# Оглавление

[Оглавление 1](#_Toc420416647)

[О видеоадаптерах 2](#_Toc420416648)

[История развития GPU 3](#_Toc420416649)

[Первое поколение GPU 3](#_Toc420416650)

[Второе поколение GPU 4](#_Toc420416651)

[Третье поколение GPU 4](#_Toc420416652)

[Четвертое поколение GPU 5](#_Toc420416653)

[Пятое поколение GPU 5](#_Toc420416654)

[Особенности архитектуры современных GPU 7](#_Toc420416655)

[NVIDIA 8](#_Toc420416656)

[Семейство GeForce 8](#_Toc420416657)

[Семейство Quadro 9](#_Toc420416658)

[Семейство Tesla 9](#_Toc420416659)

[Семейство Fermi 9](#_Toc420416660)

[Семейство Kepler 10](#_Toc420416661)

[Семейство Maxwell 11](#_Toc420416662)

[Особенности архитектуры 12](#_Toc420416663)

[Мультипроцессоры 12](#_Toc420416664)

[Организация памяти 12](#_Toc420416665)

[Вычислительные возможности 13](#_Toc420416666)

[Программная модель CUDA. Основные принципы 14](#_Toc420416667)

[Основные принципы 14](#_Toc420416668)

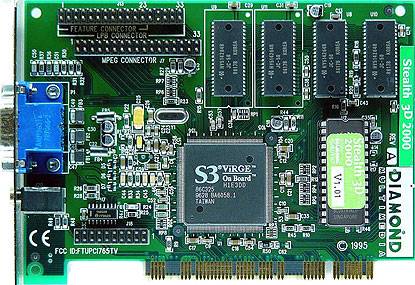
# О видеоадаптерах

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. 

## 

## История развития GPU

Трудно, наверное, сейчас представить, что были времена, когда видеоадаптеры, такие, например, как S3 ViRGE (Рис. 1), специализировались в основном на ускорении вывода 2D-графики. В то время считалось, что обработка трехмерных данных просто не целесообразна.

  
Модель S3 ViRGE (1995 г.)

Существует два противоположных мнения насчет того, верно ли связывать подобные модели видеоадаптеров и графические процессоры первого поколения, или же история развития GPU начинается с момента появления в середине 90-х годов прошлого столетия 3D-акселераторов. Сам термин GPU впервые был использован в августе 1999 года в отношении главного чипа видеокарты модели nVidia GeForce 256, основная функция которого заключалась в ускорении вывода трехмерной графики. Поэтому условимся, что в дальнейшем повествовании речь пойдет о поколениях GPU как 3D-ускорителях, а не об этапах развития видеоадаптеров как таковых. Однако не исключается, что между развитием и тех, и других существует достаточно тесная связь.

Каждый новый виток развития GPU представляет собой некое поколение, поэтому для начала следует ввести стандартизацию поколений. Понимать поколения можно по-разному – ниже предлагается лишь один из вариантов.

### Первое поколение GPU

GPU первого поколения являлись специализированными процессорами для ускорения операций с трехмерной графикой и предназначались для построения двумерных изображений трехмерных сцен в режиме реального времени. Для увеличения скорости данных операций использовались аппаратная реализация алгоритмов, в том числе отсечения невидимых поверхностей при помощи буфера глубины, и аппаратное распараллеливание. GPU первого поколения принимали на вход описание трехмерной сцены в виде массивов вершин и треугольников, а также параметры наблюдателя, и формировали по ним на экране двумерное изображение сцены для этого наблюдателя. Еще графические процессоры могли осуществлять текстурирование объектов, задание цвета вершин, а также интерполяционную закраску. Все предшествующие этапы выполнялись на CPU.

### Второе поколение GPU

Основоположником GPU следующего поколения принято считать главный чип видеокарты nVidia GeForce 256 (Рис. 8), появившийся в августе 1999 года, и привнесший в 3D-графику принципиально новые возможности. От предшествующих графических чипов его отличала поддержка технологии Transform&Lighting (T&L). Эта технология заключается в преобразовании координат вершин в плоские координаты, отображаемые на мониторе, и вычислении их освещенности. Это весьма ресурсоемкие и сложные вычисления, особенно при большом количестве вершин. Ранее они выполнялись на центральном процессоре, что отнимало значительную часть процессорного времени, либо на отдельных процессорах освещения и трансформации. Поэтому, благодаря появлению графических процессоров второго поколения, с одной стороны, с CPU снималась часть нагрузки, что позволяло использовать его для решения других, не менее важных задач. С другой стороны, появилась возможность увеличения количества объектов и степени их прорисовки, что позволило добиться нового уровня реалистичности в 3D-приложениях, а особенно в компьютерных играх.

### Третье поколение GPU

GPU третьего поколения добавили возможности программирования к графическим процессорам предыдущего поколения. Изначально фиксированный алгоритм вычисления освещенности и преобразования координат вершин был заменен на алгоритм, задаваемый пользователем. Позже появилась возможность писать программы для вычисления цвета пикселя на экране. По этой причине программы для GPU стали называть шейдерами (вершинные и пиксельные). Вершинные шейдеры позволяют определять параметры пикселя (освещенность, прозрачность, отражающую способность, координаты, текстуру), исходя из параметров вершин треугольника, содержащего его. Пиксельные шейдеры позволяют работать с каждым пикселем индивидуально, уже после проведения геометрических преобразований. Шейдеры для третьего поколения GPU писались на ассемблере графического процессора, их длина не превосходила 20 команд, не было поддержки команд переходов, а вычисления производились в формате с фиксированной запятой.

Обозначились основные производители дискретных графических процессоров: компании nVidia и ATI. К представителям 3D-акселераторов третьего поколения можно отнести nVidia GeForce 2 – 4 (Рис. 12), ATI Radeon 8500 – 9200.

### Четвертое поколение GPU

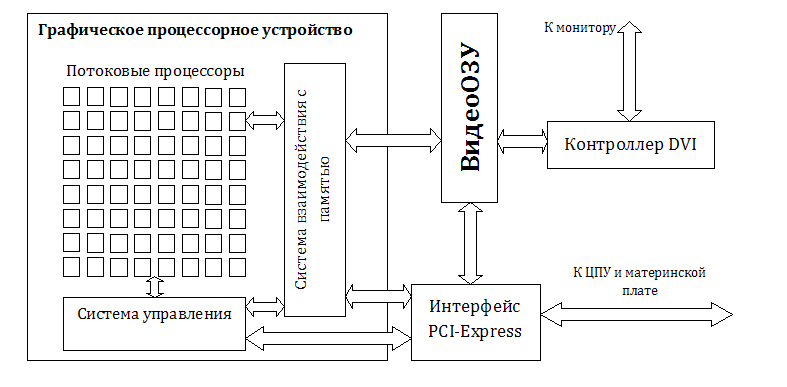
Представители следующего поколения GPU стали уже полностью программируемыми. В качестве основного интерфейса программирования выделился Direct3D, который первым обеспечил поддержку шейдеров (язык HLSL). OpenGL, начиная с версии 2.0, также добавляет поддержку высокоуровневого языка программирования шейдеров GLSL. Не стоит забывать и о языке Cg от nVidia. Появление операций ветвления и циклов позволило создавать более сложные шейдеры. В 2003 году на GPU впервые появилась поддержка вычислений с 32-разрядной точностью. Появились приложения, использующие графические процессоры для высокопроизводительных вычислений, таким образом, начало складываться направление общих вычислений GPGPU (англ. General Purpose GPU). Для программирования GPU был предложен SIMD-подход (англ. Single Instruction Multiple Data). Этот подход предполагает, что группа параллельно работающих процессоров осуществляют действия над разными данными, но при этом все они в произвольный момент времени должны выполнять одинаковую команду. В качестве примеров GPU четвертого поколения можно назвать графические процессоры моделей nVidia GeForce 5 – 7 и ATI Radeon 9500 – X800 (Рис. 14). На этом этапе наблюдается повсеместное вытеснение стандарта AGP более быстрым PCI Express.

### Пятое поколение GPU

GPU пятого поколения характеризуются расширенными возможностями программирования. На этом этапе GPU начинает поддерживать геометрические шейдеры, которые, в отличие от вершинных, позволяют обрабатывать не только одну вершину, но и целый примитив. Также появляется полная поддержка унифицированной шейдерной архитектуры, за счет которой осуществляется более гибкое использование ресурсов графического процессора. Например, в условиях с симуляцией тяжелой геометрии сцены унифицированная шейдерная архитектура может задействовать все блоки GPU для вычисления вершинных и геометрических шейдеров. И наоборот; когда геометрия не является сложной, а симулируется множество сложных пиксельных эффектов, все вычислительные блоки могут быть направлены на выполнение только пиксельных шейдеров. Кроме этого, GPU пятого поколения начинают поддерживать целочисленные операции, а также операции с двойной точностью. Появляются специализированные средства, позволяющие взаимодействовать с GPU напрямую, минуя уровень интерфейса программирования трехмерной графики. Поддержка 32-битных вычислений с плавающей запятой становится повсеместной, и это способствует активному росту направления GPGPU, для которого создаются средства программирования. Появляются потоковые библиотеки программирования GPU (RapidMind, Accelerator), а также первые коммерческие применения GPGPU (nVidia CUDA, AMD FireStream). Более того, отпадает необходимость в использовании специализированного физического ускорителя (англ. Physics Processing Unit, PPU) PhysX, поскольку GPU получили возможность аппаратно ускорять физические расчеты и освобождать, тем самым, CPU от излишних вычислений. Этот этап продолжается по настоящее время.

## 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

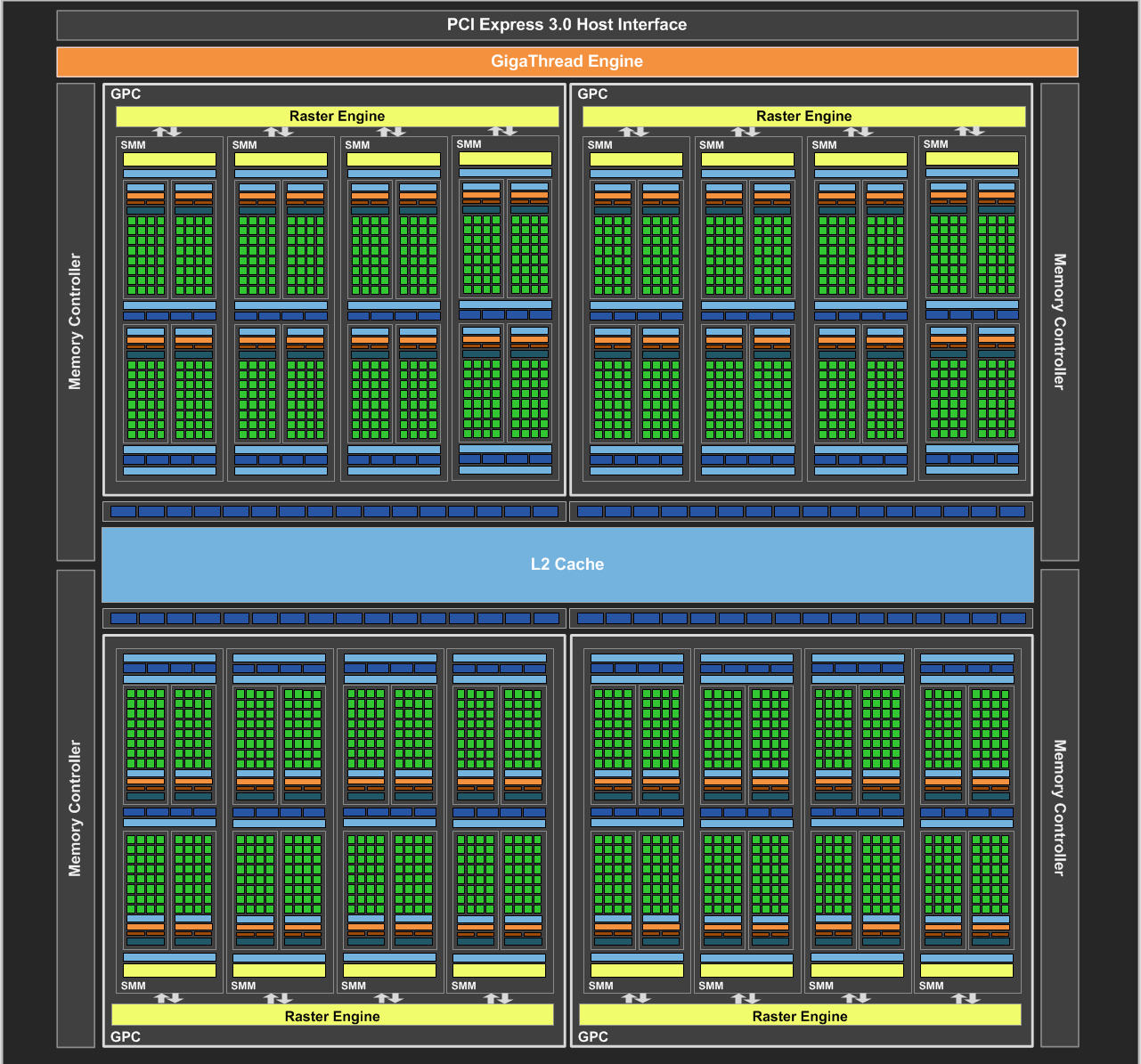
Kepler – архитектура, созданная для высокопроизводительных вычислений, с акцентом на энергоэффективности. Тогда как направленностью предыдущей архитектуры, Fermi, была чистая производительность, Kepler рассчитан на энергоэффективность, программируемость и производительность.

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 в еще 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.

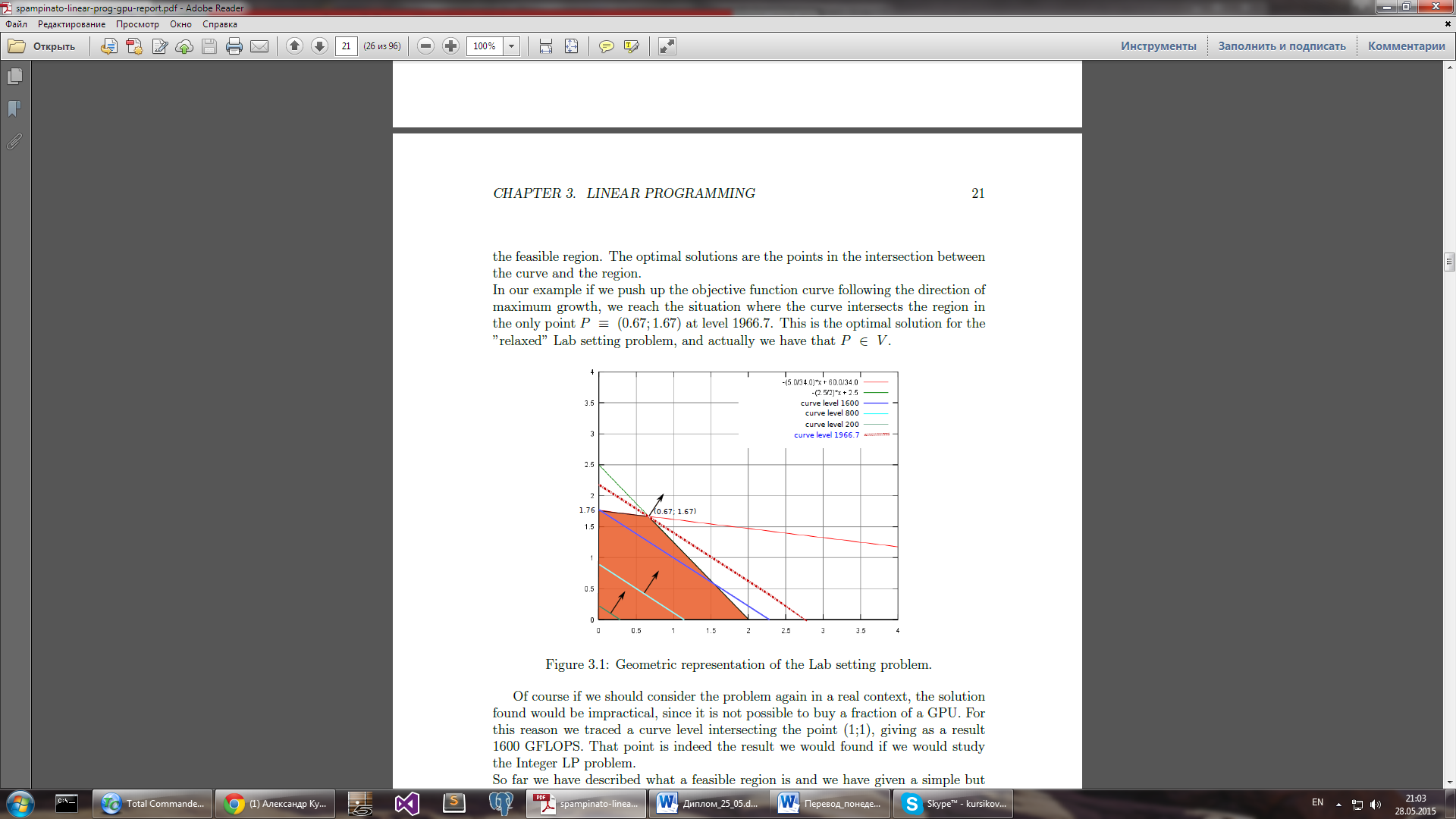
## Геометрическая интерпретация

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.

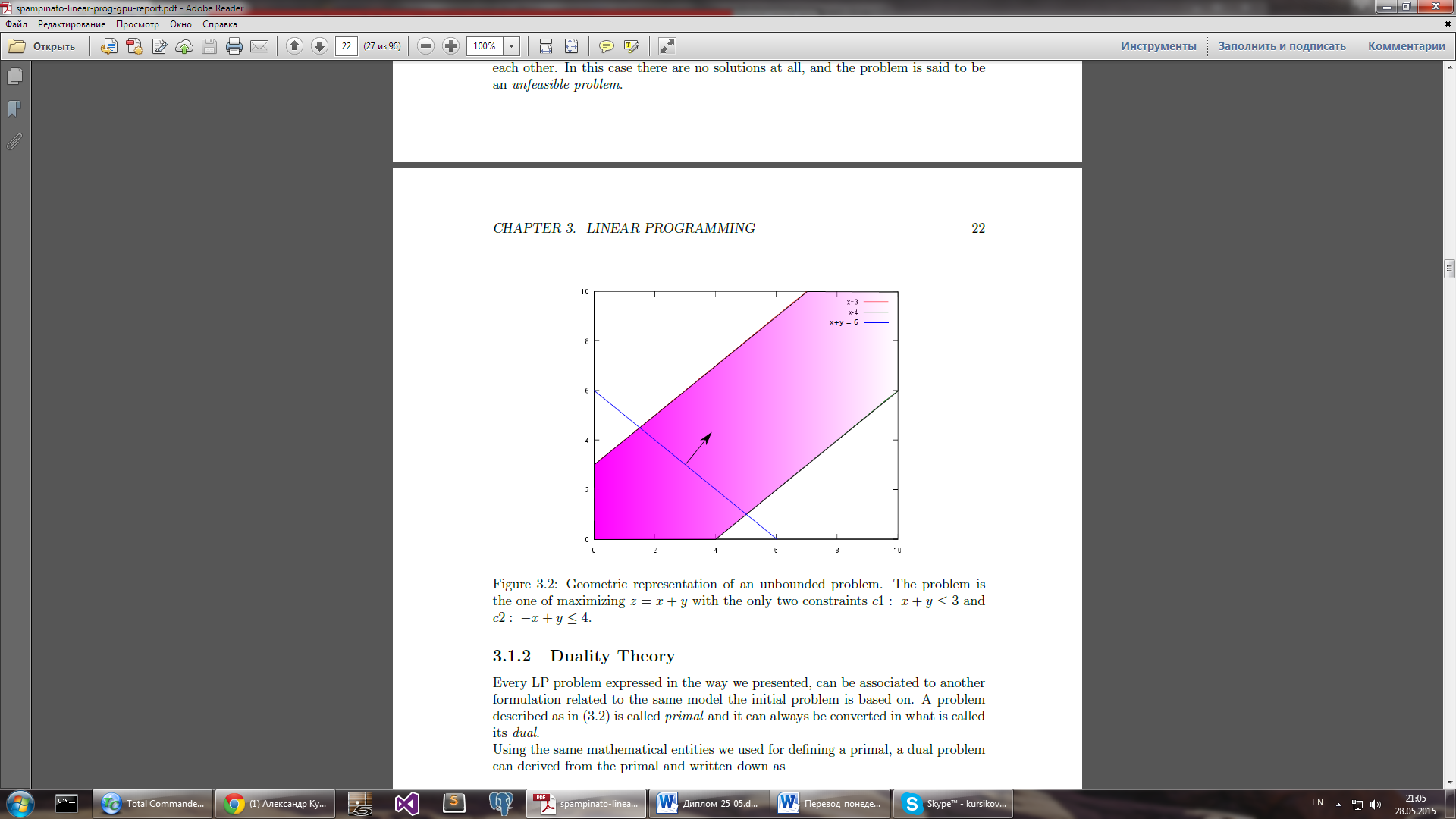
Посмотрим на Рис.1. В нашем случае множество вершин *V* образовано четырьмя точками . На рисунке изображено направление кривой целевой функции, перемещенной вдоль ее градиента. Мы искали максимум целевой функции. Двигаясь вдоль градиента в направлении роста, достигается ситуация, когда кривая целевой 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 задачей в канонической форме. 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.

В дальнейшем в работе не будет акцента на это отношение, поскольку будут рассматриваться техники, не использующие двойственность. Тем не менее, для глубины понимания приведем некоторые важные теоремы.

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 .

НУЖНЫ ПРАВИЛЬНЫЕ ФОМУЛИРОВКИ

## Методы решения задачи линейного программирования.

Теперь, когда мы сформировали представление о задаче линейного программирования, рассмотрим методы ее решения. Рассмотрим две группы методов: симплекс-методы и метод 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-методы

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 поиска решения. Симплекс-метод – итерационный метод, проходящий через вершины области поиска решения 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 измененная матричная версия, которая подходит для реализации на GPU.

Симплекс-метод

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:

Формулааааа

На основе системы мы можем записать симплекс-таблицу:

Формулааааа

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.

Формулааааа

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.

Алгоритм симплекс-метода, максимизирующий целевую функцию, представлен в Алгоритме 1:

Алгоритм

Можно разделить алгоритм на три шага:

1. Выделение опорного столбца.

Выбираем столбец с максимальным положительным значением из вектора коэффициентов целевой функции. В нашем случае нам придется выбрать наибольший из отрицательных коэффициентов, так как они все отрицательны. Это произошло из-за того что в (3.2.1) мы перенесли слагаемые из правой части в левую.

1. Выделение опорной строки.

Делим каждый элемент столбца *b* на соответствующее значение из опорного столбца. За опорный вектор берем тот, который дал наименьшее значение при делении.

1. Перестроение таблицы.

Проводим гауссово преобразование таблицы, при этом на пересечении опорного вектора и опорного столбца должен быть нуль.

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 поиска решений с минимальным шагом. Это важно для минимизации риска выйти из области поиска решения. На третьем шаге мы переписываем таблицу в свете выбранных опорного вектора и опорного столбца. Важно, что алгоритм меняет базис, но это нормально.