# ОГЛАВЛЕНИЕ

[ОГЛАВЛЕНИЕ 1](#_Toc422290690)

[ВВЕДЕНИЕ 2](#_Toc422290691)

[1. ПОСТАНОВКА ЗАДАЧИ 3](#_Toc422290692)

[2. О ВИДЕОАДАПТЕРАХ 5](#_Toc422290693)

[2.1. ОСОБЕННОСТИ АРХИТЕКТУРЫ СОВРЕМЕННЫХ GPU 6](#_Toc422290694)

[2.2. NVIDIA 7](#_Toc422290695)

[2.3. ОСОБЕННОСТИ АРХИТЕКТУРЫ 11](#_Toc422290696)

[2.4. ПРОГРАММНАЯ МОДЕЛЬ CUDA. ОСНОВНЫЕ ПРИНЦИПЫ 14](#_Toc422290697)

[3. ЛИНЕЙНОЕ ПРОГРАММИРОВАНИЕ 16](#_Toc422290698)

[3.1. ЗАДАЧА ЛИНЕЙНОГО ПРОГРАММИРОВАНИЯ 18](#_Toc422290699)

[3.1.1. ГЕОМЕТРИЧЕСКАЯ ИНТЕРПРЕТАЦИЯ 21](#_Toc422290700)

[3.1.2. ДВОЙСТВЕННОСТЬ ЗАДАЧИ ЛИНЕЙНОГО ПРОГРАММИРОВАНИЯ 24](#_Toc422290701)

[3.2. СИМПЛЕКС-МЕТОД 25](#_Toc422290702)

[3.3. МОДИФИЦИРОВАННЫЙ СИМПЛЕКС-МЕТОД 29](#_Toc422290703)

[3.3.1. ОБОСНОВАНИЕ МОДИФИЦИРОВАННОГО СИМПЛЕКС-МЕТОДА 29](#_Toc422290704)

[3.3.2. АЛГОРИТМ МОДИФИЦИРОВАННОГО СИМПЛЕКС-МЕТОДА 34](#_Toc422290705)

[3.4. МЕТОД КАРМАРКАРА 36](#_Toc422290706)

[3.4.1. ОСНОВНАЯ ИДЕЯ 36](#_Toc422290707)

[3.4.2. АЛГОРИТМ КАРМАРКАРА 38](#_Toc422290708)

[4. РЕАЛИЗАЦИЯ НА CUDA 41](#_Toc422290709)

[4.1. ВЫБОР МЕТОДА 42](#_Toc422290710)

[4.2. ОСОБЕННОСТИ ПРОГАММИРОВАНИЯ НА CUDA 44](#_Toc422290711)

[4.3. ПОСЛЕДОВАТЕЛЬНАЯ РЕАЛИЗАЦИЯ 48](#_Toc422290712)

[4.3.1. ВСПОМОГАТЕЛЬНЫЕ БИБЛИОТЕКИ 48](#_Toc422290713)

[4.3.2. ОСНОВНОЙ МОДУЛЬ 51](#_Toc422290714)

[4.4. CUDA РЕАЛИЗАЦИЯ 52](#_Toc422290715)

[4.4.1. БИБЛИОТЕКА CUBLAS 52](#_Toc422290716)

[4.4.2. СТРУКТУРА ПРОГРАММЫ 53](#_Toc422290717)

[5. ЧИСЛЕННЫЕ ИССЛЕДОВАНИЯ 56](#_Toc422290718)

[5.1. МЕТОДОЛОГИЯ ТЕСТИРОВАНИЯ 56](#_Toc422290719)

[5.2. РЕЗУЛЬТАТЫ ИЗМЕРЕНИЙ 58](#_Toc422290720)

[ЗАКЛЮЧЕНИЕ 64](#_Toc422290721)

# ВВЕДЕНИЕ

Развитие архитектуры вычислительны систем – это история постоянного поиска баланса устройств, оптимального для множества целевых приложений. Пока не был исчерпан ресурс основных факторов роста, массовое производство и экономическая выгода сдерживали сколь-нибудь значительную специализацию основных вычислительных структур. Однако каждое новое инженерное решение в своем развитии со временем обнаруживало соответствующий противовес: частота и тепловыделение, многоядерность и когерентность кэшей, общая память и неоднородный доступ, конвейерность и ветвление и т.д. В условиях недостатка новых идей фактором роста в настоящее время становятся специализированные вычислители. Наибольший успех графических ускорителей (GPU) в этом качестве связан с их устойчивым положением в основной сфере применения.

Устройства архитектуры GPU можно кратко охарактеризовать как “макроархитектуру вычислительного кластера, реализованную в микромасштабе”. GPU состоит из однородных вычислительных элементов с общей памятью. Каждый вычислительный элемент способен исполнять тысячи потоков, переключение между которыми не имеет накладных расходов. Потоки могут быть сгруппированы в блоки, имеющие общий кэш и быструю разделяемую память, явно контролируемую пользователем. Данная реализация в сочетании с расширениями для процедурных языков программирования носит название Compute Unified Device Architecture.

В данной работе рассматривается эффективность применения технологии CUDA для решения задачи линейного программирования. Зачастую размеры задачи оптимизации крайне велики, или требуется производить расчеты за ограниченное время (например, при краткосрочном прогнозировании), в таком случае необходимо максимально эффективно использовать технические ресурсы вычислительной системы, производить оптимизацию алгоритма и техники вычислений, основываясь на особенностях конкретной задачи.

В результате работы было рассмотрено текущее состояние проблемы решения задачи линейного программирования, произведен обзор GPU-адаптеров и их технических особенностей, а также разработано ПО для решения задачи линейного программирования с использованием технологии CUDA и произведен анализ эффективности полученного решения.

# ПОСТАНОВКА ЗАДАЧИ

Итак, одной из целей данной работы является разработка программного обеспечения для решения задачи линейного программирования. При этом необходимо разработать как классическую последовательную реализацию, так и параллельную реализацию на CUDA. Опишем формат входных данных для разрабатываемого ПО:

*2 2*

*5 3*

*1 1 4*

*5 2 10*

Это пример входных данных, соответствующий задаче

В первой строке

*2*

записано число переменных и число ограничений. В тестовых задачах эти значения будут равны. Во второй строке

*5 3*

указаны коэффициенты оптимизируемой функции. Имеется ввиду, что ведется поиск именно максимума функции. Далее следует описание ограничений по числу, указанному в первой строке

*1 1 4*

*5 2 10,*

где каждой строке соответствует одно линейное ограничение. В строке указаны коэффициенты ограничения.

Так же требуется произвести анализ эффективности применения технологии CUDA. Будет произведен сравнительный анализ как общего времени работы, так и времени, затрачиваемого на выполнение отдельных операций. Для визуализации экспериментальных данных будем использовать Gnuplot – свободное программное обеспечение для создания двух- и трехмерных графиков.

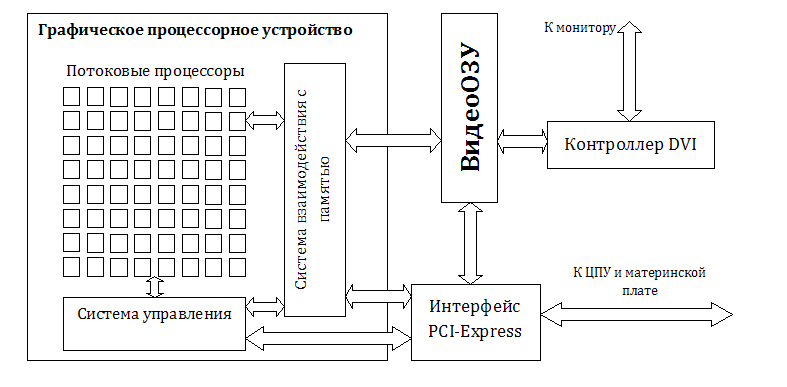
Запуск и тестирование разработанного ПО будет происходить на системе с двухъядерным процессором Intel Core2 Quad (Q9550) 2,83 Ггц с 12 Мб кэш-памяти и 1333МГц шиной. Видеодаптер – Nvidia GeForce GTX 280.

# О ВИДЕОАДАПТЕРАХ

Вычислительная математика благодаря развитию компьютерных технологий получила возможность решать важные практические задачи за достаточно приемлемое время. К тому же появляются все новые и более быстрые способы обработки данных. Сегодня становится популярным использование современных графических процессоров для осуществления высокопроизводительных математических вычислений. Это позволяет значительно увеличить скорость вычислений по сравнению с теми, что обычно выполняются на центральном процессоре компьютера.  
Графическое процессорное устройство (англ. Graphics Processing Unit, GPU) – программируемое вычислительное устройство, изначально предназначенное для обработки графической информации. Оно занимается расчётами выводимого изображения, освобождая от этой обязанности центральный процессор (англ. Central Processing Unit, CPU), а также производит расчёты для обработки команд трёхмерной графики (геометрическая трансформация, моделирование освещения). В силу ряда особенностей архитектуры GPU стали использоваться как платформа для высокопроизводительных математических вычислений.

## ОСОБЕННОСТИ АРХИТЕКТУРЫ СОВРЕМЕННЫХ GPU

Современные графические процессоры пятого поколения имеют довольно схожую архитектуру (Рис. 21), они содержат набор одинаковых вычислительных устройств (потоковых процессоров, ПП), работающих с общей памятью графического процессора (видеоОЗУ). Число ПП, а также размер видеоОЗУ может быть различным, в зависимости от модели GPU. Все ПП синхронно исполняют одну и ту же команду, что позволяет отнести GPU к классу SIMD. Система команд ПП включает арифметические команды для вещественных и целочисленных вычислений с 32-разрядной точностью, команды управления (ветвления и циклы), а также команды обращения к памяти. Из-за высоких задержек команды доступа к оперативной памяти выполняются асинхронно. С целью сокрытия задержек в очереди выполнения GPU может одновременно находиться несколько сотен потоков, и если текущий поток блокируется по доступу к памяти, на исполнение ставится следующий. Поскольку контекст потока полностью хранится на регистрах графического процессора, переключение осуществляется за один такт. За переключение потоков отвечает диспетчер потоков, который не является программируемым.



рис

Тактовые частоты GPU ниже, чем у центрального процессора компьютера. Однако благодаря большому количеству потоковых процессоров производительность GPU весьма значительна, а если еще учесть, что существует возможность установки на одну машину двух графических карт, то это позволяет получить пиковую производительность до нескольких ТФлопс. Кроме того, на некоторых практических задачах может достигаться значительный процент пиковой производительности (до 70%). Одновременно с этим, в сравнении с классическими кластерными системами, графические процессоры обладают значительно лучшими характеристиками, как по цене, так и по энергопотреблению.

## NVIDIA

NVIDIA Corporation (рис. 38) – транснациональная компания, занимающаяся разработкой графических процессоров для широкого круга направлений, включая серверные установки, рабочие станции, ПК, мобильные устройства. Это одна из крупнейших компаний в своей области наряду с Intel и AMD. Основана компания была в 1993 году. Создали ее трое: Джен-Хсун Хуанг (Jen-Hsun Huang), Картис Прэм (Curtis Priem) и Крис Малачовски (Chris Malachowsky). Первые графические решения компании появились в 1995 году — NV1. Видеокарты на базе NV1 производила компания SGS-THOMSON Microelectronics.

Название компании — весьма любопытное сочетание. Литера «N» пришла из математики, где обычно она означает натуральное число. «VIDIA» – это производная от латинского слова «videre» (смотреть, увидеть). В итоге, получилось нечто вроде «многократного виденья», «n-кратного виденья». Название NVIDIA созвучно с испанским словом «envidia» (зависть), что подало идею для необычного рекламного слогана для графических процессоров серии GeForce 8 – «Green with envy» (Зеленый от зависти), с намеком на конкурентов и их продукцию.

В 2004 году появляется 3D медиапроцессор GeForce 3D 4500 для мобильных устройств — первый в мире. Вообще, вся история компании – это история создания все более и более “продвинутых” процессоров. А так же наград и премий, которых они удостаивались. Сама компания также не раз была отмечена – к примеру, в 2007 году она стала “компанией года”, по мнению Forbes.

Ниже представлен обзор развития графический процессоров от NVIDIA.

#### Семейство GeForce

GPU семейства GeForce были разработаны под нужды игровой индустрии. Последней видеокартой из этого семейства была GeForce GTX 200. Самой производительной моделью была GeForce GTX 280. В ее основе были 240 процессорных ядер, каждое из которых работало на частоте 1,296 ГГц. Видеокарта обладала 1 ГБ оперативной памяти, работающей по 512-битному GDDR3 интерфейсу. Так как семейство видеокарт былло разработано для отрисовки видеоигр, наиболее интересной характеристикой для них является скорость заполнения текстур (texture fill rate). Например, для топовой версии GTX 280 скорость заполнения текстур составляет 280 млрд./с.

#### Семейство Quadro

Семейство Quadro является следующей ступенью развития графических процессоров. Основные улучшения заключаются в оптимизации CAD, DCC (Digital Content Creation) и производительности визуализации. На первый взгляд это семейство очень похоже на предыдущее поколение GeForce, и в некотором смысле это верно. Многие из Quadro видеокарт имеют тот же чипсет, что и GeForce видеокарты. Но есть существенная деталь, определяющая границу между семействами. Изначально ориентированные на игровую индустрию, в картах семейства GeForce ради скорости вычислений жертвовалась точность. Новое семейство Quadro было спроектировано с учетом требований к точности вычислений, что позволило использовать видеокарты на их основе в таких задачах как обработка аудио и видео данных. Флагманская видеокарта из семейства Quadro FX 5600 показывала скорость заполнения текстур на уровне 76,8 млрд./с.

#### Семейство Tesla

GPU семейства Tesla разрабатывались непосредственно под нужды высокоскоростных вычислений. Видеокарты этого семейства обладали большой вычислительной мощностью и отличались большим объемом оперативной памяти. Например, C1060 обладала 240 процессорными ядрами с частотой 1,296 ГГц и выдавала 933 ГФпс. Видеокарты, обладающие 4 ГБ оперативной памяти имели пропускную способность 102 ГБ/с. Также GPU семейства Tesla не имели непосредственного соединения с устройствами отображения. Было заявлено, что Tesla не ориентированы не на обработку графики, а на высокопроизводительные вычисления.

#### Семейство Fermi

Архитектура Fermi предполагает, что обработка компьютерной графики больше не является  единственной задачей графических процессоров, хотя и остаётся одним из приоритетных направлений. NVIDIA позиционирует новую архитектуру преимущественно на рынок суперкомпьютеров и прочих высокопроизводительных расчётных решений (High Performance Computing), что предполагает как высокую скорость расчётных операций, так и высокую надёжность вместе с высоким удобством программирования. Для этого рынка ключевым требованием является поддержка вещественных вычислений двойной точности (Double Precision Floating Point) и механизмов нахождения и коррекции ошибок (ECC, Error Checking and Correcting) в оперативной памяти и подсистемах кэш-памяти для повышенной отказоустойчивости.

Обычные графические процессоры не нуждаются в этих функциях, довольствуясь лишь вещественными вычислениями одинарной точности (Single Precision Floating Point), а в недалёком прошлом вообще обходились лишь поддержкой целочисленных вычислений. Справедливости ради стоит заметить, что графический процессор GT200 мог использоваться для вещественных вычислений двойной точности, но его производительность на таких задачах оставляла желать лучшего.

#### Семейство Kepler

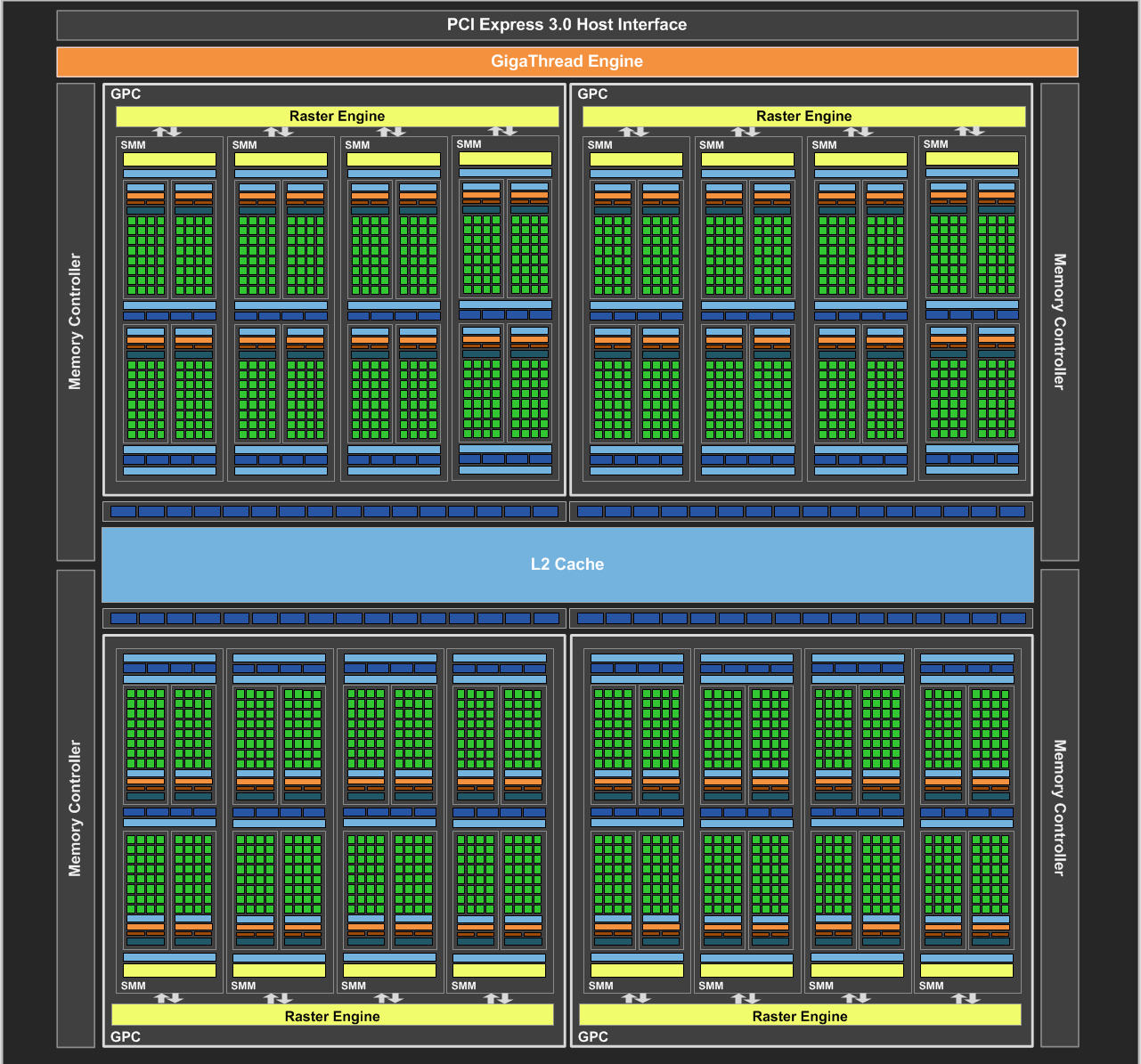
Kepler – архитектура, созданная для высокопроизводительных вычислений, с акцентом на энергоэффективности. Тогда как направленностью предыдущей архитектуры, Fermi, была чистая производительность, Kepler рассчитан на энергоэффективность, программируемость и производительность.

Энергоэффективность была достигнута за счет использования унифицированной тактовой частоты (шейдерные блоки работают на одной частоте с ядром). Отказ от модели с независимой частотой шейдерных блоков, которая использовалась в предыдущих GPU NVIDIA, позволяет снизить энергопотребление даже при том, что для достижения производительности на уровне предыдущих разработок, требуется использовать большее количество шейдерных ядер. Уменьшение энергопотребления происходит не только от того, что новая архитектура более энергоэффективна, чем архитектура предыдущего поколения (два шейдерных ядра Kepler используют около 90% питания, необходимого одному ядру Fermi), но и потому, что унификация тактовой частоты приводит к снижению частоты шейдерных блоков, что в свою очередь серьёзно снижает энергопотребление[[3]](https://ru.wikipedia.org/wiki/%D0%9A%D0%B5%D0%BF%D0%BB%D0%B5%D1%80_(%D0%BC%D0%B8%D0%BA%D1%80%D0%BE%D0%B0%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0)#cite_note-ref3-3).

Улучшенная программируемость была достигнута за счёт введения новой модели обработки текстур, которая не требует привязки к CPU. Улучшение же производительности было достигнуто за счёт внедрения абсолютно новых контроллера памяти и шины. В свою очередь это позволило поднять тактовую частоту памяти до 6 ГГц, что всё ещё ниже, чем теоретически максимальные дляGDDR5 7 ГГц, но значительно больше, чем частота памяти в 4 ГГц при архитектуре предыдущего поколения.

#### Семейство Maxwell

Это самое последнее на данный момент семейство видеокарт от компании NVIDIA. Схематично GPU можно представить так:



Отметим наличие 2048 потоковых процессоров, которые объединены в 16 потоковых мультипроцессоров Maxwell, каждый содержит 128 ядер CUDA. Отсюда понятно наличие 128 текстурных блоков и 64 конвейеров растровых операций, но, опять же, поговорим об этом позже. Самая впечатляющая техническая деталь – тепловой пакет (TDP) всего 165 Вт. При сохранении прежнего техпроцесса и усложнении GPU (по сравнению с GK104), с повышением тактовых частот NVIDIA удалось существенно снизить энергопотребление.

## ОСОБЕННОСТИ АРХИТЕКТУРЫ

#### Мультипроцессоры

Видеочипы от NVIDIA состоят из нескольких кластеров текстурных блоков (*Texture Processing Cluster*). Каждый кластер в свою очередь состоит из блока текстурных выборок и нескольких мультипроцессоров. Мультипроцессор состоит из 8 вычислительных устройств и двух функциональных блоков. Все инструкции в GPU выполняются по принципу SIMD, т.е. одна инструкция применяется ко всем потокам в *warp* (группа из 32 потоков). На каждый из мультипроцессоров выделяется от 8192 до 16384 регистров (в зависимости от модели видеокарты), которые являются общими для всех потоков, исполняемых на нем. Также на каждом процессоре имеется зависящий от модели объем быстрой разделяемой памяти, которая доступна на запись и чтение из любого потока внутри одного блока. Мультипроцессоры также имеют доступ к видеопамяти, но необходимо быть крайне осторожным, поскольку доступ к ней сопряжен со значительными временными задержками. Для ускорения доступа и снижения количества обращений к видеопамяти все мультипроцессоры оснащаются небольшим кэшем для констант и текстур. Мультипроцессоры GPU могут выполнять до восьми блоков и до 24-х warp, каждый из которых включает в себя 32 потока. Иными словами, на мультипроцессор приходится до 768 потоков.

#### Организация памяти

На GPU можно выделить два различных типа памяти: непосредственно на чипе и в оперативной памяти. Каждый процессор имеет свой объем памяти следующих типов;

* Регистровая (*register*) память. Является самой быстрой из всех видов памяти. В CUDA нет явных способов использования регистровой памяти, всю работу по размещению данных в регистрах берет на себя компилятор;
* Локальная (*local*) память. Может быть использована компилятором при большом количестве локальных переменных в какой-либо функции. По скоростным характеристикам локальная память значительно медленнее, чем регистровая;
* Глобальная (*global*) память. Самый медленный тип памяти, из доступных GPU. Глобальная память в основном служит для хранения больших объемов данных, поступивших на device с host’а. В алгоритмах, требующих высокой производительности, количество операций с глобальной памятью необходимо свести к минимуму;
* Разделяемая (*shared*) память, которую можно сравнить с кэшем первого уровня на CPU. Относиться к быстрому типу памяти. Разделяемую память рекомендуется использовать для минимизации обращение к глобальной памяти, а так же для хранения локальных переменных функций. Адресация разделяемой памяти между нитями потока одинакова в пределах одного блока, что может быть использовано для обмена данными между потоками в пределах одного блока;
* Константная (*constant*) память. Отличительной особенностью константной памяти является возможность записи данных с хоста, но при этом в пределах GPU возможно лишь чтение из этой памяти, что и обуславливает её название;
* Текстурная (*texture*) память. Как и следует из названия, предназначена главным образом для работы с текстурами. Текстурная память имеет специфические особенности в адресации, чтении и записи данных.

#### Вычислительные возможности

Одной из немаловажных характеристик являются вычислительные возможности (*compute capability, CC*)*.* Они были введены для разделения GPU с идентичной архитектурой. Для примера: устройства с CC 1.0 не умели делать атомарных операций вообще, с СС 1.1 – умели в глобальной памяти а с СС 1.2 – и в глобальной и в разделяемой, с версии 2.0 появились операции на числами двойной точности. Версия вычислительных возможностей всегда записывается парой цифр: мажорным и минорным номерами. Мажор говорит об архитектуре GPU, а минор – о наличии тех или иных возможностей в рамках основной версии.

## ПРОГРАММНАЯ МОДЕЛЬ CUDA. ОСНОВНЫЕ ПРИНЦИПЫ

CUDA (*Compute Unified Device Architecture*) – это программная модель, включающая описание вычислительного параллелизма и иерархичной структуры памяти непосредственно в язык программирования. С точки зрения программного обеспечения, реализация CUDA представляет собой кроссплатформенную систему компиляции и исполнения программ, части которых работают на GPU и CPU. CUDA предназначена для разработки GPGPU-приложений без привязки к графическим API и поддерживается всеми GPU NVIDIA, начиная с GeForce 8.

#### Основные принципы

Концепция CUDA отводит GPU роль массивно-параллельного сопроцессора. В литературе о CUDA основная система, к которой подключен GPU, для краткости называют хост (*host*), аналогично сам GPU по отношению к хосту часто называют устройством (*device*). CUDA-программа задействует как CPU, так и GPU. На CPU выполняется последовательная часть кода и подготовительные стадии GPU-вычислений. Параллельные участки кода могут быть перенесены на GPU, где будут одновременно выполняться большим количеством нитей (*threads*). Важно отметить ряд принципиальных различий между обычными потоками CPU и нитями GPU:

* Нить GPU чрезвычайно легковесна, ее контекст минимален, регистры распределены заранее;
* Для эффективного использования ресурсов GPU программе необходимо задействовать тысячи отдельных нитей, в то время как на многоядерном CPU максимальная эффективность обычно достигается при числе потоков, равном или в несколько раз большем количества ядер.

В целом работа нитей на GPU соответствует принципу SIMD, однако есть существенное различие. Только нити в пределах одной группы (для GPU архитектуры Fermi – 32 нити) называемой варпом (*warp*), выполняются *физически одновременно.* Нити различных варпов могут находиться на разных стадиях выполнения программы. Такой метод обработки данных обозначается термином SIMT (*Single Instruction Multiple Threads*). Управление работой варпов производится на аппаратном уровне.

Важным преимуществом CUDA является использование для программирования GPU языков высокого уровня. В настоящее время существую компиляторы C++ и Fortran. Эти языки расширяются небольшим множеством новых конструкций: атрибуты функций и переменных, встроенные переменные и типы данных, оператор запуска ядра.

# ЛИНЕЙНОЕ ПРОГРАММИРОВАНИЕ

Под линейным программированием мы понимаем математические модели и техники, используемые для обучения и решения целого семейства прикладных задач. В одной из постановок требуется оптимизировать функцию, удовлетворяющую некоторому набору условий.

Линейное программирование – предмет изучения такой отрасли математики как исследование операций. Первые методы и модели были разработаны во время Второй Мировой войны в военных целях. Одни из самых известных математиков того времени внесли свой вклад в решение проблемы. Джордж Бернард Данциг, например, американский математик, разработавший симплекс-метод в 1947 году, или Джон фон Нейман, разработавший теорию двойственности, к которой мы обратимся в дальнейшем.

Чтобы понять идею проблемы задачи линейного программирования, представим такой сценарий. Требуется собрать небольшой, но мощный вычислительный кластер GPU. Главная цель – вычислительная мощность. И есть несколько поставщиков процессоров. Например, продукция двух из них в целом устраивает: модель *A*, выдающая 700 гФлопс, и модель *B*, выдающая 900 гФлопс. При отсутствии иных условий, конечно же, второй вариант наиболее предпочтителен, в силу большей производительности. Но в реальности всегда есть ограничения. Допустим, есть ограничение по бюджету на закупку оборудования и требуемой кластером мощностью сети. Пусть бюджет составляет 6000 евро и электрическая сеть ограничена 500 Вт, в то время как процессор *A* стоит 500 евро и потребляет 250 Вт, а процессор *B* стоит 3 400 евро и потребляет 200 Вт.

В примере, приведенном выше, ставится вполне очевидная задача: максимизировать конечную мощность кластера, варьируя количество закупленных процессоров каждого типа. При этом, разумеется, необходимо учитывать все ограничения.

Схематично задачу можно представить в виде таблицы:

|  |  |  |  |
| --- | --- | --- | --- |
|  | GPU модель *A* | GPU модель *B* |  |
| Производительность (ГФлопс) | 700 | 900 |  |
| Ограничение бюджета (евро) | 500 | 3400 | 6000 |
| Ограничение энергопотребления (Вт) | 250 | 200 | 500 |

Этот пример крайне прост для понимания, но он отражает суть задачи линейного программирования. В дальнейшем в этой главе мы будем стараться абстрагироваться от специфических прикладных ограничений и формализуем задачу, а так же предложим алгоритмы решения. Проанализировав алгоритмы, можно будет определить техники, наиболее подходящие для реализации алгоритма на GPU.

## ЗАДАЧА ЛИНЕЙНОГО ПРОГРАММИРОВАНИЯ

Как уже было сказано, линейное программирование используется в различных областях науки. Как следствие, она была широко изучена, и было издано много работ на эту тему. К сожалению, как это часто бывает, терминология отличается от источника к источнику. Сформулируем формально задачу линейного программирования и будем придерживаться терминологии, принятой здесь.

Задача линейного программирования состоит из следующих основных элементов:

1. Целевая функция ,удовлетворяющая условиям и ;
2. Конечное множество линейных ограничений, где каждое ограничение представлено в виде , где .

Цель линейного программирования оптимизировать, т. е. минимизировать или максимизировать целевую функцию. Задачу линейного программирования можно записать в виде

Последнее ограничение требует, чтобы все переменные были неотрицательны, что напрямую вытекает из смысла задачи. Например, в задаче о покупке GPU для кластера, само собой, нельзя купить отрицательное число процессоров. Задача линейного программирования также часто представляется в матричном виде. Положим *c* и *x* векторами из , *b* – вектор в , и *A* – матрица из , то есть

Перепишем формулировку задачи в этих терминах:

Для примера запишем в этой формулировке задачу, сформулированную ранее:

Справедливости ради отметим, что этот пример является далеко не самым наглядным. Здравый смысл накладывает ограничение на решение, а именно требует его целочисленности. Действительно, с практической точки зрения невозможно купить дробное число процессоров. Такого рода задачи относятся к семейству задач целочисленного линейного программирования – особой категории задач, имеющий свои методы решения, отличные от методов решения обычной задачи линейного программирования.

Взглянем еще раз на формулировку задачи линейного программирования. Она представлена в *канонической* форме. Это одна из возможных форм представления. Есть так называемая *стандартная* форма, которая лишь изменяет форму представления записи:

Разумеется, этим варианты записи задачи линейного программирования не ограничиваются. Например, постановку задачи можно переписать еще и в таком виде:

Переменные называются *фиктивными*, они вводятся для того, чтобы заменить нестрогое неравенство равенством.

В дальнейшем будет использоваться форма записи задачи линейного программирования, которая может быть получена путем добавления фиктивной переменной к канонической записи задачи, получая в результате

Добавление в представление фиктивной переменной будет полезно при формулировании численного алгоритма решения задачи, так как намного удобнее оперировать равенствами, чем неравенствами.

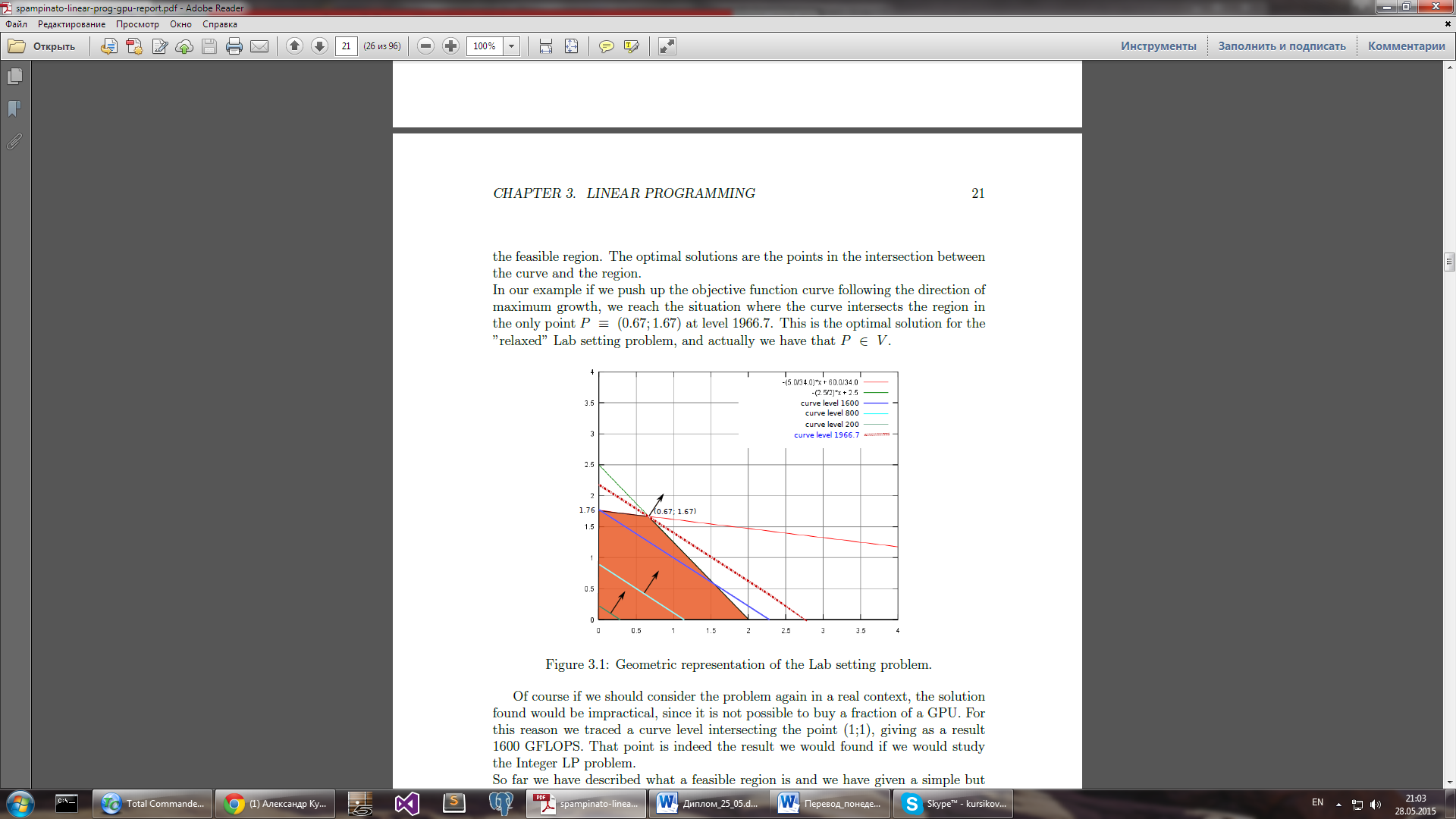
### ГЕОМЕТРИЧЕСКАЯ ИНТЕРПРЕТАЦИЯ

Для большего понимания задачи линейного программирования и техник ее решения, представим ее геометрический смысл. Свяжем понятия, через которые мы формулировали задачу, с их геометрическим отражением. В пространстве оптимизируемых переменных, ограничения в виде неравенств образуют полупространство, в то время как ограничения, заданные в виде равенств, образуют гиперплоскости. Пересечение всех полуплоскостей или гиперплоскостей, порожденных ограничениями, называется областью допустимых решений. Эта область представляет собой набор точек, каждая из которых удовлетворяет набору ограничений задачи. В аналитической геометрии объект, определенный пересечением полупространств называется многогранником. Причем по построению многогранник является выпуклым. Выпуклые многогранники также часто называют симплекс-многогранниками. Тривиальными примерами симплекс-многогранников могут служить точка, отрезок, треугольник. На примере этих простых объектов продемонстрируем геометрический смысл задачи линейного программирования.

Вновь обратимся к примеру с оптимальной закупкой процессоров. Рис. 1 демонстрирует графический смысл задачи линейного программирования. Оранжевая область – выпуклый многогранник, образованный пересечением четырех полупространств, порожденных ограничениями задачи. Зеленая и красная линии – наборы точек, удовлетворяющие соответственно стоимости и мощности, выраженные через равенства. Они образуют два полупространства, направленные к центру области в декартовой системе координат. Оставшиеся два полупространства образованы условием неотрицательности оптимизируемых переменных, что позволяет нам рассматривать только первый квадрант системы координат.

Оранжевая область и является областью допустимых решений. Важнейшим результатом в области исследования задачи линейного программирования является тот факт, что если для задачи существует конечное решение, то оно соответствует одной из вершин области допустимых решений.

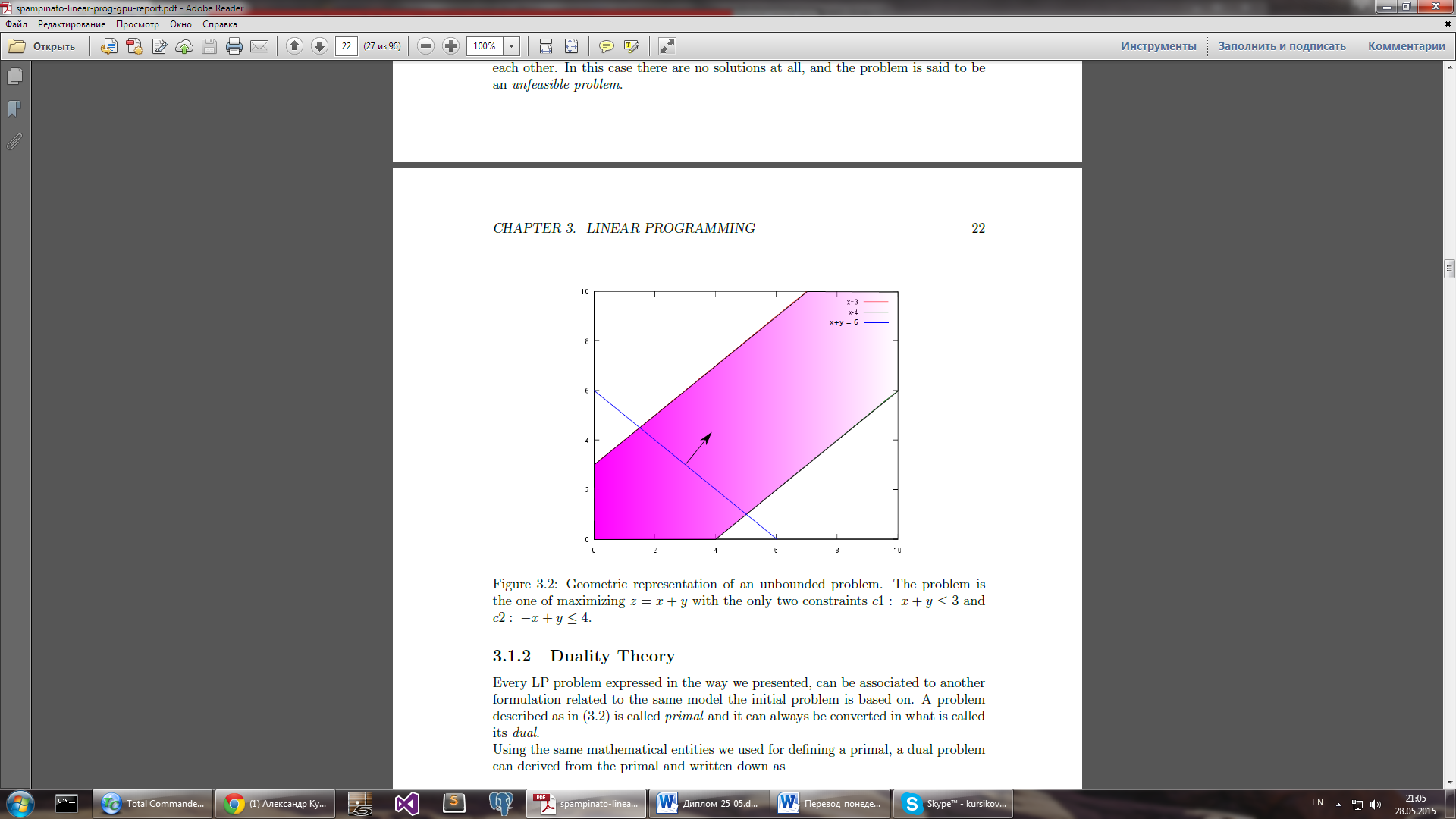
Посмотрим на Рис.1. В нашем случае множество вершин *V* образовано четырьмя точками . На рисунке изображено направление кривой целевой функции, перемещенной вдоль ее градиента. Мы искали максимум целевой функции. Двигаясь вдоль градиента в направлении роста, достигается ситуация, когда кривая целевой функции касается области допустимых решений в единственной точке. Это и есть оптимальное решение нашей задачи.



В рассмотренном примере нас интересует только целочисленное решение, поэтому выбираем ближайшее к оптимальному, в нашем случае это точка (1;1).

Итак, мы получили простое, но наглядное представление задачи линейного программирования. Мы рассмотрели только случай, область допустимых решений ограничена. Однако это не единственная ситуация. Зачастую ограничения задачи порождают область допустимых решений. Продемонстрируем также и эту ситуацию. В этом случае возникает проблема, что целевая функция не имеет экстремума. На практике к таким ситуациям приводит, как правило, неполное или неверное понимание изучаемого объекта. На рис. 2 представлено геометрическое представление задачи линейного программирования с областью допустимых решений. Двигая кривую целевой функции вдоль направления градиента, значение функции растет неограниченно.

Третья и последняя возможная ситуация, когда нет ни одной точки, удовлетворяющей условиям задачи. В этом случае решения задачи не существует.



На этом рисунке изображено геометрическое представление следующей задачи: , при ограничениях и .

### ДВОЙСТВЕННОСТЬ ЗАДАЧИ ЛИНЕЙНОГО ПРОГРАММИРОВАНИЯ

Задача линейного программирования, представленная в каноническом виде, может быть переформулирована. При этом смысл задачи совершенно не изменяется. Используя те же понятия, что и при построении канонической формы задачи линейного программирования, сформулируем двойственную к ней:

Нетрудно показать, что двойственная к двойственной задаче является прямой задачей. Как уже было сказано двойственность дает другой взгляд на ту же задачу. И эта связь между постановками задачи вовсе не формальность. Некоторые методы решения задачи линейного программирования используют оба варианта постановки для получения результатов.

Одна из таких теорем называется теоремой о слабой двойственности, гласящая, что если задача линейного программирования сформулирована в канонической и двойственной форме, и они имеют допустимые значения и соответственно, то . Отношение можно усилить, если эти допустимые значения оптимальны. Об этом говорит теорема сильной двойственности. Т. е. если и – решения задач, то справедливо .

Так же можно доказать, что если значение функции *z,* соответствующее любому допустимому решению прямой задачи, не меньше значения функции *w*, соответствующего допустимому решению двойственной задачи.

Еще одна теорема гласит, что если двойственная задача имеет конечное решение , то прямая задача имеет конечное решение . Значения симплекс-множителей оптимального решения двойственной задачи являются значениями переменных в оптимальном решении прямой задачи.

## СИМПЛЕКС-МЕТОД

Теперь, когда мы сформировали представление о задаче линейного программирования, рассмотрим методы ее решения. Рассмотрим две группы методов: симплекс-методы и методы внутренней точки. Эти методы наиболее широко используются на практике при решении задач линейного программирования. Оба подхода к решению используют сформулированную ранее постановку задачи, разница заключается в способе поиска решения в области допустимых решений.

Симплекс-метод был первым алгоритмом решения задачи линейного программирования, примененным на практике. Он был разработан Д. Б. Данцигом, одним из основоположником линейного программирования, в 40-х годах XX века.

Как уже было сказано ранее, если задача линейного программирования имеет решение, то оно расположено в одной из вершин области допустимых решений. Симплекс-метод – итерационный метод, проходящий через вершины области допустимых решений, пока не достигнет экстремума, на каждом шаге получая более оптимальное решение.

Целая группа методов основывается на этой идее. В этой главе будут рассмотрены классический симплекс-метод, а также модифицированная версия, которая подходит для реализации на GPU.

Для полноты понимания вновь обратимся к задаче о закупке процессоров и рассмотрим на ее примере ход решения задачи симплекс-методом. Используя фиктивную переменную, мы можем переписать целевую функцию и ограничения в виде системы уравнений:

На основе системы мы можем записать симплекс-таблицу:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *z* |  |  |  |  | *b* |
| 1 | -700 | -900 | 0 | 0 | 0 |
| 0 | 500 | 3400 | 1 | 0 | 6000 |
| 0 | 250 | 200 | 0 | 1 | 500 |

В таблице можно выделить две группы переменных. Первая группа – базовые переменные, или просто базис. Переменная является базисной, если она имеет только одно неотрицательное значение в своем столбце. Базис обеспечивает допустимое решение на каждом шаге алгоритма. Основа подхода к решению заключается в том, чтобы свести значения всех небазисных переменных к нулю. На первом шаге базис представлен фиктивными переменными.

Вторая группа переменных, в столбцах которых более одного неотрицательного значения, называется небазисными переменными. Графическое отделение столбцов в таблице было использовано исключительно для наглядного разделения реальных переменных от фиктивных и никак не связано с тем, являются они базисными или нет.

Опишем алгоритм симплекс-метода, максимизирующий целевую функцию:

1. **Проверка оптимальности или нахождение ведущего столбца.**

* Если все коэффициенты в выделенной строке при небазисных переменных неотрицательны (коэффициенты в z-уравнении), то текущее базисное решение является оптимальным.
* В противном случае на следующей итерации в число базисных переменных вводим небазисную переменную , номер которой находится по правилу:

Столбец под номером s называется опорным столбцом симплексной таблицы.

1. **Проверка условия неограниченности решения задачи линейного программирования и нахождение опорной строки (опорного элемента).**

* Если в ведущем столбце симплексной таблицы s нет положительных коэффициентов, то значение задачи ЛП неограниченно (нет оптимального решения).
* В противном случае (в ведущем столбце имеются положительные элементы) в качестве базисной переменной, которая исключается из числа базисных, выбирается та переменная , для которой

Строка под номером r называется ведущей строкой, а элемент – опорным элементом.

1. **Преобразование симплексной таблицы.**

* Используя эквивалентные преобразования таблицы (процедуру Гаусса) пересчитываем таблицу так, чтобы ведущий элемент новой симплекс-таблицы стал равным 1, а все остальные элементы ведущего столбца – равными 0. Обозначим верхним индексом 1 элементы новой симплексной таблицы. Тогда формулы пересчета коэффициентов примут вид:
* Перейти к исследованию новой симплексной таблицы (новая итерация).

Смысл первого шага в том, чтобы найти переменную, которая может увеличить значение целевой функции. На втором шаге выбирается строка, которая позволит перейти к следующей вершине области допустимых решений с минимальным шагом. Это важно для минимизации риска выйти из области допустимых решений. На третьем шаге мы переписываем таблицу в свете выбранных опорного вектора и опорного столбца.

Приведем пример расчета простой задачи линейного программирования симплекс-методом.

Приводим задачу к каноническому виду:

Запишем симплекс-таблицу:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  | 0 | -5 | -3 | 0 | 0 |
|  | 4 | 1 | 1 | 1 | 0 |
|  | 10 | 5 | 2 | 0 | 1 |

Далее применяем описанный выше алгоритм:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  | 10 | 0 | -1 | 0 | 1 |
|  | 2 | 0 | 3/5 | 1 | -1/5 |
|  | 2 | 1 | 2/5 | 0 | 1/5 |

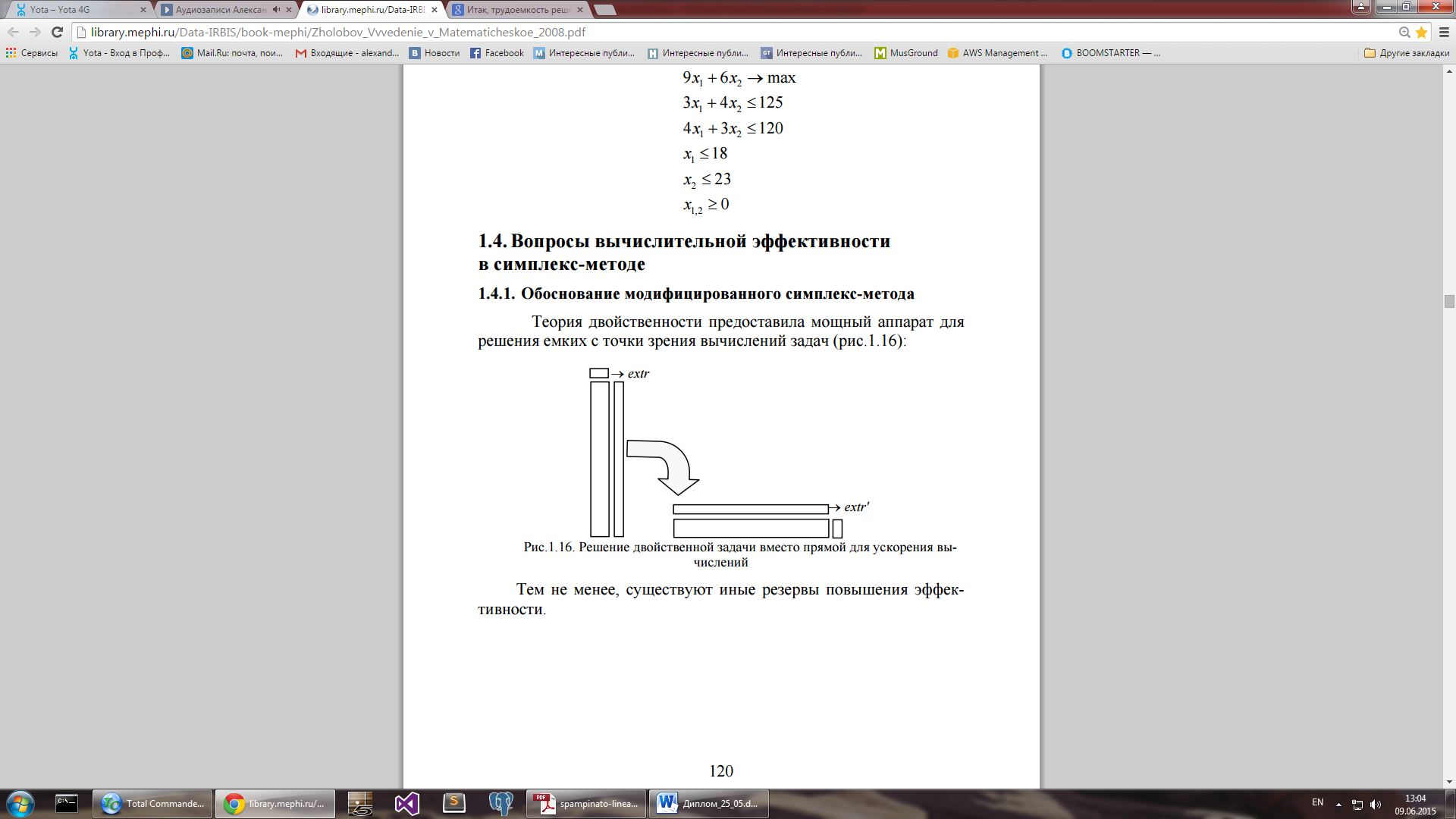
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  | 40/3 | 0 | 0 | 5/3 | 2/3 |
|  | 10/3 | 0 | 1 | 5/3 | -1/3 |
|  | 2/3 | 1 | 0 | -2/3 | 1/3 |

Ответом задачи будет:

## МОДИФИЦИРОВАННЫЙ СИМПЛЕКС-МЕТОД

### ОБОСНОВАНИЕ МОДИФИЦИРОВАННОГО СИМПЛЕКС-МЕТОДА

Теория двойственности предоставила мощный аппарат для решения емких с точки зрения вычислений задач (рис):



Тем не менее, существуют иные способы повышения эффективности. Рассмотрим пример.

не ограничены в знаке

Если решать задачу “в лоб”, то общее число переменных составит:

* (2+2) – наложение на переменные требования неотрицательности;
* 10 – количество дополнительных переменных для перехода от нестрогих неравенств к равенствам;
* 4 – количество искусственных переменных для получение полного набора единичных векторов

Всего – 18 переменных, таким образом, размерность задачи будет (10×18). Но можно поменять знаки смысла “≤” на противоположные (“≥”) и перейти к двойственной задаче

Размерность этой задачи составляет (2×10)

Итак, трудоемкость решения задач линейного программирования существенным образом зависит от соотношения количества ограничений *m* (или количества строк в матрице системы ограничений) и количества неизвестных *n* (или количества столбцов в этой матрице). Наиболее технологичными являются задачи, у которых , так как для решения подобных задач требуется значительно меньшее количество дополнительных (и искусственных) переменных, необходимых для придания им канонической формы и получения единичной базисной матрицы исходного решения. В этой связи использование результатов теории двойственности позволяет резко повысить эффективность решения задач линейного программирования. Действительно, вместо решения задачи , у которой , всегда можно перейти к двойственной задаче с и, решив последнюю, найти решение прямой задачи. Рассмотрим модифицированный симплекс-метод, ориентированный на эффективное решение именно тех задач, у которых . Этот метод в настоящее время является основой многих коммерческих пакетов линейного программирования.

Рассмотрим задачу линейного программирования:

Пусть известно некоторое опорное решение задачи и базис этого решения . В тех случаях, когда количество неизвестных *n* существенно превышает количество ограничений *m*, основной объем вычислений приходится на пересчет коэффициентов разложения небазисных векторов при переходе от одного опорного решения к другому – от одного базиса к другому, смежному с ним.

Зададимся вопросом, с какой целью осуществляется этот пересчет? Это делается только для того, чтобы вычислить оценки небазисных векторов с использованием известного выражения:

или

где – вектор коэффициентов целевой функции при базисных переменных задачи; – вектор-столбец коэффициентов разложения вектора по базису.

В дальнейшем будем считать, что – значение целевой функции ().

Учитывая, что где – обратная базисная матрица, выражение (1.88) можно переписать следующим образом:

. (1.89)

Если знать обратную базисную матрицу вычисление оценок? можно проводить по следующей схеме: сначала вычислить вектор

после чего – вектор оценок:

(1.91)

Вектор называется *вектором симплексных множителей*. В соответствии с приведенной схемой, для вычисления оценок не требуется знать координаты разложения соответствующих векторов по базису – требуется лишь знать обратную матрицу.

Учитывая тот факт, что решение начинается с единичной базисной матрицы, достаточно, начиная с первой итерации симплекс-метода, хранить разложения всех исходных единичных векторов по текущему базису. В этом случае в каждой симплекс-таблице будет находиться обратная матрица.

Для вычисления же обратных матриц в соседних базисах можно использовать основные формулы обычного симплекс-метода.

Пусть, например, из базиса выводится вектор . Вместо него в базис вводится вектор . Тогда элементы новой обратной матрицы вычисляются по формулам:

Таким образом, для вычисления новой обратной матрицы необходимо иметь только коэффициенты разложения вводимого в очередной базис вектора. В данном случае это

Итак, зная обратную базисную матрицу очередного опорного решения, можно, используя приведенную выше схему вычислений (1.90),(1.91), определить оценки всех небазисных векторов.

Если все оценки неотрицательны , процесс закончен: получено оптимальное решение задачи, координаты которого определяются коэффициентами разложения вектора .

Новая ситуация возникает в том случае, когда среди оценок есть отрицательные оценки.

В обычном симплекс-методе для каждой отрицательной оценки проверяется признак неограниченности сверху целевой функции: проверяются знаки всех коэффициентов . Если обнаруживается, что среди этих коэффициентов нет ни одного строго положительного, принимается решение о неразрешимости задачи. То есть для принятия этого решения нужно иметь разложения всех векторов с отрицательной оценкой по базису. В модифицированном же симплекс-методе коэффициенты неизвестны, поэтому используется несколько упрощенная схема проверки на неограниченность целевой функции.

Выберем одну из отрицательных оценок (максимальную по абсолютной величине или, что используется чаще, первую обнаруженную отрицательную оценку). Пусть это будет .

Найдем коэффициенты разложения вектора по текущему базису (соответствующая процедура называется *генерацией столбца*):

Если все , целевая функция не ограничена сверху на допустимом множестве. В противном случае по правилам симплекс-метода определяется вектор, который должен быть выведен из базиса: коэффициенты разложения вектора известны; известны и коэффициенты разложения вектора . Новизна ситуации заключается в том, что факт неограниченности сверху целевой функции на данной итерации может быть и не установлен, так как проводится проверка только одного вектора . Однако, рано или поздно, этот факт обязательно обнаружится.

### АЛГОРИТМ МОДИФИЦИРОВАННОГО СИМПЛЕКС-МЕТОДА

Дана задача линейного программирования:

Известно некоторое опорное решение задачи и базис этого решения . Алгоритм модифицированного симплекс-метода рассмотрим по шагам.

**Шаг 1**. Вычисляется матрица – обратная базисная матрица. Ввиду того, что в качестве исходного опорного решения обычно принимается решение с единичным базисом, это вычисление не производится, так как в этом случае .

Вычисляются коэффициенты разложения вектора по базису B: .

**Шаг 2.** С использованием выражений и вычисляются оценки , где – значение целевой функции.

**Шаг 3.** Если все то получено оптимальное решение задачи. Это решение представлено коэффициентами разложения вектора по базису *B*. Известно также оптимальное значение целевой функции – .

**Шаг 4.** Если среди оценок есть оценка , то в качестве претендента на введение в базис принимается вектор . Обычно это первый вектор, для которого вычисленная оценка отрицательная.

**Шаг 5.** Определяются коэффициенты разложения вектора по базису *B*. Для этого используется выражение:

Если все , то конец: задача не имеет решения, так как ее целевая функция не ограничена сверху на допустимом множестве. В противном случае для всех вычисляются отношения и из этих отношений выбирается минимальное. Пусть

т.е. из базиса следует выводить вектор .

**Шаг 6.** В базис *B* вместо вектора вводится вектор . По основным формулам пересчитываются элементы новой обратной матрицы, а также коэффициенты разложения вектора по новому базису и новое значение целевой функции. Далее выполняется шаг 2.

## 

## МЕТОД КАРМАРКАРА

### ОСНОВНАЯ ИДЕЯ

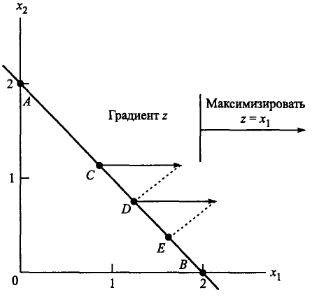
Рассмотрим очень простой пример.

 , при выполнении ограничения 

Вводим дополнительную (остаточную) переменную *.* Задача в стандартной форме будет записана следующим образом.

, при ограничениях 

Эта задача представлена на рис. 1. Ее пространство решений совпадает с от­резком прямой *АВ.* Направление возрастания целевой функции *z* совпадает с по­ложительным направлением оси *x1.*



Начнем поиск оптимального решения с произвольной *внутренней* (не крайней) точки *С* пространства допустимых решений (отрезок *АВ).* Градиент целевой функ­ции , в точке *С* показывает направление скорейшего возрастания функции *z.* Если мы зафиксируем какую-нибудь точку вдоль градиента и опустим из нее пер­пендикуляр на пространство допустимых решений, то получим новую точку *D* с лучшим значением целевой функции. Такое улучшение значения целевой функ­ции получено вследствие движения по направлению проекции *СD* градиента. Если мы повторим описанную процедуру для точки *D*, получим новую точку *Е,* которая будет еще ближе к точке оптимума *В.* Таким образом, если мы будем двигаться (но осторожно) в направлении проекции градиента, то, вероятно, рано или поздно "споткнемся" о точку оптимума *В.* Если же необходимо найти минимум целевой функции (вместо ее максимума), следует перемещаться в направлении, противопо­ложном проекции градиента, т.е. от точки *В* к точке *A*, где .

Данную процедуру нельзя считать алгоритмом (в обычном смысле), но она демонстрирует очень интересную идею. Чтобы воплотить эту идею в алгоритм, необходимо выпол­нить модификации приведенной процедуры, гарантирующие, что, во-первых, по­следовательность шагов, сделанных вдоль проекции градиента, не "перескочит" через точку оптимума, во-вторых, в общем *n*-мерном случае направления, указан­ные проекциями градиентов, не приведут алгоритм к неоптимальной точке (т.е. не "зациклят" алгоритм на неоптимальной точке). Если реализовать эти условия, то получим работоспособный алгоритм.

### АЛГОРИТМ КАРМАРКАРА

Существует несколько вариантов алгоритма Кармаркара. Мы рассмотрим ис­ходный вариант, предложенный его автором. Кармаркар предполагал, что задача ЛП приведена к следующему виду.

, при ограничениях 

Здесь все ограничения представлены в виде однородных уравнений, за ис­ключением ограничения , которое определяет *n*-мерный пра­вильный симплекс. *Симплексом* (*n*-мерным) называется выпуклая оболочка *n* точек X1, X2,..Хn, не лежащих на одной (n - 2)-мерной плоскости. Точки X1, X2,..Хn являются вершинами симплекса, при этом любую точку симплекса можно представить как выпуклую комби­нацию его вершин. Двухмерный симплекс — это отрезок, трехмерный — треугольник, четырехмерный — тетраэдр. В данном случае вершинами симплекса являются точки . Такой симплекс является частным случаем *правильного* симплекса.

Обоснованность алгоритма Кармаркара "покоится" на вы­полнении двух условий:

1. Вектор  удовлетворяет ограничениям .

2. .

Кармаркар предложил алгебраические преобразования, приводящие общую за­дачу ЛП к виду, представленному выше. Эти преобразования показаны в следую­щем примере.

Рассмотрим следующую задачу.

, при ограничениях 

С помощью дополнительной переменной *у3 > 0* преобразуем ограничение  вравенство: 

Теперь введем неравенство 

где *U* – достаточно большое положительное число, но такое, которое не удаляло бы ни одной допустимой точки исходного пространства решений. В данном приме­ре, исходя из равенства , достаточно взять *U,* равное 5. После введе­ния еще одной дополнительной переменной *у4* получаем



Теперь можно сделать уравнение  однородным, умножив его правую часть на *,* поскольку последнее соотношение равно 1. Таким обра­зом, после приведения подобных членов получаем



Чтобы преобразовать равенство  в уравнение, определяющее симплекс, введем новые переменные . Получаем следующую задачу ЛП.

, при ограничениях 

Теперь обеспечим выполнение условия, что точка , являю­щаяся центром симплекса, будет удовлетворять однородному уравнению. Для этого от левой части каждого однородного уравнения отнимем искусст­венную переменную с коэффициентом, равным алгебраической сумме всех коэффициентов левой части уравнения (в данном случае имеем *3+8+3-2=12*). Эта искусственная переменная также прибавляется к уравнению симплекса и в ви­де штрафа появляется в выражении целевой функции. В нашем примере искусст­венная переменная *x5*, войдет в задачу ЛП следующим образом.

,при ограничениях 

Для этой системы уравнений новый центр симплекса (*1/5, 1/5, .., 1/5*) является допустимым решением однородного уравнения. Значение константы *М* в выраже­нии целевой функции должно быть достаточно большим, чтобы привести переменную , к нулевому значению.

Последний пример показывает, что любую задачу ЛП можно с помощью преоб­разований привести к виду, который необходим для выполнения алгоритма Кармаркара. Но эти преобразования громоздкие и неочевидные, поэтому их редко ис­пользуют на практике. Существуют различные модификации алгоритма, которые не требуют обязательного выполнения второго условия ().

# РЕАЛИЗАЦИЯ НА CUDA

Ранее мы рассмотрели платформу GPU-адаптеров и особенности алгоритмов для параллельной реализации. В этой главе будут освещены особенности реализации алгоритма решения задачи линейного программирования на CUDA.

Но, во-первых, надо ответить на вопрос: для чего это нужно? Надо понимать, что задача линейного поддается распараллеливанию не самым лучшим образом. Так зачем искать параллельное решение этой задачи? Во-первых, это научный интерес. Понимая, что задача слабо поддается распараллеливанию, проведя эту работу можно сделать некоторые выводы относительно квази-параллельности задачи. С другой стороны, нельзя исключать вероятности, что в ходе исследования будут выделены такие части алгоритма, которые эффективно распараллеливаются.

Вторая причина более практическая. Мы увидели, что GPU обладают очень большой вычислительной мощностью, поэтому реализация на них алгоритма решения задачи линейного программирования может дать значимый прирост производительности, что очень важно, так как решение многих задач сводится к решению задачи линейного программирования.

Помимо основной цели, есть еще несколько проблем, которые надо будет разрешить. Во-первых, надо сделать расхождение между алгоритмом и его реализацией на GPU минимальным. Код должен давать понятное представление об алгоритме, который он реализует, и, конечно же, должен быть легко читаемым. С другой стороны, возникает вопрос, какой большой прирост производительности можно получить, применяя простой последовательно параллельный подход в программировании.

## ВЫБОР МЕТОДА

Выберем метод решения задачи линейного программирования, с лучшей производительностью при реализации на GPU, нежели в случае последовательной реализации. Джанг и Гриф [литература] публиковали хорошие результаты применения модифицированного симплекс-метода.

Как уже было сказано, метод внутренней точки имеет меньшую алгоритмическую сложность, но далеко не всегда на практике он оказывается быстрее симплекс-метода. Коммерческие вычислительные продукты основаны на обоих алгоритмах. Нам нужен алгоритм, подходящий для реализации на параллельной платформе видео-адаптеров. Для принятия решения о выборе алгоритма воспользуемся критериями, сформулированными Оуэнсом в [литература].

**Большие объемы вычислений**. Нас интересуют только большие задачи с сотнями переменных и ограничений, ибо именно они представляют практический интерес.

**Задача распараллеливается**. В алгоритме применяются алгебраические операции, к которым уже разработаны параллельные реализации.

**Масштабируемость важнее времени вычисления отдельной операции**.

Модифицированный симплекс-метод соответствует всем этим критериям. Важно учитывать, что отказ от стандартного симплекс-метода влечет за собой необходимость реализации операции инвертирования матрицы, которая необходима как для метода внутренней точки, так и для модифицированного симплекс-метода.

Приведем применяемый алгоритм симплекс-метода, записанный на псевдокоде:

## ОСОБЕННОСТИ ПРОГАММИРОВАНИЯ НА CUDA

#### В области параллельных вычислений принята следующая терминология:

* **Хост (host)** — CPU. Выполняет управляющую роль — запускает задачи на устройстве, выделяет память на устройстве, перемещает память на/с устройства. Использование CUDA предполагает, что как устройство так и хост имеют свою отдельную память.
* **Устройство (device)** — GPU. Не производит вычислений самостоятельно, запускается посредством вызова с CPU.

**Ядро (kernel)** — задача, запускаемая хостом на устройстве.

Таким образом, хотя речь идет о “параллельных вычислениях на видеокарте”, на самом деле вычисления ведутся и на CPU, и на GPU, причём GPU инструкции вызываются из CPU. Таким образом, можно вести не только параллельные вычисления на GPU, но и одновременно с этим проводить вычисления на центральном процессоре, запуская код на видеоадаптере асинхронно.

#### Основные этапы CUDA-программы:

1. Хост выделяет нужное количество памяти на устройстве.
2. Хост копирует данные из своей памяти в память устройства.
3. Хост стартует выполнение определенных ядер на устройстве.
4. Устройство выполняет ядра.
5. Хост копирует результаты из памяти устройства в свою память.

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

При этом оптимизацию можно вести с двух сторон. Во-первых, не все алгоритмы параллельны по своей природе. Некоторые могут не параллелиться вовсе, другие содержат некоторые этапы вычислений, которые можно вести параллельно. Поэтому многие привычные алгоритмы, которые показывают высокую эффективность в классических последовательных реализациях, могут быть совершенно непригодны для реализации на CUDA. В этом случае нужно либо подобрать новый алгоритм для решения задачи, либо реализовать существующий таким образом, что некоторые этапы вычислений велись параллельно, а некоторые последовательно.

Вторым способом оптимизации является изменение формата хранения данных. Зачастую изменение способа доступа к данным в объекте может значительно сократить время выполнения программы, так как несколько обращений к памяти по соседним регистрам на аппаратном уровне может быть объединено в одну транзакцию, за счёт чего и достигается прирост производительности.

#### Ядра

Рассмотрим более детально процесс написания кода для ядер и их запуска. Важный принцип – ядра пишутся как (практически) обычные последовательные программы – то есть создания и запуска потоков в коде самих ядер нет. Вместо этого, для организации параллельных вычислений GPU запустит заданное (как правило, достаточно большое) количество копий одного и того же ядра в разных потоках. И, возвращаясь к вопросу эффективности использования GPU – чем больше потоков вы запускаете (при условии, что все они будут выполнять полезную работу) – тем лучше. Код для ядер отличается от обычного последовательного кода в таких моментах.

1. Внутри ядер имеется возможность узнать «идентификатор» или, проще говоря, позицию потока, который сейчас выполняется — используя эту позицию мы добиваемся того, что одно и то же ядро будет работать с разными данными в зависимости от потока, в котором оно запущено. Кстати, такая организация параллельных вычислений называется SIMD (Single Instruction Multiple Data) — когда несколько процессоров выполняют одновременно одну и ту же операцию но на разных данных.
2. В некоторых случаях в коде ядра необходимо использовать различные способы синхронизации.

Каким же образом задается количество потоков, в которых будет запущено ядро? Поскольку GPU это все-таки **Graphics** Processing Unit, то это, естественно, повлияло на модель CUDA, а именно на способ задания количества потоков.

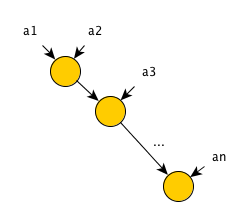
* Сначала задаются размеры так называемой сетки (grid), в 3D координатах: *grid\_x, grid\_y, grid\_z*. В результате, сетка будет состоять из *grid\_x\*grid\_y\*grid\_z* блоков.

Потом задаются размеры блока в 3D координатах: *block\_x, block\_y, block\_z*. В результате, блок будет состоять из*block\_x×block\_y×block\_z* потоков. Итого, имеем *grid\_x×grid\_y\*grid\_z×block\_x×block\_y\*block\_z* потоков. Важное замечание — максимальное количество потоков в одном блоке ограничено и зависит от модели GPU — типичны значения 512 (более старые модели) и 1024 (более новые модели).

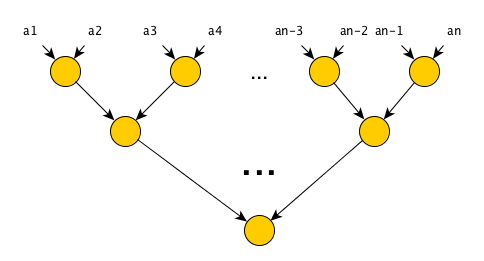
* Потом задаются размеры блока в 3D координатах: *block\_x, block\_y, block\_z*. В результате, блок будет состоять из*block\_x × block\_y × block\_z* потоков. Имеем *grid\_x × grid\_y × grid\_z × block\_x × block\_y × block\_z* потоков. Важное замечание – максимальное количество потоков в одном блоке ограничено и зависит от модели GPU – типичны значения 512 (более старые модели) и 1024 (более новые модели).
* Внутри ядра доступны переменные *threadIdx* и *blockIdx* с полями *x, y, z*– они содержат 3D координаты потока в блоке и блока в сетке соответственно. Также доступны переменные *blockDim* и *gridDim* с теми же полями — размеры блока и сетки соответственно.

Данный способ запуска потоков действительно подходит для обработки 2D и 3D изображений: например, если нужно определенным образом обработать каждый пиксел 2D либо 3D изображения, то после выбора размеров блока (в зависимости от размеров картинки, способа обработки и модели GPU) размеры сетки выбираются такими, чтобы было покрыто все изображение, возможно, с избытком — если размеры изображения не делятся нацело на размеры блока.

Для простоты понимания рассмотрим реализацию алгоритма *свертки* (reduce). Операция свертки выполняется над некоторым массивом элементов и определяется оператором свертки. Оператор свертки должен быть *бинарным*и *ассоциативным* – то есть принимать на вход 2 элемента и удовлетворять равенству *a× (b×c)=(a×b) ×c*, где *×* – обозначение оператора. Операция свертки над массивом из элементов *a1,...,an* определяется как *(...((a1×a2) ×a3)... ×an)*. Дерево выполнения последовательного алгоритма для реализации операции свертки имеет следующий вид:



Очевидно, что для данного алгоритма количество операций равно количеству шагов и равно *n-1=O(n)*. Для параллелизации алгоритма достаточно учесть свойство ассоциативности оператора свертки, и переставить скобки местами:*(...((a1×a2) ×a3)... ×an)=(a1×a2) × (a3×a4) ×...× (an-1×an)*. То есть, мы можем параллельно посчитать значения *(a1×a2)*, *(a3×a4)* и так далее, после чего произвести операцию свертки на результирующих значениях. Дерево выполнения такого алгоритма:



Количество шагов теперь равно *O(log(n))*, количество операций – *O(n)*. С помощью простой перестановки скобок получен алгоритм со значительно меньшим количеством шагов, при этом количество операций осталось прежним.

## ПОСЛЕДОВАТЕЛЬНАЯ РЕАЛИЗАЦИЯ

### ВСПОМОГАТЕЛЬНЫЕ БИБЛИОТЕКИ

Реализация выполнена в объектно-ориентированном стиле с использованием библиотеки BLAS (англ. Basic Linear Algebra Subprograms — базовые подпрограммы линейной алгебры)  в реализации от Intel. BLAS — стандарт де-факто интерфейса программирования приложений для создания библиотек, выполняющих основные операции линейной алгебры, такие как умножение векторов и матриц.

Впервые опубликован в 1979 году, и использован для создания больших пакетов, например LAPACK. Интенсивно используемые в высокопроизводительных вычислениях, высокооптимизированные реализации интерфейса BLAS были разработаны производителями аппаратного обеспечения, такими как Intel, а также другими авторами (например, ATLAS — переносимый самооптимизирующийся BLAS).

Также для чистоты кода и контроля ошибок была написана вспомогательная библиотека Matman.

В ней описаны функции выделения памяти:

int allocate\_array(float \*\*a, int m, int n) {

if( (\*a = (float \*)calloc(m\*n, sizeof(float))) == NULL )

return 1;

return 0;

}

int allocate\_int\_array(int \*\*a, int m, int n) {

if( (\*a = (int \*) calloc(m \* n, sizeof( int ))) == NULL )

return 1;

return 0;

}

Эти методы выделяют массивы памяти размерности под числа с плавающей точкой и целые числа соответственно, возвращая 0 в случае успешного завершения операции и 0 в противном случае.

Для промежуточной отладки и контроля за вычислениями был описан метод вывода содержимого массивов:

void display\_array(char \*name, float \*a, int m, int n) {

int i, j;

printf("Array %s:\n", name);

for(i=0;i<m;i++) {

for(j=0; j<n;j++)

printf("%f ", a[(i\*n)+j]);

printf("\n");

}

}

и аналогичный метод для массивов, содержащих целочисленные значения.

Для чтения исходных данных из файла также был написан метод:

int read\_array(FILE \*file, float \*a, int m, int n) {

int i, dim;

dim = m\*n;

for( i = 0; i < dim; i++ ) {

fscanf(file, "%f", &a[i]); //Get the ith-element of the matrix from

}

return 0;

}

Эти и ряд других вспомогательных методов были вынесены в отдельный модуль Matman.c и используются в теле основной программы в качестве вспомогательных функций.

Вторым вспомогательным модулем является Liblp.c. В него вынесены операции, специфические для алгоритма симплекс метода, например, операция определения индекса исключаемого столбца:

int leaving\_index(float \*t, int \*flag, int size) {

int i;

int minpos\_i = -1;

for(i=0; i< size; i++) {

if(minpos\_i < 0) {

if((flag[i] > 0) && (t[i] >= -EPS))

minpos\_i=i;

} else {

if((flag[i] > 0) && (t[i] >= -EPS) && (t[i] < t[minpos\_i]))

minpos\_i=i;

}

}

return minpos\_i;

}

или проведение некоторых промежуточных расчетов:

int compute\_E(float \*E, float \*a, float \*I, int size, int li) {

int i;

float qth = a[li];

if((qth >= -EPS) && (qth <= EPS)) {

printf("qth == 0....exit...\n");

return 1;

}

memcpy(E, I, size\*size\*sizeof(float));

for(i = 0; i < size; i++)

a[i] = -a[i]/qth;

a[li]=1/qth;

for(i = 0; i < size; i++)

E[(i\*size)+li] = a[i];

return 0;

}

### ОСНОВНОЙ МОДУЛЬ

Ход выполнения основной программы можно структурно представить следующим образом:

1. Объявление переменных для работы с данными (размерность задачи, коэффициенты целевой функции и т.д.)
2. Проверка корректности параметров командной строки.
3. Выделение памяти и чтение входных данных из файла:

allocate\_array(&c, 1, n);

read\_array(sourcefile, c, 1, n);

1. Вычислительный цикл, реализующий алгоритм модифицированного симплекс-метода, описанный выше.
2. Вывод результатов. Результатом является оптимальное решение или факт того, что решений нет.
3. Освобождение памяти. Происходит следующим образом:

free\_array(A);

В процессе вычислений ведутся замеры процессорного времени, затрачиваемого на операцию:

/\* Timing \*/

clock\_gettime(CLOCK\_REALTIME, &lv\_start);

/\* Timing \*/

extract\_column(A, A\_e, ev, n, m);

cblas\_sgemv(CblasRowMajor, CblasNoTrans, m, m, 1,

Binv, m, A\_e, 1, 0, alpha, 1);

compute\_theta(xb, alpha, theta, theta\_flag, m);

lv = leaving\_index(theta, theta\_flag, m);

/\* Timing \*/

clock\_gettime(CLOCK\_REALTIME, &lv\_end);

/\* Timing \*/

Учитывается время, затраченное на операции, заключенные между инструкциями clock\_gettime. Это дает возможность получить не только общее время выполнения расчетов, но и оценить ресурсозатраты на выполнение отдельных шагов алгоритма.

## CUDA РЕАЛИЗАЦИЯ

### БИБЛИОТЕКА CUBLAS

CUBLAS — реализация интерфейса программирования приложений для создания библиотек, выполняющих основные операции линейной алгебры BLAS (*Basic Linear Algebra Subprograms*) для CUDA. Он позволяет получить доступ к вычислительным ресурсам графических процессоров NVIDIA. Библиотека является самодостаточной на уровне API, то есть, прямого взаимодействия с драйвером CUDA не происходит. CUBLAS прикрепляется к одному GPU и автоматически не распараллеливается между несколькими GPU.

Основные функции библиотеки CUBLAS: создание матриц и векторных объектов в пространстве памяти GPU, заполнение их данными, вызов последовательных функций CUBLAS, и загрузка результатов из области памяти GPU обратно к хосту. Чтобы достичь этого, CUBLAS предоставляет вспомогательные функции для создания и уничтожения объектов в памяти GPU, и для записи данных и извлечения информации из этих объектов.

Для максимальной совместимости с существующими средами Fortran, CUBLAS использует хранения в столбцах и индексирование с 1. Так как C и C++ используют построчное хранение, приложения не могут использовать родные семантики массивов для двумерных массивов. Вместо этого, макросы или встроенные функции должны быть определены для того чтобы использовать матрицы при помощи одномерных массивов. Для Fortran’а код портирован на C механическим способом, при котором сохраняется индексирование с 1. В этом случае индекс массива из матричного элемента в строке *i*и в столбце *j*могут быть вычислены с помощью следующего макроса:

#define IDX2F(i,j,ld) ((((j)-1)\*(ld))+((i)-1))

Здесь ld — это размерности матрицы, в случае хранения в столбцах, является количеством строк. Для кода изначально написанного на С и С++, можно было бы использовать индексирование с 0, в этом случае макрос выглядит так:

define IDX2C(i,j,ld) (((j)\*(ld))+(i))

Из-за того что основные функции CUBLAS (в отличие от вспомогательных функций), не возвращают статус ошибки напрямую (из соображений совместимости с существующими библиотеками BLAS), CUBLAS предоставляет отдельную функцию, чтобы помочь в отладке, которая возвращает последнюю записанную ошибку.

### СТРУКТУРА ПРОГРАММЫ

Общая схема вычислений CUDA-реализации, разумеется, похожа на схему вычислений последовательной реализации, но есть особенности. Они связаны, во-первых, непосредственно с проведением вычислений на GPU, а во-вторых, с необходимостью копирования данных из оперативной памяти ПК в оперативную память GPU. Структурно схему можно представить следующим образом:

1. Объявление переменных для работы с данными (размерность задачи, коэффициенты целевой функции и т.д.)
2. Проверка корректности параметров командной строки.
3. Выделение памяти и чтение входных данных из файла:

allocate\_array(&c, 1, n);

read\_array(sourcefile, c, 1, n);

1. Объявление переменных для работы на GPU. Объявление переменных универсально для CPU и GPU, префиксы в названии переменных служат исключительно для простоты понимания кода:

float \*devA\_e, \*devalpha, \*devtheta;

Выделение памяти происходит посредством вызова библиотечных функций:

stat = cublasAlloc(m, sizeof(\*devy), (void \*\*)&devy);

if(stat != CUBLAS\_STATUS\_SUCCESS) {

if(stat == CUBLAS\_STATUS\_ALLOC\_FAILED)

printf("Memory allocation failed: lack of resources.\n");

else printf("Error in allocation.\n");

return 1;

}

При каждой попытке выделения памяти выполняется проверка успешности операции, в противном случае операция, неудачно завершившаяся на GPU может не возбудить исключения на CPU, тем самым выполнение программы будет нестабильно и может приводить к неопределенным последствиям.

1. Инициализация переменных на GPU посредством копирования данных из памяти CPU:

stat = cublasSetMatrix(m, n, sizeof(\*A), A, m, devA, m);

if(stat != CUBLAS\_STATUS\_SUCCESS) {

if(stat == CUBLAS\_STATUS\_MAPPING\_ERROR)

printf("Error accessing device memory.\n");

else printf("Setting error.\n");

return 1;

}

При каждом копировании также необходима проверка успешности операции, в противном случае результат работы программы будет непредсказуемым.

1. Вычислительный цикл, реализованный на GPU и состоящий из последовательного вызова CUDA-ядер. Ядра представляют собой реализацию некоторых операций алгоритма и выполняются параллельно, тем самым доставляя прирост производительности. Пример ядра:

\_\_global\_\_ void get\_idx(float \*f, int \*index, float \*val, int n) {

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

if(j == 0)

index[0] = -1;

\_\_syncthreads();

if(j < n) {

float diff = f[j]-val[0];

if(diff>=-EPS && diff<=EPS) atomicCAS(index, -1, j);

}

}

Ядра имеют префикс \_\_global\_\_, что сообщает о том, что код выполняется на стороне GPU. Вызываются ядра следующим образом:

get\_idx<<<numBlocks, BS>>>(a, devidx, devred, n);

где параметры в угловых скобках (<<<numBlocks, BS>>>) являются настройками топологии ядра. Иными словами здесь задается число параллельно выполняющихся операций, имеющих общий доступ к своему разделы разделяемой памяти.

1. Копирование результатов вычислений обратно на CPU и освобождение занятых ресурсов GPU посредством вызова библиотечной функции:

cublasFree(devc);

1. Вывод результатов. Результатом является оптимальное решение или факт того, что решений нет.
2. Освобождение памяти. Происходит следующим образом:

free\_array(A);

Дополнительно для GPU-реализации была написана еще одна вспомогательная библиотека cumatman.cu – аналог cumatman.c, имеющая подобную функциональность.

# ЧИСЛЕННЫЕ ИССЛЕДОВАНИЯ

## МЕТОДОЛОГИЯ ТЕСТИРОВАНИЯ

В ходе численных исследований были решены 1000 тестовых задач линейного программирования. Размерность задачи росла постепенно, чтобы можно было оценить разницу в производительности на разных размерностях задачи. Максимальный размер задачи с матрицей ограничений 2000 × 4000. Исходные данные генерируются с одинаковым числом переменных и ограничений. Также дополнительно генерируются фиктивные переменные для того, чтобы зада была в канонической форме. Таким образом, задача максимальной размерности содержала 2000 переменных, 2000 ограничений и 2000 фиктивных переменных.

Тестовые данные были записаны в файл в формате, описанном в разделе “Постановка задачи”. Время чтения данных из файла не входило ни в какие оценки скорости работы.

Важно иметь ввиду, что в условиях современного уровня развития вычислительной техники, сравнение времени выполнения ПО в миллисекундах не всегда корректно. В данной работе в оценках эффективности использовались замеры времени с точностью до наносекунд. Метод *clock\_gettime()* возвращает объект типа *timespec*, представляющий собой

struct timespec {

time\_t tv\_sec; /\* seconds \*/

long tv\_nsec; /\* nanoseconds \*/

};

Подсчет времени, затраченного на выполнение некоторых инструкций, происходит следующим образом:

...

clock\_gettime(CLOCK\_REALTIME, &start);

/\* instructions \*/

clock\_gettime(CLOCK\_REALTIME, &end);

nsec = end.tv\_nsec-start.tv\_nsec;

if(nsec < 0) {

nsec = 1E9+nsec;

end.tv\_sec-=1;

}

printf("Elapsed time: %.9f\n",

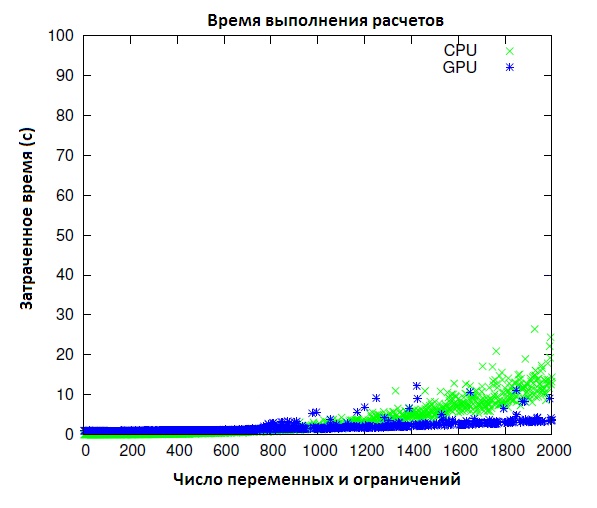
(double)(end.tv\_sec-start.tv\_sec)+(double)(nsec\*1E-9));

...

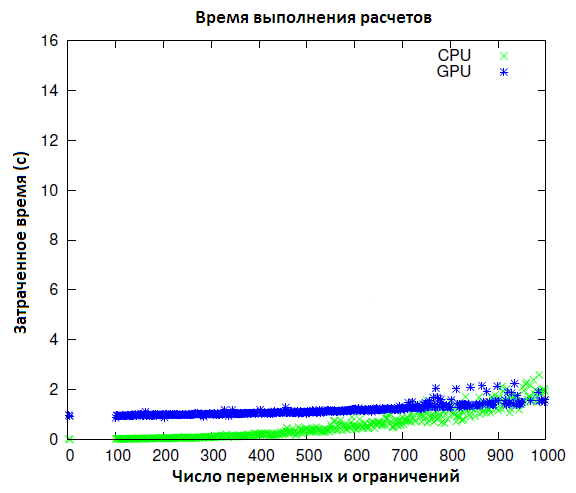
Для генерации тестовых данным был написан модуль popmat.c, принимающий на вход название файла и число ограничений, генерирующий файл с заданным названием, содержащий задачу линейного программирования в канонической форме в формате, описанном выше. Этот модуль был циклично запущен скриптом требуемое количество раз для генерации тестовых данных. Еще один скрипт запускал вычислительные модули и подавал на вход все тестовые данные по очереди. Результат визуализировался с помощью утилиты GnuPlot.

## РЕЗУЛЬТАТЫ ИЗМЕРЕНИЙ

На рис показан полный тест производительности. На нем отображена зависимость времени выполнения обеих реализаций от числа переменных и ограничений. На графике вид четкий тренд роста времени выполнения последовательного варианта программы при числе переменных больше 900, а на задачах меньшей размерности параллельная реализация несколько уступает последовательной.

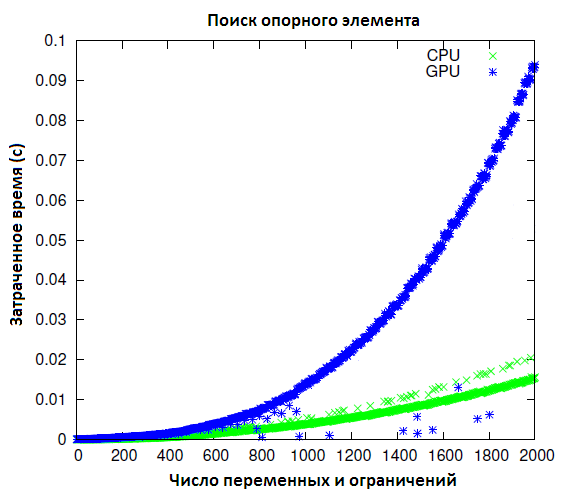


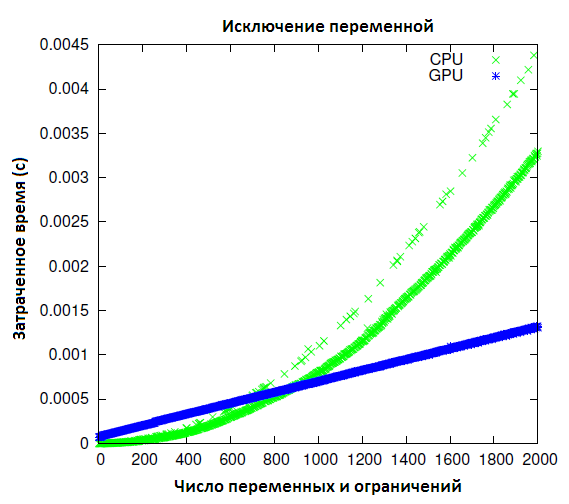
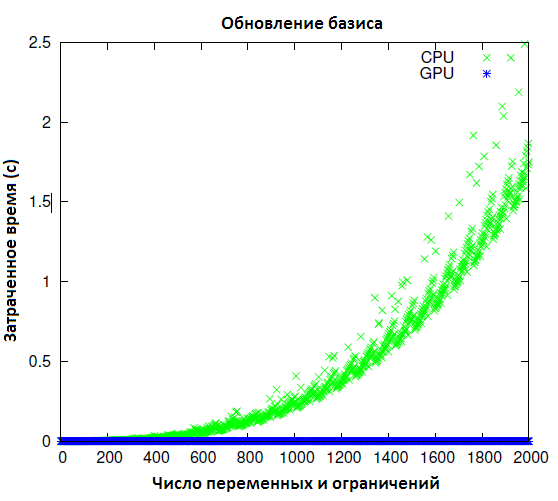
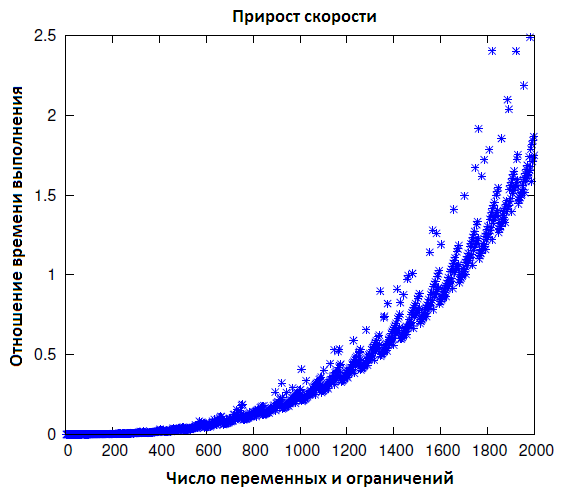
Для того чтобы понять, почему при малых размерностях CPU-решатель показывает лучшие результаты, обратимся к Рис. Это достаточно классическая картина при сравнении последовательной и параллельной реализации: в то время как время выполнения последовательной реализации растет практически линейно, поскольку большая размерность задачи требует лишь большее выполнение арифметических операций, GPU-реализация практически не показывает роста времени выполнения, но в целом работает медленнее. Это объясняется тем, что при такой размерности задачи затраты



на выделение памяти, копирование данных из памяти устройства на память хоста и наоборот сопоставимы с вычислительной трудностью алгоритма. Большая часть процессорного времени тратится не на вычисления, а на внутренние операции, никак не связанные с реализацией алгоритма. Именно поэтому при решении задач малой и средней размерности применение параллельных вычислений не всегда оправдано.

Также, как уже было сказано ранее, некоторые этапы алгоритма могут и не показывать прироста производительности при их распараллеливании. Более того, возможно даже падение производительности. Именно это и произошло на этапе поиска опорного элемента. Как видно на Рис.  
параллельная версия этого этапа алгоритма существенно проигрывает последовательной. Это также повлияло на то, что при малых размерностях параллельная реализация уступала последовательной. Но абсолютное значение затрачиваемого на эту операцию времени мало, поэтому с ростом размерности задачи этот проигрыш ощущается все в меньшей степени.



Иные же этапы алгоритма, напротив, показывали прирост производительности при использовании их параллельных реализаций. Так, например, были получены увеличения производительности на этапах исключения переменной из базиса и обновлении базиса, что отображено ра РИС Итоговый прирост производительности можно оценить по рисунку  
  
  


# ЗАКЛЮЧЕНИЕ

Все поставленные перед работой задачи, в общем, успешно выполнены. Был произведен анализ