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: **Решение классических задач параллельного программирования на технологии CUDA**

Студент А.О. Кведаравичюс _____

Руководитель А.В. Очеповский _____

Допустить к защите
Заведующий кафедрой к.тех.н, доцент, А.В. Очеповский _____

«_____»_____2016г.

Тольятти 2016

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»**

УТВЕРЖДАЮ
Зав.кафедрой«Прикладная
математика и информатика»
_____А.В.Очеповский

«____»_____2016 г.

ЗАДАНИЕ
на выполнение бакалаврской работы

Студент Кведаравичюс Артем Олегович

1. Тема Решение классических задач параллельного программирования на технологии CUDA
2. Срок сдачи студентом законченной выпускной квалификационной работы 24.06.2016
3. Исходные данные к выпускной квалификационной работе: не предъявляются.
4. Содержание выпускной квалификационной работы (перечень подлежащих разработке вопросов, разделов)

Введение
1) Параллельное программирование на GPU
2) Реализация классических задач параллельного программирования на CUDA
3) Анализ эффективности разработанных приложений

Заключение
Список литературы

5. Ориентировочный перечень графического и иллюстративного материала Презентация, графики, таблицы
6. Дата выдачи задания «11» января 2016 г.

Руководитель выпускной квалификационной работы _____ А.В. Очеповский

Задание принял к исполнению _____ А.О. Кведаравичюс

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»**

УТВЕРЖДАЮ
Зав.кафедрой«Прикладная
математика и информатика»
_____А.В.Очеповский

«____»_____2016 г.

КАЛЕНДАРНЫЙ ПЛАН
выполнения бакалаврской работы

Студента Кведаравичюс Артема Олеговича
по теме Решение классических задач параллельного программирования на технологии CUDA

Наименование раздела работы	Плановый срок выполнения раздела	Фактический срок выполнения раздела	Отметка о выполнении	Подпись руководителя
Изучение теоретических основ	20.01.2016	20.01.2016	выполнено	
Написание введения ВКР	01.02.2016	01.02.2016	выполнено	
Изучение математических основ	25.02.2016	25.02.2016	выполнено	
Написание 1 главы ВКР	23.03.2016	23.03.2016	выполнено	
Реализация рабочего кода программы	5.04.2016	5.04.2016	выполнено	
Написание 2 главы ВКР	20.04.2016	20.04.2016	выполнено	
Написание заключения ВКР	8.05.2016	8.05.2016	выполнено	

Подведение итогов, редактирование ВКР	9.05.2016	9.05.2016	выполнено	
Представление ВКР	11.05.2016	11.05.2016	выполнено	
Создание презентационного материала	11.05.2016	11.05.2016	выполнено	
Проверка на наличие заимствований (плагиата) в системе antiplagiat.ru	30.05.2016	30.05.2016	выполнено	
Предварительная защита	6.06.2016 - 11.06.2016	6.06.2016 - 11.06.2016	выполнено	
Сдача на кафедру отзыва научного руководителя и ознакомление с ним	20.06.2016	20.06.2016	выполнено	
Сдача на кафедру комплекта документов для защиты	24.06.2016	24.06.2016	выполнено	
Защита ВКР	29.06.2016	29.06.2016	выполнено	

Руководитель выпускной квалификационной работы _____ А.В. Очеповский

Задание принял к исполнению _____ А.О. Кведаравичюс

Аннотация

Темой выпускной квалификационной работы является разработка классических программ параллельного программирования на технологии CUDA. Данная работа актуальна, так как в настоящий момент большинство компьютеров обладает двумя или более процессорами, несмотря на это уже давно есть GPU, вычислительная способность которых значительно больше CPU. Поэтому компания Nvidia разработала технологию CUDA, которая позволяла реализовывать на упрощенном языке программирования C алгоритмы, выполняемые на графических процессорах GeForce 8 поколения и старше. Предметом исследования в данной работе являются параллельные алгоритмы и программная реализация решения классических задач параллельного программирования. Для этого нужно исследовать эффективность программно-аппаратной архитектуры параллельных вычислений Nvidia CUDA.

В первой главе выпускной квалификационной работы рассказывает об актуальности параллельного программирования с использованием GPU, также проводится анализ механизмов параллельных алгоритмов, описано сравнение GPU с CPU и исследование архитектуры CUDA.

Во второй главе описываются классические задачи параллельного программирования и особенности их реализации.

В третьей главе рассказывается об особенностях оптимизации параллельных программ, а также представлена реализация классической задачи параллельного программирования. На основе полученных данных в результате исследования делается вывод.

Выпускная квалификационная работа выполнена на сорок одной странице, состоит из введения, трёх глав, заключения и списка литературы, состоящего из двадцати одного литературного источника.

Оглавление

Введение

Когда-то параллельное программирование было увлечением программистов, которых интересовало решение задач на больших суперкомпьютерах. С развитием многоядерных процессоров, параллельное программирование становится технологией, которой должен владеть и уметь использовать каждый профессиональный разработчик.

Параллельное программирование возможно кажется сложным, но его легче понять, если считать не "трудным", а просто "немного другим". Оно содержит в себе все черты последовательного программирования, но в параллельном программировании имеются три дополнительных, четко определенных этапа [13].

- Определение параллелизма: анализ поставленной задачи с целью выделить в ней подзадачи, которые можно выполнять одновременно;
- Выявление параллелизма: изменение структуры задачи таким образом, чтобы подзадачи могли эффективно выполняться. Для этого нужно найти зависимости среди подзадач и реализовать исходный код так, чтобы ими можно было эффективно управлять;
- Выражение параллелизма: реализация параллельного алгоритма в исходном коде с помощью системы обозначений параллельного программирования [13].

Выбор определенной системы обозначений параллельного программирования может оказаться нелегкой задачей. У этих систем обозначений крутая кривая обучения, которая потребует огромных временных затрат. Следовательно, не имеет смысла осваивать сразу несколько систем обозначений, чтобы выбрать из них самую подходящую. Программисту всего лишь нужно предоставить возможность быстро попробовать различные системы обозначений и познакомиться с их основными особенностями на достаточном уровне для осознанного выбора определенной системы, для дальнейшего подробного изучения.

В настоящий момент большинство компьютеров обладает двумя или более процессорами, несмотря на это уже давно есть GPU, вычислительная способность которых значительно больше CPU. Поэтому компания Nvidia разработала технологию CUDA, которая позволяла реализовывать на упрощенном языке программирования C алгоритмы, выполняемые на графических процессорах GeForce 8 поколения и старше. Исполняющим устройствам GPU был разрешен доступ к памяти для чтения и записи.

В начале 2000-х GPU предназначалось для вычисления цвета каждого пикселя на экране с помощью программируемых арифметических устройств, пиксельных шейдеров. В общем случае пиксельный шейдер получает координаты точки на экране и дополнительную информацию, и на выходе должен выдать конечный цвет этой точки. Но так как арифметические действия над входными цветами и структурами полностью контролируются программистом, то в качестве «цветов» могли выступать любые данные. Программисту ничто не мешало написать шейдер выполняющий произвольные вычисления. GPU возвращал результат как окончательный «цвет» пикселя, хотя это был результат вычислений над входным данными. Таким образом, путем обмана, программист заставлял GPU проделать вычисления, не имеющие никакого отношения к рендерингу.

Скорость арифметических вычислений на GPU была очень высока, но чтобы использовать GPU в качестве вычислений, нужно было знать OpenGL или DirectX, так как никакого другого способа взаимодействия не было. Это означало, что не только данные следует хранить в виде графических текстур, но и сами вычисления записывать на специальных языках графического программирования. Несмотря на большую вычислительную мощь GPU, это оказалось серьезным препятствие на пути к признанию технологии.

В 2006 году Nvidia объявила о выпуске GPU с поддержкой стандарта DirectX 10, он был построен на архитектуре CUDA. Она включала новые компоненты, предназначенные исключительно для GPU-вычислений и

снимала многие ограничения, которые препятствовали применению графических процессоров для вычислений общего назначения.

Технология CUDA была разработана с целью, упростить работу с GPU и сделать возможным выполнение вычислений общего назначения, а не только традиционных задач компьютерной графики. Поэтому решение классических задач параллельного программирования на технологии CUDA является **актуальной.**

Новизна работы заключается в исследовании технологии CUDA посредством реализации классических задач параллельного программирования.

Объект выпускной квалификационной работы: вычислительный процесс решения классических задач параллельного программирования на CUDA C.

Предмет выпускной квалификационной работы: параллельные алгоритмы и программная реализация решения классических задач параллельного программирования.

Цели выпускной квалификационной работы: исследование эффективности программно-аппаратной архитектуры параллельных вычислений Nvidia CUDA.

Задачи выпускной квалификационной работы:

- разработка программы на языке CUDA C для решения задач параллельного программирования;
- исследование эффективности архитектуры CUDA для решения классических задач.

Выпускная квалификационная работа состоит из трех глав.

В первой главе описаны общие сведения о параллельном программировании, технологии CUDA, архитектуре GPU и ее отличия от CPU.

Во второй главе описываются классические задачи параллельного программирования, и оценивается целесообразность их реализация на CUDA.

В третьей главе производится анализ полученных результатов классической задачи параллельного программирования.

В заключении подводится общая оценка, анализ возникших при разработке трудностей, а также перспективы применения параллельных вычислений на GPU.

Глава 1 Параллельное программирование на GPU

1.1 Актуальность программирования на GPU

Параллельное программирование возникло благодаря техническим изменениям и развитию аппаратного обеспечения. Причиной развития было изобретение аппаратных модулей, названных каналами. Они позволяли выполнять операции ввода-вывода параллельно инструкциям центрального процессора и работали независимо от него. После появления каналов в программировании появилась новая проблема, так как теперь части программы могли выполняться в хаотичном порядке [2].

После изобретения каналов началась разработка многопроцессорных машин. Сейчас почти все компьютеры оснащены несколькими процессорами. Теперь программы могут одновременно выполняться на разных процессорах, это ускоряет работу приложения, если оно написано для многопроцессорной системы. При написании параллельной программы возникают некоторые возможности и трудности. Нужно решить, как разбить задачу на потоки, какого типа и сколько процессов использовать и как они будут взаимодействовать между собой. Это решение зависит от типа приложения, и аппаратного обеспечения на котором будет использоваться программа [18].

Большинство программ пишется для последовательного применения, но для того чтобы достичь более высокую производительность, создаются мультипроцессорные системы. Основа для разработки параллельных программ, использующих параллельные методы решения задач, является понятия процесса и потока. Представление программы в виде набора процессов/потоков, которые выполняются параллельно на нескольких процессорах или на одном процессоре в режиме разделения времени, помогают сосредоточиться на рассмотрении проблемы организации взаимодействия параллельных частей программы. Также позволяют определить моменты и способы обеспечения синхронизации и

взаимоисключения процессов/потоков, изучить условия возникновения или доказать отсутствие тупиков в ходе выполнения программы.

Поток определяет, в какой последовательности будет выполняться код в процессе. Большинство приложений использует единственный, первичный поток. Но если необходимо процессы могут создать дополнительные потоки, которые позволят более эффективно выполнить свою работу. Потоки будут выполняться параллельно до тех пор, пока им не понадобится общаться между собой. Для взаимодействия между потоками нужно уметь планировать и манипулировать ими в любой операционной системе. Специальные средства в ОС обеспечивают синхронизацию и общение между параллельными процессами, устанавливая определенную очередность их выполнения и взаимодействия между собой. Существуют разные механизмы синхронизации различные по способу реализации, степени эффективности и области применения в операционной системе [18].

Каждый процесс – это последовательность операторов и выполняется один за другим. При выполнении параллельной программы все процессы должны взаимодействовать между собой. Взаимодействие осуществляется с применением разделяемых переменных, один процесс производит запись в переменную, а другой считывает. Также существует пересылка сообщений, когда один процесс отправляет сообщение другому.

Существует множество различных видов синхронизации, но можно выделить два основных: условная синхронизация и взаимное исключение. Условная синхронизация останавливает процесс до тех пор, пока не выполнится определенное условие. Взаимное исключение следит за тем, чтобы критические секции операторов не выполнялись одновременно. Например, решение задачи об обедающих философах иллюстрирует реализацию взаимного исключения между процессами, конкурирующими за доступ к пересекающимся множествам разделяемых переменных, а читателей и писателей – реализацию комбинации параллельного и исключительного доступа к разделяемым переменным. Соблюдение правил

синхронизации, взаимоисключения и недопущения взаимоблокировки приводит к построению корректных параллельных программ.

Задача об обедающих философах аналогична реальным проблемам, в которых процессу необходим одновременный доступ более чем к одному ресурсу. Поэтому она часто используется для иллюстрации и сравнения различных механизмов синхронизации.

Современные графические процессорные устройства (GPU, Graphics Processing Unit) являются высокопроизводительными вычислительными системами, которые вполне сравнимы с суперкомпьютерами. По некоторым характеристикам системы, построенные на базе графических процессоров, значительно лучше традиционных суперкомпьютеров, им присущи: более высокая энергоэффективность, лучшее соотношение цена/производительность, меньшие размеры, пониженные требования к инженерной инфраструктуре, более высокая доступность и распространённость. Однако существует главная проблема, которая препятствует массовому внедрению GPU в вычислительную практику, это отсутствие нормальных высокоуровневых средств программирования для подобных архитектур.

1.2 Механизмы параллельного программирования

Главное в разработке параллельной программы, в которой реализуются параллельные методы решения задач, является обозначения процесса и потока. Понятие процесса считается одним из основополагающих в практике и теории параллельного программирования. Многие определения сводятся к пониманию процесса как «некоторая последовательность команд, одинаково конкурируют с другими процессами программы за использование процессора для своего выполнения».

Понятие процесса является чрезвычайно важным для организации вычислений на ЭВМ. Выполнение программы с точки зрения операционной системы представляет собой процессы, которые параллельно выполняются,

взаимодействуют между друг другом и претендуют на использование процессоров вычислительной системы. Однако понятие процесса является немного «тяжеловесным» – создание процесса, переключение процессоров на использование других процессов и выполнение других подобных действий требует достаточно многопроцессорного времени. Кроме того, процессы выполняются в разных адресных пространствах, и организация их взаимодействия требует определенных усилий.

Все выше описанные моменты приводят к необходимости определения потока (thread), так как это более простая альтернатива понятию процесса. Поток (точно так же, как и процесс) можно представить как последовательность команд программы, которая может конкурировать за использование процессора вычислительной системы для своего выполнения. Однако, – в отличие от процесса – потоки одной и той же программы выполняются в общем адресном пространстве и разделяют данные программы. Нужно отметить, что с одной стороны, общие данные значительно упрощают взаимодействие потоков между собой (результат, вычисленный одним потоком, сразу становится доступным всем остальным потокам программы), но, с другой стороны, требуется соблюдение определенных правил применения разделяемых данных.

Процессы и потоки в одинаковой степени используются для организации вычислений. Процессы применяются для представления отдельных программ (заданий для операционной системы) и для формирования многопроцессных программ, выполняемых на вычислительных системах с распределенной памятью. Потоки используются для представления программ как множества частей (потоков), которые могут выполняться независимо друг от друга (параллельно) [3].

Серийный программа может быть не очень хорошей, но это как правило, не столь очевидно, как тот факт, что многие из параллельных процессоров зачастую простаивают [19]. Результаты выполнения параллельных программ зависят от порядка чередования команд, так как

возможно разделение одних и тех же данных между одновременно выполняемыми потоками. Данный случай можно рассматривать как проявление главной проблемы использования разделяемых ресурсов (общих данных, файлов, устройств и т. п.). Для организации разделения ресурсов между несколькими потоками необходимо иметь возможность:

- определить доступность запрашиваемого ресурса (ресурс свободен и доступен для использования, или ресурс уже используется одним из потоков программы и не может быть, доступен дополнительно для какого-либо другого потока);
- выделение доступного ресурса одному из потоков, который запрашивает ресурс для использования;
- приостановка потоков, выдающих запросы на ресурсы, которые в данные заняты другими потоками.

Основное требование к механизмам разделения ресурсов является гарантированное обеспечение использования каждого разделяемого ресурса только одним потоком от момента предоставления ресурса этому потоку до момента освобождения ресурса. Данное требование в литературе обычно называется взаимоисключением потоков. Командные последовательности потоков, в ходе которых поток использует ресурс на условиях взаимоисключения, называется критической секцией потока. С использованием последнего понятия условие взаимоисключения потоков можно сформулировать как требование нахождения в критических секциях по использованию одного и того же разделяемого ресурса не более чем одного потока [3].

Требование взаимоисключения не является единственным к способам организации критических секций; дополнительный перечень необходимых свойств состоит в следующем:

- Отсутствие взаимной блокировки. Потоки (по отдельности или совместно) не должны мешать, другим каким-либо потокам, обращаться к выполнению своих критических секций;
- Эффективность. При наличии нескольких потоков, которые пытаются начать выполнение своих критических секций, выбор единственного потока для продолжения должен происходить за малое (конечное) время;
- Отсутствие бесконечного ожидания. Поток, который пытается начать выполнение своей критической секции, должен гарантированно рано или поздно получить такую возможность.

При разработке способов обеспечения критических секций обычно предполагается также, что относительные скорости выполнения потоков неизвестны и произвольны, а длительность нахождения потоков в своих критических секциях является конечной.

Организация синхронизации потоков также является одной из важных частей разработки параллельной программы. В самом общем виде, синхронизация может быть реализована при помощи задания необходимых логических условий. Очень важно в определении полного набора синхронизирующих действий, обеспечить гарантированное исключение недопустимых моментов параллельной программы [3].

В ходе изучения проблем параллельного программирования и разработки методов их решения сформировался набор некоторых «классических» задач, на которых принято демонстрировать результаты применения новых разрабатываемых подходов. В числе этих задач:

- Задача «Писатели - Читатели» (Readers-Writers problem);
- Задача «Производители – Потребители» (Producers-Consumers problem);
- Задача «Обедающие философы» (Dining Philosopher problem).

Реализация данных задач, с помощью технологии CUDA, поможет наглядно продемонстрировать работу и синхронизацию потоков на видеокартах Nvidia. Соблюдение правил синхронизации, взаимоисключения и недопущения взаимоблокировки приводит к построению корректных параллельных программ.

1.3 Технология CUDA

Технология CUDA – это архитектура параллельных вычислений, разработанная компанией NVIDIA, которая позволяет значительно улучшить вычислительную производительность за счет применения GPU (графических процессоров). Многие разработчики программного обеспечения, ученые и исследователи широко применяют CUDA в различных областях: вычислительная биология и химия, обработка видео и изображений, в медицине. С помощью нее моделируют динамику жидкостей, восстанавливают изображения, которые были получены при помощи компьютерной томографии, проводят сейсмический анализ, трассировку лучшей и многое другое [9].

В отличие от предыдущих поколений графических процессоров, в которых вычислительные ресурсы подразделялись на вершинные и пиксельные шейдеры, в архитектуру CUDA включен унифицированный шейдерный конвейер, позволяющий программе, выполняющей вычисления общего назначения, задействовать любое арифметически-логическое устройство (АЛУ), входящее в микросхему. Поскольку Nvidia рассчитывала, что новое семейство графических процессоров будет использоваться для вычислений общего назначения, то АЛУ были сконструированы с учетом требований IEEE к арифметическим операция над числами с плавающей точкой одинарной точности. Кроме того, был разработан набор команд, ориентированный на вычисления общего назначения, а не только на компьютерную графику. Наконец, исполняющим устройствам GPU был разрешен произвольный доступ к памяти для чтения и записи, а также доступ

к программно-управляемому кэшу, получившему название разделяемая память (shared memory). Все эти средства были добавлены в архитектуру CUDA с целью создать GPU, который отлично справлялся бы с вычислениями общего назначения, а не только с традиционными задачами компьютерной графики.

В настоящий момент происходит эволюция вычислений от обработки данных на центральном процессоре до совместной обработки на CPU и GPU. Для реализации новых вычислительных возможностей компания NVIDIA разработала архитектуру параллельных вычислений CUDA, на данный момент представленную в графических процессорах GeForce, ION, Quadro и Tesla и обеспечивающую необходимую базу разработчикам программного обеспечения.

CUDA обеспечивает сотрудничество ядер между собой. Для запуска многопоточных приложений нет необходимости потоковых вычислений в GPU, так как ядра могут взаимодействовать и обмениваться информацией друг с другом. Многопоточность – это свойство платформы или приложения, при котором, созданный в ОС процесс, может состоять из нескольких потоков, которые будут выполняться параллельно. Смысл многопоточности заключается в многозадачности на уровне одного исполняемого процесса, то есть все потоки выполняются в адресном пространстве процесса и имеют общие дескрипторы файлов. Выполняющий процесс имеет как минимум один главный поток. Преимущества многопоточности заключается в уменьшении временных затрат на создание потока, также за счет распараллеливания вычислений и операций ввода-вывода, повышается производительность.

CUDA хорошо подходит и полезен для высоко параллельных алгоритмов. Если вы хотите увеличить производительность вашего алгоритма во время работы на GPU, то вам нужно иметь много потоков. Обычно большее количество потоков дает более высокую производительность. Для большого количества последовательных

алгоритмов, польза CUDA не велика. Если решение задачи не может быть разбито, по крайней мере, на тысячи нитей, тогда использование CUDA не дает преимуществ. В таком случае мы можем попробовать, преобразовать последовательный алгоритм в параллельный, но этот способ не всегда возможен. Как уже упоминалось выше, чтобы получить наилучшую оптимизацию нужно разделить вашу проблему минимум в тысячу нитей. Тогда производительность алгоритма быстро возрастает [16].

CUDA может быть использована в полной мере при написании на C. Как указывалось ранее, основная идея CUDA, заключается в том, что нужно иметь тысячи нитей, выполняющихся параллельно. Все эти нити будут выполнять одну и ту же функцию (код), известный как ядро. Все эти потоки выполняются с использованием той же инструкции и различных данных. Каждый поток будет знать свой собственный идентификатор, и на основании этих идентификаторов, он будет определять, какие это части данных для работы. Программа CUDA состоит из одной или нескольких фаз, которые выполняются на любом хосте (CPU) или устройстве, таком как GPU [16].

Платформа параллельных вычислений CUDA обеспечивает набор расширений для языков C и C++, позволяющих выражать как параллелизм данных, так и параллелизм задач на уровне мелких и крупных структурных единиц. Программисту предоставлен выбор средства разработки: языки высокого уровня, такие как C, C++, Fortran или же открытые стандарты, такие как директивы OpenACC. По мнению разработчиков, это упростит процесс изучения архитектуры CUDA [6].

Одно из преимуществ заключается в том, что нет необходимости писать всю программу с помощью технологии CUDA. Если вы пишете большое приложение, с пользовательским интерфейсом, а также множеством других функций, и большая часть кода будет написана на C ++ или на любом другом языке на ваш выбор. И если вам понадобится сделать большие математические вычисления, то, вы можете просто написать вызов ядра для функции CUDA. Таким образом, при написании программы, вы можете

использовать GPU только для некоторой части кода, где вам понадобится произвести огромные математические вычисления. Платформа параллельных вычислений CUDA используется на сегодняшний день в тысячах GPU-ускоренных приложений и тысячах опубликованных научных статьях [15].

В области научных исследований нашли широкое применение новой технологии. К примеру, сейчас CUDA ускоряет работу AMBER, программу, которая используется для моделирования молекулярной динамики веществ, используемую более 60000 исследователями в академической среде и многими фармацевтическими компаниями по всему миру, это позволяет значительно сокращать сроки создания лекарственных препаратов [9].

На финансовом рынке компании Numerix и CompatibL объявили о поддержке CUDA в новом приложении анализа риска контрагентов, это позволило увеличить скорость работы в 18 раз. Numerix используется почти 400 финансовыми институтами [9].

Благодаря выпущенным операционным системам Microsoft Windows 7 и Apple Snow Leopard, вычисления на GPU начали обширно использоваться для массовых решений. В этих системах GPU представлена не только графическим процессором, но и универсальным процессором для параллельных вычислений общего назначения, которые работают с любым приложением.

1.4 Сравнение GPU и CPU в параллельных вычислениях.

Основное преимущество GPU перед CPU заключаются в том, что GPU проектируется для быстрого выполнения большого числа параллельно выполняемых потоков команд. В то время как, ядра CPU оптимизированы для выполнения одного потока последовательных команд, алгоритмов с максимальной производительностью. Центральные процессоры оптимизированы для достижения высокой производительности одного потока команд, который обрабатывает и целые числа и числа с плавающей точкой, при этом доступ к памяти случайный. Начиная с процессоров Intel

Pentium, разработчики центральных процессоров стараются добиваться выполнения набольшего числа команд параллельно, чтобы увеличить производительность. К сожалению, у параллельного выполнения последовательного потока команд есть определённые ограничения и увеличением количества исполнительных блоков, добиться кратного увеличения скорости не получится.

У видеочипов работа распараллелена изначально, на входе он принимает несколько полигонов, выполняет необходимые операции, и на выходе получаем пиксели. Обработка полигонов и пикселей не связана между собой, и их можно обрабатывать отдельно друг от друга, то есть параллельно. В отличие от последовательного потока команд CPU, в GPU используется большое количество исполнительных блоков, которые легко загрузить. Кроме того, современные GPU могут исполнять более одной команды за такт. Например, архитектура Nvidia Tesla в некоторых условиях одновременно запускает на исполнение операции MAD+SFU или MAD+MUL.

Разница между GPU и CPU есть также в принципах доступа к памяти. В GPU он легко предсказуемый и связанный, то есть если из памяти считывается пиксель текстуры, то спустя некоторое время будут прочитаны соседние пиксели. А при записи пиксель записывается во фрейм буфер, и спустя несколько тактов будет записывать в расположенный рядом с ним. Поэтому организация памяти различна от той, что применяется в CPU. Видеочипу, в отличие от центрального процессора, не нужна кэш-память большого размера, в для текстур требуется лишь несколько килобайт (до 128-256 в нынешних GPU). Также, не все CPU обладают встроенными контроллерами памяти, а у всех GPU существует по несколько контроллеров, вплоть до восьми 64-битных каналов в чипе Nvidia. Также на GPU используется более быстрая память, и в результате видеочипам доступна в разы большая пропускная способность памяти, что очень важно для параллельных вычислений, оперирующих с огромными потоками данных.

В современных процессорах большое количество транзисторов и огромные объемы чиповой кэш-памяти. Все это нужно для ускорения выполнения немногочисленного потока команд. Видеочипы используют транзисторы на массивы исполнительных блоков, управляющие потоками блоки, разделяемую память небольшого объёма и контроллеры памяти на несколько каналов. Все это не ускоряет выполнение отдельных потоков, но это позволяет чипу выполнять обработку несколько тысяч потоков.

Центральные процессоры для увеличения производительности за счет уменьшения времени доступа к данным используют кэш-память, а также предсказания ветвлений кода. Эти аппаратные части занимают большую площадь на чипе и потребляют много энергии. Для того чтобы увеличить пропускную способность полосы GPU использует кэш или общую память. Также обходит проблему ожидания доступа к памяти при помощи одновременного исполнения тысячи потоков – видеочип может выполнять вычисления потока без ожидания и задержек, пока другой поток ожидает данные из памяти.

Есть множество отличий в поддержке многопоточности. Центральный процессор на одно процессорное ядро выполняет всего лишь 1-2 потока вычислений, а видеочипы могут поддерживать до 1024 потоков на каждый мультипроцессор, которых в чипе несколько штук. Также если переключение с одного потока на другой для CPU стоит сотни тактов, то в GPU переключение нескольких потоков происходит за один такт.

Можно сказать, что видеочипы были предназначены для параллельных вычислений с большим количеством арифметических алгоритмов, в отличие от современных CPU. Большее число транзисторов GPU работает по прямому назначения, для обработки массивов данных, а не управляет исполнением последовательных вычислительных потоков. Наглядная демонстрация архитектуры CPU и GPU представлена на рисунке 1.1 [12].

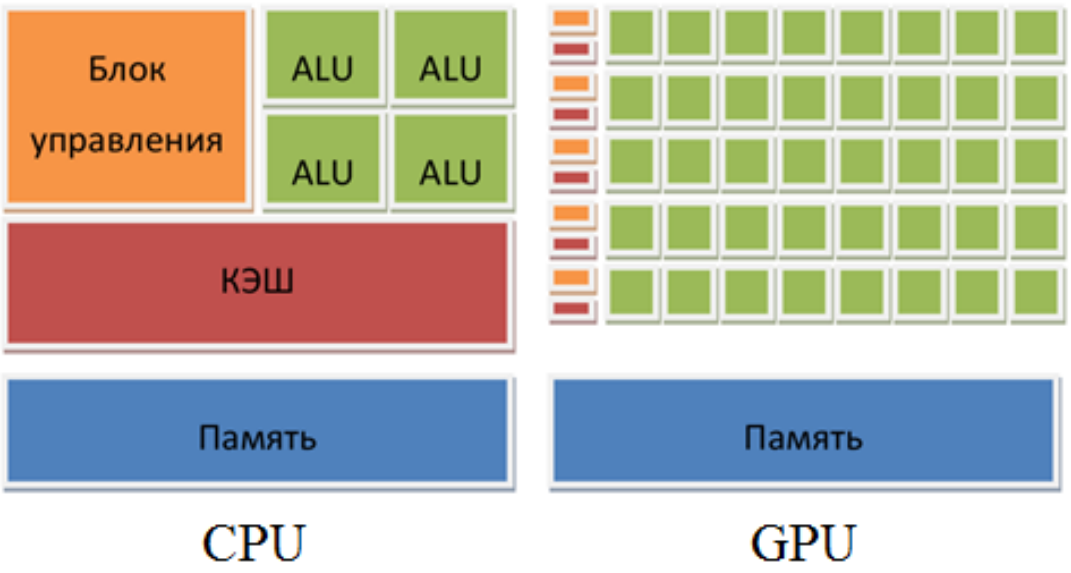

Рисунок 1.1 - Архитектура CPU и GPU

Компьютерная индустрия стоит на пороге революции, связанной с массовым переходом к параллельным вычислениям. С помощью параллельных алгоритмов и развитию архитектуры центральных процессоров можно значительно оптимизировать выполнение поставленных задач. Однако в настоящее время активно развиваются вычисления на GPU, которые при определенных условиях имеют показатели производительности гораздо выше, чем у CPU.

1.5 Описание вычислителя на GPU лаборатории распределенных вычислений ТГУ

В высшем учебном заведении города Тольятти, Тольяттинском Государственном университете (ТГУ), в 2015 году был представлен графический суперкомпьютер, который призван обеспечивать массовое внедрение технологий параллельного и распределенного программирования для ускорения вычислений в решении различных задач.

В основе графического суперкомпьютера ТГУ лежит супер сервер SuperMicro SYS-7047GR-TPRF, с двумя восьмиядерными процессорами Intel Xeon Processor E5-2670 достигающие максимальную частоту 3.3 Гц в режиме Turbo Boost. Установлено 64 Гб оперативной памяти

третьего поколения. За вычисления отвечают четыре GPU ускорителя Nvidia Tesla K10, имея в совокупности 12 288 ядер графических процессоров фирмы Nvidia, обладающие пропускной способностью 320 Гб/сек и обеспечивающие производительность в 18.32 TFlops в операциях с одинарной точностью.

GPU NVIDIA Tesla – это массивно параллельные ускорители, основанные на платформе параллельных вычислений NVIDIA CUDA. Графические процессоры Tesla созданы с нуля для экономичных, высокопроизводительных вычислений, вычислительной науки и супервычислений, обеспечивая намного более высокую скорость работы широкого круга научных и коммерческих приложений по сравнению с системой на базе CPU. Ярким преимуществом GPU Tesla K10 является превосходство его двух ключевых параметров, которые серьезно влияют на общую работу приложения: скорость операций с плавающей точкой и полоса пропускания памяти. Вместе эти параметры позволяют K10 обеспечивать заметный прирост производительности в ведущих научных, инженерных и коммерческих приложениях с минимальными усилиями со стороны разработчика.

GPU NVIDIA Tesla K10 – это вычислительный ускоритель, созданный для решения самых сложных в мире HPC задач. Архитектура Kepler создана специально для высокой производительности и рекордно низкого энергопотребления, она втрое более экономична, чем предшественница NVIDIA Fermi, создавшая новый стандарт для параллельных вычислений несколько лет назад.

Графический процессор Nvidia состоит из набора независимых мультипроцессоров, которые состоят из нескольких ядер CUDA, нескольких модулей математический функций, конвейера, разделяемой памяти и кэша. Каждый поток выполняется на отдельном CUDA- ядре, и использует собственную локальную память и набор инструкций.

Отдельные потоки образуют одинаковые по размеру блоки потоков, которые выполняются на отдельном мультипроцессоре. Внутри блока потоки

могут взаимодействовать друг с другом при помощи синхронизации, атомарных операций, глобальной памяти и общих данных в разделяемой памяти. Потоки блока группируются в варпы, в котором выполняют одинаковые инструкции параллельно. Потоки выполняют одну и ту же команду, но каждый со своими данными, а если внутри варпа из-за оператора if происходит ветвление, то все потоки варпа выполняют появившиеся после этой ветви. Ветвление может плохо отразиться на производительности, потому что различные потоки не могут выполняться параллельно [17].

Блоки потоков объединены в решетки блоков потоков, где у каждого блока есть свои координаты, также как и у потока в блоке потоков. Пример иерархии потоков показан на рисунке 1.2 [11].

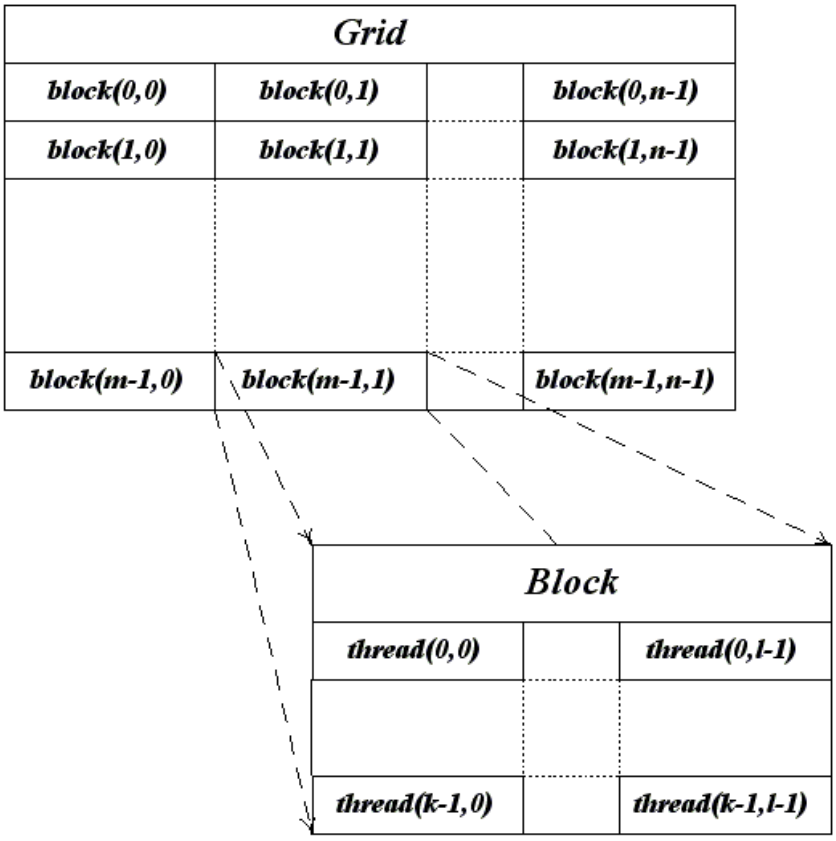

Рисунок 1.2 - Иерархия потоков

Кроме иерархии потоков также существуют несколько типов памяти, от которой зависит скорость работы приложения. Поэтому большую часть кристалла CPU занимает кэш, предназначенный для ускорения работы с

памятью. Для GPU существует несколько типов памяти, которые имеют различию между собой, наглядно показаны в таблице 1.1.

Таблица 1.1 - Типы памяти в GPU

Тип памяти	Чтение/Запись	Расположение	Скорость работы
Register (регистровая)	Чтение/Запись	мультипроцессор	высокая
Local (локальная)	Чтение/Запись	DRAM GPU	низкая
Shared (разделяемая)	Чтение/Запись	мультипроцессор	высокая
Global (глобальная)	Чтение/Запись	DRAM GPU	низкая
Constant (константная)	Чтение	DRAM GPU	высокая
Texture (текстурная)	Чтение	DRAM GPU	высокая

Регистровая память (Register memory) – это одна из самых быстрых типов памяти, потому что расположена в потоковом мультипроцессоре, доступна для чтения и записи одной нити. У каждой нити есть свой набор регистров, который не доступен соседним нитям, даже когда они не выполняются. Переменные, которые создаются в процессе выполнения автоматически и помещаются в регистры. Если в процессе выполнения будет создано большое количество локальных переменных, то это приведет к нехватке регистров доступных на одну нить. В таком случае часть переменных будет сохранена в локальной памяти.

Локальная память (Local memory) – это маленький объем памяти, доступ к которому имеет только один потоковый процессор, и значительно медленнее регистровой памяти. Каждая нить имеет свою область локальной

памяти доступной для чтения и записи. У пользователя нету доступа к локальной память, вся работа происходит на уровне системы.

Разделяемая память (Shared memory) – это 16-килобайтный блок памяти, который располагается на мультипроцессоре с общим доступом для всех потоков. Это самая быстрая память после регистровой. Доступна для чтения и записи нитями одного блока, с помощью нее можно обмениваться данными внутри блока. Обеспечивает взаимодействие потоков, управляется программистом напрямую, он может указать какие данные должны быть помещены в разделяемую память. Использование данного типа памяти один из ключевых методов оптимизации программы.

Глобальная память (Global memory) – самая большая по размеру память, доступная для мультипроцессоров на видеочипе, объем памяти на Tesla K10 составляет 8192Мб. Обладает высокой пропускной способностью, около 100 Гб/сек, однако скорость записи на глобальную память низкая. Поэтому для быстродействия приложения, желательно свести к минимуму работу с глобальной памятью.

Константная память (Constant memory) – область памяти доступная всем мультипроцессором только для чтения, расположена в глобальной памяти. Ее размер составляет 64 килобайта, кэшируется по 8 килобайт на каждый мультипроцессор.

Текстурная память (Texture memory) – область памяти доступная только для чтения. Обладает дополнительным интерфейсом обработки хранящейся в ней данных. Такая же медленная, как и глобальная память.

Правильное использование различных типов памяти, поможет оптимизировать скорость вычислений. Конечно же, написание оптимального кода для вычислений на GPU непростая задача, но выпущенная компанией Nvidia платформа CUDA со своим компилятором и библиотеками для вычислений на GPU, дает возможности и больший контроль программисту над аппаратными возможностями GPU.

Глава 2 Реализация классических задач параллельного программирования на CUDA

2.1 Описание задачи «Производители – Потребители»

Постановка данной задачи состоит в следующем, предполагается, что существует два потока, поток-производитель, который генерирует сообщение, и второй поток-потребитель, который принимает сгенерированное сообщение для дальнейшей обработки. Взаимодействие потоков происходит через некоторую область памяти (хранилище), в которой производитель размещает свое сообщение и из которой это сообщение извлекается потребителем.

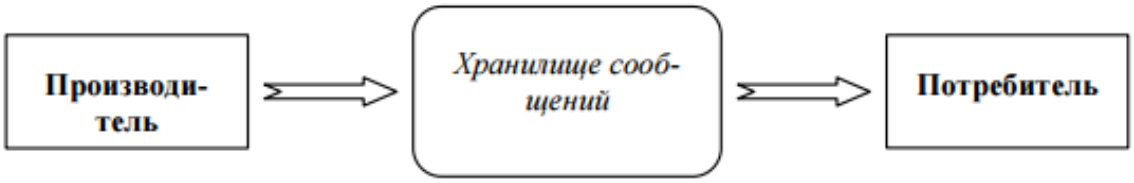

Рисунок 2.1 - Общая схема задачи "Производители - Потребители"

Рассмотрев более подробно данную задачу, можно заметить, что хранилище сообщений представляет собой общий разделяемый ресурс, и доступ к нему должен осуществляться по правилам взаимного исключения. Также при реализации нужно учесть, что если в области памяти выделенной под хранилище отсутствуют сообщения, то поток - потребитель ничего считать не может, а в случае, когда хранилище заполнено, поток – производитель не сможет оставить свое сообщение. Но данную задачу реализовывать на технологии CUDA нецелесообразно, так как в ней участвуют всего два потока, и продемонстрировать работу графического процессора в полном объеме не получится, потому что потоков должно быть гораздо больше. Поэтому мы рассмотрим задачу параллельного программирования, которая подразумевает работу большого количества потоков.

2.2 Описание задачи «Читатели – Писатели»

Главное отличие задачи «Читатели – Писатели» от «Производители – Потребители» в том, что здесь происходит работа большого количества потоков. Постановка данной задачи состоит в следующем, пусть у нас есть некоторая область памяти, с которым могут одновременно работать несколько потоков. Эти потоки можно разделить на две группы. Первая группа – это потоки – читатели, которым доступно чтение из области памяти. Вторая – это потоки - писатели, которые создают новые записи в хранилище данных (см. рис. 2.2).

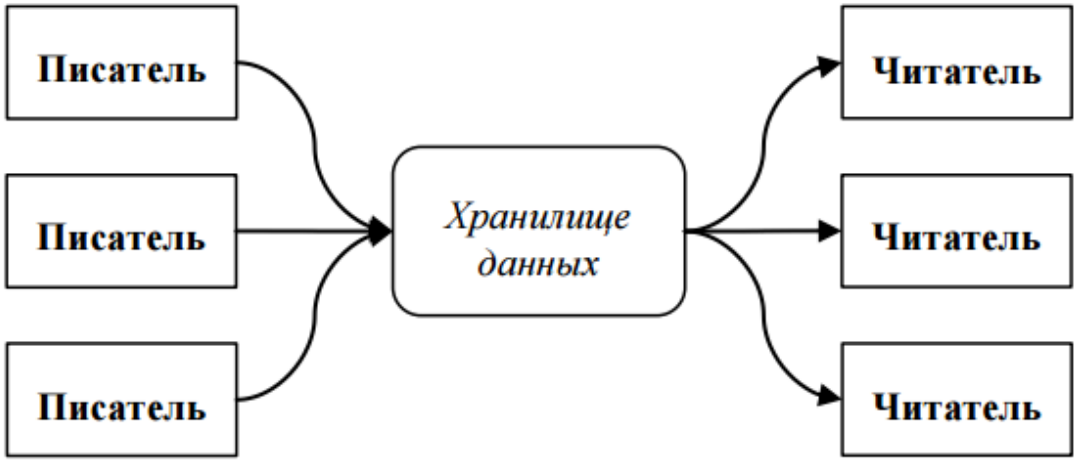

Рисунок 2.2 – Схема задачи «Читатели – Писатели»

Постановка задачи может различаться в правилах разрешения ситуации обращения потока-писателя к хранилищу. Например, в случае при попытке записи уже существуют активные потоки-читатели, то поток-писатель придется заблокировать до завершения работы потоков-читателей. Но что делать в том случае, если поступают новые запросы на чтение. Один из способов решения этой проблемы, это отдать предпочтение потокам-читателям, чтобы они могли начинать свою работу, не обращая внимания на блокированный процесс-писатель, но в этом случае блокировка потока-писателя может продолжаться бесконечно долго. Поэтому в реализации нашей задачи будет реализована поочередная блокировка новых потоков, все четные потоки будут потоки-читатели, а нечетные потоки-писатели. Потоки

по очереди будут получать доступ к хранилищу и оставлять или считывать записи в нем.

2.3 Реализация задачи «Читатели – Писатели»

Реализацию классической задачи параллельного программирования «Читатели - Писатели», будем осуществлять через мьютекс. Мьютекс – это простейший двоичный семафор, который может находиться в одном из двух состояний (закрытом, открытом). Их основная задача – организация взаимного исключения для потоков из одного или разных процессов. Когда потоку нужен доступ к критической области он вызывает метод lock. Если мьютекс находится в незаблокированном состоянии, вызов проходит успешно и поток который вызвал метод может спокойно войти в критическую область. Подразумевается, что в конкретный момент только один поток может быть защищен мьютексом. В случае когда мьютекс уже заблокирован, поток вызывающий метод lock будет ждать до тех пор, пока поток, защищенный мьютексом завершит свою работу и вызовет метод unlock [8].

К сожалению, на технологии CUDA не предусмотрена реализация мьютекса, поэтому придется реализовать структуру осуществляющую блокировку потока по типу мьютекс (рисунок 2.3). Для этого в технологии CUDA предусмотрены атомарные функции, нам понадобятся всего две из них atomicCas и AtomicExch [20].

```
struct Lock{
    int *mutex;
    Lock(void){
        int state = 0;
        cudaMalloc((void**)&mutex, sizeof(int));
        cudaMemcpy(mutex, &state, sizeof(int), cudaMemcpyHostToDevice);
    }
    ~Lock(void){
        cudaFree(mutex);
    }
    __device__ void lock(void){
        while (atomicCAS(mutex, 0, 1) != 0);
    }
    __device__ void unlock(void){
        atomicExch(mutex, 0);
    }
};
```

Рисунок 2.3 - Реализация структуры Lock

Функция AtomicExch выполняет атомарный обмен значениями – передаваемое значение записывается по указанному адресу, а предыдущее значение возвращается. При этом подобный обмен происходит как одна транзакция, то есть ни один поток не может вклиниться между шагами этого обмена [4].

Функция AtomicCas читает значение по переданному адресу и сравнивает его с параметром compare. В случае совпадения по переданному адресу записывается значение параметра value, иначе значение по адресу не изменяется. Возвращается всегда старое прочитанное значение [4].

Результат работы структуры представляет собой поочередный доступ потоков к разделяемому ресурсу. Блокировка потоков происходит в методе theKernel (рисунок 2.4). Если поток четный и записей в ресурсе больше нуля, то происходит считывание записи из ресурса. В случае если поток нечетный и записей в ресурсе не больше двадцати, то мы создаем новую запись в ресурсе.

```
__global__ void theKernel(Lock myLock){
    int index = blockIdx.x; //using only one thread per block

    myLock.lock();
    if ((index % 2 == 0) && note>0){
        note--;
        printf("Reader number = %i read a note.\n", index);
        printf("Notes in book: %i\n", note);
    }

    if ((index % 2 != 0) && note<10){
        note++;
        printf("Writer number = %i make a note. \n", index);
        printf("Notes in book: %i\n", note);
    }

    myLock.unlock();

}
```

Рисунок 2.4 - Реализация метода theKernel

2.4 Описание задачи «Обедающие философы»

Данная задача считается одной из самых популярных в области параллельного программирования. Если с помощью задачи «Читатели–Писатели» демонстрируются методы исключительного и параллельного доступа к одному общему ресурсу, то в задаче «Обедающие философы» рассматриваются способы доступа нескольких потоков(философы) к нескольким разделяемым ресурсам(вилки). Формулировка задачи предложенной Э. Дейкстрой, выглядит следующим образом.

Пять философов сидят за круглым столом. Они чередуют размышления и приемы пищи. В центре стола стоит большая тарелка со спагетти. Спагетти запутанные и длинные, их очень неудобно кушать, и потому каждому философу для удобства нужно использовать две вилки. К сожалению, у философов есть всего лишь пять вилок. Между каждым философом лежит одна вилка, поэтому они решили, что каждый будет использовать только те вилки, которые лежат рядом с ним (слева и справа), а те, у кого не будет двух вилок, продолжат размышления, пока их соседи кушают. Задача состоит в

написании программы, которая моделирует поведение философов. В ней следует учесть невозможность ситуации, в которой все философы хотят кушать, но никто из них не может взять две вилки – например, у каждого из них по одной вилке в руке, и никто не хочет отдавать ее [3].

Задача проиллюстрирована на рисунке 4. Понятно, что двое сидящих рядом философов одновременно приступить к еде не могут. Кроме того, так как вилок всего лишь пять, одновременно могут кушать не более двух философов.

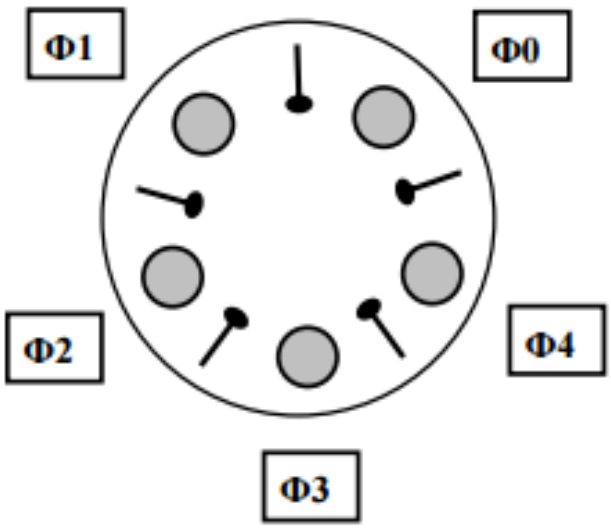

Рисунок 2.5 - Общая схема задачи «Обедающие философы»

Предположим, что периоды раздумий и приемов пищи различны – для их имитации в программе можно использовать генератор случайных чисел. Проимитируем поведение философов следующим образом.

```
for (int i = 0; i < 5; i++){
while (true) { поразмыслить; взять вилки;  покушать; положить вилки}
}
```

Для решения задачи нужно запрограммировать операции «взять вилки» и «отдать вилки». Вилки являются разделяемым ресурсом, поэтому сосредоточимся на их взятии и освобождении.

Каждая вилка похожа на блокировку критической секции: в любой момент времени владеть ею может только один философ. Следовательно,

вилки можно представить массивом семафоров, инициализированных значением 1. Семафор — это объект, ограничивающий количество потоков, которые могут войти в заданный участок кода. Семафоры используются для синхронизации и защиты передачи данных через разделяемую память, а также для синхронизации работы процессов и потоков.

Процессы, по существу, идентичны, поэтому естественно предполагать, что они выполняют одинаковые действия. Например, каждый процесс может сначала взять левую вилку, потом правую. Однако это может привести к взаимной блокировке процессов. Например, если все философы возьмут свои левые вилки, то они навсегда останутся в ожидании возможности, взять правую вилку.

Необходимое условие взаимной блокировки – возможность кругового ожидания, т.е. когда один процесс ждет ресурс, занятый вторым процессом, который ждет ресурс, занятый третьим, и так далее до некоторого процесса, ожидающего ресурс, занятый первым процессом. Таким образом, чтобы избежать взаимной блокировки, достаточно обеспечить невозможность возникновения кругового ожидания. Для этого можно заставить один из процессов, сначала взять правую вилку [7].

К сожалению, реализация этой задачи на технологии CUDA нецелесообразна, так как задаче «Обедающие философы» за доступ к ресурсу борются всего лишь пять потоков-философов, а для демонстрации преимущества выполнения задачи на CUDA, потоков должно быть значительно больше.

Глава 3 Анализ эффективности разработанных приложений

3.1 Методы оценки эффективности работы программы

Сбор характеристик программы с целью их дальнейшей оптимизации, называется профилирование. Его необходимо выполнять, чтобы понять, насколько эффективно работает программа, какие шаги и на каких этапах программы стоит выполнить для дальнейшей оптимизации.

К интересующим разработчика характеристикам относится время выполнения отдельных участков программы, число верно предсказанных условных переходов, число кэш промахов и т.д. К сожалению, у ручного способа профилирования есть некоторые недостатки:

- Вставка дополнительного кода в программу, в результате чего могут возникнуть новые ошибки, и возможно потребуется чистка кода;
- Анализ результата в текстовом виде;
- Большой объем выходных файлов;
- Отсутствие дерева вызовов.

Поэтому для упрощения анализа был разработан профилировщик – инструмент анализа производительности программы, который производит сбор информации о поведение программы во время ее выполнения. Существует много разных средств профилирования, и для сбора информации о работе программы они могут использовать разные подходы.

Наиболее распространенный подход является семплирование. Это подход, при котором через определенные промежутки времени профилировщик смотрит, где находится программа и фиксирует это положение, отождествляя его с исходным кодом.

В случае с многопоточными приложениями ситуация совершенно другая, потому что появляются новые аспекты производительности, связанные с взаимодействиями потоков, затратами на их создание и синхронизацию. Ускорение отдельного участка кода может не дать никакого

увеличения производительности, если разработчик построил свое приложение таким образом, что большую часть времени, потом ждут когда освободится разделяемый ресурс. Также в многопоточных приложениях существуют такие распространенные проблемы как неравномерное распределение нагрузки и неэффективное использование примитивов синхронизации. Данные факторы могут привести к плохой производительности приложения вплоть до того, что многопоточная версия приложения будет медленнее последовательной. Для анализа многопоточных приложений необходимы специальные инструменты, ориентированные на обнаружение проблемных ситуаций и последующей помощи программистам в их разрешении. В нашем случае, в наборе CUDA Toolkit предусмотрено приложение Visual Profiler, с помощью которого осуществляется сбор характеристик многопоточной программы [14].

Профайлер при запуске требует указать приложение (путь до exe-файла) и выбрать счетчики, которые нужно отслеживать. Каждый счетчик показывает, как часто произошло какое-то событие. После этого программа запускается, и профайлер регистрирует значения счетчиков по ходу ее выполнения для каждого ядра в отдельности. Когда выполнение программы завершено, профайлер составляет таблицу на каждый запуск каждого из ядер, в которой можно посмотреть значения счетчиков и продумать возможные оптимизации. Кроме того, счетчики регистрируются только на одном потоковом мультипроцессоре на GPU, так что некоторые значения могут не совпадать с общим числом запущенных варпов, и необходимо запускать столько потоков, чтобы гарантированно всем мультипроцессорам досталось достаточно работы.

Несмотря на наличие профайлера, входящего в состав CUDA Toolkit, важно иметь возможность точно замерять время, затраченное GPU на выполнение обычных операций. Для этого рассмотрим метод CUDA events. Эта некоторая альтернатива стандартных таймеров в C, C++ была создана, потому что применение стандартных библиотек time замедляет работу GPU.

При анализе эффективности очень важно точно замерить время выполнения различных операций на GPU. Для этого нам и понадобятся CUDA события(events). Соответствующие функции позволяют создавать и уничтожать CUDA события, с их помощью обозначается начало и конец анализируемого кода, и в результате мы получаем время в миллисекундах, прошедшее между двумя событиями. Каждое событие, привязанное к точке, характеризуется тем, пройдена GPU данная точка или нет.

3.2 Основные способы оптимизации вычислений на CUDA

Оптимизация производительности, как правило, сводится к следующим шагам:

- максимальное использование параллелизма задачи;
- оптимизация доступа в память;
- оптимизация математики.

Для достижения наилучшей производительности, нужно разбить задачу на такие подзадачи, чтобы полностью использовать параллелизм по данным. В тех участках алгоритма, где параллелизм нарушается (например, потокам необходимо синхронизироваться для разделения данных между собой), возможны два случая:

1. если потоки принадлежат одному блоку, то они могут использовать разделяемую память для эффективного обмена данных и __syncthreads() для синхронизации внутри ядра;
2. если потоки принадлежат разным блокам, то эффективнее использовать два разных ядра с промежуточной записью в глобальную память.

При условии, что алгоритм достаточно распараллелен, можно продолжить его дальнейшую оптимизацию. Для этого существуют несколько основных способов, с помощью которых можно оптимизировать работу приложения на CUDA.

1. Необходимо свести к минимуму обмен данными между памятью GPU и CPU, даже если придется запускать участки кода на GPU, которые не приведут к ускорению по сравнению с CPU. Даже если вычисление на CPU происходило бы быстрее, требовалось бы время для обмена данными.

2. Исходные данные для расчета загружаются в память GPU единожды перед началом моделирования, а обмен данным между CPU и GPU происходит только в том случае, когда нужно их сохранить.

3. Вычисления всех уравнений происходят параллельно, для вычисления одного уравнения, создается один поток. Размер сетки и блока подбирается так, чтобы скорость расчета была максимальной. Число потоков в блоке приблизительно должно равняться числу ядер в GPU.

4. Все загружаемые данные в память GPU хранятся в глобальной памяти, но GPU также обладает более быстрыми типами памяти (разделяемой, константной), однако применение данных типов памяти дают незначительное преимущество [5].

В некоторых случаях, когда много раз выполняются одинаковые вычисления, например a*b, создается переменная c=a*b, которая в дальнейшем будет использоваться. Следует учитывать, что некоторые математические функции порождают использование локальной памяти (в частности, нужно быть внимательным с sinf(), cosf() и другими функциями). Стоит избегать хаотичного обращения в разделяемую память. При работе с константной памятью, важно знать, что ее максимальная производительность достигается при обращении всеми потоками варпа по одному адресу. Таким образом, можно получить незначительный прирост производительности.

Результаты применения различных способов оптимизации могут объясняться спецификой моделируемой задачи. Поэтому стоит учитывать, что небольшая экономия времени за одно моделирование дает большую экономию в итоге.

3.3 Описание эксперимента и анализ полученных данных

В результате выполнение выпускной квалификационной работы, на технологии CUDA была реализована классическая задача параллельного программирования «Читатели – писатели» рисунок 3.1. Данная программа были запущена и протестирована на отладочной станции и суперкомпьютере Тольяттинского Государственного Университета.

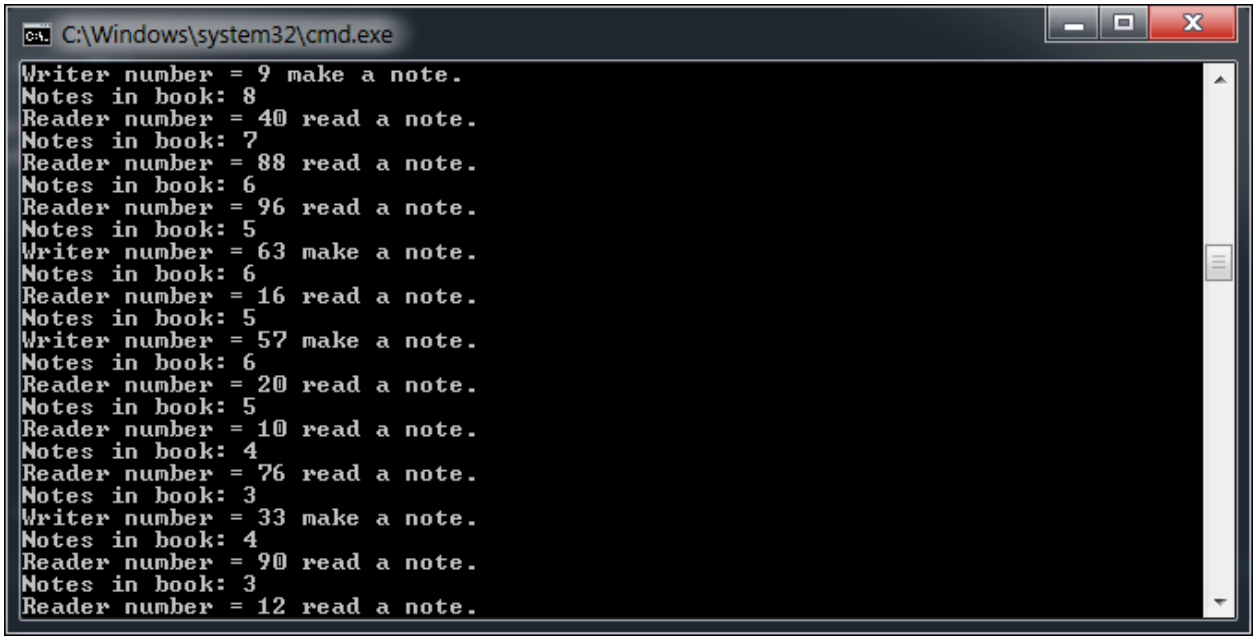

Рисунок 3.1 – Результат выполнения программы «Читатели – Писатели»

Работа программы «Читатели – писатели» заключается в том, что четный поток-читатель считывает данные из ресурса, в случае если ресурс пуст, то читатель ничего не делает. Нечетный поток-писатель производит запись данных в ресурс, если возникает ситуация, когда ресурс полностью заполнен, тогда писатель ничего не делает. Зависимость времени выполнения программы от количества потоков показана на рисунке 3.2.

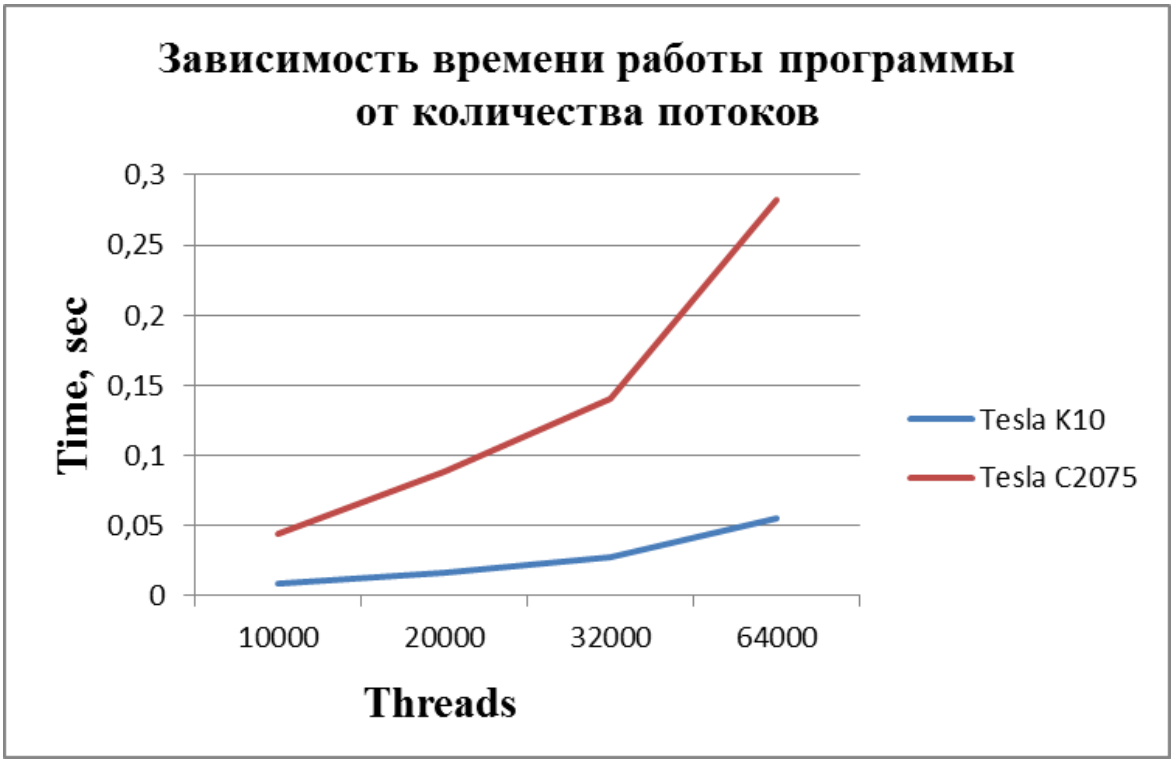

Рисунок 3.2 – Время работы программы «Читатели – Писатели» на отладочной станции (Tesla C2075) и суперкомпьютере (Tesla K10)

Из данного графика мы видим, что при увеличении количества потоков время выполнения программы увеличивается прямо пропорционально. Эксперимент был проведен на отладочной станции обладающей видеокартой Nvidia Tesla C2075. Как можно заметить скорость выполнения программы на суперкомпьютере, обладающем восьмью видеокартами Nvidia Tesla K10, значительно быстрее отладочной станции. Однако время выполнения программы по-прежнему прямо пропорционально увеличению количества потоков, примерно в 2 раза.

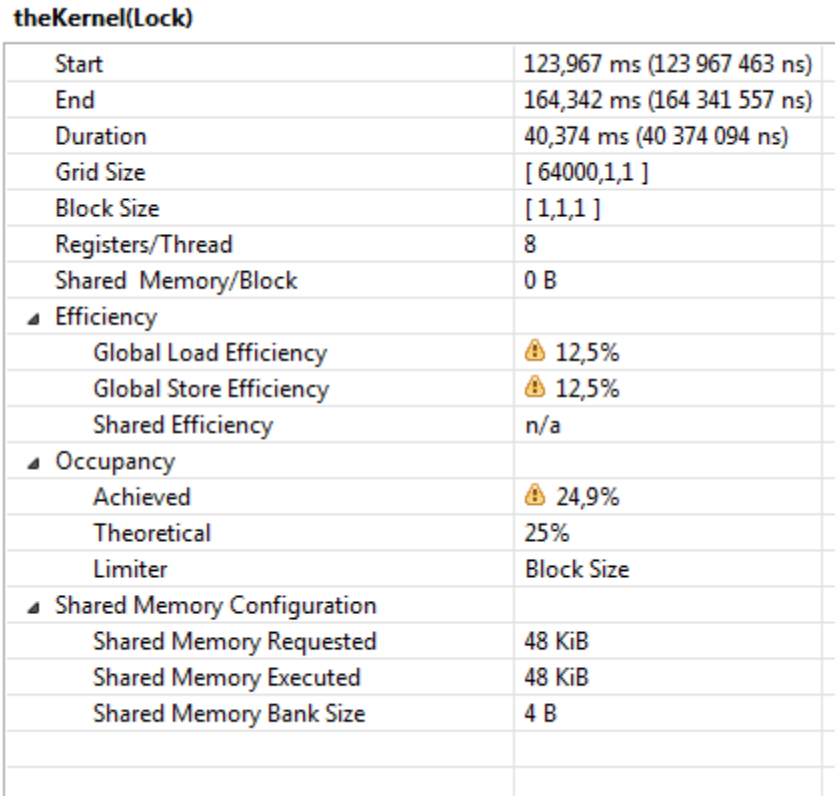

Рисунок 3.3 - Данные из профайлера

При тестировании программы через профайлер (рис. 3.3) было выявлено, что загрузка процессора видеочипа осуществляется всего лишь на 25%, что крайне неэффективно [21]. Изменение размеров хранилища данных никак не повлияло на загруженность ядра, но при большем размере хранилища и количестве потоков, время выполнения программы больше на ~2мсек, по сравнению с меньшим размером хранилища и таким же количеством потоков. Это связано с тем, что в первом случае потоки с большой вероятностью получат доступ к ресурсу, из-за большого размера, и выполнят свою задачу. Во втором случае размер ресурса меньше в 10 раз, поэтому многие потоки видят, что ресурс заполнен или пуст, и не производят никаких действий с ним. По результатам исследования и данных профайлера можно сделать вывод, что реализация классических задач параллельного программирования на технологии CUDA нецелесообразна, так как большую часть времени потоки находятся в режиме ожидания доступа к ресурсу.

Заключение

В результате исследования мы выяснили, что использование GPU и технологии CUDA для неграфических вычислений является эффективным решением для увеличения скорости самих вычислений, и как следствие уменьшения временных и материальных затрат. При разработке параллельной программы очень важно учитывать работу потоков и их синхронизацию между собой, чтобы не возникали сбои в программе. Реализация классических задач параллельного программирования на технологии CUDA демонстрирует взаимодействие потоков между собой. Из результатов эксперимента видно, что применение более мощных GPU будет эффективным при проведении сложных расчётов, в том случае если задачу можно распараллелить на большое количество потоков.

Вычислительная система NVIDIA Tesla на основе графического процессора с архитектурой CUDA, может широко применяться в научных и технических вычислениях общего назначения. Но все же Tesla пока не может полностью заменить обычный центральный процессор, но с ее помощью множества своих ядер можно использовать вычислительный ресурс для решения определенного типа ресурсоёмких задач. Tesla можно назвать своего рода сопроцессором, который можно применять для создания вычислительных систем на основе персональных компьютеров, а также в составе серверов и вычислительных кластеров. Преимущество вычислительной системы с использованием Tesla является большая энергоэффективность и меньшая стоимость, единственным ее недостатком является меньшая универсальность.

Применение GPU в параллельных вычислениях общего назначения является эффективным и целесообразным для проведения больших и сложных расчётов, также позволяет получить их результаты за наименьшее время.

Список используемой литературы

Учебники и учебные пособия

1. Параллельные вычисления на GPU. Архитектура и программная модель CUDA: Учебное пособие / Боресков А. В. и др. – М.: Издательство Московского университета, 2012 – 336с.
2. Программирование. Основы параллельного программирования / Богачев К.Ю., Издательство: М.: Бином. Лаборатория знаний, 2015 – 344с.
3. Высокопроизводительные вычисления для многоядерных многопроцессорных систем / Гергель В.П. – Издательство Нижегородского госуниверситета, 2011 – 421с.
4. Основы работы с технологией CUDA / Борееков А. В., Харламов А. А. - М.: ДМК Пресс, 2011 - 232 с.
5. Оптимизация вычислений на CUDA / Приймак А.В. - К.: Век+, 2013 – 167с.
6. 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.
7. Параллельное программирование мультикомпьютеров: Учеб. Пособие. / Малышкин В.Э., Корнеев В.Д. – М.: Изд-во НГТУ, 2011. – 296с.
8. Современные операционные системы / Таненбаум Э., Бос Х. - СПб.: Питер, 2015. – 1120с.

Электронные ресурсы

9. Параллельные вычисления с CUDA [Электронный ресурс] - Nvidia Corp.,[2015]. – Режим доступа: http://www.nvidia.ru/object/cuda-parallel-computing-ru.html
10. Новый стандарт для высокопроизводительных вычислений с выходом Tesla GPU на базе архитектуры Kepler [Электронный ресурс] – Nvidia

Corp.,[2012]. – Режим доступа: http://www.nvidia.ru/object/hpc-kepler-based-tesla-gpus-20120516-ru.html

11. Общие принципы работы GPU-ускорителей [Электронный ресурс] – Баймурзин Ч.К., [2014]. – Режим доступа: http://cuda.artisteer.net/cuda-2/%D0%BE%D0%B1%D1%89%D0%B8%D0%B5-%D0%BF%D1%80%D0%B8%D0%BD%D1%86%D0%B8%D0%BF%D1%8B-%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D1%8B-gpu-%D1%83%D1%81%D0%BA%D0%BE%D1%80%D0%B8%D1%82%D0%B5%D0%BB%D0%B5%D0%B8/
12. Nvidia CUDA — неграфические вычисления на графических процессорах [Электронный ресурс] – Берилло А. [2012] – Режим доступа: http://www.ixbt.com/video3/cuda-1.shtml
13. Введение в технологии параллельного программирования – [Электронный ресурс] – Чернышев А. [2011] – Режим доступа: https://software.intel.com/ru-ru/articles/writing-parallel-programs-a-multi-language-tutorial-introduction
14. Профилирование программ [Электронный ресурс] – Романенко А.А., [2013] – Режим доступа: http://ccfit.nsu.ru/arom/data/PP_ICaG/06_Profiling_txt.pdf

Литература на иностранном языке

15. Parallel processing with CUDA [Электронный ресурс]: [Параллельная обработка данных на CUDA] - Nvidia Corp., [2012] – Режим доступа: http://www.nvidia.com/docs/IO/55972/220401_Reprint.pdf
16. GPGPU processing in CUDA architecture [Электронный ресурс]: [GPGPU обработка данных в архитектуре CUDA] - Advanced Computing: An International Journal, [2012] – Режим доступа: http://arxiv.org/ftp/arxiv/papers/1202/1202.4347.pdf
17. Understanding the CUDA Data Parallel Threading Model [Электронный ресурс]: [Понимание модели потоков параллельных данных в CUDA] –

Nvidia Corp. PGI Insider, [2012] – Режим доступа : https://www.pgroup.com/lit/articles/insider/v2n1a5.htm

18. Parallel Programming Models [Электронный ресурс]: [Модели параллельного программирования] - Luis Moura e Silva and Rajkumar Buyya – Режим доступа: http://www.buyya.com/cluster/v2chap1.pdf
19. What is Parallel Computing? [Электронный ресурс]: [Что такое параллельные вычисления?] - Ann Arbor, [2015] – Режим доступа: http://web.eecs.umich.edu/~qstout/parallel.html
20. NVIDIA CUDA Compute Unified Device Architecture Programming Guide. Version 2.0 [Электронный ресурс] : [Единая вычислительная архитектура устройств. Руководство по программированию. Версия 2.0] – Electronic data. – NVidia Corp., [2015]. – Режим доступа: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf
21. Profiler User's Guide [Электронный ресурс]: [Профайлер руководство пользователя] – Nvidia Corp.,[2015] – Режим доступа: http://docs.nvidia.com/cuda/profiler-users-guide/

Приложение А

Листинг кода файла ReadersWriters.cu

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<cuda.h>
#include<cuda_runtime.h>
#include<cuda_profiler_api.h>

// number of blocks
#define nob 10
__device__ int note;

struct Lock{
	int *mutex;
	Lock(void){
		int state = 0;
		cudaMalloc((void**)&mutex, sizeof(int));
		cudaMemcpy(mutex, &state, sizeof(int),
cudaMemcpyHostToDevice);
	}
	~Lock(void){
		cudaFree(mutex);
	}
	__device__ void lock(void){
		while (atomicCAS(mutex, 0, 1) != 0);
	}
	__device__ void unlock(void){
		atomicExch(mutex, 0);
	}
};


__global__ void theKernel(Lock myLock){
	int index = blockIdx.x; //using only one thread per block

	myLock.lock();
	if ((index % 2 == 0) && note>0){
		note--;
		printf("Reader number = %i read a note.\n", index);
		printf("Notes in book: %i\n", note);
	}
```

```
        if ((index % 2 != 0) && note<100){
                note++;
                printf("Writer number = %i make a note. \n", index);
                printf("Notes in book: %i\n", note);
        }

        myLock.unlock();

}


int main(void)
{
        cudaProfilerStart();
        Lock myLock;
        theKernel << <nob, 1 >> >(myLock);
        cudaProfilerStop();
        return 0;

}
```