



ОТ ТРАНЗИСТОРА ДО ТРАНСФОРМЕРА

Настольная книга программиста

Железо • Компиляторы • Сети • Архитектура • ML

МАКСИМ БАЙТОВИЧ

Максим Байтович

**От транзистора до трансформера.
Настольная книга программиста**

«Автор»

2026

Байтович М.

От транзистора до трансформера. Настольная книга программиста
/ М. Байтович — «Автор», 2026

Что происходит между нажатием клавиши и пикселем на экране? Большинство программистов честно ответит: «магия». Эта книга снимает магию слой за слоем — от куска кремния, который умеет только отличать высокое напряжение от низкого, до трансформера, который пишет текст на человеческом языке. «От транзистора до трансформера» — сборник лекций профессора, выстроенный как единый путь: железо, программа, данные и системы, интеллект. Каждая лекция — это история из практики, разбор механики и ответ на вопрос «как это меняет твои решения». Почему код тормозит, хотя сложность хорошая? Почему гонка не воспроизводится на стенде? Почему компилятор не векторизовал цикл и почему база отказалась от индекса? Книга отвечает не заклинаниями, а ментальными моделями, которые можно проверить руками. Вместо таблиц и справочных простыней — связный рассказ у доски и домашние задания в духе «что измерить у себя»: снять промахи кэша, посчитать цену системного вызова, оценить стоимость одного ответа модели.

© Байтович М., 2026

© Автор, 2026

Содержание

Лекция 00. Введение. Зачем программисту кремний и внимание	5
Лекция 01. Транзистор и логика: кусок кремния, который решает	7
Лекция 02. От вентиля к процессору: сумматоры, память, такт	10
Лекция 03. Конвейер и спекуляция: как процессор обманывает время	12
Лекция 04. Иерархия памяти: кэш, латентность, локальность	14
Конец ознакомительного фрагмента.	15

Максим Байтович

От транзистора до трансформера. Настольная книга программиста

Лекция 00. Введение. Зачем программисту кремний и внимание

«Если вы не можете объяснить это просто — вы не понимаете этого достаточно хорошо.» — Ричард Фейнман

Когда студент впервые приходит ко мне на курс, он обычно уже умеет писать код. Он собирал проекты, выкладывал сервисы в облако, может быть, даже поднимал собственный сервер и гордился этим вполне заслуженно. Но если спросить его, что происходит между нажатием клавиши и пикселем на экране, он почти наверняка скажет «магия». Не потому, что он ленив, а потому, что современная разработка устроена так, чтобы этот вопрос не возникал. Фреймворк прячет сеть, сборщик прячет компилятор, облако прячет железо. Каждый слой честно делает свою работу и каждый слой делает вас чуть более беспомощным, когда что-то идёт не так.

Эта книга — попытка снять эти слои по одному. Мы пройдем путь, который инженеры прошли за семьдесят с лишним лет: от транзистора, который умеет только отличать высокое напряжение от низкого, до трансформера, который пишет текст на человеческом языке. Ни одно из этих устройств не умнее другого. Разница лишь в том, сколько раз простые операции повторили и как аккуратно их сложили вместе.

Почему это вообще нужно программисту, а не только инженеру железа? Потому что цена ошибки растёт вместе с уровнем абстракции. Пропущенный индекс в низкоуровневом коде стоит вам часа под санитайзером. Неверное предположение о кэше стоит дня профилирования. Ошибка в расчёте нагрузки на кластер стоит недели простоя и разговора, который вы не забудете. Чем дальше вы от железа, тем дороже обходится незнание железа и тем меньше шансов, что ошибку поймает за вас кто-то другой.

Я буду настаивать на одной мысли всю книгу. Программисту не нужно уметь проектировать процессор. Ему нужно понимать, какие договоры заключают между собой слои системы и что происходит, когда договор нарушен. Эти договоры не записаны в документации: они живут в головах инженеров и в поведении системы под нагрузкой. Наша задача — перевести их в слова, которые можно проверить экспериментом.

Каждая лекция устроена одинаково, и я предупреждаю об этом заранее. Сначала я рассказываю историю: реальную, из практики, часто с чужой или своей ошибкой в центре. Потом разбираю механику — что именно произошло и почему это было неизбежно. Затем показываю, как это знание меняет ваши повседневные решения. В конце каждой лекции есть домашнее задание, и это не упражнения ради упражнений: я прошу что-то измерить на вашей собственной машине или в вашем собственном проекте, потому что знание, которое вы не проверили руками, испаряется за две недели.

Про измерения скажу отдельно. В этой книге почти нет утверждений вида «это быстрее в десять раз» без оговорки, в каком контексте и на какой нагрузке. Инженерная правда всегда условна: быстрее для этого размера данных, для этого профиля доступа, на этом поколении железа. Как только вы перестаёте бояться этих оговорок, вы перестаёте спорить о технологиях в интернете и начинаете решать задачи. Я считаю это главным признаком взросления инженера: он перестаёт искать лучшую технологию и начинает искать подходящую.

Ещё одно предупреждение. Я буду говорить о вещах, которые в вашем стеке, возможно, не встречаются: о барьерах памяти, о протоколах консенсуса, о механизме внимания. Не спешите пропускать. Почти все эти темы возвращаются в неожиданном месте и обычно в тот момент, когда вы уже в проде и у вас горит дедлайн. Лучше встретить странный термин здесь, на странице, чем ночью в логах продакшена.

И последнее, о чём я прошу перед тем, как мы начнём. Держите под рукой свой проект — любой, пусть учебный. Все примеры из этой книги можно прогнать на нём. Когда вы увидите, как теория объясняет поведение вашего собственного кода, она перестанет быть теорией. Именно в этот момент книга делает свою работу, а не тогда, когда вы дочитали последнюю страницу.

Итак, двадцать лекций, один вечер на каждую. Начнём с самого низа — с куска кремния, который умеет принимать решение. От транзистора мы поднимемся к трансформеру, и по дороге вы увидите, что между ними меньше разрыва, чем принято думать.

Лекция 01. Транзистор и логика: кусочек кремния, который решает

«Компьютер — это глупейшее изобретение: он умеет только делать то, что ему велют, и делать это невероятно быстро.» — приписывается Дональду Кнуту

Начну с вопроса, который я задаю на первой лекции и на который почти никто не отвечает правильно. Что общего у процессора в вашем ноутбуке и у водопроводного крана? Студенты называют материалы, размер, цену. А общее — одно: и то и другое является клапаном, который либо пропускает поток, либо нет. Вся цифровая цивилизация построена на том, что мы научились делать такие клапаны микроскопическими, дешёвыми и переключаемыми напряжением, а не рукой. Этот клапан называется транзистор, и с него начинается всё.

Чтобы понять транзистор, не нужна квантовая механика. Достаточно одной идеи: у нас есть три вывода — исток, сток и затвор. Между истоком и стоком ток идёт только тогда, когда на затвор подано напряжение. Нет напряжения — тока нет. Есть — есть. То есть транзистор — это выключатель, которым управляет не палец, а электрический сигнал. А раз сигнал управляет выключателем, то один выключатель может управлять другим, а тот — третьим, и из этой цепочки управляемых переключателей можно собрать всё.

Почему именно два состояния? Потому что аналоговый сигнал, у которого «сколько угодно» значений, легко испортить: добавился шум — и значение уехало, и вы не отличите полезный сигнал от помехи. У сигнала с двумя уровнями есть запас прочности: пока шум не перевалил половину разницы между уровнями, мы уверенно читаем ноль или единицу. Цифровая схема — это способ получить надёжность из ненадёжных компонентов, и цена этой надёжности — отказ от всей середины между нулем и единицей. Это первое и главное решение, которое приняли инженеры, и всё остальное из него следует.

Из одного транзистора собирается простейший элемент — инвертор: подали единицу, получили ноль, и наоборот. Из двух транзисторов — элемент «и-не»: он выдаёт ноль только тогда, когда оба входа единицы. Кажется, это мелочь, но именно «и-не» — кирпич, из которого собирается всё остальное. Любой вентиль, любой сумматор, любой регистр, любой процессор — это в конечном счёте дерево из таких переключателей. Есть теорема, что любую булеву функцию можно выразить через «и-не», и на практике инженеры любят этот вентиль именно за универсальность: фабрике проще штамповать один тип ячейки.

Здесь стоит остановиться и посмотреть, что произошло. У нас был физический прибор с нелинейной характеристикой, а мы договорились считать его чёрным ящиком с двумя состояниями. Потом мы собрали из двух таких ящиков логический вентиль и договорились считать его функцией. Потом из вентиля — сумматор и считаем его арифметикой. На каждом шаге мы поднимались на уровень абстракции выше, и на каждом шаге нижний уровень продолжал честно работать. Это и есть инженерный метод: строить башню договоров, где каждый этаж опирается на предыдущий и не заглядывает ему внутрь. Запомните эту картину — мы будем возвращаться к ней в каждой части книги.

Теперь арифметика. Как из переключателей получить сложение? Очень просто, если записать сложение одного бита как таблицу: ноль плюс ноль — ноль, ноль плюс единица — единица, единица плюс единица — ноль с переносом в следующий разряд. Сумма бита — это «исключающее или», перенос — это «и». Обе функции собираются из вентилях за несколько штук. Сложите такие одноразрядные сумматоры цепочкой, передавая перенос, и вы получите сумматор на любое число бит. Всё, что делает ваш процессор с числами, в основе — эта цепочка переключателей, передающих перенос. Умножение — это сложение и сдвиги, деление — это вычитание и сдвиги. Арифметика — надстройка над переключением.

От арифметики до памяти один шаг. Если выход схемы завести обратно на вход через вентиль, управляемый тактом, получится ячейка, которая хранит бит, пока есть питание. Регистры, кэши, оперативная память — все это вариации этой идеи: цепь, которая держит собственное состояние. Разница между ними — в плотности, скорости и цене, но не в принципе. Память — это тоже переключатели, просто включённые так, чтобы сохранять, а не вычислять.

И вот тут появляется то, что определяет всю дальнейшую жизнь инженера, — цена переключения. Каждый раз, когда транзистор меняет состояние, он расходует энергию: заряжает и разряжает крошечную ёмкость затвора. Один переход — ничтожно мало, но переключателей в современном чипе десятки миллиардов, и частота — миллиарды переключений в секунду на каждый. Перемножьте — и получите десятки ватт на кристалле размером с ноготь. Отсюда главное физическое ограничение микроэлектроники: мы упираемся не в то, сколько вентилях умеем делать, а в то, сколько тепла умеем снять. Вся история последних двадцати лет — это история борьбы за то, чтобы переключать меньше и реже.

Эта борьба объясняет, почему закон Мура в прежнем виде закончился и почему вместо роста частоты появились многоядерные процессоры. Поднять частоту — значит переключать чаще, а значит, греть сильнее. Проще поставить два ядра на меньшей частоте, чем одно на большей: суммарно они сделают больше работы на том же тепловом бюджете. Отсюда и весь сдвиг в программировании: параллелизм перестал быть экзотикой и стал способом уложиться в ватты. Когда вы пишете код и он не использует все ядра, вы буквально отапливаете комнату половиной купленного вами транзисторного бюджета.

Есть и вторая цена, о которой программисты забывают, — цена перемещения, а не вычисления. Переключить вентиль дешево. Передать сигнал по проводу длиной в сантиметр — дорого по меркам транзистора: сигнал идёт *nanoseconds*, а за это время вентиль успел бы переключиться несколько раз. Поэтому на кристалле стараются держать связанные вещи рядом, и поэтому же иерархия памяти выстроена пирамидой: маленькое и быстрое рядом с вычислителем, большое и медленное — дальше. Мы подробно разберём это в лекции про память, но корень один: переключение дешевле перемещения, и вся архитектура компьютера подчинена этой асимметрии.

Вернёмся к уровню абстракции. Из вентилях мы поднялись к арифметике и памяти. Следующий этаж — автомат, который по такту читает команду, расшифровывает её и исполняет. Команда — это тоже просто биты, и расшифровка — это опять вентили, которые по коду операции открывают нужные пути данных. Никакой магии: процессор — это конечный автомат, собранный из переключателей, который читает свои собственные переключатели как данные. Граница между программой и железом — это тоже договор, и мы вернёмся к ней в лекции про ассемблер и в лекции про компилятор.

Что это знание меняет в вашей повседневной работе? Три вещи. Первая: когда вы слышите «это бесплатно, это же просто операция», вспоминайте, что за операцией стоят переключения и перемещения, и у того и другого есть цена. Вторая: когда выбираете алгоритм, спрашивайте не только про число операций, но и про то, сколько данных он гоняет туда-сюда, потому что перемещение часто дороже вычисления. Третья: когда ваш код медленный, не начинайте с микрооптимизаций — начните с вопроса, сколько раз вы заставляете железо переключаться и пересылать байты, которых можно было бы не пересылать.

Домашнее задание на эту лекцию — три измерения. Первое: найдите спецификацию своего процессора и выпишите три числа — количество ядер, базовую частоту и теплопакет. Разделите теплопакет на примерное число транзисторов и почувствуйте масштаб энергии на один переключатель. Второе: напишите на любом языке функцию, складывающую два числа побитово через логические операции, и сравните её время с обычным сложением на миллиарде итераций — вы увидите цену абстракции в обратную сторону. Третье: посчитайте, сколько

байт читает ваш простой цикл и сколько из них действительно нужно, — это первый шаг к пониманию памяти, о которой мы поговорим дальше.

Лекция 02. От вентиля к процессору: сумматоры, память, такт

«Простота — необходимое условие надёжности.» — Антуан де Сент-Экзюпери

На прошлой лекции мы остановились на том, что из переключателей собираются вентили, из вентилях — арифметика и память. Сегодня я покажу, как из этих кирпичей складывается машина, которая выполняет программу. Звучит как магия, но на деле это три идеи: состояние, такт и хранимая программа. Разберём каждую, потому что именно они отделяют калькулятор от компьютера.

Первое различие, которое нужно усвоить, — между схемами без памяти и схемами с памятью. Сумматор — схема без памяти: подали входы, подождали, пока сигнал пройдёт через вентили, получили выходы. У неё нет «до» и «после», есть только «сейчас». Она называется комбинационной. Но машина, которая выполняет программу, обязана помнить, где она находится: какую команду уже сделала и какую берёт следующей. Значит, ей нужно состояние. Состояние — это просто набор битов, которые схема хранит и обновляет. И вот тут появляется вторая идея — такт.

Зачем нужен такт? Представьте цепочку вентилях, где сигнал бежит от входа к выходу. Каждый вентиль вносит крошечную задержку. Если читать результат раньше, чем сигнал добежал до конца, вы прочитаете мусор. Можно договориться ждать «достаточно долго», но «достаточно» у каждой цепи своё, и собирать из таких цепей большую машину неудобно. Такт решает это радикально: мы вводим общий метроном и договор, что любое состояние читается и обновляется только по его удару. Между ударами сигнал успевает пройти через вентили и устаканиться. Так схема из хаотичного набора вентилях превращается в дисциплинированный автомат, где на каждом ударе метронома состояние переходит в следующее.

Элемент, который хранит один бит между ударами такта, называется регистром в узком смысле, или триггером. Группа триггеров — регистр в широком смысле. Из регистров собирается всё, что машине нужно помнить прямо сейчас. И теперь мы можем описать простейший процессор как автомат, который на каждом такте читает своё состояние, вычисляет из него следующее и записывает его обратно. Всё остальное — детали.

Третья идея — хранимая программа. До неё машины настраивались под задачу: чтобы решать другое, их перепрошивали руками, переставляя перемычки. Перелом — в том, что программу решили записывать туда же, где лежат данные, и дать машине возможность читать её как данные. Тогда «что делать дальше» — это просто число, лежащее в памяти, и машина сама выбирает его и расшифровывает. Расшифровка — опять комбинационная схема: по коду операции она открывает нужные пути данных. Граница между программой и данными исчезла, и это, пожалуй, самое важное инженерное решение в истории вычислений: машина получила универсальность.

Теперь соберём картинку целиком. У простейшего процессора есть несколько узлов. Счётчик команд хранит адрес следующей команды. Память команд по этому адресу отдаёт байты команды. Устройство управления расшифровывает их и выставляет сигналы. Арифметико-логическое устройство делает операцию над числами из регистров. Регистровый файл хранит операнды. И на каждом такте происходит один и тот же цикл: выбрать команду, расшифровать, исполнить, записать результат, сдвинуть счётчик. Этот цикл — выборка, декодирование, исполнение — и есть сердце любого процессора, от микроконтроллера до серверного чипа. Разница лишь в том, сколько команд за такт он умеет пропускать через это сердце и как хитро он их переставляет.

Здесь я всегда слышу вопрос: если всё так просто, почему процессоры такие сложные? Потому что простой цикл тратит уйму времени впустую. Пока команда исполняется, память команд простаивает. Пока результат пишется, арифметика простаивает. Инженеры заметили это и спросили: а что мешает держать в работе несколько команд одновременно, каждую на своей стадии? Это и есть конвейер — тема следующей лекции. Но сначала я хочу, чтобы вы почувствовали цену простого цикла, потому что из неё вырастает всё.

Цена измеряется в тактах. Если одна команда проходит через выборку, декодирование и исполнение за три такта, то программа из миллиарда команд займёт три миллиарда тактов. При частоте в гигагерц — три секунды. Кажется нормально, но мы платим за простой: на каждом такте две трети машины бездельничают. Умножьте простой на миллиарды транзисторов — и вы увидите, почему индустрия одержимо боролась за то, чтобы загружать всё одновременно. Конвейер, спекуляция, многоядерность — всё это ответы на один и тот же вопрос: как не давать транзисторам простаивать, не сжигая при этом лишние ватты.

Отдельно скажу про память, потому что именно она в следующий раз вас удивит. В нашей простейшей схеме мы молчаливо предполагали, что чтение из памяти укладывается в такт. На реальных машинах это правда только для самой ближней памяти. Стоит данным оказаться чуть дальше — и процессору приходится ждать. Это ожидание, а не вычисление, определяет производительность большинства реальных программ. Мы посвятим этому целую лекцию, но запомните уже сейчас: процессор — быстрая штука, которая большую часть жизни ждёт данные.

Что это знание меняет в вашей работе? Первое: когда вы пишете код, представляйте, что каждая ваша строка распадается на десятки машинных команд, и каждая команда проходит один и тот же цикл. Второе: когда вам говорят «это одна операция», уточняйте, сколько тактов и сколько обращений к памяти за ней стоит, потому что «одна операция» на языке языка программирования и «одна операция» на языке железа — разные вещи. Третье: когда вы видите, что программа медленная, первым делом спросите, не заставляете ли вы машину простаивать — ждать память, гонять лишние байты, — а уже потом оптимизируйте вычисления.

Домашнее задание на эту лекцию — два упражнения. Первое: соберите на бумаге или в любом логическом симуляторе одноразрядный сумматор и триггер и посчитайте, сколько вентиля ушло, — вы почувствуете, из чего складывается «железо». Второе: выпишите из спецификации своего процессора частоту и посчитайте, сколько тактов уходит на одну миллисекунду, а затем прикиньте, сколько команд он может исполнить за это время в идеальном случае. Сравните с тем, что показывает ваш код, и вы увидите разрыв между идеалом и реальностью — тот самый разрыв, который мы будем закрывать в следующих лекциях.

Лекция 03. Конвейер и спекуляция: как процессор обманывает время

«Преждевременная оптимизация — корень всех зол.» — Дональд Кнут

Начну с истории, которая отлично показывает, почему эта лекция существует. Студент принёс мне код, где один и тот же цикл на его ноутбуке работал вдвое медленнее, чем у одноклассника, при одинаковых флагах компилятора и одинаковых данных. Разница была в одной строке: у одного данные шли в одном порядке, у другого — в перемешанном. Алгоритм, сложность, число операций — всё одинаковое. А время разное. Причина — не в алгоритме, а в том, как процессор предсказывает ветвления и гоняет данные через конвейер. Перемешанный порядок ломал предсказатель и кэш одновременно. Эта лекция — про то, почему так происходит.

Вспомним простейший цикл из прошлой лекции: выбрать команду, расшифровать, исполнить, записать. Пока команда идёт через эти стадии, остальные узлы простаивают. Конвейер — это способ загрузить их все: на одном такте одна команда исполняется, следующая в это время расшифровывается, третья выбирается из памяти. Как на сборочной линии: каждая станция делает свой шаг, и с линии каждые такт сходит готовая команда, хотя каждая отдельная команда тратит несколько тактов. Пропускная способность вырастает в несколько раз при той же частоте — и без роста напряжения и тепла.

Но конвейер хруп. Он даёт полную скорость, только пока команды идут ровным потоком, как вагоны. Стоит потоку споткнуться — и вся линия останавливается. Спотыкания бывают трёх родов, и инженеры называют их конфликтами. Первый — конфликт по данным: команде нужен результат предыдущей, который ещё не готов. Второй — по структуре: две команды хотят одну и ту же шину или память. Третий, самый дорогой, — конфликт управления: программа дошла до ветвления, и неизвестно, какую ветку брать следующей, а конвейер уже должен загружать следующие команды и не может ждать.

С конфликтом по данным справляются просто: либо ждут, вставляя «пузырь» — пустой такт, либо пересылают результат напрямую из стадии исполнения в следующую команду, минуя регистр. Это называется пересылкой, и современные процессоры делают её почти всегда, так что зависимость по данным, отстоящим на две-три команды, почти бесплатна. Конфликт по структуре лечат дублированием шин и отдельными кэшами команд и данных. А вот конфликт управления — это то, где начинается настоящее волшебство и где ваш код может выиграть или потерять в разы.

Смотрите, в чём проблема. Около каждой пятой команды в типичной программе — ветвление. Когда конвейер доходит до него, следующие команды ещё не известны, но конвейер пустым быть не может — он теряет такты. Инженеры спросили: а что если угадывать? Если ветвь шла в одну сторону несколько раз подряд, вероятно, пойдёт туда и снова. Процессор заводит таблицу предсказаний, смотрит в неё и заряжает в конвейер команды с угаданного адреса. Если угадал — конвейер ни на такт не остановился. Если не угадал — всё, что успело загрузиться по неверной ветке, нужно выбросить, и конвейер начинает заново. Цена промаха — десяток-другой потерянных тактов.

Теперь посчитаем, почему это важно для вас. Допустим, предсказатель угадывает девять из десяти ветвей. Тогда на каждых десяти ветвях один промах ценой, скажем, пятнадцать тактов. Это в среднем полтора такта сверху на ветвь — заметная надбавка, но терпимая. А теперь представьте код, где ветвление зависит от данных, которые невозможно предсказать, — например, сравнение со случайным порогом. Тогда угадывание падает до половины, и вы платите промахами постоянно. Именно это и случилось у моего студента: перемешанные данные сле-

лали ветвление непредсказуемым, и половина конвейера уходила в сброс. Отсюда первое правило: горячие циклы любят предсказуемые ветви.

Второе правило следует из того же: данные, идущие ровным потоком, дружат не только с предсказателем, но и с памятью. Процессор не ждёт данные пассивно — он заранее тянет их в ближний кэш, предполагая, что вы пойдёте дальше по тому же адресу. Ровный порядок — предсказуемые адреса — попадание в кэш. Перемешанный порядок — промахи и ожидание. Два механизма, предсказание ветвей и предвыборка данных, оба построены на одной вере: будущее похоже на прошлое. Ваш код либо подтверждает эту веру, и тогда машина летит, либо разрушает её, и тогда машина спотыкается на каждом шагу.

Отсюда практические выводы, которые я прошу запомнить. Во-первых, в горячем коде предпочитайте данные, идущие последовательно, а структуры — компактные и выровненные. Во-вторых, если у вас есть ветвление, зависящее от данных, и вы знаете, что одна ветка сильно вероятнее, скажите об этом прямо: во многих языках есть подсказки ветвления, а иногда выгоднее переписать ветвление в арифметику без перехода. В-третьих, не бойтесь измерять: счётчики промахов предсказателя и промахов кэша есть в каждом современном процессоре, и profiler умеет их показывать.

Здесь уместно перейти к тому, что происходит, когда конвейера и предсказания мало, — к спекуляции и внеочередному исполнению. Идея: пока команда ждёт данные из памяти, процессор не обязан стоять — он может исполнять следующие команды, которые от этих данных не зависят, а результат придержать и подставить, когда данные придут. Для этого он переименовывает регистры, чтобы не затереть то, что ещё нужно, и ведёт учёт, какие команды завершены, но ещё не «обнародованы». Если спекуляция пошла по неверной ветке, все её результаты просто отбрасываются, и архитектурное состояние не меняется. Машина выглядит последовательной, а внутри — хаос из сотен команд, и это нормально.

Заметьте красивую симметрию: снаружи процессор — это скучный автомат, исполняющий команды строго по порядку. Внутри — параллельный, спекулятивный, переименовывающий регистры хаос, который лишь в конце притворяется последовательным. Граница между ними — договор, называемый архитектурным состоянием. Пока договор держится, программа корректна. Нарушить его может ошибка в железе, и такие ошибки — самые громкие аппаратные баги последних лет — возникают именно на этой границе, когда спекуляция оставляет след там, где следа быть не должно. Мы вернёмся к этому в лекции про безопасность.

И последнее, что связывает эту лекцию с вашей повседневностью, — почему мы не гоним частоту, а добавляем ядра. Конвейер и спекуляция дают выигрыш, пока программа умеет их кормить. Но у последовательного кода есть предел: часть команд зависит от предыдущих, и эту часть невозможно ускорить никаким конвейером. Когда упёрлись в этот предел и в тепловую стену, индустрия пошла вширь: больше ядер, каждое со своим конвейером. Поэтому современный выигрыш даёт не «более быстрый цикл», а «больше независимых потоков работы». Отсюда вывод для вас: параллелизм — это не мода, а единственный оставшийся способ тратить транзисторный бюджет, и код, который его не использует, оставляет половину машины без работы.

Домашнее задание — три измерения, от простого к сложному. Первое: найдите в своём профиле-анализаторе счётчик промахов предсказателя ветвлений и снимите его для двух версий одного цикла — с предсказуемым и со случайным ветвлением. Второе: прогоните один и тот же обход массива в прямом и в случайном порядке и сравните время и промахи кэша. Третье: запустите свою программу на одном ядре и на всех, посмотрите на ускорение и честно посчитайте, какая доля кода осталась последовательной. Эти три числа скажут вам о вашей программе больше, чем любые бенчмарки из интернета.

Лекция 04. Иерархия памяти: кэш, латентность, локальность

«Преждевременная оптимизация — корень всех зол, но преждевременный пессимизм о памяти — корень всех тормозов.» — из лекции

Начну с цифры, которая переворачивает представление о скорости. Операция в регистре процессора занимает около одного такта. Чтение из ближнего кэша — несколько тактов. Из дальнего кэша — десятки. Из оперативной памяти — сотни. Это не градация «быстро–медленно», это разные порядки величин: если обращение к регистру — это секунда, то обращение в память — это несколько дней. Программа, которая постоянно ходит в память, — это программа, которая большую часть времени ждёт. Вся эта лекция — о том, как устроена лестница, по которой данные спускаются к процессору, и как не заставлять её подниматься лишний раз.

Почему вообще нужна лестница, а не одна быстрая память? Потому что быстрые ячейки дорогие и крупные, а дешёвые — медленные. Инженеры давно поняли, что можно обмануть эту дилемму, если использовать свойство реальных программ — локальность. Программа обращается не ко всей памяти равномерно, а к небольшим участкам, и недавно использованное скорее всего использует снова, а рядом с использованным скорее всего лежит нужное дальше. Первое называется временной локальностью, второе — пространственной. Если это так, держим малый быстрый слой рядом с вычислителем для горячих данных, а большой медленный — для остальных. Так рождается пирамида: регистры, несколько уровней кэша, оперативная память, и дальше — диски и сеть.

Важно понять единицу, которой память разговаривает с процессором, — линию кэша. Процессор не тянет из памяти один байт: он тянет блок, обычно шестьдесят четыре байта, и кладёт его в кэш. Поэтому, если вы читаете массив подряд, первое обращение стоит сотни тактов, а следующие десятки элементов — почти бесплатно, потому что они приехали вместе с первым. Это и есть эксплуатация пространственной локальности. А если вы ходите по массиву большими прыжками, то на каждый прыжок тянете новую линию и выбрасываете предыдущую, и платите полную цену за каждый элемент. Отсюда первое правило: порядок обхода данных решает.

Здесь я показываю классический эксперимент, который прошу вас повторить. Возьмите двумерный массив и просуммируйте его двумя способами: по строкам и по столбцам. В одном языке и с одними данными разница достигает десятков раз, хотя число операций одинаково. Причина в том, как массив лежит в памяти: подряд по строкам. Обход по строкам тянет линию и использует её целиком. Обход по столбцам берёт из каждой линии один элемент и выбрасывает её. Машина одна и та же, данные одни и те же, а скорость разная — вся разница в локальности. Этот пример объясняет больше, чем любые лекции о микрооптимизациях.

Конец ознакомительного фрагмента.

Текст предоставлен ООО «Литрес».

Прочитайте эту книгу целиком, [купив полную легальную версию](#) на Литрес.

Безопасно оплатить книгу можно банковской картой Visa, MasterCard, Maestro, со счета мобильного телефона, с платежного терминала, в салоне МТС или Связной, через PayPal, WebMoney, Яндекс.Деньги, QIWI Кошелек, бонусными картами или другим удобным Вам способом.