

Санкт-Петербургский государственный университет

Кафедра информационно-аналитических систем

Группа 21.Б10-мм

# Оптимизация библиотеки XXHASH для архитектуры RISC-V

*Пономарев Николай Алексеевич*

Отчёт по учебной практике  
в форме «Эксперимент»

Научный руководитель:  
ст. преподаватель кафедры ИАС К. К. Смирнов

Санкт-Петербург  
2023

# Оглавление

<b>Введение</b>	<b>3</b>
<b>1. Постановка задачи</b>	<b>4</b>
<b>2. Обзор</b>	<b>5</b>
2.1. Векторное расширение RISC-V . . . . .	5
2.2. Библиотека xxHash . . . . .	5
<b>3. Реализация</b>	<b>7</b>
3.1. Умножение с ручным расширением . . . . .	7
3.2. Сложение с переносом . . . . .	8
3.3. Различия в версиях RVV . . . . .	8
3.4. Загрузка по невыровненному адресу . . . . .	9
<b>4. Эксперимент</b>	<b>10</b>
<b>Заключение</b>	<b>12</b>
<b>Список литературы</b>	<b>13</b>

# Введение

xxHASH<sup>1</sup> — современная библиотека для хеширования, целью которой является генерация хеша со скоростью, сравнимой с пропускной способностью оперативной памяти. Высокую скорость работы библиотеки, в частности, хешей XXH3 и XXH128, обеспечивает реализация алгоритмов хеширования с помощью векторных расширений процессора, уже имеется поддержка таких векторных расширений как SSE2, AVX512, NEON и других.

RISC-V, в свою очередь, это молодая и активно развивающаяся архитектура, которая получила поддержку от таких компаний как GOOGLE [1], IMAGINATION TECHNOLOGIES [4] и ALIBABA [3]. Некоторые процессоры архитектуры RISC-V имеют поддержку векторных инструкций. К сожалению, долгое время спецификация векторного расширения находилась в разработке, что привело к наличию в продаже процессоров с разными версиями расширения [2].

Библиотека xxHASH ещё не содержит в себе оптимизаций с помощью векторного расширения для платформы RISC-V (сокращенно RVV). На момент написания данной работы в продаже можно было найти только процессоры с «урезанной» поддержкой RVV. Изучение вопроса о том, возможно ли ускорение алгоритмов при переносе на процессоры с «ограниченными» возможностями на примере библиотеки xxHASH, является целью данной работы.

В работе приводится изучение различий в версиях RVV, разбор трудностей, встреченных при переносе на процессор с «ограниченными» возможностями, и исследование результатов измерения производительности.

---

<sup>1</sup><https://github.com/Cyan4973/xxHash>

# 1. Постановка задачи

Целью данной работы является реализация хеш-функций из библиотеки XXHASH при помощи векторных расширений архитектуры RISC-V.

Для достижения данной цели были поставлены следующие задачи:

- Сравнить возможности разных версий векторного расширения RISC-V;
- Выбрать целевую платформу для адаптации кода и проведения измерений;
- Адаптировать одну из существующих реализаций под выбранную платформу;
- Выполнить замеры производительности оптимизированного кода.

## 2. Обзор

### 2.1. Векторное расширение RISC-V

Векторное расширение RISC-V находилось в разработке довольно долго, из-за чего производители чипов начали выпускать свои решения основываясь на недоработанной версии стандарта 0.7.1. Один из таких чипов — ALLWINNER D1, получил большое распространение в недорогих одноплатных ПК, таких как Sipeed Litee RV и MangoPi MQ-Pro.

Последней и официальной ратифицированной версией является версия 1.0, она поддерживается современными версия LLVM и GCC. Версия 0.7.1 не была ратифицирована, и поддерживается лишь в специализированных версиях компиляторов, предназначенных для конкретных аппаратных платформ. Насколько известно, единственным таким компилятором является компилятор от компании ALIBABA на основе GCC 10<sup>2</sup>.

Для облегчения жизни программиста в компиляторах существуют так называемые `intrinsic` функции, которые позволяют использовать специфичные возможности процессора из высокоуровневых языков. Спецификация RVV определяет множество типов (в смысле языка C), что порождает множество функций с практически идентичным названием. Современные версии LLVM и GCC поддерживают перегрузку `intrinsic` функций, что делает код гораздо более читаемым — строго типы нужно указывать малому количеству операции, например загрузке и выгрузке векторов в память. Сравнить читаемость кода можно на листинге 1. К сожалению, в компиляторе от ALIBABA данная возможность отсутствует. Далее во всех листингах для компактности будут использоваться перегруженные версии `intrinsic` функций.

### 2.2. Библиотека xxHash

Библиотека содержит в себе четыре алгоритма хеширования: XXH32, XXH64, XXH3, XXH128. Первые две хеш-функции были включены в

---

<sup>2</sup><https://occ.t-head.cn/community/download?id=4090445921563774976>

## Листинг 1: Сравнение перегруженных и не перегруженных функций

```
// Код без использования "перегрузки"  
vbool32_t carry_bits = vmadc_vvm_u32m1_b32(prod_even,  
    ↪ prod_odd, xzero_mask, VL);  
// Код с использованием "перегрузки"  
vbool32_t carry_bits = vmadc(prod_even, prod_odd,  
    ↪ xzero_mask, VL);
```

состав библиотеки с первых версий, однако их алгоритм работы не позволяет использовать векторные возможности процессоров. ХХНЗ и ХХН128, в свою очередь, разрабатывались с учетом векторных операций, поэтому они представляют интерес в данном исследовании.

ХХНЗ и ХХН128 по сути используют один алгоритм, но возвращают 64- или 128-битные хеши соответственно. Внутри они используют функции `ХХНЗ_accumulate` и `ХХНЗ_scrambleAcc`. Первая из них получает на вход указатели на данные, секрет и аккумулятор, затем, особым образом изменив данные с помощью секрета, добавляет их в аккумулятор. Вторая же, получая на вход секрет и аккумулятор после обработки большого блока данных, перемешивает аккумулятор для усиления хеширования.

### 3. Реализация

Интерес для оптимизации представляют только две функции, используемые в XXH3 и XXH128: `XXH3_accumulate` и `XXH3_scrambleAcc`. Сама библиотека написана на языке C, а для оптимизации применяются `intrinsic` функции.

Уже существующие реализации оперируют компонентами вектора размером в 64 бита, что в терминологии RISC-V называется `Selected Element Width (SEW)`. А длина вектора разнится от 128 бит до 512 бит, в терминологии RISC-V — `Vector Length (VL)`.

В качестве эталонной реализации был выбран вариант для набора команд SSE2, т. к. целевой процессор имеет VL в 128 бит, как и SSE2.

По спецификации векторного расширения RISC-V, минимальный VL равен 128 битам, также должна присутствовать поддержка SEW равного 64 битам. К сожалению, в устройствах на чипе ALLWINNER D1, отсутствует поддержка 64-битных элементов вектора, поэтому была предпринята попытка использовать SEW в 32 бита из-за чего потребовалось некоторое количество ухищрений, о которых будет рассказано далее.

#### 3.1. Умножение с ручным расширением

Одной из первых проблем стала операция умножения. В алгоритме требуется перемножить 2 вектора, используя только младшие 32 бита каждого элемента, в результате чего получаются 64-битные числа. При SEW в 32 бита необходимо перемножить между собой нулевой и второй элементы каждого вектора. Для совершения данной операции необходимо использовать отдельные инструкции для вычисления младших и старших битов произведения, затем сдвигать старшие биты в первый и третий элемент вектора, а после объединять их по маске, таким образом, чтобы в нулевой и второй элемент попали младшие биты произведения, а в первый и третий старшие соответственно. Данную операцию иллюстрирует листинг 2.

## Листинг 2: Умножение с ручным расширением до 64 бит

```
vuint32m1_t product_hi = vmulhu(data_key, data_key_lo, VL);  
    ↪ // Вычисление старших битов произведения  
vuint32m1_t product_lo = vmul(data_key, data_key_lo, VL);  
    ↪ // Вычисление младших битов произведения  
vuint32m1_t product_hi_sl = vrgather(product_hi,  
    ↪ xlshift_mask, VL); // Сдвиг старших битов  
vuint32m1_t product = vmerge(xmerge_mask, product_hi_sl,  
    ↪ product_lo, VL); // Объединение двух векторов в один
```

### 3.2. Сложение с переносом

Сложение тоже требует модификации: необходимо сохранить бит переноса, т. к. мы складываем 64-битные числа, представленные как пары 32-битных. RISC-V имеет инструкцию, которая по паре векторов заполняет битовую маску: если при сложении необходим перенос, соответствующий бит становится единицей. Однако команды для сдвига маски не существует, поэтому используется такая последовательность команд: сначала вычисляется маска переноса, затем, с помощью операции И, в маске оставляются только биты на нулевом и втором местах, после этого выполняется обычное сложение. Для того чтобы прибавить бит переноса, элементы вектора попарно переставляются местами, производится прибавление переноса, а затем возвращение элементов на свои места. Эту операцию иллюстрирует листинг 3.

### 3.3. Различия в версиях RVV

К сожалению, версия RVV 0.7.1 гораздо менее доработана, нежели версия 1.0. Одной из проблем версии 0.7.1, является отсутствие операции загрузки маски из памяти. Для обхода этого ограничения используется команда `vmseq.vx`, она принимает на вход вектор и число, и если элемент вектора равен заданному числу, то бит в маске устанавливается в единицу.



### Листинг 3: Сложение с переносом

```
vbool32_t carry_bits = vmadc(prod_even, prod_odd,  
    ↪ xzero_mask, VL); // Вычисляем маску битов переноса  
carry_bits = vmand(carry_bits, xmerge_mask, VL);  
    ↪ // Оставляем в маске только биты на 0 и 2 местах  
vuint32m1_t prod = vadd(prod_even, prod_odd, VL);  
    ↪ // Складываем числа без переноса  
prod = vrgather(prod, xswap_mask32, VL);  
    ↪ // Меняем попарно местами элементы вектора  
prod = vmadc(prod, xzero_vector, carry_bits, VL);  
    ↪ // Прибавляем биты переноса  
prod = vrgather(prod, xswap_mask32, VL);  
    ↪ // Возвращаем элементы вектора на свои места
```

Операция вычисления битов переноса при сложении в версии 0.7.1 сделана таким образом, что ей всегда требуется маска, даже если это первое вычисление, что требует создавать маску из нулей для корректной работы.

### 3.4. Загрузка по невыровненному адресу

Спецификация векторных расширений RISC-V не определяет корректное поведение при загрузке данных из памяти по невыровненному адресу. Так например в функции `XXH3_accumulate`, два из трех указателей, принимаемых на вход, являются невыровненными. К счастью, эту проблему легко обойти: достаточно загружать данные, как если бы вектор был из 8-битных элементов, а затем интерпретировать его как вектор нужной размерности, что демонстрирует листинг 4.

## Листинг 4: Загрузка по невыровненному адресу

```
vuInt32m1_t key_vec =  
    ↪  vreinterpret_v_u8m1_u32m1(vle8_v_u8m1((uint8_t*)(xsecret  
    ↪  + VL * i), VL * 4));
```

## 4. Эксперимент

Измерения проводились на одноплатном компьютере SIPEED LICHEE RV 86 со следующими характеристиками:

- Процессор ALLWINNER D1 с частотой 1 ГГц;
- Оперативная память DDR3 объемом 512 Мб с частотой 792 МГц;
- Операционная система DEBIAN SID с последними обновлениями на момент тестирования.

Для компиляции использовался компилятор от компании ALIBABA с флагами `-O3` и `-march=rv64gcv0p7`. Для выбора набора функций использовались флаги `-DXXH_VECTOR=XXH_SCALAR` и `-DXXH_VECTOR=XXH_RVV` соответственно. Данные измерений были получены с помощью поставляемой вместе с библиотекой утилиты `xxhsum`.

Таблица 1: Сравнение производительности хеш-функции XXHASH3 на входных данных размером в 1000 Кб

Набор функций	Скорость работы, Мб/с
Скалярный	169.3
Векторный	116.0

Результаты замеров приведены в таблице 1. По ним можно сделать вывод, что в результате скорость работы алгоритма не только не увеличилась, более того, стала заметно меньше. Можно предположить, что причиной этого являются способы обхода ограничений. Функции

XXH3\_accumulate и XXH3\_scrambleAcc являются ключевыми для вычисления хеша, а следовательно вызываются постоянно. Использование 32-битных элементов требует предварительной загрузки большого количества масок и вспомогательных векторов в память, что затрачивает время перед основной обработкой данных. Кроме того, операцию объединения векторов и операцию перестановки элементов в векторе можно считать сложными и длительными для исполнения, но именно они используются в большом объеме, что так же вносит значительный вклад в замедление алгоритма.

Встреченные трудности и результаты измерений позволяют сделать вывод о том, что использование 32-битных элементов векторов для обхода ограничений платформы приводит не только к сложным конструкциям в коде программы, но и не дает прироста производительности.

# Заключение

В рамках данной учебной практики были достигнуты следующие результаты:

- Изучены возможности разных версий векторного расширения RISC-V;
- Выбрана целевая платформа для адаптации кода и проведения измерений;
- Произведена адаптация одной из существующих реализаций под выбранную платформу;
- Выполнены замеры производительности оптимизированного кода.

Исходный код расположен по адресу: <https://github.com/WoWaster/xxHash>.

## Список литературы

- [1] Google Announces Official Android Support for RISC-V | Ars Technica. — URL: <https://arstechnica.com/gadgets/2023/01/google-announces-official-android-support-for-risc-v/> (дата обращения: 2023-05-14).
- [2] Implications of Widely Distributed Draft 0.7.1 Implementation(s) · Issue #667 · Riscv/Riscv-v-Spec, GitHub. — URL: <https://github.com/riscv/riscv-v-spec/issues/667> (дата обращения: 2023-05-14).
- [3] Robinson Dan. Alibaba Launches RISC-V Developer Platform for Edge SoCs. — URL: [https://www.theregister.com/2022/08/25/alibaba\\_riscv\\_developer\\_platform/](https://www.theregister.com/2022/08/25/alibaba_riscv_developer_platform/) (дата обращения: 2023-05-14).
- [4] Why We've Levelled up on RISC-V. — URL: <https://blog.imaginationtech.com/why-weve-levelled-up-on-risc-v> (дата обращения: 2023-05-14).