений в научных областях, а также используются при решении практических проблем реальной жизни, тем самым решая значительную часть практических задач.

Ниже приводятся некоторые из основных применений матриц:

- в приложениях, связанных с физикой, матрицы применяются при изучении электрических схем, квантовой механики и оптики,

- при вычислении выходов мощности батареи, преобразовании резисторов электрической энергии и так далее, матрицы играют важную роль в расчетах. Особенно при решении проблем с использованием законов Кирхгофа напряжения и тока существенны матрицы матриц;

- в компьютерных приложениях матрицы играют жизненно важную роль в проектировании трехмерного изображения на двумерный экран, создавая движения кажущиеся реалистичными.

Стохастические матрицы и алгоритмы вычисления собственных значений используются в алгоритмах, которые задействованы в ранжировании веб-страниц в поиске Google.

Матричное вычисление используется при обобщении аналитических понятий, таких как экспоненты и производные, в больших размерах.

Одним из наиболее важных применений матриц в компьютерных приложениях является шифрование кодов сообщений.

Матрицы и их обратные матрицы используются программистами для кодирования или шифрования сообщений. Сообщение выглядит как последовательность чисел в двоичном формате.

Они используются для построения графиков, статистических данных, а также для исследований в самых различных областях.

Матрицы используются для представления данных реального мира, таких как черты характера населения, привычки и т. д.

Они являются наилучшими методами представления для построения общих опросов.

Также широко используются для расчета валовой внутренней продукции в экономике, что в конечном итоге помогает в эффективном исчислении товаров.

Во многих организациях для записи данных экспериментов; в робототехнике и автоматизации матрицы являются базовыми элементами движений роботов.

Движения роботов программируются с вычислением строк и столбцов матриц. Данные для управляющих роботов приведены на основе вычислений из матриц.

Никто не может отрицать, что матричное умножение является важной частью нашей жизни. Матрицы помогают нам в разных областях достичь

невероятных результатов, выполняя простые операции, которые облегчают жизнь человека. Хорошо известно, что для сложных вычислений необходимы более мощные компьютеры, чем ПК, и суперкомпьютеры приходят сюда на помощь.

В настоящее время в области суперкомпьютеров мы наблюдаем феномен, похожий на персональные компьютеры, заменяющие мейнфреймы 25 лет назад. Они были только в исследовательских лабораториях и были недоступны для простых смертных. Ключевым словом для этого процесса является «личный», что означает предназначение для индивидуального использования. Такие персональные супер калькуляторы стали приемлемыми для широкой аудитории, открывая возможности для профессионального и индивидуального использования. Архитектура современных суперкомпьютеров достигла высокого уровня эффективности. Стало возможным построение кластеров национального метеоцентра, или использование в качестве базового элемента для реализации идей электронного искусства.

Похоже, что с ростом производительности настольных персональных компьютеров, рабочих станций и серверов потребность в суперкомпьютерах будет уменьшаться, что неверно. С одной стороны, на рабочих станциях теперь можно успешно запускать несколько приложений, но, с другой стороны, время показало, что появляется устойчивая тенденция к появлению новых приложений, требующих использования суперкомпьютера.

Прежде всего, необходимо указать на процесс, когда суперкомпьютеры проникают в коммерческую сферу, которая ранее была для них абсолютно недоступной. Речь идет не только о графических приложениях для кино и телевидения, где требуется высокая эффективность операций с плавающей запятой, это прежде всего задачи, связанные с интенсивной обработкой (также оперативной обработкой) транзакций для супер больших баз данных. В этом классе могут быть внедрены системы поддержки принятия решений и организация информационных складов. Некоторые люди утверждают, что необходима высокая эффективность процессов ввода и вывода и скорость при

выполнении целочисленных операций для работы с аналогичными приложениями, а также для таких приложений оптимальными являются компьютерные системы. Например, система Himalaya MPP компании Tandem, компьютеры SMP SGI CHAL ENGE и Alpha Server 8400 от DEC не являются полностью суперкомпьютерами. Необходимо помнить, что такие требования возникают, в частности, из ряда применений ядерной физики, например, при обработке результатов экспериментов на ускорителях частиц. Хорошо известно, что ядерная физика является классическим полем для применения суперкомпьютеров с их первого появления.

Традиционные сферы применения суперкомпьютеров всегда были научные изыскания: физика плазмы и статистическая механика, физика конденсированных сред, молекулярная и атомная физика, теория элементарных частиц, Газовой динамики и теории турбулентности и физической астрономии. В области химии существуют различные области вычислительной химии: квантовая химия (в том числе расчеты электронной структуры для разработки новых материалов, например, катализаторы и сверхпроводники), молекулярная динамика, химическая кинетика, теория поверхностных явлений и химия твердого тела и медицина. Естественно, что количество полей использования зависит от соответствующих союзов соответствующих наук, например, от химии и биологии, а также от совпадений с техническими приложениями. Таким образом, метеорологические проблемы, такие как изучение атмосферных явлений и проблемы долгосрочного прогноза погоды, для которых не хватает мощностей для современных суперкомпьютеров для поиска решений, тесно связаны с решениями перечисленных выше физических проблем.

Суперкомпьютеры традиционно используются для военных целей. За исключением очевидных проблем разработки оружия массового уничтожения и проектирования самолетов и ракет, важно упомянуть о разработке тихоходных подводных лодок и т. Д. Наиболее известным примером является американская программа SDI (Стратегическая оборонная инициатива). Компьютер MPP

Министерства энергетики США для моделирования ядерного оружия позволит отменить ядерные испытания в этой стране в целом.

Матричное умножение является одной из основных математических операций вне повседневной арифметики. К нему можно эффективно сэкономить много других существенных матричных операций, таких как гауссово исключение, разложение LUP, детерминант или обратная матрица. Матричное умножение также используется как подпрограмма во многих вычислительных задачах, которые на первый взгляд не имеют ничего общего с матрицами.

Матрицы - одна из важнейших частей современного образования. Невозможно представить квантовую физику без матричной квантовой механики, которая помогает выполнять сложные вычисления. Например, матрицу можно описать известной физической величиной, удовлетворяющей уравнению движения в любой момент.

Матрицы широко используются в технике. Например, каждое изображение на экране представляет собой двумерную матрицу, элементы которой являются точками разных цветов.

С помощью матриц можно записать некоторые экономические зависимости. Например, таблица распределения ресурсов отдельных отраслей экономики.

Поскольку требуется невероятная емкость, суперкомпьютеры снова спасаются.

В современном мире трудно представить научные исследования без компьютеров. Использование суперкомпьютеров поможет провести такие исследования в областях, где прямой эксперимент или простая разработка моделей сложны или невозможны. В таких ситуациях незаменимым методом исследований и разработок является компьютерное моделирование. Например, исследование процессов в планете подпочвы при сверхвысоких температурах и давлении или анализ сейсмической активности или развитие современных

вооружений и средств их нейтрализации. Без суперкомпьютеров разработка новых многоцелевых материалов была бы невозможна.

Использование персональных суперкомпьютеров также обеспечивает трехмерное производство контента высокой четкости. В частности, использование суперкомпьютера для этой цели позволяет людям наслаждаться разнообразными визуально богатыми и вычислительно-интенсивными приложениями, такими как видеоигры, фильмы, мультимедиа и т. д.

Быстрота решения задач находится в сильной зависимости от тактовых частот основного процессора(CPU) и памяти. Процессоры со значительной частотой и огромным числом встроенной памяти обладают отличной производительностью, однако не считаются массовыми из-за очень значительной стоимости.

Вследствие чего в замену CPU для решения задач, которые обладают параллелизмом, нередкого применяют графические ускорители(GPU).

Впервые всеми известный термин GPU(Graphics Processing Unit) был использован компанией NVIDIA. Он означал, что первоначально используемый графический ускоритель применялся только для улучшения быстродействия трехмерной графики. Позже данное определение было расширенно и теперь, GPU является пригодным для решения вычислительных задач более широкого класса.

Функционал первых графических ускорителей оставлял желать лучше, они использовались лишь для наложения текстур и растеризации. Центральный процессор выполнял обработку и выдавал на выход ускорителю лишь вершины. Несмотря на это универсальным центральным ускорителям требовалось больше времени на решение простейших задач, что и дало начало широкому распространению и использованию GPU.

В наши дни графические ускорители GPU являются массивно-параллельными вычислительными устройствами с производительностью порядка TFLOPS (1012 FLOPS или 1012 операций в секунду), которые

обладают большим объемом собственной высокоскоростной памяти и возможностью обработки данных без обращения к CPU.

1.2 Общий вид умножения матриц

Дана матрица A (m × r) из m строк и r столбцов, где каждый из ее элементов обозначается aij (1 ≤ i ≤ m и 1 ≤ j ≤ r), и матрица B (r × n), в которой r строк и n столбцов, где каждый элемент обозначается bij (1 ≤ i ≤ r и 1 ≤ j ≤ n). Матрица C, полученная в результате операции умножения матриц A и B, C = A × B. Каждый ее элемент обозначается cij (1 ≤ i ≤ m и 1 ≤ j ≤ n), и вычисляется следующим образом

$$c_{ij} = \sum_{k=1}^{r} (a_{ik} \times b_{kj})$$

$$\begin{matrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{matrix} \qquad \begin{matrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{33} \end{matrix}$$

Простейший способ умножения двух матриц занимает около n3 шагов.

Количество операций, необходимых для умножения A x B:

$$m \times n \times (2r-1)$$

Для простоты анализа обычно рассматривают квадратные матрицы порядка n. Количество основных операций между скалярами вычисляется следующим образом:

$$2n3 - n2 = O(n3)$$

Таким образом для реализации матричного умножения необходимо, чтобы количество строк матрицы A совпадало с количеством столбцов матрицы B, в противном случае они произведение неопределенно и не имеет места быть.

1.3 Математическое описание матричного умножения

Дано:

1. Для простоты мы будем работать с квадратными матрицами размера nxn.

2. Матрицы для умножения - A и B. Оба они будут рассматриваться как плотные матрицы, результат будет сохранен в матрице C.

3. Предполагается, что узлы обработки являются однородными, благодаря этой однородности можно добиться балансировки нагрузки.

Последовательный алгоритм умножения матриц показан на рисунке 1.1:

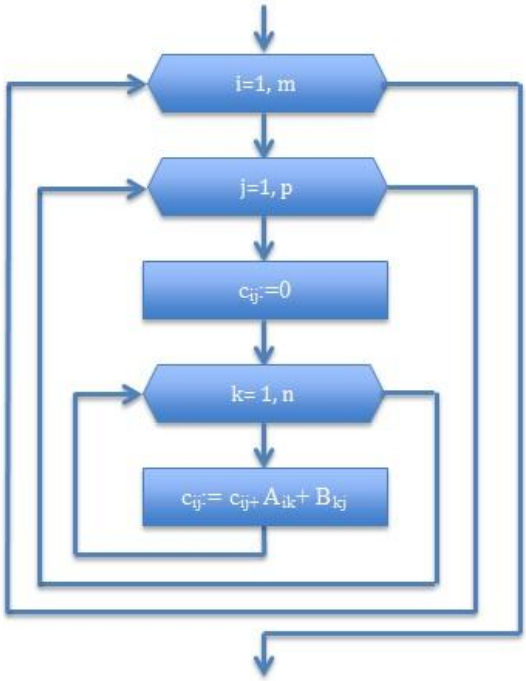

Рисунок 1. 1-Последовательный алгоритм умножения матриц

Реализация. Рассмотрим две квадратные матрицы A и B размера n, которые необходимо умножить:

1. Необходимо поместить эти матрицы в квадратные блоки p, где p - количество доступных процессов.

2. Необходимо создать матрицу процессов размером p1 / 2 x p1 / 2, чтобы каждый процесс мог поддерживать блок матрицы A и блок матрицы B.

3. Каждый блок отправляется каждому процессу, а скопированные субблоки умножаются вместе, результаты добавляются к частичным результатам в субблоках C.

4. Подблоки A перекатываются на один шаг влево, а подблоки B на один шаг вверх.

5. Повторить шаги 3 и 4 sqrt(p) раз.

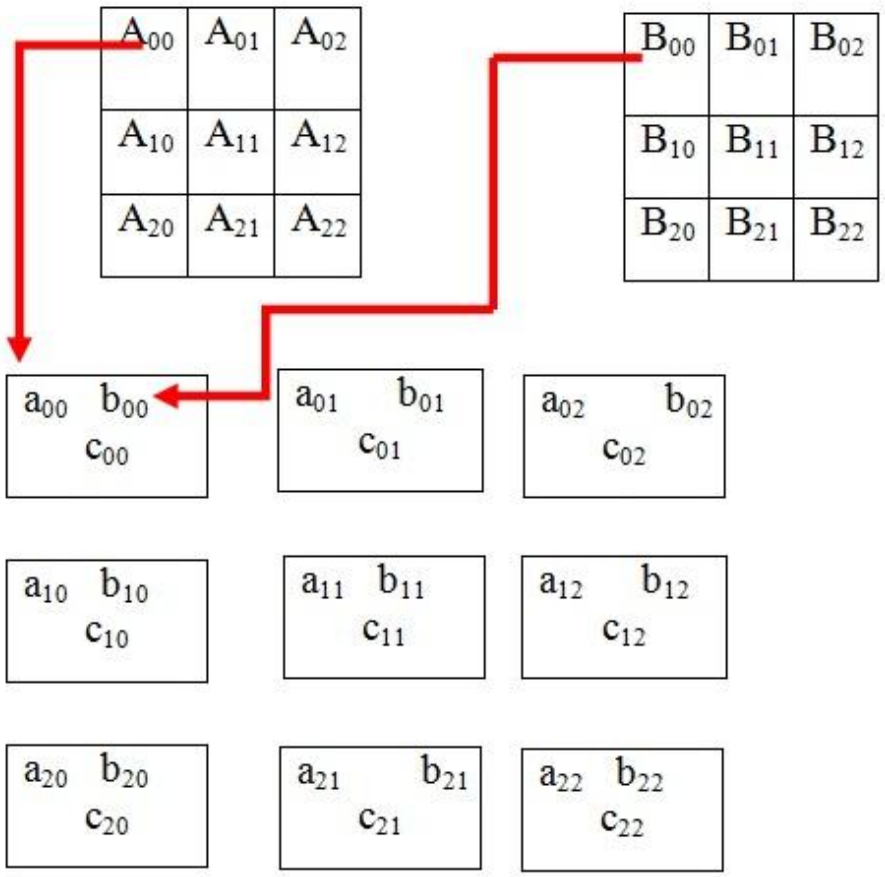

Рисунок 1. 2 - Матрица A и матрица B

Рассмотрим умножение двух матриц:

Эти матрицы разделены на 4 квадратных блока, как показано на рисунке 1.3:

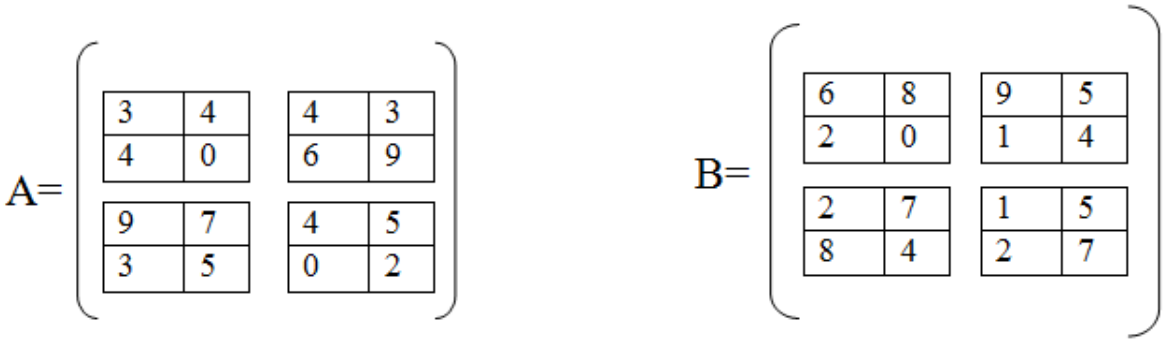

Рисунок 1. 3- Матрица A и матрица B

Матрицы A и B после первоначального выравнивания (рисунок 1.4):

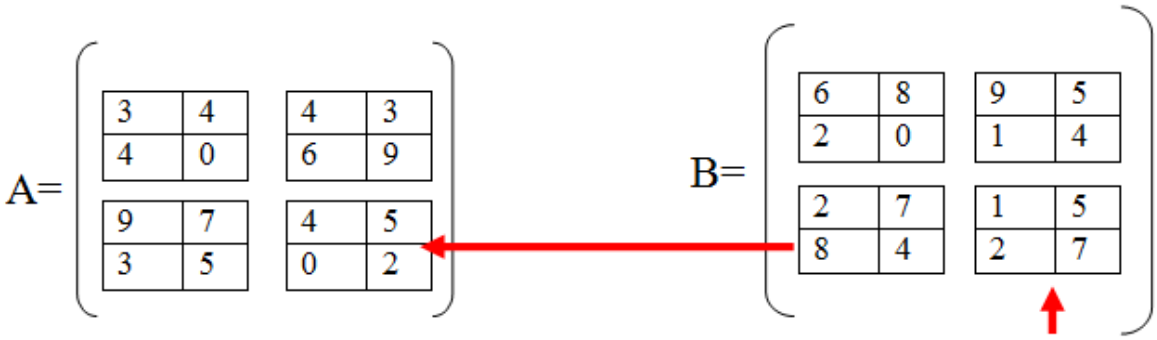

Рисунок 1. 4- Первоначальное выравнивание

Локальное матричное умножение (рисунок 1.5).

$C_{0,0}=$ | 3 | 4 | | 4 | 0 | x | 6 | 8 | | 2 | 0 | = | 26 | 24 | | 24 | 32 |

$C_{0,1}=$ | 4 | 3 | | 6 | 9 | x | 9 | 5 | | 1 | 4 | = | 39 | 32 | | 63 | 66 |

$C_{1,0}=$ | 9 | 7 | | 3 | 5 | x | 2 | 7 | | 8 | 4 | = | 74 | 91 | | 46 | 41 |

$C_{1,1}=$ | 4 | 5 | | 0 | 2 | x | 1 | 5 | | 2 | 7 | = | 14 | 55 | | 4 | 14 |

Рисунок 1. 5- Матричное умножение

Сдвиг матрицы A на один шаг влево, сдвиг B один шаг вверх (рисунок 1.6).

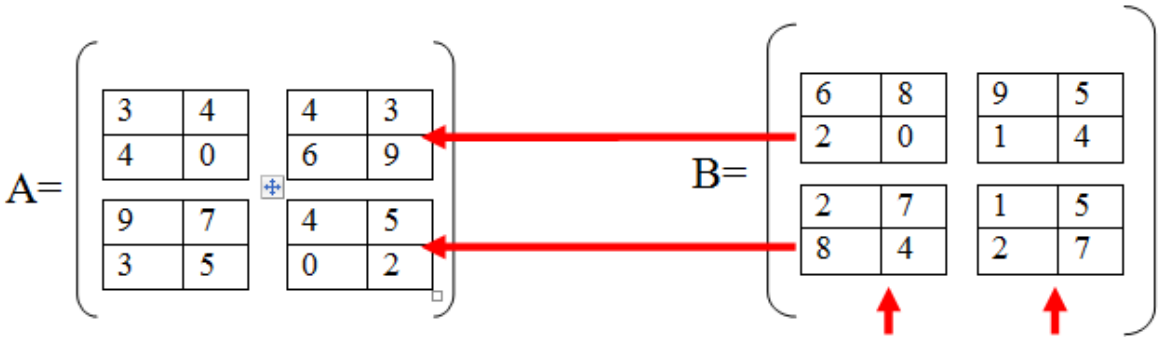

Рисунок 1. 6- Смещение матриц A и B

Локальное матричное умножение (рисунок 1.7).

$C_{0,0}$= $C_{0,0}$+ | 4 | 3 | / | 6 | 9 | x | 2 | 7 | / | 8 | 4 | = | 26 | 24 | / | 24 | 32 | + | 32 | 40 | / | 84 | 78 | = | 58 | 64 | / | 108 | 110 |

$C_{0,1}$=$C_{0,1}$+ | 3 | 4 | / | 4 | 0 | x | 1 | 5 | / | 2 | 7 | = | 39 | 32 | / | 63 | 66 | + | 11 | 43 | / | 4 | 20 | = | 50 | 75 | / | 67 | 86 |

$C_{1,0}$=$C_{1,0}$+ | 4 | 5 | / | 0 | 2 | x | 6 | 8 | / | 2 | 0 | = | 74 | 91 | / | 46 | 41 | + | 34 | 32 | / | 4 | 0 | = | 108 | 123 | / | 50 | 41 |

$C_{1,1}$=$C_{1,1}$+ | 9 | 7 | / | 3 | 5 | x | 9 | 5 | / | 1 | 4 | = | 14 | 55 | / | 4 | 14 | + | 88 | 73 | / | 32 | 35 | = | 102 | 128 | / | 36 | 49 |

Рисунок 1. 7-Матричное умножение

Мы получаем результат (рисунок 1.8).

C=

| 58 | 64 | / | 108 | 110 | | 50 | 75 | / | 67 | 86 |

| 108 | 123 | / | 50 | 41 | | 102 | 128 | / | 36 | 49 |

Рисунок 1. 8 -Матрица C

В данном пункте были рассмотрены матрицы A и B размерностью 4x4.Из рисунка 1.7 можно сделать вывод о неэффективности последовательного

алгоритма матричного умножения, связанного с большой затратой времени на его реализацию. Наблюдается необходимость улучшения быстродействия данного процесса.

1.4 Архитектура графического процессора

Как и в центральных процессорах (CPU), модули графической обработки (GPU) также имеют разнообразные архитектуры и функции. Рассмотрим формальную и общепринятую фундаментальную архитектуру. Архитектура параллельных вычислений GPU включает в себя многопроцессорное аппаратное обеспечение, обладающее сложными технологиями, а также сложное управление аппаратными задачами, поскольку ранние графические процессоры предназначены для обработки графических рабочих нагрузок высокого уровня.

Графические процессоры состоят из множества параллельных многопроцессорных процессоров, которые также называются потоковыми мультипроцессорами. Каждый потоковый мультипроцессор содержит набор процессорных ядер, как показано на рисунке 1.9.

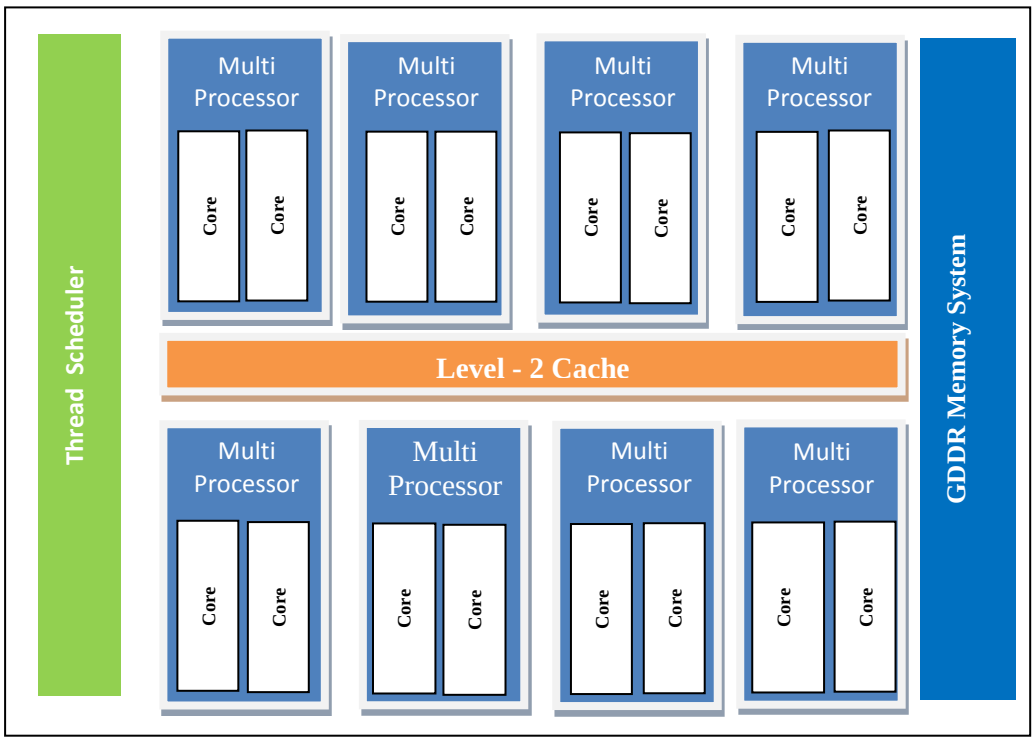

Рисунок 1. 9- Архитектура графического процессора

Планировщик потоков, показанный на рисунке 1.9, работает на основе аппаратного обеспечения. Этот планировщик потоков отвечает за планирование потоков по кластерам обработки потоков, что достигается почти на 100%. Если поток ожидает доступа к памяти, планировщик может выполнить незначительную операцию и немедленно переключиться на другой поток. Поскольку управление созданием потоков осуществляется по выделенному оборудованию многие потоки могут быть созданы, а также легко поддержаны и эффективно спланированы.

Графические процессоры также содержат различные уровни моделей памяти. Иерархия выглядит следующим образом:

- регистры частной памяти;
- локальная память или встроенная память с нуля;
- общий глобальный кеш данных постоянной памяти;
- глобальная память;
- память хоста.

Все вышеупомянутые виды памяти работают с разными тактовыми частотами; Обычно память, которая намного ближе к процессору или ядру, быстрее, подобно процессорам. В вышеупомянутом порядке памяти, когда мы переходим от (1) Регистры к (5) Память хоста, временные циклы для доступа к данным в этой памяти увеличиваются, поэтому латентность памяти увеличивается. Регистры GPU похожи на регистры процессора и работают с очень небольшим количеством тактовых циклов, обычно 1-2. Локальная память или память блокнота Onchip могут использоваться всеми рабочими элементами в рабочей группе. Эта локальная память также является оперативной памятью. В отличие от кеша, память с помощью блокнотной памяти управляется программным обеспечением. Таким образом, память с нуля может быть запрограммирована разработчиком для достижения максимальной производительности. Общий кэш данных глобальной памяти - это аппаратный управляемый кеш для глобальной и постоянной памяти. Постоянная память также является частью глобальной памяти. Глобальная память содержит очень

большой размер памяти с очень высокой задержкой. Глобальная память может использоваться всеми вычислительными устройствами в устройстве.

Ускорение GPU широко используется во всех возможных приложениях, например GPUdb. Это распределенная база данных для многих основных устройств. GPUdb - это масштабируемая распределенная база данных с возможностью запросов в стиле SQL для больших данных. Полный набор возможностей геопространственного расчета. Широкое использование графического процессора было найдено в области проектирования и визуализации. Одним из самых популярных приложений является Autodesk-3ds Max. Трудно представить себе разработку 3D-изображений без такого приложения. 3D-анимационный набор инструментов для моделирования, анимации, моделирования и рендеринга для продуктов и строительных конструкций. Решение научных проблем в области науки и образования не имеет особого значения. Например, приложение NMath Premium, где GPU-ускоренная математика и статистика для .NET, автоматически обнаруживает присутствие графического процессора с поддержкой CUDA во время выполнения и плавно перенаправляет соответствующие вычисления на него.

1.5 Описание вычислителя на графическом процессоре лаборатории распределенных вычислений ТГУ

В 2014 году в ведущем вузе города Тольятти, Тольяттинском Государственном Университете, была создана лаборатория распределённых вычислений.

Серверная платформа SuperMicroSYS-7047GR-TPRF, с двумя восьми ядерными CPU Intel Xeon E5-2670, которая функционирует на максимальной частоте 3.3Гц. Она легла в основу Тольяттинского суперкомпьютера, оперативная память которого составляет 64 Гб высокоскоростной памяти третьего поколения.

Четыре ускорителя NVIDIA Tesla K10 имеют 12288 ядер графических процессоров, обеспечивают производительность в 18.32 TFLOPS с одинарной точностью.

GPU NVIDIA Tesla – это массивно параллельные ускорители, основанные на платформе параллельных вычислений NVIDIA CUDA. GPU Tesla обеспечивают наилучшую производительность для эффективных экономичных и высокопроизводительных вычислений, а также служат для научных и коммерческих приложений, что отличает их от CPU [14].

Tesla K10 основана на архитектуре третьего поколения от NVIDIA для вычислений на CUDA – Kepler, который состоит из 7,1 млрд транзисторов, Kepler и является самым быстрым и эффективным в мире высокопроизводительным процессором для вычислений на суперкомпьютере. Данная архитектура предназначена для максимизирования производительности вычислений с наивысшей энергоэффективностью, делая гибридные вычисления значительно проще, доступнее и применимее к более широкому набору приложений и является одной из самых быстрых и эффективных на сегодняшний день [13, 14].

Потоковый мультипроцессор SM (Streaming Multiprocessor) или SMX (Next Generation Streaming Multiprocessors), имеющий 192 вычислительных ядра, напоминает CPU ядро и является основной единицей построения видеокарт [21].

За счет близкого расположения ядер в архитектуре Kepler было достигнуто максимальное количество нитей и блоков, одновременное число которых возросло с 8 до 16, что теоретически увеличило скорость ее работы [20].

Рисунок 1.10 иллюстрирует 16 потоковых мультипроцессоров, содержащиеся в TeslaK10 [14,15].

Рисунок 1. 10– Кластеры SMX в архитектуре Kepler

Рассмотрим рисунок 1.11, иллюстрирующий структуру одного мультипроцессора [14]. В каждом из 16 находятся по 192 вычислительных ядра CUDA core, которые выполняют код параллельно, что в сумме дает 3072 CUDA core. 64 DP Unit ядра исполняют инструкции для типа double. Отношение DP Unit к количеству CUDA cores определяет производительность вычислений с типом float, двойной точностью как 1/3 от производительности с одинарной точностью [17,18].

Компания NVIDIA разработала компилятор для GPU – технология GPGPU (General-Purpose computing on Graphics Processing Units), которая дает возможность программистам реализовывать алгоритмы на более простом языке, например, C, C++, Fortran и OPENACC. Реализация на CUDA представляет собой кроссплатформенную систему компиляции и исполнения программ, части которых работают на CPU (последовательная часть кода и подготовительные стадии) и GPU (вычисления с большим количеством нитей, являющиеся исполнителями вычислений Для повышения эффективности

загрузки Tesla K10 желательно запускать значительно большее количество нитей, чем количество CUDA ядер, в связи с быстрым переключением контекста нитей [23,24].

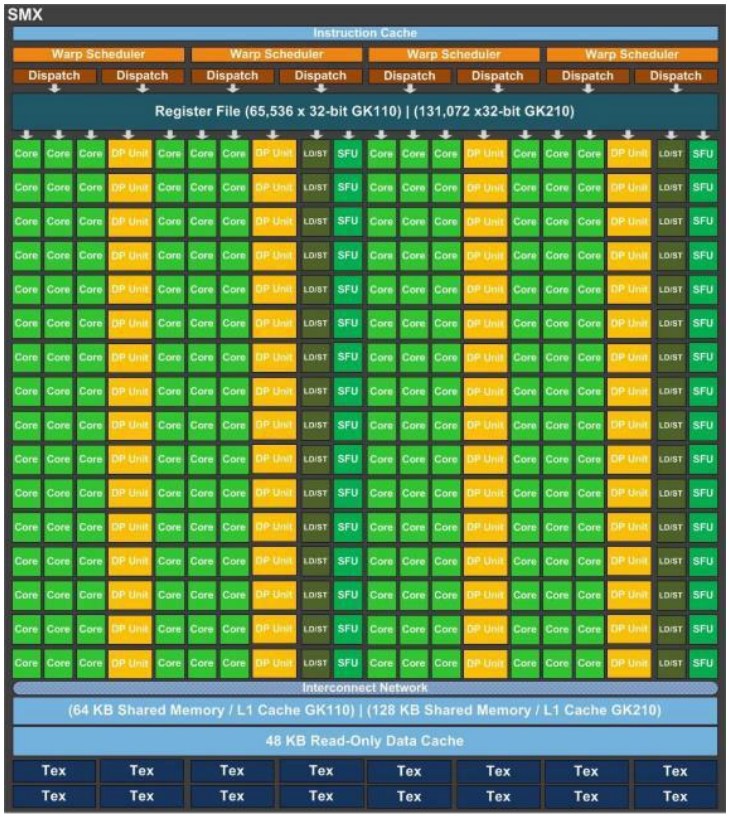

Рисунок 1. 11– Структура SMX

Нити кооперируются в два блока, в каждом из которых по 1536 нитей, имеющих доступ к общей памяти и выполняющихся параллельно друг другу[22].

Рисунок 1.12 показывает блочную архитектуру CPU, ключевым различием между нитями в одном и разных блоках является возможность их синхронизации, она доступна только нитям одного блока. Взаимодействие нитей между собой достигается за счет использования разделяемой памяти.

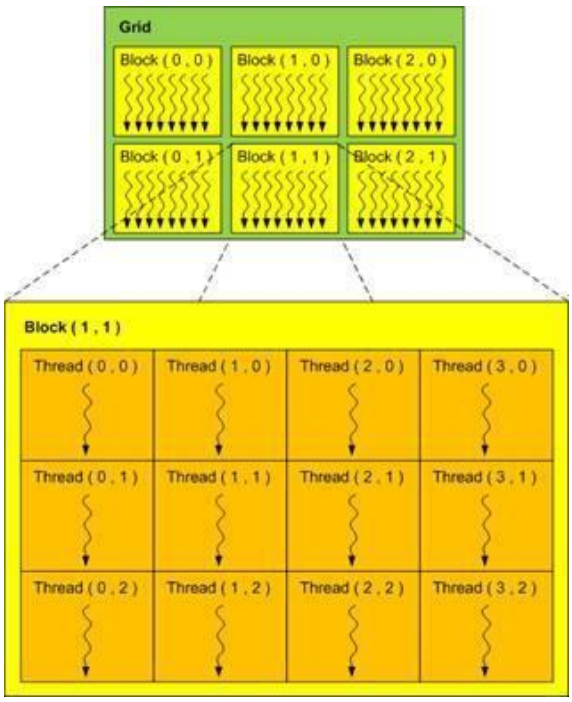

Рисунок 1. 12– Иерархия нитей в CUDA

SIMT (Single Instruction – Multiple Threads) -«одна инструкция и много потоков», предполагающая нахождения нитей различного блока на разных стадиях их выполнения, что показано на рисунке 1.13 [4].

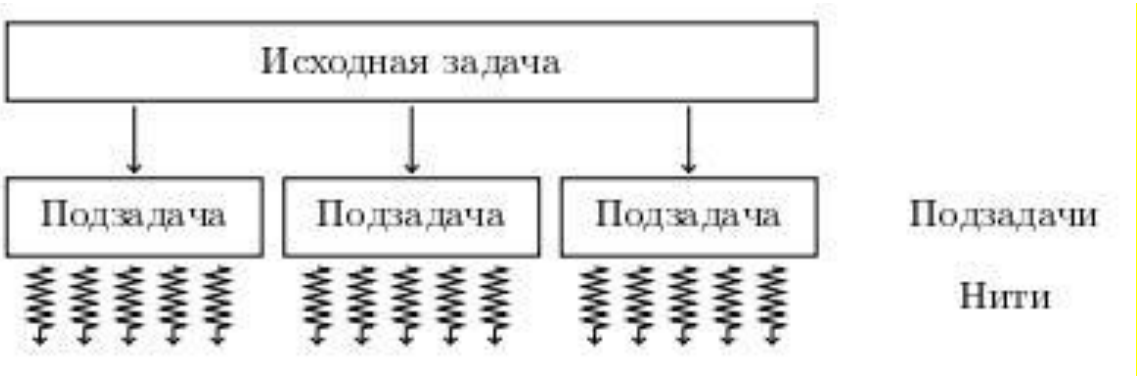

Рисунок 1. 13– Разбиение исходной задачи на набор подзадач

Под варпом подразумевается последовательное разделение блоков на группы по 32 нити в соответствии с их линейным индексом в блоке [17]. Специальные планировщики прослеживают выполнение варпов и выбирают готовые к запуску на CUDA ядрах.

Изначально, задачи, предполагающие интенсивную работу с большими объемами, решаются на TeslaK10, именно поэтому необходимо правильно распределить нагрузки и временя обращения в память видеокарты [23,24].

Существует несколько основных типов памяти, продемонстрированные в таблице 1.1.

Таблица 1. 1 - Типы памяти в GPU NVIDIA

Тип памяти	**Расположение**	**Чтение/запись**	**Скорость работы**
Регистровая	мультипроцессор	чтение и запись	высокая
Локальная	DRAM GPU	чтение и запись	низкая
Разделяемая	мультипроцессор	чтение и запись	высокая
Глобальная	DRAM GPU	чтение и запись	низкая
Константная	DRAM GPU	только чтение	высокая
Текстурная	DRAM GPU	только чтение	высокая

Рассмотрим более детально каждый тип, представленный в таблице 1.1.

Располагаясь в потоковом процессоре, регистровая память относится к наиболее быстрым. Более того она доступна для чтения и записи в рамках одной нити, у которой существует свой набор 32-х битовых регистров, недоступных соседним нитям. Переменные, созданные в процессе выполнения нитей, отправляются в регистры, таким образом, некая часть локальных переменных сохраняется в локальную память.

Следующий вид памяти, рассмотренный в таблице 1.1, является локальная память, находящаяся в оперативной памяти GPU и ограниченная объемом данной памяти, что значительно замедляет ее работу. Каждая нить окружена своей областью локальной памяти, которая доступна для чтения и записи. Данный вид памяти является малоэффективным и во многом проигрывает регистровой памяти.

Третьим рассмотренным видом памяти является разделяемая память, располагающаяся на мультипроцессоре, в месте вычислений. Доступная для чтения и записи, обладающая задержкой на 1-2 порядка меньше, нежели в глобальной памяти, разделяемая память позволяет делить пространство с

кэшем первого уровня. Совместная работа данных структур кэширует медленную глобальную память, так как на обе выделяется только 64 кб. Программист, работающий с разделяемой памятью, имеет возможность напрямую указать какие именно данные могут быть отправлены в нее, именно это и являет основным методом оптимизации программ.

Перейдем к следующему виду памяти, которым является глобальная память, передающая данные между TeslaK10 и ЭВМ, запускающим компилятор. Процесс копирования данных идет через PCI-Express 3 x16 шины, к которым подключены Tesla K10 в серверной платформе SuperMicro SYS-7047GR-TPRF. Объем памяти Tesla K10 составляет 8192 Мб. В сравнении с копированием данных внутри графического ускорителя GPU, скорость записи данных будет гораздо ниже, таким образом необходимо уменьшить количество копирований.

Последующие два типа памяти крайне редки в практики, но не будет лишним их рассмотреть.

Располагаясь в глобальной памяти, размеров 64 кб, константная память доступна только для чтения с устройства всем нитям сети и для записи с хоста.

Последняя, представленный в таблице 1.1 вид памяти, является текстурная память, которая представляет собой участок глобальной памяти доступной только для чтения и расположенная в глобальной памяти.

Разумное использование представленных видов памяти способно увеличить скорость производительности вычислений.

Глава 2 РЕАЛИЗАЦИЯ МАТРИЧНОГО УМНОЖЕНИЯ

2.1 Сравнение различных технологий внедрения параллельного программного обеспечения

Таблица 2.1 показывает, насколько хорошо разные алгоритмы отображают шаблоны программирования. Результаты этих таблиц показывают близкое приближение предсказаний Матсона к результатам проекта, выделенного цветом

Таблица 2. 1- Сравнение OpenMP, MPI и CUDA

	OpenMP	MPI	CUDA
SPMD	☺☺☺	☺☺☺☺ ☺☺☺	☺☺☺☺ ☺☺☺
Loop Parallel	☺☺☺☺ ☺☺	☺	
Master/Slave	☺☺	☺☺☺ ☺☺	
Fork/Join	☺☺☺ ☺		

Результаты этого сравнения не полностью описывают все особенности этой технологии. Начиная с версии 7 CUDA появился динамический параллелизм, позволяющий пользователям делать рекурсивные запросы параллельных ядер. Используя эти результаты, можно с уверенностью сказать, что:

- парадигмы OpenMP подходят для шаблонов уровня «Loop level», и в плане алгоритмов лучше всего подходят алгоритмы «Геометрическое разложение» и «Разделяй и властвуй»;

-потоки подходят для шаблонов «Loop level» и «Fork and Join», а также алгоритмов «Разделяй и властвуй» и алгоритмов «Геометрическое разложение»;

-MPI лучше всего подходит для шаблонов SPMD и «Master-Worker», и с точки зрения алгоритмов намного лучше решать алгоритмы «геометрического разложения», а не парадигмы разделяемой памяти;

- CUDA хорошо подходит только для SPMD (супер-набор SIMD). Было также очевидно, что CUDA - лучшая парадигма алгоритмов «Геометрическая декомпозиция».

В данной работе будет продемонстрирована эффективность технологии CUDA в примере матричного умножения. С этой целью будет рассмотрена архитектура GPU, будут проанализированы алгоритмы параллельного умножения матрицы для обнаружения зависимости данными, будут реализованы алгоритмы на языке CUDA C, а затем будут выполнены экспериментальные результаты эксперимента.

2.2 Особенности решения систем линейных алгебраических уравнений на CUDA

Отсутствие прямого доступа графического процессора к компьютерной памяти приводит к невозможности полного отказа от работы с CPU,так как оно отвечает за постобработку данных.

Общая концепция CUDA строится в том, что GPU устройство выступает в роли массивно-параллельного сопроцессора к GPU хосту. Программа на CUDA включает работу как CPU, так и GPU, где GPU отвечает за подготовку и копирование данных на девайс, а CPU за задание параметров для ядра.

Графический процессор не может напрямую обратиться к памяти компьютера, также невозможно обойтись простыми указателями при работе с данными, поэтому в CUDA существует специально разработанный набор CUDA SDK.

Так как на графических устройствах наблюдается падение производительности при обращении к типу double, предпочтительнее использование типа float. CUDA поддерживает все стандартные математические функции библиотек языков C,C++ [12], но существует

особенность употребления условного оператора if. Нити одного варпа будут выполнять разные функции при выполнении программы, имеющей несколько ветвлений. Удивителен факт простойки незадействованных нитей. В случае наличия более одного ветвления, производительность GPU может упасть в половину. Задача усложняется с использованием многоуровневого ветвления.
Все нити варпа должны идти по одной ветви условного оператора, другая ветвь считается, не будет и в этом случае не будет потери производительности. Таким образом, в CUDA программах предпочтительно использовать те реализации алгоритмов, в которых меньше ветвления в рамках одного варпа.

Для достижения максимальной производительности на архитектуре Kepler и его мультипроцессоре (SMX) необходим запуск ядра с полным заполнением Kepler нитями и блоками.

Существует несколько основных шагов при работе с CUDA,один из которых подготовка памяти. Для распределения данных по типам памяти в GPU вызывается одна из функций: cudaMalloc, cudaMemcpy и cudaFree все операции проводятся над видеопамятью. У функции cudaMemcpy присутствует дополнительный параметр, отвечающий направление копирования информации.

Следующий шаг, конфигурация блоков. На данном этапе следует найти оптимальный баланс между размером блоков и их количеством за счет снижения количества обращений к глобальной памяти через увеличение количества потоков

Запуск ядра. Для вызова ядра необходимо объявить идентификатор globa. Следует заранее передать ему размерность сетки и блока.

Получение результатов. Для копирования результатов в хост используется функция cudaMemcpy с указанием обратного направления.

Освобождение памяти является последним действием, которое помогает предотвратить утечку памяти и освободить все выделенные ресурсы.

2.3 Реализация и тестирование алгоритмов матричного умножения

2.3.1 Умножение матриц при ленточной схеме разделения данных

Вызывается ядро GPU N раз из кода хоста, чтобы умножить заданную матрицу N раз.

Мелкомодульный подход: основная подзадача - вычисление одного элемента матрицы C

$$c_{ij} = a_i, b_j^T \ , a_i = a_{10}, a_{i1}, \dots, a_{in-1} \ , b_j^T = b_{0j}, b_{1j}, \dots, b_{n-1j}^{\ T}$$

Количество базовых подзадач равно n2. Повторяющийся параллелизм избыточен! Как правило, количество доступных процессоров меньше n2 (p <n2), поэтому необходимо выполнить масштабирование подзадачи.

Агрегированная подзадача: Вычисление одной строки матрицы C (количество подзадач равно n)

Распределение данных: разбиение по строкам с блочной полосой для матрицы A и разбиение по столбцу с блочной полосой для матрицы B (рис. 2.1, 2.2).

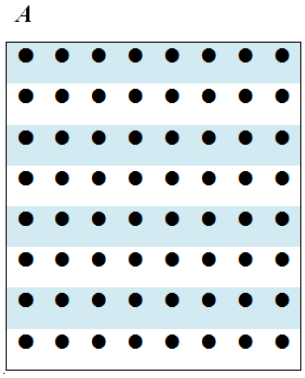

Рисунок 2. 1- Разложение по блочным полоскам для матрицы A

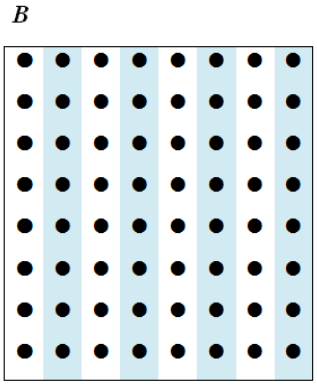

Рисунок 2. 2 -Разбиение на столбцы с блочным разложением для матрицы B

Анализ информационных зависимостей:

- каждая подзадача содержит одну строку матрицы A и один столбец матрицы B;

- каждая итерация каждой подзадачи выполняет вычисление внутреннего продукта своей строки и столбца, в результате получается соответствующий элемент матрицы C;

-тогда каждая подзадача i, $0 \leq i < n$, передает свой столбец матрицы B для подзадачи с номером $(i + 1) \bmod n$.

После того как все итерации алгоритма были получены, все столбцы матрицы B расположены внутри каждой подзадачи один за другим.

На рисунке 2.3 представлена схема информационных зависимостей.

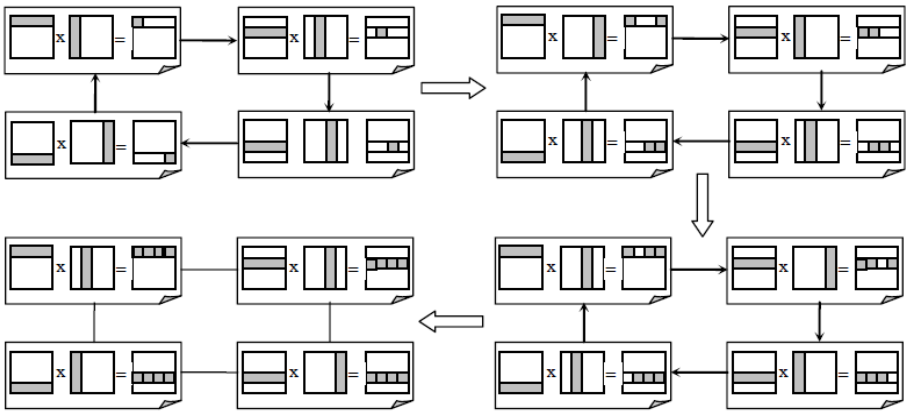

Рисунок 2. 3-Схема информационных зависимостей

Агрегирование и распределение подзадач среди процессоров:

– в случае, если количество процессоров p является меньше, чем количество основных подзадач n, вычисления могут быть агрегированы таким способом, которым каждый процессор выполнил бы несколько внутренних продуктов матрицы строки и матричные столбцы B. В этом случае после завершения вычисления, каждая агрегированная основная подзадача определяет несколько строк получающейся матрицы C,

- в таких условиях начальная матрица A разбивается на p горизонтальных полос, а матрица B разлагается на p вертикальных полос,

– распределение подзадачи среди процессоров должно удовлетворить требования эффективного представления кольцевой структуры зависимостей от информации о подзадаче.

Реализация алгоритма CUDA C. В листинге (рис. 2.4) показана реализация алгоритма матричного умножения для умножения матриц при ленточном разделении данных.

```
typedef struct {
            int width;
            int height;
            int stride;
            t_matrix* elements;
} Matrix;

__global__ void MatMulKernel(Matrix A, Matrix B, Matrix
C)
{
     t_matrix Cvalue = 0;
     int row = blockIdx.y * blockDim.y + threadIdx.y;
     int col = blockIdx.x * blockDim.x + threadIdx.x;
     for (int m = 0; m < A.width; ++m)
          Cvalue += A.elements[row * A.width + m]
                    * B.elements[m * B.width + col];
         C.elements[row * C.width + col] = Cvalue;
}
```

Рисунок 2. 4- Реализация алгоритма матричного умножения для умножения матриц при ленточном разделении данных

Память для хранения исходных матриц и матрицы результатов распределяется на CPU. Чтобы получить доступ к работе на ядре GPU, матрицы копируются в глобальную память GPU. После вычисления результирующая матрица копируется в ОЗУ.

В вычислительном ядре MatMulKernel каждый поток считывает одну строку A и один столбец B и вычисляет соответствующий элемент C, как показано на рисунке 2.5.

Поэтому A считывается B.width раз из глобальной памяти, а B -A. Height раз.

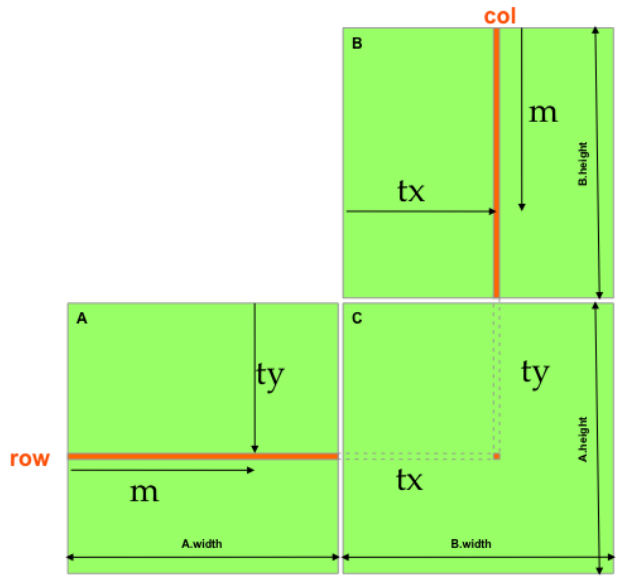

Рисунок 2. 5-Матричное умножение

Такая реализация умножения матриц не показывает, что CUDA имеет возможность оптимизировать использование памяти, варпов и потоков. Существуют аспекты, которые следует учитывать при оптимизации реализации данного алгоритма.

2.3.2 Метод Фокса

Распределение данных: схема шахматной доски представлена на рисунке 2.6.

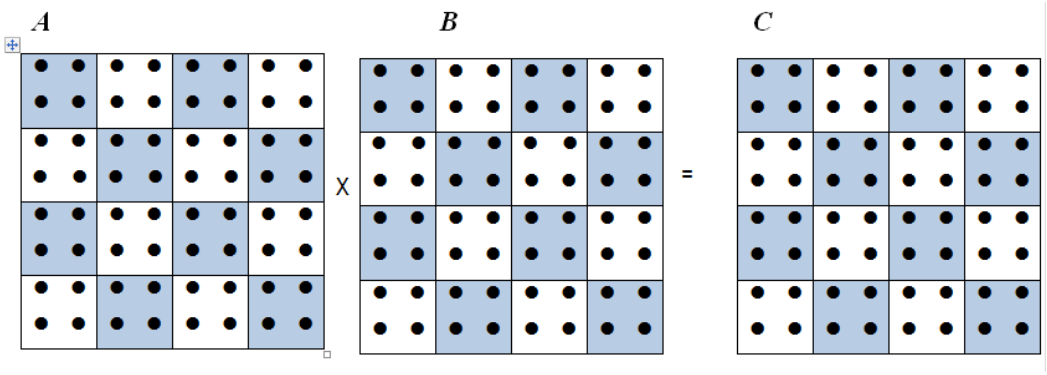

Рисунок 2. 6- Схема шахматной доски из матриц A, B и C

Основная подзадача - это процедура, которая вычисляет все элементы одного блока матрицы.

$$\begin{matrix} A_{00} & A_{01} \dots A_{0q-1} \\ \dots & \\ A_{q-10} & A_{q-11} \dots A_{q-1q-1} \end{matrix} \times \begin{matrix} B_{00} & B_{01} \dots B_{0q-1} \\ \dots & \\ B_{q-10} & B_{q-11} \dots B_{q-1q-1} \end{matrix} = \begin{matrix} C_{00} & C_{01} \dots C_{0q-1} \\ \dots & \\ C_{q-10} & C_{q-11} \dots C_{q-1q-1} \end{matrix}$$

$$C_{ij} = \sum_{s=0}^{q-1} A_{is} B_{sj}$$

Анализ информационных зависимостей включает следующие характеристики:

- подзадача с номером (i, j) вычисляет блок Cij результирующей матрицы C. В результате подзадачи образуют qxq двумерную сетку;

- каждая подзадача содержит 4 блока матрицы:

  • блок Cij результирующей матрицы C, который вычисляется в подзадаче,

  • блок Aij матрицы A, который был помещен в подзадачу до начала расчета,

  • блоки Aij 'и Bij' матрицы A и матрицы B, которые принимаются подзадачей во время вычислений.

Во время итерации l, 0≤l <q выполняется алгоритм:

1. Подзадача (i, j) передает свой блок Aij матрицы A всем подзадачам одной и той же горизонтальной строки i сетки; Индекс j, который определяет положение подзадачи в строке, может быть получен при помощи уравнения:

$$j = (i+l) \bmod q,$$

где операция mod - это процедура вычисления остатка целочисленного деления.

2. Каждая подзадача выполняет умножение принятых блоков Aij 'и Bij' и добавляет результат к блоку Cij:

$$C_{ij} = C_{ij} + A'_{ij} \times B'_{ij}.$$

3. Каждая подзадача (i, j) передает свой блок Bij соседнему, который является предыдущим в той же вертикальной линии (блоки подзадач первой строки передаются подзадачи последней строки сетки).

На рисунке 2.7 представлена схема информационных зависимостей.

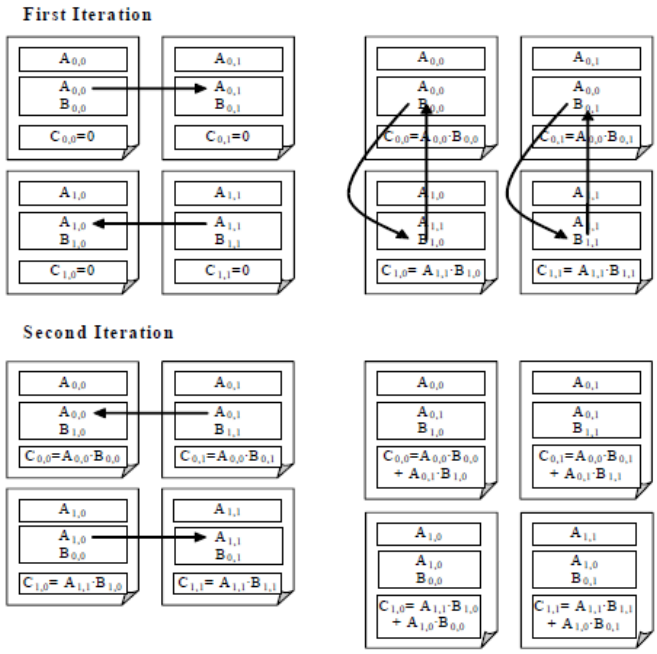

Рисунок 2. 7- Схема информационных зависимостей

Масштабирование и распределение подзадач среди процессоров включает следующие характеристики:

- размеры блоков матриц могут быть выбраны так, чтобы количество подзадач совпадало с количеством доступных процессоров p;

- наиболее эффективное выполнение алгоритма Фокса может быть достигнуто, когда топология сети связи представляет собой двумерную сетку;

- в этом случае подзадачи могут быть распределены между процессорами естественным образом: подзадача (i, j) должна быть помещена в процессор pi, j.

В листинге показана реализация на CUDA C алгоритма матричного умножения для метода Фокса (рисунок 2.8).

```
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix
C) {
      __shared__ t_matrix As[BLOCK_SIZE][BLOCK_SIZE];
      __shared__ t_matrix Bs[BLOCK_SIZE][BLOCK_SIZE];
      int bx = blockIdx.x, by = blockIdx.y,
            tx = threadIdx.x, ty = threadIdx.y,
            Row = by * BLOCK_SIZE + ty, Col = bx *
BLOCK_SIZE + tx;
      t_matrix Pvalue = 0;
      for (int m = 0; m < (A.width - 1) / BLOCK_SIZE +
1; ++m) {
            if (Row < A.height && m * BLOCK_SIZE + tx <
A.width)
                  As[ty][tx] =
                        A.elements[Row * A.width + m *
BLOCK_SIZE + tx];
            else
                  As[ty][tx] = 0;
            if (Col < B.width && m * BLOCK_SIZE + ty <
B.height)
                  Bs[ty][tx] =
                        B.elements[(m * BLOCK_SIZE + ty)
* B.width + Col];
            else
                  Bs[ty][tx] = 0;
            __syncthreads();
            for (int k = 0; k < BLOCK_SIZE; ++k)
                  Pvalue += As[ty][k] * Bs[k][tx];
            __syncthreads();
                              }
      if (Row < C.height && Col < C.width)
            C.elements[Row * C.width + Col] = Pvalue;
}
```

Рисунок 2. 8- Реализация алгоритма матричного умножения для метода Фокса

Эта реализация матричного умножения помогает использовать преимущества разделяемой памяти. В данной реализации каждый блок потока отвечает за вычисление одной квадратной подматрицы Csub C, и каждый поток в блоке отвечает за вычисление одного элемента Csub. Как показано на рис. 2.9, Csub равен произведению двух прямоугольных матриц: подматрицы A размерности (A.width, block_size), которая имеет те же индексы строк, что и Csub, и подматрицу B размерности (Block_size, A.width), который имеет те же индексы столбцов, что и Csub. Чтобы вписаться в ресурсы устройства, эти две прямоугольные матрицы делятся на столько квадратных матриц размерности block_size, сколько необходимо, а Csub вычисляется как сумма продуктов этих

квадратных матриц. Каждый из этих продуктов выполняется путем первой загрузки двух соответствующих квадратных матриц из глобальной памяти в общую память с одним потоком, загружающим один элемент из каждой матрицы, а затем путем вычисления каждого потока одного элемента продукта. Каждый поток накапливает результат каждого из этих продуктов в регистр после этого записывает результат в глобальную память.

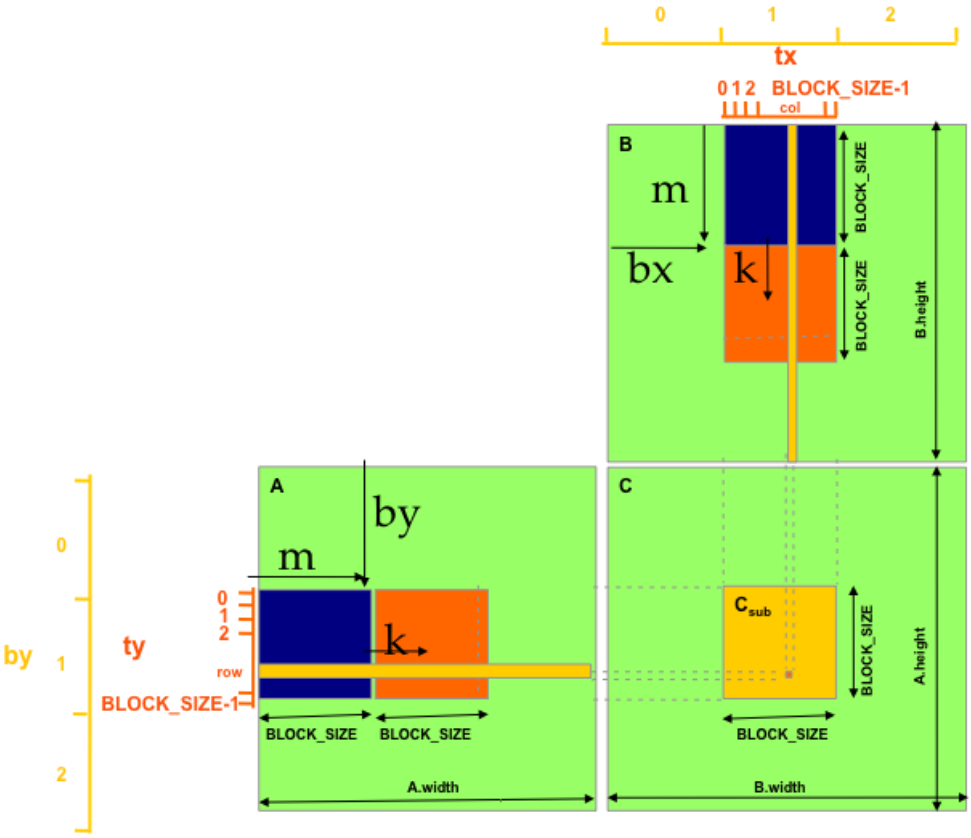

Рисунок 2. 9- квадратная подматрица Csub.

Блокируя вычисления таким образом, мы используем быструю разделяемую память и сохраняем большую пропускную способность глобальной памяти, поскольку A считывается только (B.width / block_size) раз из глобальной памяти, а B (A.height / block_size) раз.

В данной главе было проведен сравнительный анализ различных технологий внедрения параллельного программного обеспечения, выявлены, реализованы и протестированы особенности решения линейных алгебраических уравнений на CUDA.

Глава 3 АНАЛИЗ ЭФФЕКТИВНОСТИ РАЗРАБОТАННЫХ АЛГОРИТМОВ

3.1 Разработка технологии анализа эффективности алгоритмов

Важной характеристикой работы алгоритма на GPU является знание загруженности GPU. Отношение числа активных варпов на мультипроцессоре к максимальному количеству возможных активных варпов называется загруженностью, которая используются для прослеживания эффективности выполнения алгоритмов.

Время выполнения приложения является не менее важной характеристикой проверки эффективности методов. Появляется необходимость сравнивания выполнения на GPU и распределения ресурсов, поэтому будет считаться время, затраченное на создание переменных для GPU, перенос переменных из памяти CPU в память GPU, вычисления на GPU и обратный перенос переменных из памяти GPU в память CPU.

Один из методов анализа эффективности служит Visual Profiler необходимый для профилирования создаваемого приложения. Профилирование показывает затраченное на выполнение каждого ядра время, количество запущено блоков и расходящихся ветвей было выполнено в варпах, объединялись ли операции доступа к памяти со стороны ядра и многое другое.

При запуске профайлер регистрирует значения счётчиков, показывающих определенное событие, по ходу ее исполнения для каждого ядра в отдельности. По окончанию выполнения приложения, профайлер создает таблицу с результатами по запуску каждого ядра, с помощью, которой можно выявить узкие места кода.

При ручном профилировании используется переменная среды CUDA_PROFILE=1. Информация записывается в лог-файл.

Высокая загруженность не всегда является показателем высокой производительности, в то время как низкая загруженность приводит к её ухудшению [19].

CUDA events – метод использования облегченной альтернативы таймеров, улучшающий эффективность работы GPU. Данные события cudaEvent_t используются для того, что измерить время выполнения операций.

Надлежащие функции дают возможность формировать и уничтожать CUDA события, определять позиции начала и окончания рассматриваемого части программного кода, контролировать появление установленного действия либо ожидать его прихода, а кроме того получать период выполнения в миллисекундах. Событие определяется прохождением точки GPU.

CUDA runtime API гарантирует точное измерение периода времени, разрешая приложению разновременно делать запись действия в каждой точке проекта, запрашивать наступило ли это явление, ожидать появления действий и извлекать промежуток периода в миллисекундах среди событий.

Для разработки эффективной реализации параллельного алгоритма на GPU необходимо учитывать ряд особенностей:

- количество потоков;
- использование многопроцессорной системы, в зависимости от:
- количества регистров в потоке;
- суммы разделяемой памяти для блочных потоков;
- количества потоков в блоке;
- конфликтов разделяемой памяти;
- объединения памяти.

Умножение плотных матриц: оптимальный размер блока зависит от емкости как памяти, так и арифметических графических устройств. Оценить реализацию можно с помощью специального калькулятора от разработчиков Nvidia CUDA.

На рисунке 3.1 представлен расчет эффективности для реализации блочного алгоритма умножения матриц.

CUDA GPU Occupancy Calculator

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	3,0	(Help)
1.b) Select Shared Memory Size Config (bytes)	49152	

2.) Enter your resource usage:		
Threads Per Block	1024	(Help)
Registers Per Thread	20	
Shared Memory Per Block (bytes)	8192	

3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	2048	(Help)
Active Warps per Multiprocessor	64	
Active Thread Blocks per Multiprocessor	2	
Occupancy of each Multiprocessor	100%	

Physical Limits for GPU Compute Capability:	3,0
Threads per Warp	32
Warps per Multiprocessor	64
Threads per Multiprocessor	2048
Thread Blocks per Multiprocessor	16
Total # of 32-bit registers per Multiprocessor	65536
Register allocation unit size	256
Register allocation granularity	warp
Registers per Thread	63
Shared Memory per Multiprocessor (bytes)	49152
Shared Memory Allocation unit size	256
Warp allocation granularity	4
Maximum Thread Block Size	1024

Allocated Resources		Per Block	Limit Per SM	= Allocatable Blocks Per SM
Warps	(Threads Per Block / Threads Per Warp)	32	64	2
Registers	(Warp limit per SM due to per-warp reg count)	32	84	2
Shared Memory (Bytes)		8192	49152	6

Note: SM is an abbreviation for (Streaming) Multiprocessor

Maximum Thread Blocks Per Multiprocessor	Blocks/SM	* Warps/Block	= Warps/SM
Limited by Max Warps or Max Blocks per Multiprocessor	2	32	64
Limited by Registers per Multiprocessor	2	32	64
Limited by Shared Memory per Multiprocessor	6		

Note: Occupancy limiter is shown in orange

Physical Max Warps/SM = 64
Occupancy = 64 / 64 = 100%

CUDA Occupancy Calculator	
Version:	5,1
Copyright and License	

Рисунок 3. 1- Калькулятор занятости графического процессора CUDA

Используя блоки из 1024 потоков, разделенная память заполняется 8192 байтами и 20 регистрами, что позволяет использовать 2 блока в каждом

мультипроцессоре. Из-за этого мультипроцессоры не выдерживают холостого хода из-за ограничений.

Мы проведем серию компьютерных экспериментов для оценки эффективности использования технологии CUDA для умножения матриц. Для этого необходимо определить время работы программы Texp.

Матрицы чисел 512x512, 1024x1024, 2048x2048, 4096x4096 и 8192x8192 будут рассмотрены с использованием одного и двухточечного измерения

Чтобы продемонстрировать возможности умножения матриц для реализации алгоритмов необходимо учитывать :

• реализацию алгоритма блока матричного умножения на устройствах с несколькими GPU;

• реализацию матричного умножения на одном устройстве GPU с использованием алгебраической вычислительной ускоренной библиотеки CUBLAS;

• реализацию матричного умножения на нескольких устройствах с использованием ускоренных библиотек GPU алгебраических вычислений CUBLAS-XT.

Поскольку последовательный алгоритм умножения матриц без оптимизации требует гораздо больше времени, чем параллельные алгоритмы для компиляции сравнительных характеристик, и реализации будем использовать библиотеку последовательных алгебраических вычислений BLAS.

Реализация алгоритма блока матричного умножения на устройствах с несколькими GPU.

Чтобы использовать все доступные функции устройства GPU через cudaGetDeviceCount, необходимо получить количество устройств, поддерживающих CUDA. Матрица A делится на блоки строк, затем каждое устройство будет вычислять соответствующие блоки строк результирующей матрицы (рисунок 3.2).

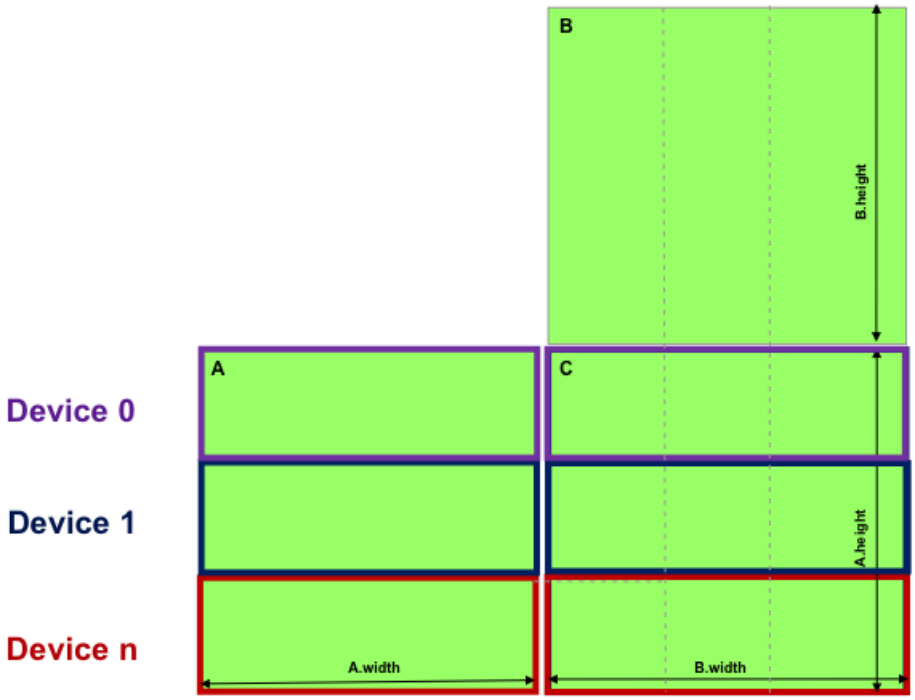

Рисунок 3. 2- Вычисление результирующей матрицы

Выбор ядер для каждого устройства производится асинхронно с процессором.

Код реализации представлен на рисунке 3.3.

```
int deviceCount = 0;
cudaError_t error_id = cudaGetDeviceCount(&deviceCount);
int nrows = A.height / deviceCount;
int lastnrows = A.height - nrows * (deviceCount - 1);
for (dev = 0; dev < deviceCount; ++dev) {
    cudaSetDevice(dev);
    int locnrows = nrows;
    if (dev == deviceCount - 1)
        locnrows = lastnrows;

    int gridrows = locnrows / BLOCK_SIZE;
    int gridcols = B.width / BLOCK_SIZE;
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(gridcols, gridrows, 1);

    d_A.elements = multi_ad[dev];
    d_B.elements = multi_bd[dev];
    d_C.elements = multi_cd[dev];
    d_A.height = d_C.height = locnrows;

    MatMulKernel<<<dimGrid, dimBlock,
    BLOCK_SIZE*BLOCK_SIZE*sizeof(t_matrix)*2>>>(d_A, d_B, d_C);
}
```

Рисунок 3. 3- Асинхронный выбор ядер

Реализация матричного умножения на одном устройстве GPU с использованием ускоренной библиотеки алгебраических вычислений CUBLAS.

Библиотека cuBLAS представляет собой реализацию BLAS (базовых подпрограмм линейной алгебры) поверх среды NVIDIA®CUDA ™. Он позволяет пользователю получать доступ к вычислительным ресурсам графического процессора NVIDIA (GPU).

Начиная с CUDA 6.0, библиотека cuBLAS предоставляет два набора API - обычный API cuBLAS, который называется просто cuBLAS API и API CUBLASXT.

Чтобы использовать API cuBLAS, приложение должно распределять необходимые матрицы и векторы в пространстве памяти GPU, заполнять их данными, извлекать последовательность желаемых функций cuBLAS и затем загружать результаты из памяти GPU обратно на хост. API cuBLAS также предоставляет вспомогательные функции для записи и извлечения данных с графического процессора.

Чтобы использовать API CUBLASXT, приложение должно хранить данные в хосте и библиотеке, следить за отправкой операции на один или несколько графических процессоров, присутствующих в системе, в зависимости от запроса пользователя.

Чтобы произвести умножение матриц, были использованы следующие функции:

- CublasStatus_t cublasSgemm (cublasHandle_t handle, cublasOperation_t transa, cublasOperation_t transb, int m, int n, int k, const float * alpha, const float * A, int lda, const float * B, int ldb, const float * beta, float * C , Int ldc);

- CublasStatus_t cublasDgemm (cublasHandle_t handle, cublasOperation_t transa, cublasOperation_t transb, int m, int n, int k, const double * alpha, const double * A, int lda, const double * B, int ldb, const double * beta, double * C , Int ldc).

Таким образом, реализация матричного умножения на одном устройстве GPU с использованием ускоренной библиотеки алгебраических вычислений CUBLAS

показывает эффективность использования библиотек аппаратно-программной платформе CUDA.

3.2 Описание эксперимента и их анализ полученных данных

3.2.1 Умножение матриц при ленточной схеме разделения данных

Анализ эффективности: обобщенные оценки ускорения и эффективности вычисляются по формулам:

$$S_p = \frac{n^3}{(n^3 \div p)} = p \qquad E_p = \frac{n^3}{p*(n^3 \div p)} = 1.$$

Завершение метода параллельных вычислений обеспечивает идеальные характеристики ускорения и эффективности.

Анализ эффективности (подробные оценки): время выполнения параллельного алгоритма, соответствующее вычислениям процессора, вычисляется по формуле:

$$T_p\ calc\ =\ n^2 \div p\ *\ 2n-1\ *\tau.$$

Время выполнения операций передачи данных может быть получено с помощью модели Хокни:

$$T_p\ comm\ =\ p-1\ *(\alpha + w*n*(n \div p) \div \beta).$$

Предполагается, что все операции передачи данных между процессорами в течение одной итерации могут выполняться параллельно.

Общее время выполнения параллельного алгоритма:

$$T_p = \ n^2 \div p\ *\ 2n-1\ *\tau +\ p-1\ *(\alpha + w*n*(n \div p) \div \beta).$$

Полученные показатели являются низкими, что говорит о быстродействии данного алгоритма. Зачастую на практике получаются относительные показатели, несовпадающие с идеальными с связи с затратами на выполнение операций передачи данных между процессорами, служебных затрат и т.д.

3.2.2 Метод Фокса

Анализ эффективности: обобщенные оценки ускорения и эффективности вычисляется по формуле:

$$S_p = \frac{n^2}{(n^2 \div p)} = p \qquad E_p = \frac{n^2}{p*(n^2 \div p)} = 1.$$

Завершение метода параллельных вычислений обеспечивает идеальные характеристики ускорения и эффективности.

Анализ эффективности (подробные оценки): время выполнения параллельного алгоритма, соответствующее вычислениям процессора, вычисляется по формуле:

$$T_p\ calc\ = q[\ n^2 \div p\ *\ 2n \div q - 1\ +\ n^2 \div p\ ] * \tau.$$

При каждой итерации один из процессоров в каждой строке сетки передает свой блок матрицы остальным процессорам в строке:

$$T_p^1\ comm\ = \log_2 q * (\alpha + w * (n^2 \div p) \div \beta).$$

После матричного блочного умножения каждый процессор передает свои блоки матрицы B его соседу в вертикальном столбце:

$$\mathrm{T}_\mathrm{p}^2\ comm\ = \alpha + w * (n^2 \div p) \div \beta.$$

Общее время выполнения параллельного алгоритма:

$$T_p = q\ \ n^2 \div p\ *\ 2n \div q - 1\ +\ n^2 \div p\ \ * \tau + (\mathrm{q} * \log_2 q + (q-1)) * (\alpha + w * (n^2 \div p) \div \beta).$$

Показатели эффективности в разы возрастают, что дает возможность сделать вывод о высокой эффективности работы алгоритма Фокса по сравнению с алгоритмом последовательного матричного умножения.

3.3 Проведение экспериментов по эффективности

Результаты вычислительных экспериментов представлены в Приложение А. На рисунке 3.4 показана диаграмма, отображающая время умножения матрицы с использованием чисел с плавающей запятой.

На рисунке 3.5 показано время умножения матрицы с использованием чисел типа double.

На рисунке 3.6 показано время ускорения умножения матрицы с использованием чисел с плавающей запятой. Все результаты вычислительных экспериментов представлены в Приложении Б.

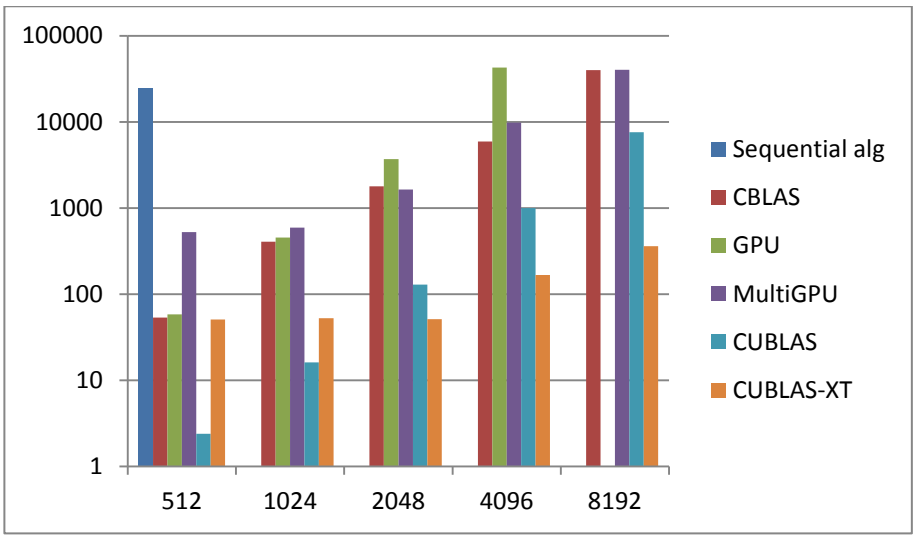

Рисунок 3.4-Время (float)

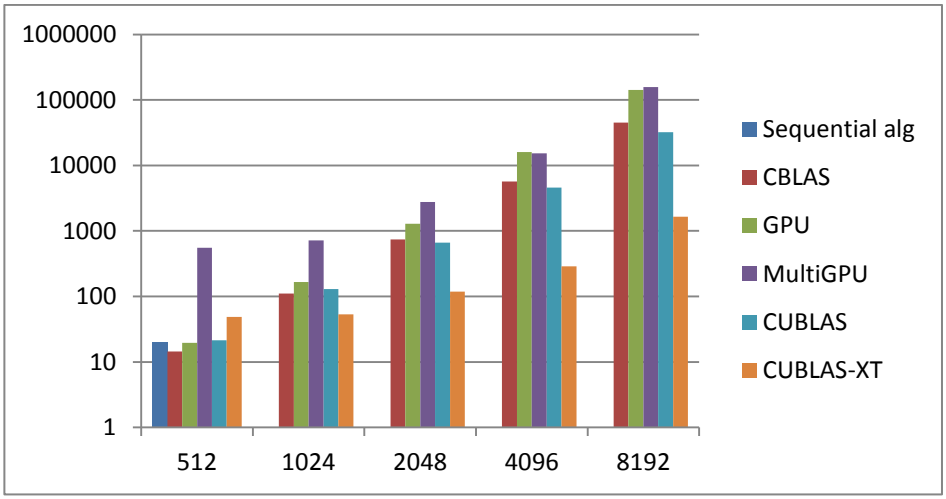

Рисунок 3.5- Время(double)

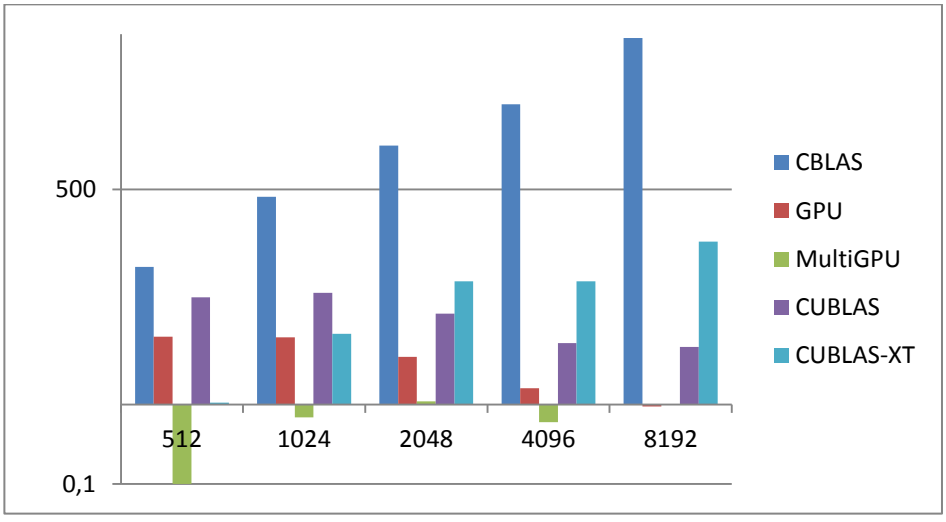

Рисунок 3.6-Ускорение (float)

На рисунке 3.7 показано время ускорения матричного умножения с использованием чисел типа double.

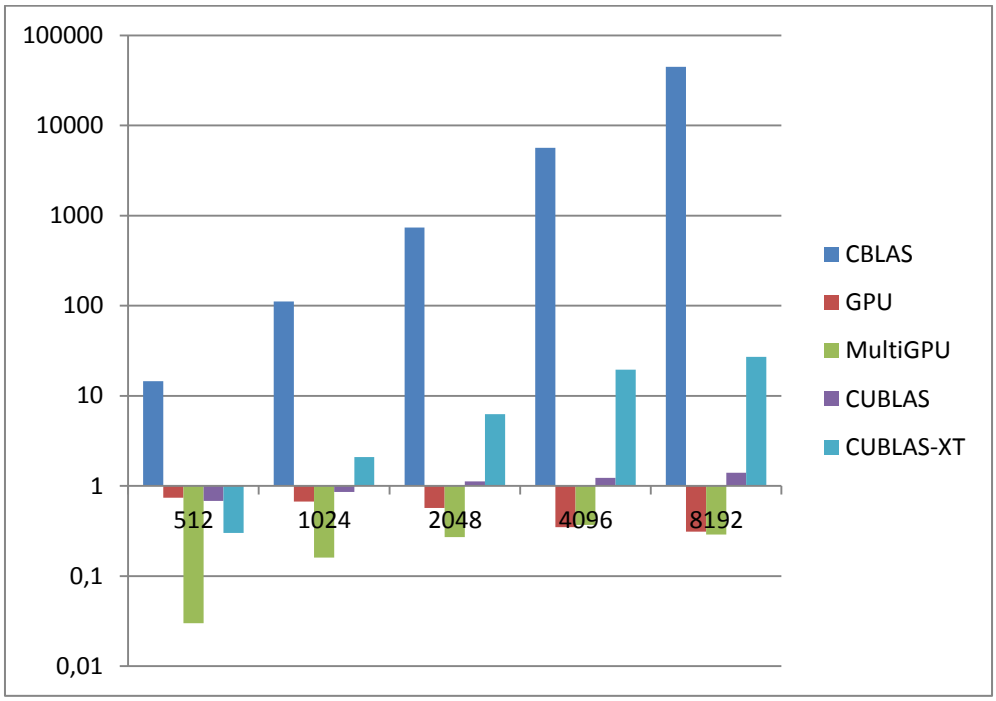

Рисунок 3. 7-Ускорение (double)

Эксперименты проводились на вычислительном узле на базе двух процессоров Intel Xeon E5-2670, 4 Nvidia Tesla K10, 64Gb DDR3, работающих под управлением операционной системы Ubuntu 14.04 LTE. Программы были разработаны в среде программирования NSight Eclipse Edition, CUDA Toolkit v7.5 был использован для компиляции с ее полной оптимизацией.

Время рассчитывается с момента запуска вычислительных ядер, остановка была основана на завершении расчета. Время передачи данных из памяти произвольного доступа для памяти графического процессора и обратно не учитывалось.

Параметры теоретических зависимостей в этой вычислительной системе имеют значения: a - латентность - 44 микросекунды; B - емкость - 39,73 МБ / с, а w - размер, равный 8 байтам.

3.4 Анализ эффективности разработанных алгоритмов

По результатам экспериментов был построен график так, чтобы полностью отражались сильные и слабые стороны библиотек (рис. 3.8).

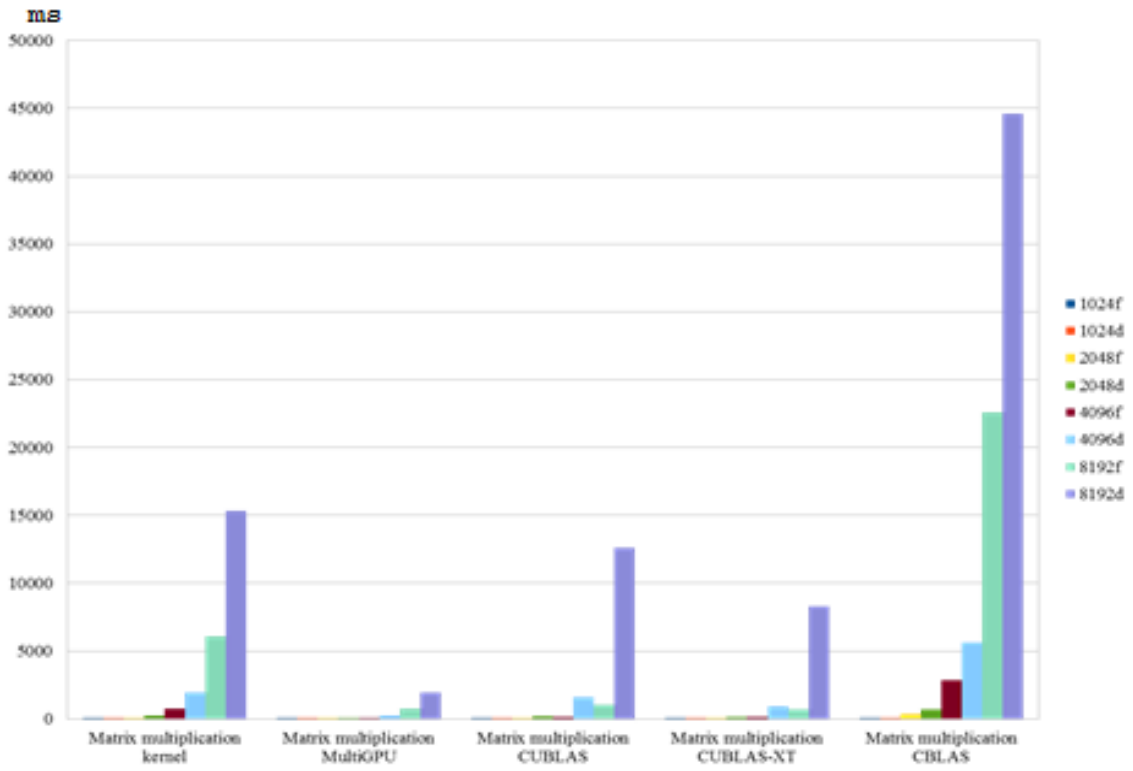

Рисунок 3. 8-Расчет времени

Из рисунка 3.8 видно снижение скорости последовательных алгоритмов Матричного умножения даже с использованием оптимизированной библиотеки CBLAS с увеличением количества элементов, время вычисления быстро увеличивается. Однако, если количества элементов недостаточно, то нет необходимости использовать библиотеки ускорения Cuda. На матрицах размером до 4096x4096 время вычислений существенно не отличается. Однако для матриц размером более 4096x4096 показаны преимущества параллельных алгоритмов и технологии CUDA. Среди анализируемых реализаций кратчайшие расчеты для одиночных и двойных матриц точности - CUBLAS-XT для 8 устройств.

На рисунках 3.9-3.10 показаны результаты матричного умножения при одиночной и двойной точности.

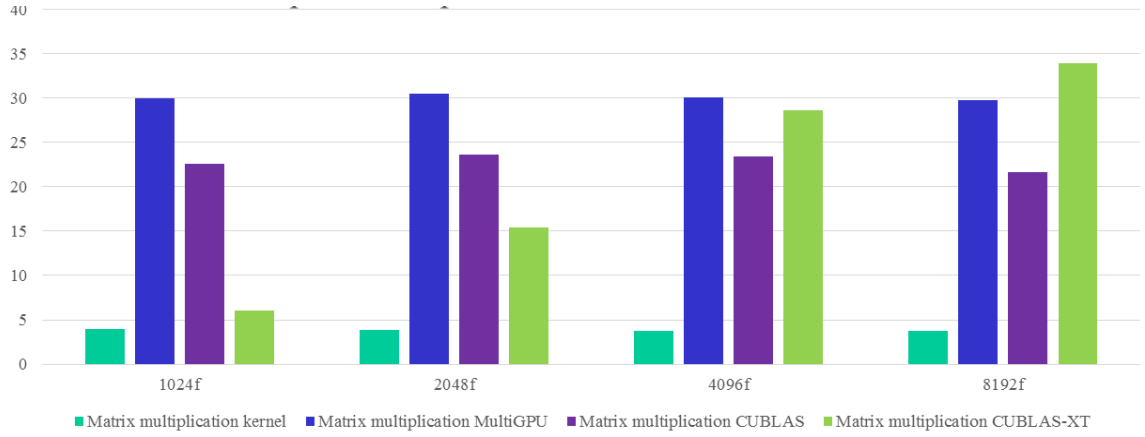

Рисунок 3. 9-Матричное умножение одиночной точности

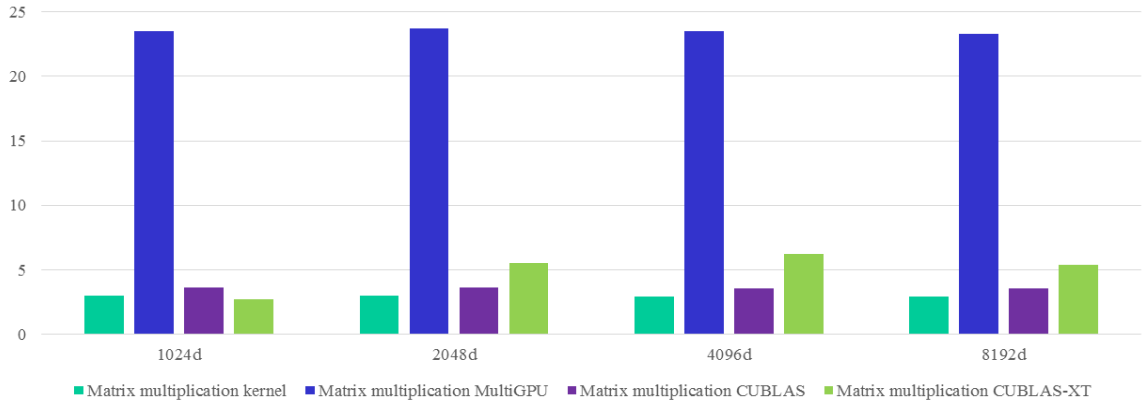

Рисунок 3. 10- Матричное умножение двойной точности

Выполнение одномерного умножения матрицы, а также умножение с двойной точностью привело к выводу, что по сравнению с матричным умножением CBLAS использование GPU выгодно при умножении больших матриц.

ЗАКЛЮЧЕНИЕ

В ходе проведенной работы была изучена архитектура видео ускорителей от NVidia, были изучены и реализованы методы повышения производительности алгоритмов под графические процессоры NVidia Tesla K10, А также проведен сравнительный анализ методов матричного умножения, были представлены параллельные реализации матричного умножения с использованием как общих, так и распределенных подходов к памяти. Матричное умножение является превосходным алгоритмом параллельной обработки, как показали показатели ускорения / эффективности. Чтобы избежать конфликтов памяти при использовании подхода с разделяемой памятью, важно перенести данные на встроенную / локальную память для выполнения. Поскольку межпроцессорная связь требуется только изначально, она не оказывает сильного влияния на производительность подхода с распределенной памятью; Но с использованием методов двойной буферизации, это можно свести к минимуму еще больше. Также необходимо учитывать балансировку нагрузки.

Для хорошо распараллеливаемых вычислений и сложных алгоритмов использование CUDA дает лучший результат.

На примере умножения матриц мы видим эффективность и способы оптимизации реализации параллельных алгоритмов на технологии Nvidia Cuda. Серия экспериментов выявила один небольшой недостаток, заключающийся в том, что передача данных в видеопамять и обратно требует много времени.

Графические процессоры от NVIDIA и программно-аппаратная архитектура параллельных вычислений NVidia CUDA позволяют программистам использовать тысячи CUDA ядер для параллельных вычислений. Для простых вычислений глубокие знание архитектуры GPU не требуются. Однако только реализация программ под конкретную архитектуру конкретного GPU даст максимальную производительность.

СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ

Научная и методическая литература

1. Белоусов И.В. Матрицы и Определители: Учеб. Пособие-Кишинев,2006-101 с.

2. Боресков А.В. Параллельные вычисления на GPU. Архитектура и программная модель CUDA: Учеб.пособие / Боресков А.В. и др. Предисл.: Садовничий В.А. – М.: Издательство Московского университета, 2012. – 336с.

3. Варыгина М.П. Основы программирования в CUDA: Учеб. Пособие/Варыгина М.П. – М.: Краснояр. гос. пед. ун-т. В.П. Астафьева. – Красноярск, 2012. – 138с.

4. Малышкин В.Э. Параллельное программирование мультикомпьютеров: Учеб. Пособие. / Малышкин В.Э., Корнеев В.Д. – М.: Изд-во НГТУ, 2011. – 296с.

5. Сандерс Дж. Технология CUDA в примерах: введение в программирование графических процессоров:Пер. с англ. Слинкина А.А., научный редактор Боресков А.В./ Сандерс Дж., Кэндрот Э. – М.:ДМК Пресс, 2011. – 232с.

6. Умнов А.Е. Аналитическая геометрия и линейная алгебра: Учеб. Пособие/ Умнов А.Е./ – 3е. изд., испр. и доп. – М.: МФТИ, 2011. – 544 с.

Электронные ресурсы

7. GTC 2012: финальные подробности GK110 [Электронный ресурс] – Шиллинг А. [2012]. – Режим доступа: http://www.hardwareluxx.ru/index.php/news/hardware/grafikkarten/22110-gtc-2012-gk110.html

8. NVIDIA представила Tesla K20 и K20X на основе GK110 [Электронный ресурс] Шиллинг А. [2012].– Режим доступа: http://www.hardwareluxx.ru/index.php/news/hardware/grafikkarten/23874-nvidia-tesla-k20-k20x-gk110.html

9. NVIDIA Tesla K10 GPU ускоряет поиски залежей нефти и газа и обработку сигналов и изображений для военных [Электронный ресурс] –

NVidia Corp., [2012]. – Режим доступа: http://www.nvidia.ru/object/nvidia-tesla-k10-gpu-accelerator-20120516-ru.html

10. Казённов А.М. Основы технологии CUDA. / Казённов А.М. // Компьютерные исследования и моделирование [Электронный ресурс]. – Электрон. журн. – 2010. – Т.2 №3. – С.295-308. – Режим доступа: http://crm.ics.org.ru

11. Капорин, О.Ю. Милюкова, Ю.Г. Бартенев. – Электрон.дан. – Режим доступа: http://2013.nscf.ru/TesisAll/Section%205/04_2010_KaporinIE_S5.pdf

12. Фролов А.В. Еще один метод распараллеливания прогонки с использованием ассоциативности операций / Фролов А.В. // Суперкомпьютерные дни в России 2015 [Электронный ресурс]. – Электрон. журн. – 2015. – Режим доступа: http://russianscdays.org/files/pdf/151.pdf

13. Шарый С.П. Курс вычислительных методов [Электронный ресурс] / Шарый С.П. – Электрон.дан. – М.: Новосибирск: НГУ, 2016. – 529. – Режим доступа: http://www.ict.nsc.ru/matmod/files/textbooks/SharyNuMeth.pdf

Литература на иностранном языке

14. CUDA Toolkit 4.1 CURAND Guide [Electronic resource]: [Руководство по CUDA Toolkit 4.1 CURAND]. – Electronic data. – NVidia Corp., [2012]. – Mode of access: https://hpc.oit.uci.edu/nvidia-doc/sdk-cuda-doc/CUDALibraries/doc/CURAND_Library.pdf

15. CUDAProgramming[Electronicresource]:[CUDAпрограммирование]. – Romero. M., Urra. R. [2012]. – Mode of access: http://cuda.ce.rit.edu/cuda_overview/cuda_overview.htm

16. Dynamic Parallelism in CUDA 5.0 [Electronic resource]:[Динамический Параллелизм в CUDA 5.0]. – Electronic data. – NVidia Corp.,[2012].–Mode of access: http://developer.download.nvidia.com/assets/cuda/docs/TechBrief_Dynamic_Parallelism_in_CUDA_v2.pdf

17. Fastest, Most Efficient HPC Architecture [Electronic resource]: [Самая быстрая и самая эффективная архитектура СуперЭВМ]. – NVidia Corp., [2012].

18. Fastest, Most Efficient HPC Architecture Ever build [Electronic resource]: [Самая быстрая и самая эффективная GPU архитектура когда-либо созданная]. –NVidia Corp., [2012]. – Mode of access: http://www.nvidia.com/content/tesla/pdf/NV_DS_TeslaK_Family_May_2012_LR.pdf

19. NVIDIA CUDA Compute Unified Device Architecture Programming Guide. Version 2.0 [Electronic resource]: [Единая вычислительная архитектура устройств. Руководство по программированию. Версия 2.0]. – Electronic data. – NVidia Corp., [2015]. – Mode of access: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

20. OpenCL Best Practices Guide [Electronic resource]: [Руководство по OpenCL]. – Electronic data. – NVidia Corp., [2011]. – Mode of access: http://camlunity.ru/swap/Library/Conflux/OpenCL/NVIDIA_OpenCL_Best_Practices_Guide.pdf

21. Tuning CUDA Applications for Kepler [Electronic resource]: [Настройка CUDA приложений для архитектуры Kepler]. – Electronic data. – NVIDIA Corp., [2015]- Mode of access: http://docs.nvidia.com/cuda/pdf/Kepler_Tuning_Guide.pdf

22. Understanding the CUDA Data Parallel Threading Model [Electronic resource]: [Общие сведения о параллельной модели потоков данных CUDA]. – Electronic data. – Technical News from The Portland Group, [2012]. – Mode of access:https://www.pgroup.com/lit/articles/insider/v2n1a5.htm

ПРИЛОЖЕНИЕ А

Результаты вычислительных экспериментов при ленточной схеме разделения данных

Matrix Size	Texp(msec)											
	Sequential algorithm (without optimization)		CBLAS		GPU (block)		MultiGPU (block)		CUBLAS		CUBLAS-XT	
	float	double	float	double	float	double	Float	double	float	double	float	double
512	24693,81	19,7	53,21	14,51	7,5	19,6	522,64	551,16	2,39	21,39	50,65	48,78
1024	-	-	407,33	111,08	58,45	165,22	591,85	715,42	16,19	129,04	52,62	53,15
2048	-	-	1791,85	739,28	453,57	1293,69	1640,23	2757,52	129,5	659,52	51,04	118,14
4096	-	-	5892,44	5660,05	3690,22	15992,61	9838,17	15240,7	999,49	4598,58	167,17	289,3
8192	-	-	39961,76	44845,47	42535,41	142455,9	40121,78	157197,5	7582,92	32142,44	360,77	1651,08

ПРИЛОЖЕНИЕ Б

Результаты вычислительных экспериментов (ускорение) при ленточной схеме разделения данных

Matrix Size	Time		Speedup							
	CBLAS		GPU (block)		MultiGPU (block)		CUBLAS		CUBLAS-XT	
	float	double	float	double	float	double	float	double	float	double
512	53,21	14,51	7,09	0,74	0,1	0,03	22,27	0,68	1,05	0,3
1024	407,33	111,08	6,97	0,67	0,69	0,16	25,16	0,86	7,74	2,09
2048	1791,85	739,28	3,95	0,57	1,09	0,27	13,84	1,12	35,11	6,26
4096	5892,44	5660,05	1,6	0,35	0,6	0,37	5,9	1,23	35,25	19,56
8192	39961,76	44845,47	0,94	0,31	1	0,29	5,27	1,4	110,77	27,16