ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ
ФГАОУ ВО НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет компьютерных наук
Образовательная программа «Прикладная математика и информатика»

Отчет о программном проекте

на тему: Портирование VCL на Rust

Выполнил:
Студент группы БПМИ 223 _____ Р.А. Гундарин
Подпись И.О. Фамилия

07.02.2024
Дата

Принял:
Руководитель проекта Дмитрий Витальевич Трушин
Имя, Отчество, Фамилия
доцент, к.ф.-м.н.
Должность, ученое звание
ФКН НИУ ВШЭ
Место работы (Компания или подразделения НИУ ВШЭ)

Дата проверки: 07.02 2024 _____ _____
Оценка по 10-ти бальной шкале Подпись

Москва 2024

Содержание

Аннотация

Современные высокопроизводительные процессорные архитектуры, такие как x86-64, ARM, PowerPC, поддерживают специальные инструкции для SIMD (**S**ingle **I**nstruction, **M**ultiple **D**ata — одиночный поток команд, множественный поток данных) вычислений [1]. Такие инструкции позволяют параллельно обрабатывать несколько наборов данных. Использование подобных инструкций позволяет значительно увеличить производительность вычислений, однако проблематично из-за платформоспецифичности и низкоуровневости. VCL [2] предоставляет удобный интерфейс для использования SIMD-инструкций в C++ на x86-совместимых платформах и программные реализации некоторых математических функций, использующие SIMD-вычисления. Данный проект посвящен написанию аналогичной по возможностям и интерфейсу библиотеки на языке Rust.

Ключевые слова: SIMD, Vector Class Library, Rust, кроссплатформенность, x86, ARM, AArch64, SSE, AVX, NEON.

1. Введение

1.1. Постановка задачи

Целью данного проекта является портирование на Rust части библиотеки Vector Class Library (VCL) [2]. В отличие от VCL, полученная библиотека должна быть кроссплатформенной. Архитектура написанной библиотеки должна быть достаточно гибкой, чтобы в нее можно быть добавлять поддержку других частей VCL в будущем. Код требуется покрыть тестами, написать сопутствующую документацию, сравнить по производительности с VCL и другими альтернативами.

1.2. Полученные результаты

В процессе выполнения проекта были изучены:
- общие принципы работы с SIMD (Раздел 2);
- наборы доступных SIMD-инструкций и интринсиков на архитектурах x86/x86-64 (SSE, AVX) и ARM (NEON);
- устройство библиотеки VCL, код которой использовался в качестве основы для данного проекта;
- доступные на текущий момент в языке Rust способы работы с SIMD (Раздел 3).

Непосредственно программная часть работы разбита на две независимые части:
- библиотека Vector Rust Library (VRL), содержащая портированные под Rust классы `Vec4f` и `Vec8f` из VCL, реализованная на базе стабильных интринсиков из модуля `core::arch` (Раздел 4);
- пакет `portable_simd_addons`, содержащий имплементации некоторых математических функций (тригонометрические и экспоненциальные на текущий момент) из VCL на основе существующего в Rust модуля для SIMD-вычислений `std::simd` (Раздел 5).

1.2.1. Vector Rust Library

На текущий момент в VRL реализованы типы `Vec4f` и `Vec8f` — векторы из 32-битных вещественных чисел размеров 4 и 8 соответственно. Общие методы вынесены в отдельные типажи. Большинство функций безопасные, но также доступны и `unsafe` методы, дающие пользователю больший контроль и в теории позволяющие добиться большей производительности.

VRL, в отличие от VCL, не привязана к x86-совместимой архитектуре, а является платформонезависимой. Это достигается с помощью условной компиляции: в зависимости от платформы и доступного набора SIMD-инструкций, используются различные имплементации структур и функций. Отдельные реализации есть для наборов инструкций AVX, SSE (x86/x86-64) и NEON (ARM). На остальных платформах векторные операции эмулируются скалярными.

Одним из основных недостатков модуля `std::simd`, который поддерживает большую часть функционала VRL, является его нестабилизированность. Как следствие, данный модуль нельзя использовать на стабильной версии компилятора. Поэтому VRL разрабатывалась без использования экспериментальных возможностей Rust, чтобы работать со стабильной версией компилятора.

Функционал VRL почти полностью покрыт юнит-тестами:
- для методов, общих для векторов разных размеров, написаны generic тесты;
- на все публиные методы есть доктесты — примеры из документации, которые проверяются на корректность;
- есть тест-бенчмарк, считающий скалярное произведение векторов разными способами.

Набор тестов автоматически запускается при каждом изменении на нескольких конфигурациях платформы и набора доступных инструкций в Github CI и Travis CI. Все публичные части библиотеки сопровождаются документацией.

На примере подсчета скалярного произведения сравниваются производительность безопасного и небезопасного интерфейсов VRL с VCL, крейтом `wide` и модулем `std::simd`.

1.2.2. `portable_simd_addons`

На текущий момент этом пакете реализованы:
- портированные из VCL тригонометрические и экспоненциальные функции;
- типаж битовых манипуляций с действительными числами в векторах;
- макрос для векторного вычисления значений многочлена по схеме Эстрина [3].

Так как `simd_addons` базируется на кроссплатформенном модуле `std::simd`, то так же не привязан к конкретной архитектуре.

Все написанные функции покрыты тестами: они сравниваются с исходными реализациями из VCL, а также со скалярными аналогами из стандартной библиотеки Rust. Тесты запускаются автоматически в Github CI. Прозводительность полученной библиотеки сравнивается с производительностью VCL и скалярных функций.

2. SIMD-вычисления

В этом разделе изложены базовые принципы SIMD-вычислений. В качестве примера используется набор инструкций SSE, однако описанные концепции применимы и к другим распространенным наборам SIMD-инструкций, такие как AVX, NEON (ARM), AltiVec (PowerPC) и другие. При написании использовался источник: [4].

2.1. Введение

SIMD-инструкции позволяют обрабатывать нескольно потоков данных параллельно одной инструкцией. Рассмотрим простой пример попарного сложения нескольких чисел:

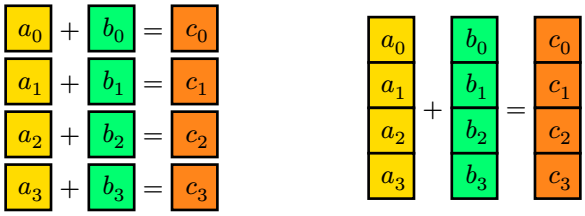

Рис. 1. Сложение четырех чисел: скалярно (слева) и векторно (справа)

«Обычные» (скалярные) операции складывают четыре числа по одному, тогда как SIMD-инструкция (векторная) выполняет все четыре сложения сразу. Векторные инструкции обычно оперируют специальными длинными регистрами, которые хранят несколько элементов одного типа («вектор»). Например, в 128-битный регистр можно записать два 64-битных целых числа или 4 вещественных числа одинарной точности.

SIMD-инструкции, в которых выполняются операции над элементами из разных векторов, называются *вертикальными*. В примере 1 показано вертикальное сложение векторов.

Не любую последовательность инструкций можно векторизовать. Чаще всего в одной SIMD-инструкциии над элементами векторов выполняется одна и таже операция. Так, последовательность инструкций на Рис. 2 нельзя заменить одной векторной.

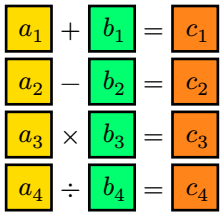

Рис. 2. SIMD-инструкция не может
выполнить несколько разных операций

2.2. Горизонтальные операции

Операции над элементами одного вектора, называются *горизонтальными*. Например, вычисление суммы элементов вектора — горизонтальная операция. Реализовывать такие операции можно с помощью доступных инструкций для перестановки элементов вектора:

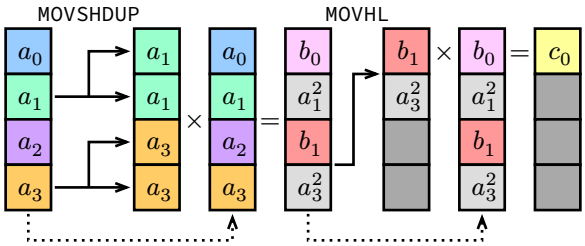

Рис. 3. Горизонтальное умножение вектора из четырех элементов

В последнем действии в примере 3 используется не векторное умножение, а скалярное: в наборе инструкций SSE можно делать скалярные операции над первыми элементами векторов.

2.3. Ускорение вычислений

Использование векторных инструкций позволяет значительно увеличить производительность вычислений. Простой пример: подсчет суммы элементов массива a вещественных 32-битных чисел. Для простоты будем считать, что размер n массива делится на 4 и больше нуля.

Обычный алгоритм делает $n - 1$ шагов цикла, по одной операции сложения за итерацию:

```rust
fn sum(a: &[f32]) -> f32 {
    let mut result = a[0];
    for i in 1..a.len() {
        result += a[i];
    }
    result
}
```

Используя векторные вычисления, можно уменьшить количество итераций в 4 раза. Будем отдельно считать суммы s_r элементов массива с индексами, сравнимых с r по модулю 4:

$$s_r := \sum_{i=0}^{n/4} a_{4i+r}, \quad 0 \le r \le 3$$

Разбив массив на векторы по 4 элемента, можно подсчитать все s_r в одном цикле длины $n/4$, выполняя по одному (векторному) сложению за итерацию (см. Рис. 4). Наконец, сумму всех элементов массива $s = \sum_{r=0}^{3} s_r$ можно получить как горизонтальную сумму полученного вектора сумм.

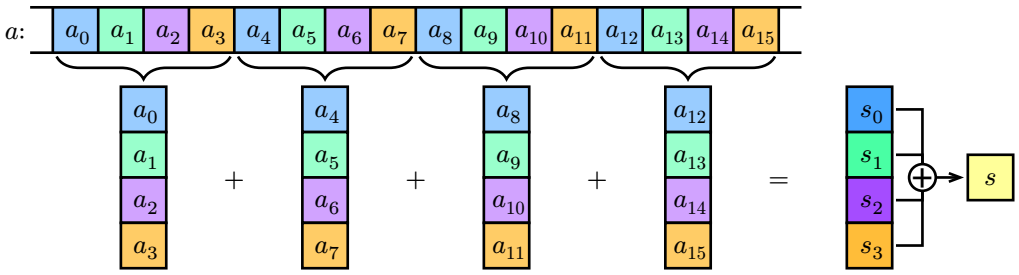

Рис. 4. Подсчет суммы элементов массива с использованием SIMD-инструкций

Использования такого подхода дает четырехкратное увеличение производительности на современных x86-совместимых процессорах: хотя количество подсчитанных сложений осталось тем же, уменьшилось количество выполняемых для этого инструкций.

Стоит отметить, что из-за неассоциативности сложения чисел с плавающей запятой описанные скалярныи и векторные подходы дают разный результат. При этом в векторном погрешность вычислений в будет меньше — $(1 + \varepsilon_{\text{machine}})^{n/4+O(1)}$ против $(1 + \varepsilon_{\text{machine}})^{n+O(1)}$ — благодаря использованию четырех переменных для суммирования вместо одной. В C/C++ можно заставить компиляторы `clang` и `GCC` считать вещественную арифметику ассоциативной с помощью флага `-funsafe-math-optimizations`. В этом скалярный код будет векторизован компилятором автоматически. В Rust аналогичного поведения можно добиться, используя `core::intrinsics::fadd_fast`. В более сложный случаях компилятор не всегда может автоматически векторизовать скалярный код.

2.4. Условные вычисления

Часто вычисления требуют использования условных переходов. Однако их использование в SIMD-вычислениях невозможно, т.к. для разных элементов одного вектора может потребоваться выполнять разные инструкции. Поэтому условные вычисления в парадигме SIMD нужно организовывать без ветвлений («branchless»).

Векторные инструкции сравнения возвращают *маски*, содержащие результат поэлементного сравнения векторов (см. Рис. 5). Обычно маски — это векторы того же размера, булевые значения в которых занимают по столько же бит, сколько занимали элементы исходных операндов, причем биты одного булевого значения одинаковы. Такое представляние логических значений позволяет использовать побитовые операции например. Например, по такой маске можно одним побитовым «И» занулить число по условию, записанному в маске.

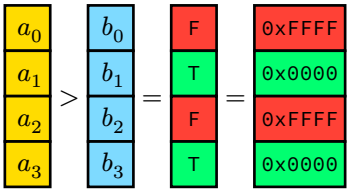

Рис. 5. Векторное сравнение векторов из четырех элементов по 32 бита: `if a[i] > b[i]`
`T` — логическое `True`, `F` — логическое `False`

Самый простой способ организации вычислений без ветвлений — вычислять все возможные ветви, а затем по маске условия выбирать нужные элементы из полученных векторов (см. Рис. 6). В SSE есть семество инструкций `BLENDV` для выбора элементов из двух векторов по маске.

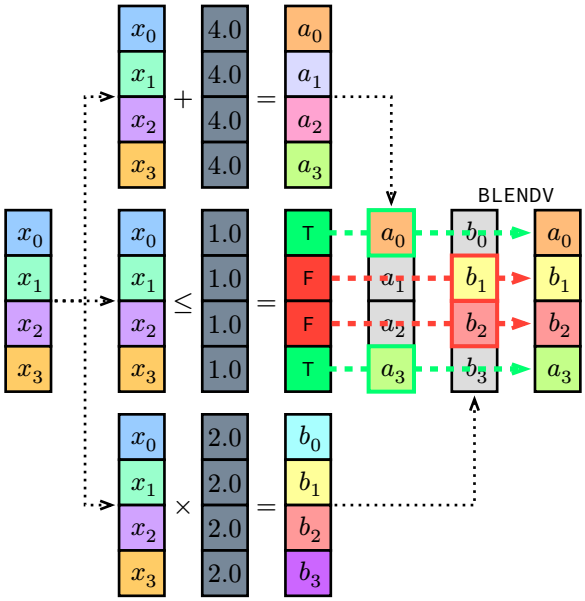

Рис. 6. Векторизированное вычисление функции с условием: $f(x) = \begin{cases} x+4, & x \le 1 \\ 2x, & x > 1 \end{cases}$

Благодаря суперскалярности современных CPU — возможности выполнять несколько инструкций параллельно на одном ядре — время вычисления обоих ветвей вместо одной будет меньше суммы временён вычисления этих же ветвей по отдельности. Если ветви небольшие и используют различные инструкции, то вычисление дополнительной ветви может вообще никак не отразиться на производительности. Так, в примере на Рис. 6 векторные сравнение, сумма и произведение на современных процессорах будут вычисляться параллельно.

3. Обзор существующих решений

3.1. Portable SIMD

Стадартная библиотека языка предоставляет удобный кроссплатформенный API для доступа к SIMD-вычислениям — модуль `std::simd` [5]. Под капотом используются интринсики, предоставляемые LLVM, что дает компилятору обширные возможности для оптимизаций. На текущий момент у данного решения есть один недостаток: `portable_simd` является эксперементальным API, что не позволяет использовать его со стабильной версией компилятора.

Данный модуль предоставляет структуры и базовые операции для векторов всевозможных примитивных типов и масок. Однако для вещественных векторных типов доступны только простые математические функции, такие как извлечение квадратного корня или взятие обратного, которые поддерживаются аппаратно на распространенных архитектурах. Поэтому вторая часть (Раздел 5) этой курсовой работы заключается в реализации более сложных функций на основе `std::simd`.

3.2. `wide`

В открытом доступе есть крейт `wide` [6], предоставляющий `safe` интерфейс для SIMD вычислений. Он кроссплатформенный и оптимизирован для платформ x86, NEON и WASM. Однако данное решение не поддерживает работу с масками (булевыми векторными типами), которые являются ключевым инструментом для организации

условных SIMD вычислений. Этот крейт сравнивается с написанной в ходе данного проекта библиотекой в бенчмарке, считающем скалярное произведение (см. Раздел 4.4).

3.3. Использования существующей C/C++ библиотеки

В C++, помимо библиотеки VCL, существует еще ряд SIMD-библиотек [7], [8]. В Rust существует множество способов для обращения к коду, написанному на C/C++: например, крейты `cxx` [9] и `cpp` [10]. Однако такой подход будет крайне неэффективным, т.к. компилятор не сможет инлайнить функции, написанные на другом ЯП, что является основным способом оптимизации векторизованного кода. Кроме того, на текущий момент передача SIMD-объектов напрямую в функции из C/C++ невозможно/вызывает неопределенное поведение, а RFC [11], добавляющий такую возможность, не реализован.

4. Vector Rust Library

4.1. Функциональные требования

Бибилиотека должна предоставлять пользователю структуры `Vec4f` и `Vec8f`, представляющие векторы из 4 и 8 вещественных 32-битных чисел соответственно, поддерживающие:

- базовые арифметические операции (`+`, `-`, `*`, `/`);
- совмещенное умножение-сложение (FMA);
- округление;
- вычисление горизонтальной суммы;
- различные способы загрузки из памяти:
  - безопасные: загрузка из массива, слайса, префикса слайса;
  - небезопасные (`unsafe`): по указателю, по выровненному указателю;
- аналогичные способы записи вектора в память;
- обращение к отдельным элементам вектора;
- преобразование в соответствующий нативный для платформы SIMD-тип из него;
- конструкторы:
  - поэлементные;
  - заполняющий вектор одним значением;
  - для `Vec8f`: «склеивающий» два `Vec4f`.

4.2. Нефункциональные требования

Библиотека написана на языке Rust и не использует нестабильные возможности языка, что позволяет использовать её со стабильной версией компилятора. Библиотека собирается и тестируется `stable` версии компилятора `rustc` версии `1.75`.

Разработка ведется с помощью системы контроля версий Git. Код форматируется `rustfmt` версии `1.7` и проверяется линтером `clippy` версии `0.1.75`. Документация генерируется с помощью `rustdoc` версии `1.75`.

- Исходный код выложен на Github: https://github.com/NamorNiradnug/vector-rust-library
- Крейт опубликован на `crates.io`: https://crates.io/crates/vector-rust-library
- Документация:
  - К текущей версии из репозитория: https://namorniradnug.github.io/vector-rust-library/vrl/index.html
  - К последней опубликованной версии: https://docs.rs/vector-rust-library/0.1.0/vrl/

4.3. Описание структуры библиотеки

Архитектура библиотеки продумана таким образом, чтобы минимизировать количество платформоспецифичного кода с одной стороны, и иметь возможность использовать наиболее эффективные интринсики для всех функций с другой.

Одной из основных проблем стала необходимость унифицировать интерфейс структур при сборке под все поддерживаемые платформы. У данной проблемы были разные решения.

1. Писать реализации всех методом под все платформы с одном модуле и использовать условную компиляцию локально в каждом методе. Такой подход в использовался изначально.

```rust
/// vec4f.rs
struct Vec4f(...);
impl Vec4f {
    fn broadcast(value: f32) -> Self {
        #[cfg(target_feature = "sse")]
        Self(core::arch::x86_64::...)
        #[cfg(target_feature = "neon")]
        Self(core::arch::neon::...)
    }
}
```

При таком подходе падала читаемость кода про увеличении числа поддерживаемых платформ, так как очень много места занимали инструкции условной компиляции, а не основной код.

2. Имплементация структуры и её методов, использующий интринсики, пишется в отдельном модуле. Затем этот платформозависимый модуль подключается в общий, структура вектора «прокидывается» и затем в этом платформонезависимом модуле пишутся методы, не использующие интринсики напрямую.

```rust
/// vec4f/neon.rs
use core::arch::neon::*;
pub struct Vec4f(float32x4_t);
impl Vec4f {
    pub fn broadcast(value: f32) -> Self { ... }
    pub unsafe fn load_ptr(addr: *const f32) -> Self { ... }
    // ...
}
```

```rust
/// vec4f/sse.rs
use core::arch::x86_64::*;
pub struct Vec4f(__m128);
impl Vec4f {
    pub fn broadcast(value: f32) -> Self { ... }
    pub unsafe fn load_ptr(addr: *const f32) -> Self { ... }
    // ...
}
```

```rust
/// vec4f/mod.rs
cfg_if::cfg_if! {
    // в зависимости от платформы подключаем нужный модуль
    if #[cfg(sse)] {
        mod sse;
        pub use sse::Vec4f;
    } else if #[cfg(neon)] {
        mod neon;
        pub use neon::Vec4f;
    }
}
impl Vec4f {
    pub fn load_checked(array: &[f32]) -> Self { /* check and load_ptr */ }
    // ...
}
```

У такого подхода есть серьезные проблемы:
- единость интерфейса между платформами никак не гарантируется инструментами языка и может быть проверена только вручную или написанными тестами;
- документацию придется либо дублировать, либо писать только для какой-то одной платформоспецифичной реализации, но с таком случае в IDE она не будет отображаться на остальных платформах.

3. Доработка предыдущего подхода. Для каждой платформы имплементируется своя непубличная структура (`VecXBase`). Публичная структура содержит в себе платформзасисимую и «пробрасывает» её публичные методы:

```rust
cfg_if::cfg_if! {
    // в зависимости от платформы подключаем нужную приватную структуру
    if #[cfg(sse)] {
        mod sse;
        use sse::Vec4fBase;
    } else if #[cfg(neon)] {
        mod neon;
        use neon::Vec4fBase;
    }
}
#[repr(transparent)]
pub struct Vec4f(Vec4fBase);
impl Vec4f {
    pub fn broadcast(value: f32) -> Self {
        Self(Vec4fBase::broadcast(value))
    }
    // ...
}
```

Такой подход решает обе проблемы предыдущего: теперь код не скомпилируется, если у платформоспецифичной стукруты не написать метод, который публичная структура должна прокинуть, а документация пишется только для методов публичной структуры. Однако при этом много места занимает boilerplate-код пробрасывания методов.

4. Используемый на текущий момент подход. Упрощенно, интерфейс оборачивается в общий для всех платформ типаж. Для методов, которые можно реализовать без использования интринсиков, пишется имплементация по умолчанию в этом типаже. Структуры под разными платформами должны реализовать этот типаж. Документация пишется к методам в типаже, а компилятор будет выдавать ошибку, если какой-то метод из требуемых в типаже не будет реализован. Кроме того, такой подход позволяет переопределять под отдельные платформы методы с реализацией по умолчанию, что было невозможно в других описанных выше подходах.

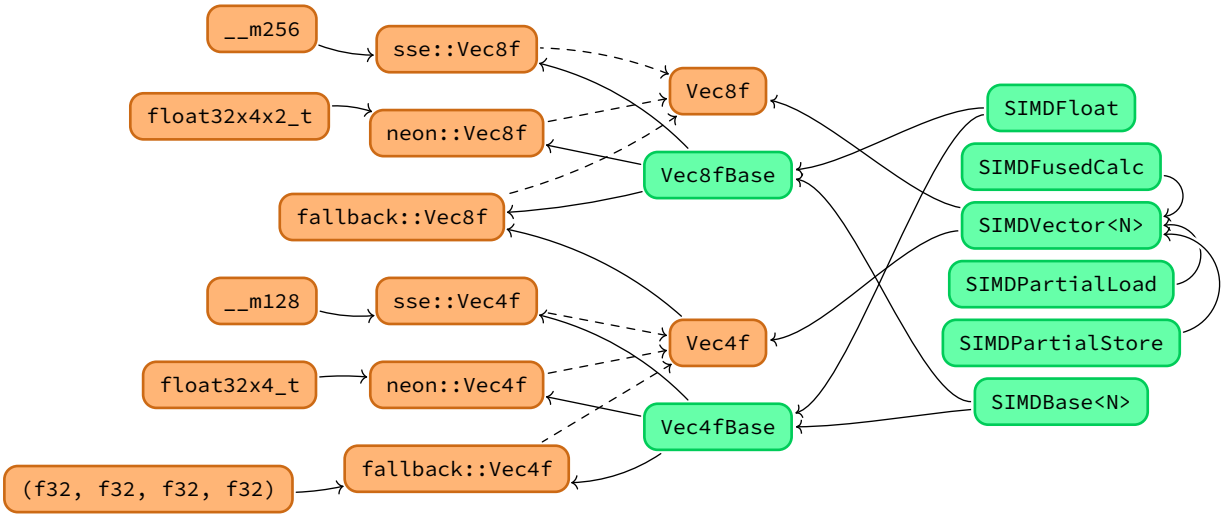

Рис. 7. Структура VRL

4.4. Бенчмарк

Для сравнения производительности полученного кода с аналогами использовалась задача подсчета скалярного произведения двух массивов вещественных чисел. Сравнивались несколько версий реализации этой задачи:

- использующая скалярные операции;
- на основе VRL, использующий `Vec8f`:
  1. разбивающая массивы на чанки с помощью `std::vec::Vec::chunks` и использующая `load_partial` для загрузки чисел в `Vec8f` на каждой итерации;
  2. написаный вручную цикл, который так же разбивает массивы на блоки по 8 элементов и использует `load_checked` в цикле, а `load_partial` только для обработки «хвоста»;
  3. аналогично предыдущему, но в цикле используется `load_ptr` для избежания ненужной проверки — сравнивается производительность безопасного и небезопасного методов;
  4. аналогично третьему, но использует совмещенное сложение-умножение;
- использующие `std::simd` [5]: с совмещенным сложением-умножением и без;

- на основе крейта `wide` [6];
- написанная на C++ с оригинальным VCL: так же с совмещенным сложением-умножением и без.

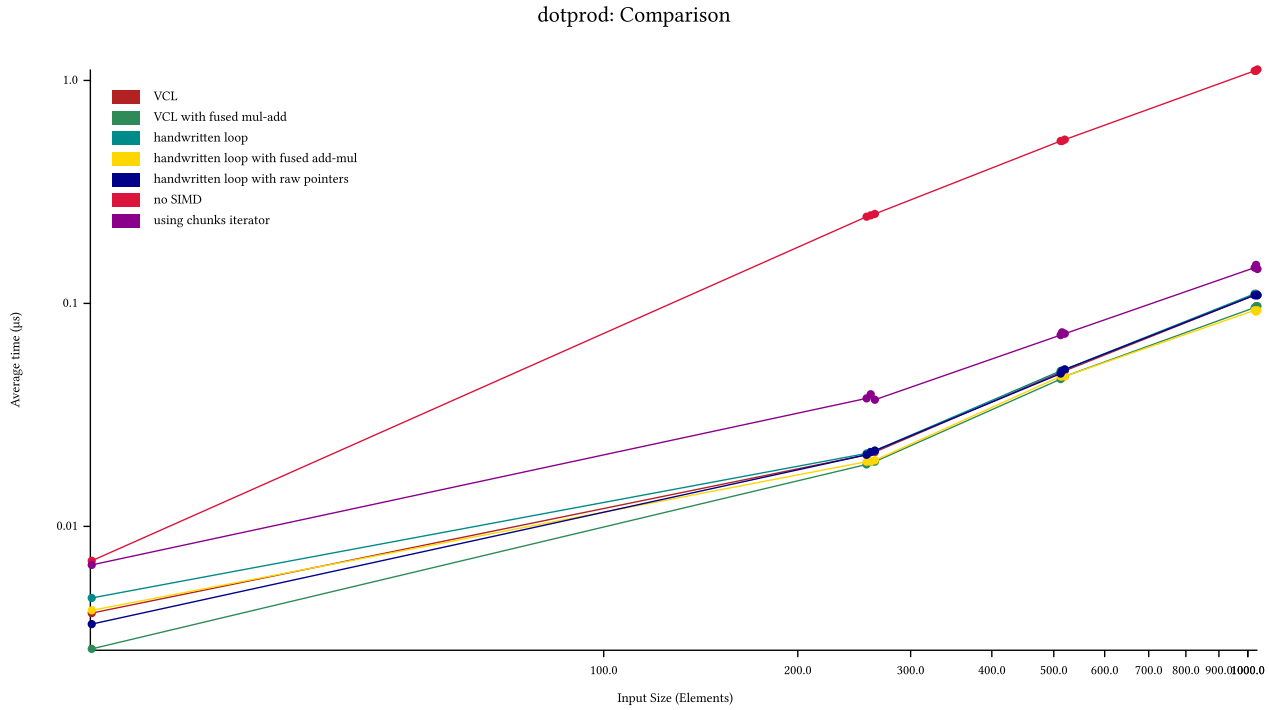

Рис. 8. Результаты бенчмарков на сервере в Яндекс.Облаке. CPU: Intel Xeon Processor (Icelake). Самая левая точка соответствует длине 8 входных массивов.

Для замера производительности используетсяся крейт `Criterion.rs` [12]. Сравнивается производительность на векторах разных размеров, в том числе не делящихся на 8: такие тесты проверяют эффективность обработки «хвоста» с помощью `load_partial` против обработки хвоста с помощью скалярных операций. Тестовые массивы заполняются случайными числами в интервале $(-1, 1)$. Всем тестируемым версиям на вход подаются одни и те же массивы.

Реализации на C++ вызываются прямо из кода на Rust, для этого используется крейт `cxx` [9]. В отличие от реализаций на Rust, функции из C++ не могут быть заинлайнены, а значит создается лишняя индирекция при вызове этий функций:

```
/* эта функция вызывается из бенчмарка напрямую, прочие слои индиреции заинлайнены */
0000000000067fb0 <benches$cxxbridge1$DotprodVec8fVCL>:
   67fb0:       48 83 ec 08             sub    $0x8,%rsp
   67fb4:       e8 17 00 00 00          call   67fd0 <benches::DotprodVec8fVCL(...)>
   67fb9:       48 83 c4 08             add    $0x8,%rsp
   67fbd:       c3                      ret
```

Таким образом, у реализаций на Rust есть преимущество в 4 инструкции. Понятно, что на результаты бенчмарка это почти не влияет. Также отметим, что код на C++ собирается компилятором `clang`, который использует ту же систему оптимизации (LLVM), что и `rustc`. Опытным путем было установленно, что `GCC` генерирует принципиально отличные от генерируемого `clang` инструкции. Подробнее об этом здесь.

В общем бенчмарки показывали следущее:
- совмещенное умножение-сложение работает быстрее, чем раздельное:
  - на Intel Xeon (x86-64) в Яндекс.Облаке на 5-13%,
  - на MediaTek 6771V/WT (AArch64) на 20%;
- использование небезопасного `load_ptr` вместо `load_checked` практически не повышает производительность: компилятор успешно убирает лишние проверки в `load_checked` и генерирует примерно одинаковые инструкции для основного цикла в обоих случаях (см. Листинг 1).
- версия с `std::vec::Vec::chunks` стабильно работает медленнее;
- версия на C++/VCL работает примерно так же на данных размера больше 8.

```
5fc00:       vmovups (%rdi,%r8,1),%ymm1
5fc06:       vmulps (%rdx,%r8,1),%ymm1,%ymm1
5fc0c:       add    $0xfffffffffffffff8,%r9
5fc10:       add    $0x20,%r8
5fc14:       vaddps %ymm1,%ymm0,%ymm0
5fc18:       cmp    $0x7,%r9
5fc1c:       ja     5fc00
```

```
5fdd0:       vmovups (%rdi,%rcx,1),%ymm1
5fdd5:       vmulps (%rdx,%rcx,1),%ymm1,%ymm1
5fdda:       add    $0x20,%rcx
5fdde:       dec    %rax
5fde1:       vaddps %ymm1,%ymm0,%ymm0
5fde5:       jne    5fdd0
```

Листинг 1. Ассемблерный код основного цикла `dotprod`: версия с `load_checked` (слева) и с `load_ptr` (справа)
Скомпилированно под AMD Ryzen 7 5700U

4.5. Fucntion inlining

Инлайнинг функций является одним из основным приемов оптимизаций в SIMD-вычислениях. В VRL это особенно важно, поскольку большинство методов очень короткие и дополнительная индирекция в виде вызова функции и передачи векторов через стек будет занимать много времени относительно полезной инструкции. Поэтому все методы в библиотеке помечены аттрибутом `#[inline]`.

4.6. Тестирование

Почти все методы библиотеки покрыты юнит- и доктестами. Тесты собираются и запускаются автоматически с несколькими поддерживаемыми конфигурациями доступных инструкций в Github CI (x86-совместимые) и Travis CI (ARM):

- AArch64 с набором векторных инструкций NEON;
- x86 с набором инструкций SSE и SSE2;
- x86 без векторных инструкций;
- x86-64 с набором инструкций SSE4.1 и ниже (здесь проверяется `fallback`-реализация `Vec8f`);
- x86-64 с AVX2 и ниже.

Доступные инструкции регулируются опцией компилятора `target-feature`.

5. Portable Simd Addons

Крейт `simd_addons` — это набор программных реализаций математических функций для векторных типов. Как и другие векторные операции, эти функции применяются в элементам вектора поэлементно. На текущий момент портированы из VCL портированны реализации следующих функций для вещественных чисел одинарной и двойной точности:

- тригонометрические: `sin`, `cos`, `sin_cos` и `tan`;
- обратные тригонометрические: `asin`, `acos`, `atan` и `atan2`;
- экспоненциальные: `exp`, `exp_m1` и `exp2`;
- вычисление значения многочлена по схеме Эстрина: в отличие от VCL, где доступны лишь написанные вручную реализации схемы для многочленов степени от 2 до 13, в `simd_addons` доступен макрос, который работает для многочленов любого размера;

5.1. Детали имплементации

Реализации для чисел одинарной и двойной точности часто существенно различаются, так как для 64-битной арифметики выше требования к точности. Далее за x обозначается значение, на котором вычисляется рассматриваемая функция.

Обычно вычисление $f(x)$ организованно следующим образом:

- заменить (*редуцировать*) x на x', близкое к нулю;
- вычислить $f(x')$, используя полиномиальное или рациональное приближение f в окрестности нуля;
- восстановить $f(x)$ по $f(x')$.

5.1.1. Тригонометрические функции

Во всех тригонометрических функциях используется *тригонометрическая редукция*: x заменяется на минимальное по модулю x', сравнимое с $|x|$ по модулю $\frac{\pi}{4}$:

$$q := \left\lfloor |x| \cdot \frac{2}{\pi} \right\rceil, \quad x' := x - \frac{q\pi}{4}$$

Вычисления sin и cos основаны на следующих тождествах:

$$\sin x = \operatorname{sgn} x \cdot (-1)^{\lfloor \frac{q}{2} \rfloor} \cdot \begin{cases} \sin x', & 2 \mid q \\ \cos x', & 2 \nmid q \end{cases}, \qquad \cos x = (-1)^{\lfloor \frac{q+1}{2} \rfloor} \cdot \begin{cases} \sin x', & 2 \nmid q \\ \cos x', & 2 \mid q \end{cases}$$

$\sin x'$ и $\cos x'$ вычисляются как значения некоторых многочленов по схеме Эстрина, дающих хорошую аппроксимацию этих функций на отрезке $\left[-\frac{\pi}{4}, \frac{\pi}{4}\right]$. Домножение на ± 1 делается с помощью битовых операций: так как знак числа — это его старший бит, то смену знака можно делать операцией побитового `XOR`:

$$\operatorname{bits}\left((-1)^k \cdot a\right) = \operatorname{bits}(a) \oplus ((k \ \texttt{\&}\ 1) \ \texttt{<<}\ \texttt{SIGN\_BIT\_POS}), \qquad \operatorname{bits}(\operatorname{sgn} x \cdot a) = \operatorname{bits}(a) \oplus (\operatorname{bits}(x) \ \texttt{\&}\ \texttt{SIGN\_BIT\_MASK}),$$

где $\operatorname{bits}(a)$ — битовое представление числа с плавающей запятой a.

Можно заметить, что вычисление $\sin x$ и $\cos x$ одновременно не сильно медленнее (примерно на 8% на процессоре AMD Ryzen 7) вычисления лишь одной из этих функций, так как требуется вычислять оба значения $\sin x'$ и $\cos x'$.

Вычисление $\tan x$ для вещественных чисел одинарной точности — вычисление частного $\frac{\sin x}{\cos x}$. Вычисление для чисел двойной точности опирается на равенство:

$$\tan x = \operatorname{sgn} x \cdot \begin{cases} \tan x', & 2 \mid q \\ \frac{1}{\tan x'}, & 2 \nmid q \end{cases}.$$

Значение $\tan x'$ вычисляется через рациональную (отношение многочленов) аппроксимацию на отрезке $\left[-\frac{\pi}{4}, \frac{\pi}{4}\right]$.

5.1.2. Обратные тригонометрические функции

Вычисление $\arcsin x$ опирается следующее тождество:

$$x' := \begin{cases} |x|, & |x| < 0.5 \\ \frac{\sqrt{1-|x|}}{2}, & |x| \geq 0.5 \end{cases} \qquad \arcsin x = \operatorname{sgn} x \cdot \begin{cases} \frac{\pi}{2} - 2\arcsin x', & |x| \geq 0.5 \\ \arcsin x', & |x| < 0.5 \end{cases}$$

Видно, что редуцированный аргумент $0 \leq x' \leq 0.5$. Значение $\arcsin x'$ аппроксимируется полиномиально для 32-битных и рационально для 64-битных числел с плавающей запятой.

$\arccos x$ вычисляется как $\frac{\pi}{2} - \arcsin x$.

Вычисление $\arctan x$ использует следующую редукцию:

$$x' := \begin{cases} \frac{-1}{|x|}, & |x| < \tan \frac{3\pi}{8} = \sqrt{2} + 1 \\ \frac{|x|-1}{|x|+1}, & \sqrt{2} - 1 = \tan \frac{\pi}{8} \leq |x| < \tan \frac{3\pi}{8} \\ \frac{|x|}{1}, & |x| < \tan \frac{\pi}{8} \end{cases} \qquad \arctan x = \operatorname{sgn} x \cdot \begin{cases} \arctan x' + \frac{\pi}{2}, & |x| > \tan \frac{3\pi}{8} \\ \arctan x' + \frac{\pi}{4}, & \tan \frac{\pi}{8} \leq |x| \leq \tan \frac{3\pi}{8} \\ \arctan x', & |x| < \tan \frac{\pi}{8} \end{cases}$$

Можно показать, что $0 \leq x' \leq \sqrt{2} - 1$. Вычисление x' выполняется с помощью лишь одной инструкции деления: деление занимает больше тактов процессора, чем иные арифметические операции, а потому важно минимизировать количество делений. $\arctan x'$ вычисляется полиномиальным приближением $\arctan$ для 32-битных чисел с плавающей запятой и рациональным для 64-битных.

5.1.3. Экспонента

Для вычисления e^x используется следующая редукция:

$$x' := x - \lfloor x \log_2 e \rceil \cdot \ln 2, \qquad e^x = e^{x'} \cdot 2^{\lfloor x \log_2 e \rceil}.$$

$\exp(x')$ аппроксимируется суммой нескольких первых членов ряда Тейлора экспоненты в нуле. Вычисление вещественного представления целой степени двойки вычисляется следующим образом. Напомним, что битовая запись нормального числа с плавающей запятой выглядит следующим образом:

$$\operatorname{bits}(a) = s e_1 e_2 ... e_{\texttt{EXPONENT\_BITS}} a_1 a_2 ... a_{\texttt{MANTISSA\_BITS}},$$

где

$$a = (-1)^s \cdot \overline{1.a_1 a_2 ... a_{\texttt{MANTISSA\_BITS}}}_2 \cdot 2^{\overline{e_1 e_2 \cdots e_{\texttt{EXPONENT\_BITS}}}_2 - \texttt{BIAS}}, \quad \texttt{BIAS} = 2^{\texttt{EXPONENT\_BITS}} - 1.$$

Тогда битовое представление степени не слишком большой целой степени двойки представляется как

$$\text{bits}(2.0^n) = 0\overbrace{n_1 n_2 ... n_{\texttt{MANTISSA\_BITS}}}^{\text{двоичная запись } n+\texttt{BIAS}}\underbrace{0\ 0\ ...\ 0}_{\texttt{MANTISSA\_BITS}} =$$

$$= (n + \texttt{BIAS}) \texttt{ << MANTISSA\_BITS} = \text{bits}(n + 2^{\texttt{MANTISSA\_BITS}} + \texttt{BIAS}) \texttt{ << MANTISSA\_BITS},$$

где $\text{bits}(n)$ в правой части интерпретируется как целое беззнаковое число с такой битовой записью. Таким образом, целую вещественную степень двойки можно вычислить используя лишь битовые операции вещественное сложение, не переводя n из числа с плавающей запятой в целое.

5.2. Тестирование

Написанные функции сравниваются со скалярными аналогами и реализацией из VCL с точностью до нескольких значащий знаков на всех значениях отрезка, покрывающем большую часть области определения функции, с некоторых шагом, а так же на специальных значениях, таких как NaN, $\pm\infty$, -0.

Библиография

[1] «SIMD — Википедия». [Онлайн]. Доступно на: https://ru.wikipedia.org/wiki/SIMD

[2] A. Fog, «Vector Class Library». [Онлайн]. Доступно на: https://github.com/vectorclass/version2

[3] «Eistrin's Scheme — Wikipedia». [Онлайн]. Доступно на: https://en.wikipedia.org/wiki/Estrin's_scheme

[4] «Basics of SIMD Programming». Просмотрено: 24 апрель 2024 г. [Онлайн]. Доступно на: http://ftp.cvut.cz/kernel/people/geoff/cell/ps3-linux-docs/CellProgrammingTutorial/BasicsOfSIMDProgramming.html

[5] Rust Portable SIMD Project Group, «Portable SIMD». [Онлайн]. Доступно на: https://github.com/rust-lang/portable-simd

[6] Lokathor, «wide». [Онлайн]. Доступно на: https://crates.io/crates/wide

[7] «Highway». Просмотрено: 24 апрель 2024 г. [Онлайн]. Доступно на: https://github.com/google/highway

[8] «xsimd». Просмотрено: 24 апрель 2024 г. [Онлайн]. Доступно на: https://github.com/xtensor-stack/xsimd

[9] D. Tolnay, «cxx». [Онлайн]. Доступно на: https://crates.io/crates/cxx

[10] N. Layzell и O. Goffart, «cpp». [Онлайн]. Доступно на: https://crates.io/crates/cpp

[11] «simd-ffi — The Rust RFC Book». [Онлайн]. Доступно на: https://rust-lang.github.io/rfcs/2574-simd-ffi.html

[12] J. Aparicio, «Criterion.rs». [Онлайн]. Доступно на: https://github.com/bheisler/criterion.rs