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

Введение

xxHash — современная библиотека для хеширования, целью которой является генерация хеша со скоростью, сравнимой со скоростью оперативной памяти [1]. Высокую скорость работы, в частности, для хешей XXH3 и XXH128, обеспечивает реализация алгоритмов хеширования с помощью векторных расширений процессора.

1. Постановка задачи

2. Обзор

3. Реализация

Библиотека содержит в себе четыре алгоритма хеширования: XXH32, XXH64, XXH3, XXH128. Первые два из них не поддаются оптимизации с помощью векторных операций, поэтому интерес представляют только последние. Внутри они используют функции `XXH3_accumulate` и `XXH3_scrambleAcc`. Именно эти функции могут использовать векторные возможности процессора: в библиотеке уже имеется поддержка SSE2, AVX512, NEON и других. Сама библиотека написана на языке C, а для оптимизации применяются intrinsic функции.

Уже существующие реализации оперируют компонентами вектора размером в 64 бита, в терминологии RISC-V это называется SEW[1]. А длина вектора разнится от 128 бит до 512 бит, в терминологии RISC-V — VL[2].

По спецификации векторного расширения RISC-V, минимальный VL равен 128 битам, также должна присутствовать поддержка SEW равного 64 битам. К сожалению, на момент написания данной работы в продаже можно было найти лишь устройства на чипе Allwinner D1, в котором отсутствует поддержка 64-битных элементов вектора, поэтому была предпринята попытка использовать SEW в 32 бита из-за чего потребовалась некоторое количество ухищрений.

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

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 пришлось 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,

[1]Selected Element Width

[2]Vector Length

Листинг 1: Пример для умножения, выдранный из библиотеки

```
vuint32m1_t product_hi = vmulhu_vv_u32m1(data_key,
 ↪  data_key_lo, VL);
vuint32m1_t product_lo = vmul_vv_u32m1(data_key,
 ↪  data_key_lo, VL);
vuint32m1_t product_hi_sl = vrgather_vv_u32m1(product_hi,
 ↪  xlshift_mask, VL);
vuint32m1_t product = vmerge_vvm_u32m1(xmerge_mask,
 ↪  product_hi_sl, product_lo, VL);
```

Листинг 2: Пример для умножения, попытка использовать перегрузки

```
vuint32m1_t product_hi = vmulhu(data_key, data_key_lo, VL);
vuint32m1_t product_lo = vmul(data_key, data_key_lo, VL);
vuint32m1_t product_hi_sl = vrgather(product_hi,
 ↪  xlshift_mask, VL);
vuint32m1_t product = vmerge(xmerge_mask, product_hi_sl,
 ↪  product_lo, VL);
```

чтобы в нулевой и второй элемент попали младшие биты произведения, а в первый и третий старшие соответственно.

Кроме того, 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...

Allwinner D1 поддерживает векторное расширение RISC-V экспериментальной версии 0.7.1, которая более не имеет официальной поддержки, а единственный компилятор со встроенной поддержкой

Листинг 3: Пример для умножения, попытка использовать перегрузки и комментарии

```
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
```

поддерживается компанией Alibaba на основе GCC 10. Актуальная и ратифицированная версия расширения — 1.0, она поддерживается современными версия LLVM и GCC. К сожалению, в версия 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.

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

Современные версии LLVM и GCC поддерживают «перегрузку» интринсик функций, что делает код гораздо более читаем — строго типы нужно указывать малому количеству операции, например загрузке и выгрузке векторов в память. К сожалению, в компиляторе от Alibaba данная возможность отсутствует.

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 указа-

Листинг 4: 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);
```

Листинг 5: Загрузка по невыровненному адресу

```
vuint32m1_t key_vec =
 ↪  vreinterpret_v_u8m1_u32m1(vle8_v_u8m1((uint8_t*)(xsecret
 ↪  + VL * i), VL * 4));
```

телей, 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.

4. Эксперимент

Заключение

Список литературы

[1] GitHub - Cyan4973/xxHash: Extremely fast non-cryptographic hash algorithm — github.com. — https://github.com/Cyan4973/xxHash. — [Accessed 10-May-2023].