Санкт-Петербургский государственный университет

Романов Алексей Александрович

Выпускная квалификационная работа

Исследование производительности векторного расширения для RISC-V ядер

Уровень образования: бакалавриат

Направление *02.03.03 «Математическое обеспечение и администрирование информационных систем»*

Основная образовательная программа *СВ.5006.2019 «Математическое обеспечение и администрирование информационных систем»*

Научный руководитель:
доцент кафедры системного программирования, к.т.н. Ю. В. Литвинов

Консультант:
старший преподаватель кафедры системного программирования Я. А. Кириленко

Рецензент:
ведущий инженер-программист ООО «Синтакор» Д. Н. Петров

Санкт-Петербург
2023

Saint Petersburg State University

Aleksei Romanov

Bachelor's Thesis

Performance Study of Vector Extension for RISC-V Cores

Education level: bachelor

Speciality *02.03.03 "Software and Administration of Information Systems"*

Programme *CB.5006.2019 "Software and Administration of Information Systems"*

Scientific supervisor:
C.Sc., System Programming Chair Docent Y. V. Litvinov

Consultant:
Senior Lecturer, Department of System Programming, Y. A. Kirilenko

Reviewer:
Lead Software Engineer at "Syntacore" D. N. Petrov

Saint Petersburg
2023

Оглавление

Введение

RISC-V является молодой открытой процессорной архитектурой, созданной в 2010-м году. Она быстро набирает популярность: в 2018-м году The Linux Foundation поддержал данную архитектуру, а в начале 2022-го года Intel стал одним из спонсоров проекта RISC-V. Любая компания может стать производителем процессоров на данной архитектуре. Таких производителей зачастую называют вендорами. RISC-V состоит из обязательного набора инструкций и дополнительных расширений [13]. Каждый вендор обязан реализовывать базовые команды для работы с целыми числами, а также может добавлять дополнительные возможности, описанные в стандарте. Таким образом, вендоры могут предлагать потребителям процессоры, созданные для решения какой-либо конкретной задачи.

В большинстве современных процессоров присутствуют векторные расширения. Данные расширения реализуют принцип SIMD[1]. Такой подход позволяет ускорять не только научные вычисления, но и программы общего назначения. Например, функция memcpy стандартной библиотеки языка C, которая используется для копирования данных, может быть реализована с помощью векторных инструкций. В x86_64 присутствуют семейства векторных инструкций SSE и AVX, а в ARM — Neon, SVE. В сентябре 2021-го года вышла первая версия векторного расширения (RVV) для архитектуры RISC-V. Ввиду того, что с первого релиза прошло немного времени, векторные инструкции в RISC-V недостаточно исследованы.

Компания «Syntacore» является одним из разработчиков RISC-V-ядер в России[2]. Также в Syntacore идет активная разработка векторного расширения для своих процессоров. Реализация любых расширений требует поддержки не только аппаратной части, но и программного обеспечения: компиляторов, наборов инструментов программирования, приложений для тестирования и измерения производительности. Каж-

[1]SIMD (Single Instruction Multiple Data) – принцип, при котором обработка массива данных происходит за одну инструкцию.

[2]https://riscv.org/members/

дому вендору необходимо настраивать компиляторы так, чтобы они порождали оптимальный для конкретной микроархитектуры код. При поддержке векторного расширения необходимо учитывать особенности алгоритмов автовекторизации — методов преобразования скалярных инструкций в векторные. При этом изменения, внесенные в компилятор, необходимо отслеживать и тестировать. Кроме того, для улучшения микроархитектуры необходимо производить измерение производительности процессоров на различных алгоритмах. Для этого необходимо использовать ПЛИС, на которых загружена прошивка — скомпилированное формальное описание работы процессора на специальном языке. В данной работе предлагается автоматизация и получение результатов тестирования компилятора и измерение производительности векторизованных алгоритмов на процессоре.

Дата сборки: 1 июня 2023 г.

1. Постановка задачи

Целью работы является исследование производительности векторного расширения RISC-V с точки зрения компилятора и аппаратной части. В свете этого были поставлены следующие задачи:

1. обзор векторного расширения RVV;
2. разработка приложения для тестирования автовекторизации под архитектуру RISC-V;
3. разработка приложения для измерения производительности алгоритмов, векторизованных с помощью RVV;
4. тестирование и проведение экспериментов на основе разработанных приложений.

2. Обзор векторного расширения RISC-V

2.1. Обзор существующих исследований

RISC-V является модульной архитектурой, которая предполагает подход к проектированию процессоров с упрощенной системой команд[3]. В [17] описываются различия между компьютерами с упрощенной и полной[4] системами команд. Например, в RISC-V размер всех инструкций равен 32-м битам в отличие от архитектуры x86_64, где инструкции могут быть различной длины. С этим связаны определенные особенности в ассемблерном коде. В некоторых случаях осложняется работа с непосредственными операндами — в [12] описывается то, как загружать в регистры 32-битные целые числа.

Разработчики RISC-V проектировали RVV в противопоставление традиционному «упакованному» SIMD-подходу к векторизации [5], в котором инструкции всегда используют фиксированный размер векторных регистров. Обычно векторные расширения процессоров имеют множество различных инструкций для поддержки различных типов данных. Так как в RISC-процессорах множество кодов инструкций ограничено, RISC-V использует полиморфное поведение векторных операций. Длина векторных регистров и типы данных определяются во время исполнения программы. Такие регистры называются масштабируемыми. Кроме того, RISC-V вводит лишь минимальные ограничения на векторное расширение, позволяя разработчикам аппаратного обеспечения самостоятельно задавать длину векторных регистров. Как заявлено в [2], похожие особенности присутствуют в расширении SVE для архитектуры Arm.

В силу того, что RISC-V является свободной архитектурой, у данной архитектуры появилось множество вендоров. При этом в общем случае компиляторы не могут делать предположения о микроархитектуре процессора. Для того чтобы ассемблерный код не зависел от реа-

[3]Reduced Instruction Set Computer — URL: https://en.wikipedia.org/wiki/Reduced_instruction_set_computer (дата обращения: 2023-04-29)

[4]Complex Instruction Set Computer — URL: https://en.wikipedia.org/wiki/Complex_instruction_set_computer (дата обращения: 2023-04-29)

лизации процессора, в RVV введено несколько механизмов. Первый из них — длина векторного регистра в байтах (*vlenb*). *vlenb* является константой, которая определяется во время проектирования процессора и содержится в специальном управляющем регистре[5], доступном только для чтения. Другим механизмом является семейство инструкций *vset{i}vl{i}*[6], которые позволяют динамически вычислять количество элементов для последующей обработки. Данные инструкции основываются на нескольких параметрах:

- количество запрашиваемых элементов N;
- SEW — размер обрабатываемых элементов в битах;
- $LMUL$ — мультипликатор длины векторного регистра, параметр для объединения векторных регистров в группы.

Листинг 1: Использование инструкции vsetvli

```
vsetvli t0, 32, e32, m1, ta, ma
// vlenb=16
// N = 32, SEW = 32, LMUL = 1
// t0=min(32, LMUL * vlenb / (SEW / 8))
```

Все векторные инструкции, исполняющиеся после инструкции *vset{i}vl{i}*, так или иначе используют параметры SEW и $LMUL$ в своих вычислениях. SEW — достаточно очевидный параметр, необходимый лишь для дальнейших вычислений в последующих инструкциях, таких как векторные арифметические операции. $LMUL$ же определяет поведение ассемблерных команд, позволяя группировать регистры или, наоборот, уменьшать их длину.
Рассмотрим примеры из [15]:

- $LMUL = 1$ — инструкции оперируют одним регистром, указанным как операнд;

[5]Vector Byte Length — URL: https://github.com/riscv/riscv-v-spec/blob/master/v-spec.adoc#vector-byte-length-vlenb (дата обращения: 2022-12-15)
[6]Configuration-Setting Instructions — URL: https://github.com/riscv/riscv-v-spec/blob/master/v-spec.adoc#sec-vector-config (дата обращения: 2023-04-29)

- $LMUL > 1$ — инструкции используют в вычислениях $LMUL$ последовательных регистров, начиная с указанного как операнд;

```
1 // vlenb = 16, SEW = 32, LMUL = 2
2 // int arr[8] = { 0, 1, 2, 3, 4, 5, 6, 7 }
3 // a0 = arr
4 vle32.v v2, (a0)
```

  На строке 4 происходит загрузка массива 4-байтных значений в векторные регистры $v2 = [3, 2, 1, 0]$, $v3 = [7, 6, 5, 4]$.

- $LMUL < 1$ — в данном случае используется только соответствующая $LMUL$ часть векторного регистра.

```
1 // vlenb = 16, SEW = 32, LMUL = 1/2
2 // int arr[4] = { 0, 1, 2, 3 }
3 // a0 = arr
4 vle32.v v2, (a0)
```

  Результатом выполнения инструкции на строке 4 будет векторный регистр $v2 = [\_, \_, 1, 0]$.

Кроме того, сами инструкции могут задавать размер операнда (EEW). Так инструкции могут переопределять поведение или группировку регистров. Группировка регистров для текущей команды вычисляется следующим образом:

$$EMUL = \frac{EEW * LMUL}{SEW}$$

Такой подход может эмулировать поведение «упакованного» SIMD, поэтому обладает всеми его свойствами, но при этом имеет множество преимуществ по сравнению с традиционными векторными расширениями. Например, в [6] сравниваются RVV и AVX и выделяются следующие достоинства:

- размер кода становится намного меньше за счет того, что RVV позволяет компиляторам не генерировать дополнительную обработку «хвостовых» элементов;

- также размер кода уменьшается в связи с тем, что RVV предусматривает аппаратную раскрутку циклов с помощью параметра *LMUL*. Таким образом нет необходимости дублировать ассемблерный код внутри циклов;

- бинарные файлы могут исполняться на других процессорах без перекомпиляции, которая может быть очень долгой для больших проектов. Единственным ограничением является наличие поддержки стандартного векторного расширения на устройстве, где планируется запускать программу.

В статье [11] сравниваются различные векторные расширения на примере функции «DAXPY»:

```
void daxpy(size_t n, double a, const double *x, double *y) {
    for (size_t i = 0; i < n; i++) {
        y[i] = a*x[i] + y[i];
    }
}
```

В данной работе представлена таблица, в которой подсчитано количество инструкций, полученных при компиляции исследуемой функции. RISC-V использует меньшее количество инструкций, чем x86. Тем не менее в основном цикле оказывается больше команд. Это связано с тем, что RISC-V не позволяет использовать смещения в операндах типа «память». Исходя из этого, был сделан вывод, что RVV может стать векторным расширением, которое выигрывает у современных архитектур по следующим показателям: соотношению производительности и энергопотребления, легкости кодогенерации. Тем не менее статья была написана еще до выхода первой версии RVV, а алгоритмы векторизации все еще исследуются учеными [3].

В статье [4] описываются алгоритмы векторизации кода, применяемые в современных компиляторах. Один из методов основан на SLP-стратегии[7], которая используется в большинстве компиляторов для век-

[7]SLP — Superword Level Parallelism

торизации выражений вне циклов. Данная стратегия состоит в нахождении скалярных инструкций и объединении их в векторные. Задача с такой постановкой является NP-полной, поэтому существуют эвристики, которые применяют в компиляторах. Из этого следует, что компиляторы не всегда способны получить оптимальный векторизованный код. Одной из эвристик является задача упаковки попарных инструкций. В результате решения данной задачи, все инструкции разбиваются на пары, образуя векторные команды. На такие пары налагаются следующие ограничения:

1. инструкции должны быть изоморфны: выполняя одну и ту же операцию с одними и теми же данными, команды дают одинаковые результаты;

2. инструкции должны быть независимы: операнды одной инструкции не должны зависеть от операндов другой;

3. инструкции должны обращаться к смежным участкам памяти.

Данные ограничения важны при дальнейшем анализе автовекторизации для RISC-V.

На данный момент *clang* является единственным компилятором, поддерживающим автовекторизацию для архитектуры RISC-V. При этом данный компилятор не всегда порождает оптимальный код. Например, в исследовании [1] проводится эксперимент, в ходе которого выясняется, насколько хорошо *clang-15* векторизует циклы при помощи масштабируемых векторов. Оказывается, что масштабируемые векторы в данном компиляторе работают хуже, чем «упакованные» векторные регистры. Также в данной работе проводятся измерения производительности на симуляторе *gem5* при помощи подсчета количества инструкций, которые были выполнены симулятором. В результате оказывается, что в некоторых алгоритмах масштабируемые векторы оптимальнее «упакованных».

Некоторые разработчики уже пробовали применять RVV для решения каких-либо конкретных задач. Одним из применений векторных

расширений являются вычисления в области криптографии. В работе [16] показана реализация метода Гаусса с помощью RVV. Как следует из результатов, авторам удалось добиться прироста производительности в 8-16 раз по сравнению со скалярным кодом. RISC-V позволяет производителями процессоров вводить собственные расширения в свои процессоры. Так, в [7] были разработаны дополнительные инструкции для вычисления SHA3 хеш-функций. После реализации алгоритма на модифицированных ядрах, авторы получили программу, которая ускоряет вычисление криптографической функции более чем в 100 раз. Такое решение может быть подвергнуто критике: в данном случае пропадает совместимость с процессорами других производителей процессоров.

В исследовании [10] используется стандартное векторное расширение RVV. При этом отслеживается производительность реализации RVV на ПЛИС и интегральных схемах специального назначения на протяжении всей эволюции RVV. Как оказалось, первый выпуск векторного расширения улучшил производительность процессоров в среднем на 10%. При этом бенчмарком служило матричное умножение на числах с плавающей запятой.

2.2. Сравнение RVV с AVX512

Как было показано ранее, производительность процессора в конкретных задачах в первую очередь зависит от его микроархитектуры. Поэтому некорректно сравнивать производительность векторных расширений по количеству инструкций или иным количественным метрикам. Чтобы понять, какие преимущества дает RVV по сравнению с другими векторными расширениями, нужно найти классы задач, для которых векторизация с помощью набора команд RISC-V дает явные преимущества по сравнению с другими архитектурами. x86_64 - одна из наиболее популярных архитектур, которая обладает своим набором векторных инструкций SSE/AVX. Для сравнения было выбрано расширение AVX512, так как оно присутствует на большинстве современных

процессоров x86_64.

Для исследования кодогенерации под архитектуру x86_64 используется инструмент «Godbolt Compiler Explorer». В качестве компилятора был выбран *clang 16.0*, так как согласно [9] clang имеет хорошие метрики для векторизации кода под архитектуру x86_64. Так, появляется гарантия, что векторизованный данным компилятором код является оптимальным. Для автовекторизации при помощи AVX512 были использованы следующие опции компилятора:

```
-O2 -march=skylake-avx512
```

Программы для архитектуры RISC-V были написаны при помощи языка ассемблера, так как автовекторизация для масштабируемых векторов недостаточно хорошо интегрирована в современные компиляторы.

Для того чтобы понять, какое преимущество дают масштабируемые векторы, рассмотрим следующую программу на языке C:

Листинг 2: Сложение целых чисел

```
1 void vadd_int(int *restrict a, const int *b, int len) {
2     for (int i = 0; i < len; ++i) {
3         a[i] += b[i];
4     }
5 }
```

Компилятор *clang* с указанными опциями порождает 36 инструкций для архитектуры x86_64. Это примерно 162 байта бинарного кода.
Для RISC-V данная программа векторизуется следующим образом:

Листинг 3: Векторизация сложения целых чисел для RISC-V

```
1 vadd_int:
2 loop:
3     vsetvli t0, a2, e32, m2, ta, ma
4     vle32.v v0, (a0)
5     vle32.v v2, (a1)
6     vadd.vv v0, v0, v2
```

```
7     vse32.v v0, (a0)
8     slli t1, t0, 2
9     sub a2, a2, t0
10    add a0, a0, t1
11    add a1, a1, t1
12    bnez a2, loop
13    ret
```

RISC-V для векторизации того же кода требуется 44 байта. Более того, на строке 3 задается параметр *m2*, обозначающий, что последующие векторные инструкции (строки 4-7) будут оперировать группой из двух регистров. Так, в RVV реализуется раскрутка цикла[8]. Как обсуждалось ранее, RVV-код и правда должен очень хорошо кэшироваться.

Рассмотрим иную функцию на языке C, которая считает количество элементов, пока не встретит нулевой элемент.

Листинг 4: Нахождение длины строки

```
1 int strlen(const char* a) {
2     int i = 0;
3     while (a[i] != 0) { ++i; }
4     return i;
5 }
```

Архитектура x86_64 не позволяет векторизовать подобные вычисления, так как векторные регистры в AVX512 имеют фиксированную длину. Таким образом, при загрузке значений из памяти на конце массива есть шанс попасть на не принадлежащую текущему процессу страницу памяти. В таком случае на процессоре произойдет прерывание и программа аварийно завершится. Поэтому компилятор для архитектуры x86_64 не векторизует данную функцию:

Листинг 5: Нахождение длины строки для x86_64

```
1 strlen:
2     mov     eax, -1
```

[8]Размотка цикла — URL: https://en.wikipedia.org/wiki/Loop_unrolling (дата обращения: 2023-04-29)

```
3 .LBB2_1:
4     inc     eax
5     cmp     dword ptr [rdi], 0
6     lea     rdi, [rdi + 4]
7     jne     .LBB2_1
8     ret
```

В архитектуре RISC-V есть специальные инструкции загрузки (Fault-Only-First Loads), предотвращающие аварийное завершение программы.

Листинг 6: Векторизация нахождения длины строки для RISC-V

```
1  strlen:
2      mv a3, a0
3  loop:
4      vsetvli a1, x0, e8, m8, ta, ma
5      vle8ff.v v8, (a3)
6      csrr a1, vl
7      vmseq.vi v0, v8, 0
8      vfirst.m a2, v0
9      add a3, a3, a1
10     bltz a2, loop
11     add a0, a0, a1
12     add a3, a3, a2
13     sub a0, a3, a0
14     ret
```

На строке 5 используется Fault-Only-First инструкция загрузки. Далее на строке 6 в регистр *a1* помещается количество успешно загруженных байт. Таким образом вычисляется длина строки. При этом условием выхода из цикла является то, что в векторном регистре есть нулевой байт (строки 7, 8, 10).

Одной из основных проблем в современных реализациях SIMD явля-

ется выбор подхода к расположению данных в памяти: массив структур или структура массивов. В руководстве по оптимизации Intel [8] приводятся весомые аргументы в пользу структур массивов, так как их легче векторизовать. Тем не менее набор векторных инструкций RVV предоставляет методы для работы с массивами структур. Для этого в RVV были введены сегментные инструкции загрузки/сохранения. Рассмотрим их использование на примере. Попробуем сложить соответствующие значения полей в двух структурах: RGB и BGR.

Листинг 7: Сложение соответствующих полей структур RGB и BGR

```
1  struct RGB {
2      unsigned char r, g, b;
3  };
4  struct BGR {
5      unsigned char b, g, r;
6  };
7  void add_bgr_to_rgb(struct RGB *rgb, const struct BGR *bgr,
8                      int len) {
9  #pragma clang loop unroll(disable)
10     for (int i = 0; i < len; ++i) {
11         rgb[i].r += bgr[i].r;
12         rgb[i].g += bgr[i].g;
13         rgb[i].b += bgr[i].b;
14     }
15 }
```

Основной векторизованный цикл, полученный при компиляции кода под архитектуру x86_64, состоит из 53-х инструкций. С помощью сегментных инструкций в RVV можно получить следующий компактный код:

Листинг 8: Векторизация сложения полей структур RGB и BGR для RISC-V

```
1  add_bgr_to_rgb:
2  loop:
3      vsetvli t0, a2, e8, m1, ta, ma
4      slli t1, t0, 1
5      add t0, t0, t1
6      vlseg3e8.v v0, (a0)
7      vlseg3e8.v v3, (a1)
8      vadd.vv v0, v0, v5
9      vadd.vv v1, v1, v4
10     vadd.vv v2, v2, v3
11     vsseg3e8.v v0, (a0)
12     add a0, a0, t0
13     add a1, a1, t0
14     sub a2, a2, t0
15     bnez a2, loop
16     ret
```

Инструкции на строках 6, 7, 11 оперируют с группой векторных регистров, загружая каждое поле структуры в соответствующий регистр. Таким образом можно векторизовать алгоритмы, работающие с массивами структур.

3. Приложение для тестирования автовекторизации

В свете того, что RISC-V начал развиваться относительно недавно, не все инструменты разработки работают оптимально. В частности это касается компиляторов, являющихся наиболее сложными и важными программами для разработчиков. Некоторые изменения в основном репозитории компилятора могут вносить неисправности или ухудшение производительности высокоуровневого кода на разрабатываемом процессоре. В ходе данной работы было разработано приложение для тестирования автовекторизации. Оно призвано помочь разработчикам компилятора отслеживать регрессии, которые ухудшают кодогенерацию для процессоров Syntacore.

3.1. Обзор используемых технологий

Как было сказано ранее, единственным компилятором, который поддерживает автовекторизацию для архитектуры RISC-V, является *clang*. Поэтому при разработке приложения использовался данный компилятор.

В качестве программы, тестирующей кодогенерацию, использовалась утилита *FileCheck*[9]. В репозитории *LLVM* она используется для тестирования генерации промежуточного представления из кода на языках высокого уровня, так как данный фреймворк поддерживает множество различных архитектур. *FileCheck* является достаточно гибкой утилитой и подходит для проверки любых файлов. На стандартный поток она принимает какой-либо ввод и проверяет, что данный ввод соответствует меткам в файле, переданном как аргумент командной строки. Если какая-то метка не совпадает, утилита завершается с кодом ошибки и выводит, в каком месте есть несоответствие. Пример использования *FileCheck*:

[9]FileCheck — URL: https://llvm.org/docs/CommandGuide/FileCheck.html (дата обращения: 2023-05-05)

```
clang --march=rv64gv -O2 -S main.c -o- | FileCheck main.c
```

Пример проверяемого файла с исходным кодом:

Листинг 9: Пример программы с метками FileCheck

```
1 // CHECK-LABEL: Increase:
2 // CHECK-NEXT: addiw    a0, a0, 1
3 // CHECK-NEXT: ret
4 int Increase(int X) {
5     return X + 1;
6 }
```

В данном примере *FileCheck* сначала проверит, что в ассемблерном файле есть метка *Increase*, а затем за ней следуют две соответствующие инструкции.

Кроме того, для автоматической генерации меток используется программа update_cc_test_checks.py[10]. Для ее работы в тестируемый файл с исходным кодом необходимо добавлять метки, описывающие работу утилиты *FileCheck*:

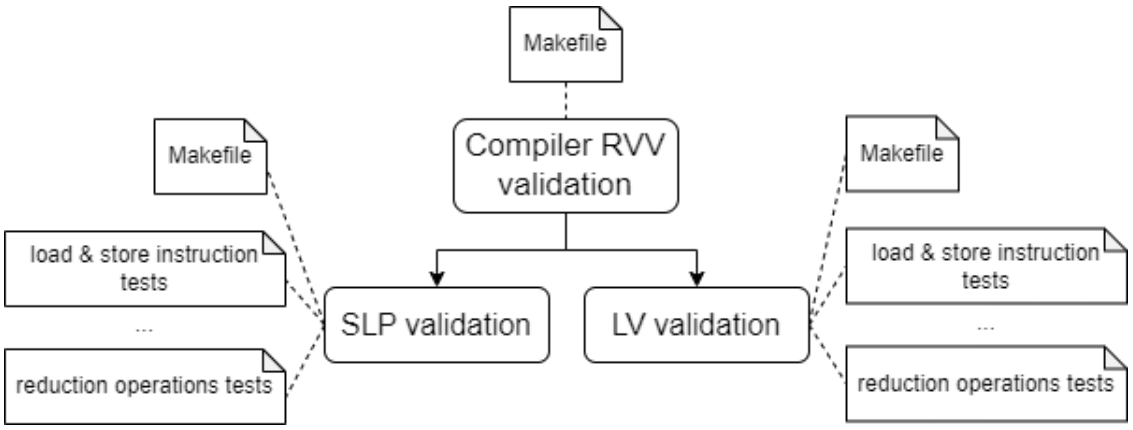
```
// RUN: %clang --target=riscv64-unknown-elf -march=rv64gv
-O2 -S %s -o- | FileCheck %s
```

3.2. Особенности реализации

Приложение для тестирования автовекторизации необходимо, чтобы отслеживать регрессии в кодогенерации. Ухудшение производительности компилятора для процессоров Syntacore могут происходить из-за изменений, приходящих из основного репозитория *LLVM*, и внутренних модификаций. Разработчикам компилятора необходимо поддерживать порождаемый код в оптимальном состоянии, поэтому особенности реализации направлены на простоту анализа изменений.

Диаграмма архитектуры приложения выглядит так:

[10]update_cc_test_checks.py — URL: https://github.com/llvm/llvm-project/blob/main/llvm/utils/update_cc_test_checks.py (дата обращения: 2023-05-05)



В приложении необходимо поддерживать два типа автовекторизации: SLP для векторизации выражений вне циклов, LV для векторизации циклов. Для этого весь проект был разбит на две соответствующие директории. В этих директориях расположены файлы с тестами на языке C. Каждый файл соответствует главе из [15]. Вместе с файлами расположены Makefile'ы с правилами для выбора языка и набора тестов. Выбор алгоритма для тестирования автовекторизации контролируется Makefile'ом, находящимся в корневой директории проекта.

Для того чтобы количество меток не было большим, в тестах на LV-автовекторизацию, использовалась следующая директива:

```
#pragma clang loop unroll(disable)
```

Так как в RVV присутствует аппаратная раскрутка цикла, в большинстве случаев нет необходимости позволять компилятору раскручивать циклы. Помимо этого, тесты становятся менее громоздкими, что позволяет легче анализировать код, сгенерированный в ходе обновления тестов.

Векторизация языка C чувствительна к тому, чтобы области памяти, на которые указывают соответствующие указатели, не пересекались. Поэтому в любых тестах необходимо указывать ключевое слово *restrict* у указателей. Пример использования данного ключевого слова приведен в листинге 10.

В целях упрощения внешнего вида высокоуровневого кода, в тестах SLP-векторизации можно обнаружить циклы. Дело в том, что одним из проходов компилятора является «полная раскрутка цикла», при которой условные переходы и операции сравнения полностью убираются

из программы. Сам цикл заменяется последовательностью однотипных инструкций.

Листинг 10: Пример теста SLP-векторизации

```
1  // CHECK-LABEL: slp_vadd:
2  // CHECK: vsetivli          zero, 4, e32, m1, ta, ma
3  // CHECK-NEXT: vle32.v v8, (a1)
4  // CHECK-NEXT: vle32.v v9, (a0)
5  // CHECK-NEXT: vadd.vv v8, v9, v8
6  // CHECK-NEXT: vse32.v v8, (a0)
7  // CHECK-NEXT: ret
8  void slp_vadd(int *restrict dest, const int *src) {
9      for (int i = 0; i < 4; ++i) {
10         dest[i] += src[i];
11     }
12 }
```

Кроме того, по умолчанию SLP-векторизация отключена для RISC-V. Чтобы включить ее, необходимо указать в опциях компилятора следующие два флага:

`-mllvm --riscv-v-slp-max-vf=0`

4. Приложение для автоматического измерения производительности RVV

Для того чтобы запускать программы, реализованные при помощи RVV, на FPGA, необходимо создать приложение, которое будет интегрировано в существующую систему сборки и запуска приложений.

4.1. Требования к приложению

Данное приложение призвано упростить анализ производительности векторных инструкций. Информация, полученная с помощью данного приложения, должна передаваться разработчикам аппаратного обеспечения. Исходя из этого, были сформированы следующие требования.

Данное приложение предназначено для запуска на процессоре без операционной системы. Поэтому оно должно использовать разработанный в Syntacore пакет поддержки платформы (BSP), включающий в себя начальный код для инициализации процессора. В данном пакете необходимо обеспечить работу векторных инструкций при помощи специальных управляющих регистров.

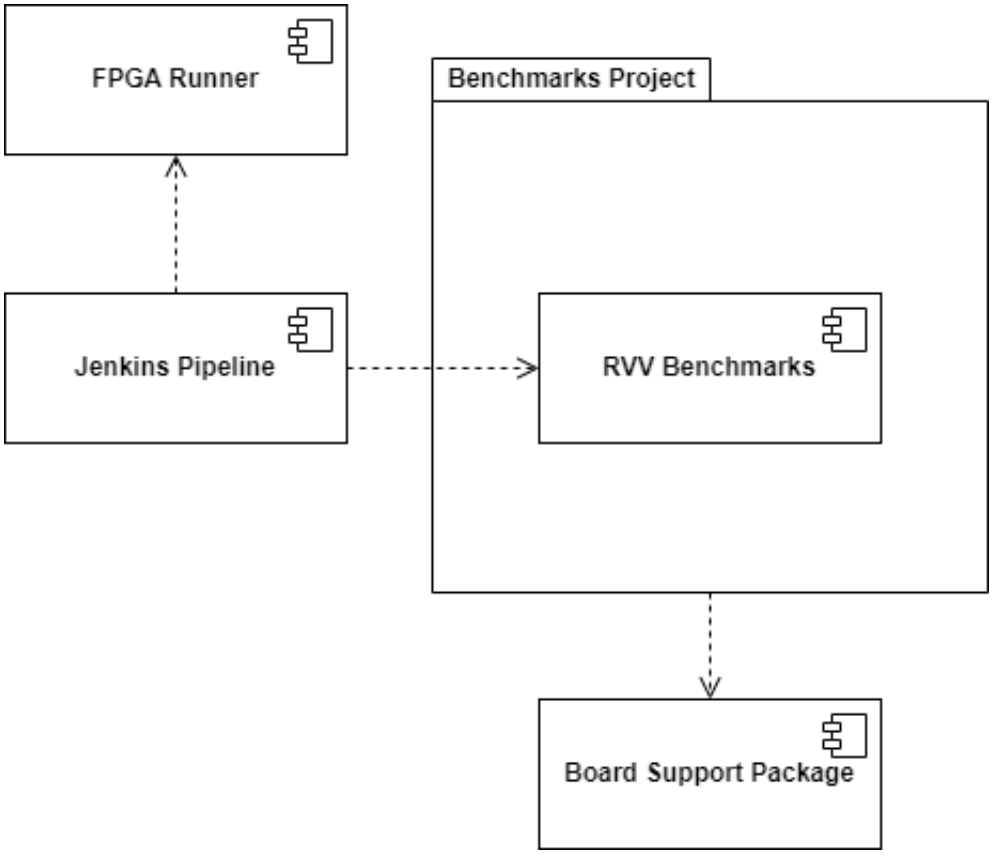
Необходимо считать среднее количество тактов за операцию, среднее количество тактов на байт данных и количество инструкций, исполняемых за такт. Приложение будет запускаться на реализации процессора с частотой порядка 60-ти МГц. В связи с маленькой частотой и запуском на процессоре без операционной системы результаты измерения производительности зачастую повторяемы и различаются не более, чем на доли процента. Поэтому нет необходимости измерять погрешность вычислений. Кроме того, необходимо тестировать корректность измеряемых функций, чтобы своевременно выявлять ошибки процессора.

Так как существующая система сборки не является достаточно гибкой, нет возможности добавлять новые зависимости в проект. В компании «Syntacore» уже используется библиотека *nanobench* для измерения

производительности кода. Она реализована с помощью языка C++, что значительно увеличивает размер исполняемого бинарного файла. После измерения производительности исполняемые бинарные файлы передаются разработчикам аппаратного обеспечения для запуска исполняемого файла на симуляции процессора. Так как время симуляции зависит от размера исполняемого файла, необходимо было отказаться от библиотеки *nanobench* и языка C++. При этом была поставлена задача самостоятельно реализовать вычисления указанных ранее метрик.

Также необходимо автоматизировать сборку и запуск приложения на FPGA. Вместе с этим надо учесть то, что некоторые инструкции могут приводить к зависанию процессора из-за ошибок в аппаратном обеспечении. Поэтому необходимо обнаруживать подобные зависания и запускать бенчмарки раздельно, чтобы не терять информацию о выполненных и еще не запущенных бенчмарках. Ошибки и зависания необходимо журналировать в ходе работы программы.

4.2. Архитектура приложения



Приложение для тестирования производительности необходимо было разработать в контексте существующей инфраструктуры Syntacore. Таким образом, оно является частью проекта с бенчмарками, поставляющегося заказчикам. При этом весь проект зависит от «Board Support Package» (BSP) — пакета поддержки платформы. В нем содержатся следующие компоненты:

- код, инициализирующий механизмы процессора и вызывающий функцию *main*;
- функции для работы с I/O;
- функции-обертки над ассемблерным кодом.

Для того чтобы автоматизировать запуск и сборку бенчмарков использовался Jenkins, являющийся основным инструментом автоматизации в Syntacore.

Кроме того, для запуска приложений на ПЛИС в Syntacore есть приложение «FPGA runner». Оно необходимо, чтобы пайплайн Jenkins мог автоматически отправлять исполняемые бинарные файлы на стенд и запускать их на ПЛИС.

4.3. Особенности реализации

В ходе работы было необходимо разработать с нуля и модифицировать существующие компоненты.

Листинг 11: Включение RVV

```
1 csrr a0, misa
2 li   a1, (1 << ('V' - 'A'))
3 and  a0, a0, a1
4 beqz a0, 1f
5 li   a0, (1 << 9)
6 csrs mstatus, a0
7 1:
```

Для начала необходимо было внести изменения в BSP: поддержать инициализацию RVV в системе сборки. В листинге 11 показан код, устанавливающий RVV в начальное состояние. На строке 1 в регистр *a0* записывается битовая маска, показывающая, какие расширения присутствуют в процессоре. Номер расширения является порядковым номером соответствующей буквы в алфавите. Для RVV это $('V' -' A')$. На строках 3-4 проверяется, что на процессоре присутствует векторное расширение. При положительном результате в управляющий регистр *mstatus* устанавливается бит, соответствующий начальному состоянию RVV.

Приложение для тестирования производительности векторизованных алгоритмов было разработано на языке C с использованием ассемблера RISC-V. Для измерения производительности в наборе инструкций RISC-V есть специальные регистры и команды. Это обязательные для реализации *mcycle* и *minstret*. Для чтения значения из этих регистров необходимо пользоваться следующей командой:

Листинг 12: Доступ к регистрам для измерения производительности

```
1 csrr a0, mcycle
2 csrr a0, minstret
```

mcycle содержит количество тактов, которое прошло с момента запуска процессора, а *minstret* — количество инструкций, которые успешно завершили свое выполнение. Также в RISC-V присутствуют регистры *hpmcounter3-hpmcounter31*, которые согласно [14] могут зависеть от платформы, то есть реализованы так, как необходимо разработчикам аппаратного обеспечения. В Syntacore, например, поддержан подсчет попаданий и промахов кэшей. Данные счетчики были использованы в ходе работы. Над ассемблерными инструкциями были сделаны обертки и в последствии внесены в проект с бенчмарками для того, чтобы ими могли пользоваться другие разработчики.

Как правило, в бенчмарках присутствует тестирование корректности выполнения. Так происходит тестирование процессора в высоко-

уровневых задачах. Кроме того, приложение выводит на экран сообщение об ошибке, если тест не был пройден.

Как было сказано ранее, для автоматизации сборки и запусков бенчмарков был использован Jenkins. Для целей работы был написан пайплайн на языке Groovy. При этом в пайплайне было необходимо резервировать стенд, обрабатывать ошибки, возникающие в ходе работы бенчмарков:

1. ошибки, возникающие, если тесты не прошли:

2. зависание процессора из-за ошибок в аппаратном обеспечении.

Чтобы корректно обрабатывать второй случай, необходимо было компилировать бенчмарки в отдельные исполняемые файлы, раздельно их запускать и журналировать зависания.

5. Тестирование и эксперименты

5.1. Тестирование и анализ автовекторизации

Эксперименты с автовекторизацией проводилось с помощью разработанного приложения для тестирования кодогенерации. В ходе эксперимента использовались тесты, компилятор и опции компилятора из главы про разработку приложения для тестирования кодогенерации.

В ходе тестирования необходимо выполнить две задачи:

1. найти шаблоны кода, которые могут быть векторизованы компилятором;

2. в случае неудачи п. 1 доказать, что компилятор не может это сделать.

На данный момент компилятор векторизует несколько шаблонов алгоритмов. Во-первых, компилятор векторизует инициализацию массивов. Например:

Листинг 13: Инициализациия массива на языке C

```
1 void init_array(int *arr, int len) {
2     for (int i = 0; i < len; ++i) {
3         arr[i] = i;
4     }
5 }
```

Во-вторых, условные операторы в циклах векторизуются в циклах. Это может быть использовано для поиска минимума/максимума в массиве. В листинге 14 приводится пример поиска максимума в массиве. Помимо инструкций сравнения при векторизации используются инструкции редукции.

Листинг 14: Поиск максимума в массиве на языке C

```
1 unsigned init_array(unsigned *arr, int len) {
2     unsigned max = 0;
3     for (int i = 0; i < len; ++i) {
```

```
4           if (arr[i] > max) {
5               max = arr[i];
6           }
7       }
8       return max;
9   }
```

Кроме того, активно векторизуются FMA-операции, в которых одновременно выполняются сложение и умножение. Далее приведен пример, который будет векторизован компилятором.

Листинг 15: FMA-операции на языке C

```
1  void init_array(unsigned *restrict dest, const unsigned *src,
2                  int len) {
3      unsigned multiplier = 23;
4      for (int i = 0; i < len; ++i) {
5          dest[i] += multiplier * src[i];
6      }
7  }
```

Тем не менее у компилятора все-таки есть проблемы с векторизацией под архитекутру RISC-V. Это происходит по нескольким причинам. Во-первых, в RVV векторные регистры устроены так, что их длина не известна во время компиляции и может изменяться во время выполнения программы. В *LLVM* есть частичное решение этой проблемы — масштабируемые векторы в промежуточном представлении. Для этого определяется специальный векторный тип:

⟨vscale × 4 × i64⟩

А для того, чтобы получить *vscale* используется интринсик:

@llvm.vscale.i64()

Параметр *vscale* не известен во время компиляции и является аппаратно-зависимой константой. Это слишком сильные ограничения относительно векторного расширения RVV. Поэтому на данный момент ассемблерный код RISC-V генерируется согласно этим ограничениям. При этом

не используется динамическая настройка длины векторных регистров во время исполнения.

Также у некоторых инструкций в RVV нет аналогов в *LLVM IR*. К таким инструкциям можно отнести, например, следующие:

- сегментную загрузку/выгрузку;
- Fault-Only-First чтение;
- некоторые инструкции для работы с масками.

На данный момент для решения данной проблемы сообщество RISC-V ввело в *LLVM IR* ряд интринсиков:

`@llvm.riscv.*`

С их помощью можно создавать проходы компилятора, которые ищут определенные закономерности в промежуточном представлении и подставляют на их место интринсики[11].

Кроме того, на данный момент не настроена стоимостная модель векторных инструкций. Это связано с тем, что у архитектуры RISC-V есть множество вендоров. При этом у различных процессоров одного вендора может быть различная микроархитектура. Сейчас в репозитории *LLVM* используется общая реализация. Это делается при помощи паттерна CRTP[12].

Также есть проблемы, связанные с SLP-векторизацией. SLP-алгоритм в *LLVM* работает внутри базовых блоков, а значит работа с условными операторами не векторизуется. Более того, алгоритмы разбирает код внутри базовых блоков снизу вверх. При этом требуются последовательные по памяти инструкции выгрузки. Так, следующий код не векторизуется:

Листинг 16: Непоследовательное по памяти копирование

[11] RISCVGatherScatterLowering — URL: https://github.com/llvm/llvm-project/blob/main/llvm/lib/Target/RISCV/RISCVGatherScatterLowering.cpp (дата обращения: 2023-05-06)

[12] CRTP — URL: https://ru.wikipedia.org/wiki/Curiously_recurring_template_pattern(дата обращения: 2023-05-06)

```
1 for (int i = 0; i < 4; ++i) {
2     a[2 * i] = b[i];
3 }
```

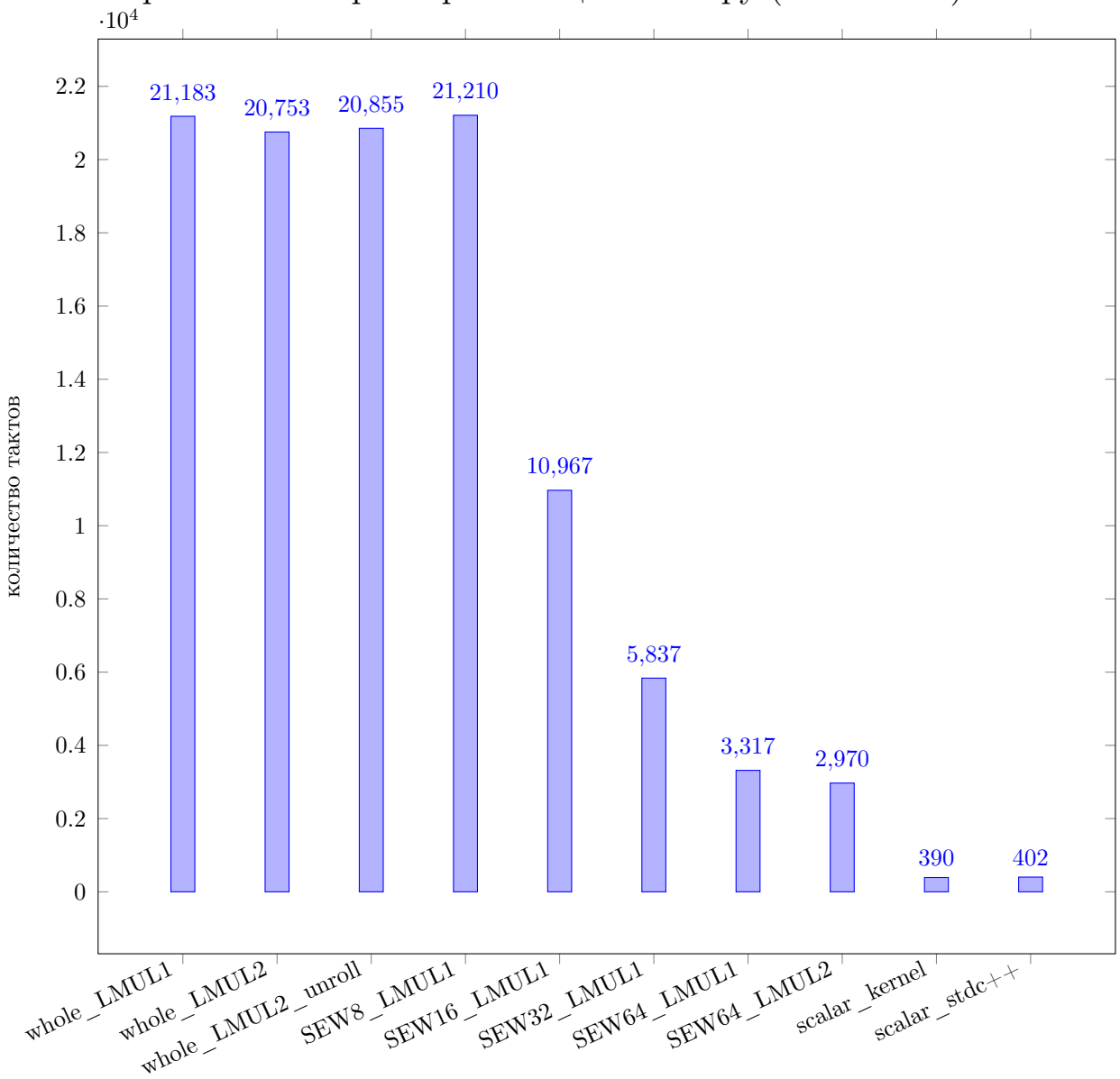
5.2. Измерение производительности RVV на процессоре SCR9

Перед анализом получившихся результатов необходимо обосновать наименования, представленные на диаграммах. Наименования содержат следующие префиксы и суффиксы:

- scalar — префикс, обозначающий скалярную версию бенчмарка;
- whole — префикс, обозначающий использование «упакованного» SIMD;
- SEW<n> — префикс, обозначающий ширину элемента;
- LMUL<n> — суффикс, обозначающий мультипликатор длины векторных регистров;
- unroll — суффикс, обозначающий, что использовалась программная раскрутка цикла;
- kernel — суффикс, обозначающий, что бенчмарк был взят из ядра Linux;
- stdc++ — суффикс, обозначающий, что бенчмарк был взят из стандартной библиотеки stdc++.

Все бенчмарки запускались на последней версии прошивки процессора SCR9 без операционной системы. Компиляция проводилась с использованием пакета поддержки платформы Syntacore. Единицей измерения является такт.

Векторные и скалярные реализации memcpy (1024 байта):

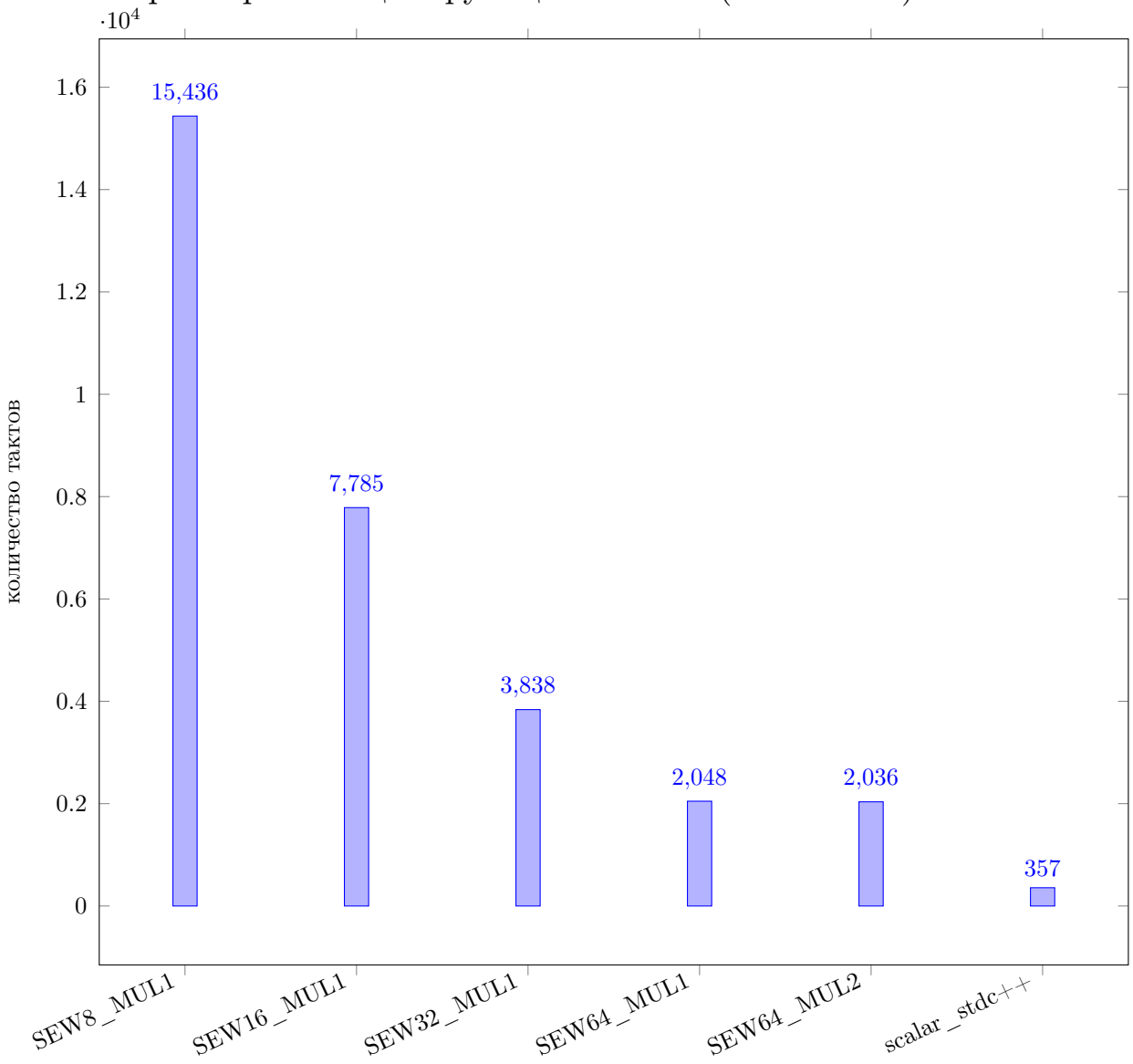


Как можно видеть по результатам измерения, ширина элемента прямо пропорционально влияет на производительность memcpy. Так, наилучший результат показали бенчмарки с шириной элемента 64 бита. Кроме того, программная раскрутка цикла ухудшает производительность, а аппаратная раскрутка, наоборот, увеличивает скорость исполнения примерно на 10%. Тем не менее скалярные версии в 10 раз лучше векторных по производительности. Это связано с двумя особенностями:

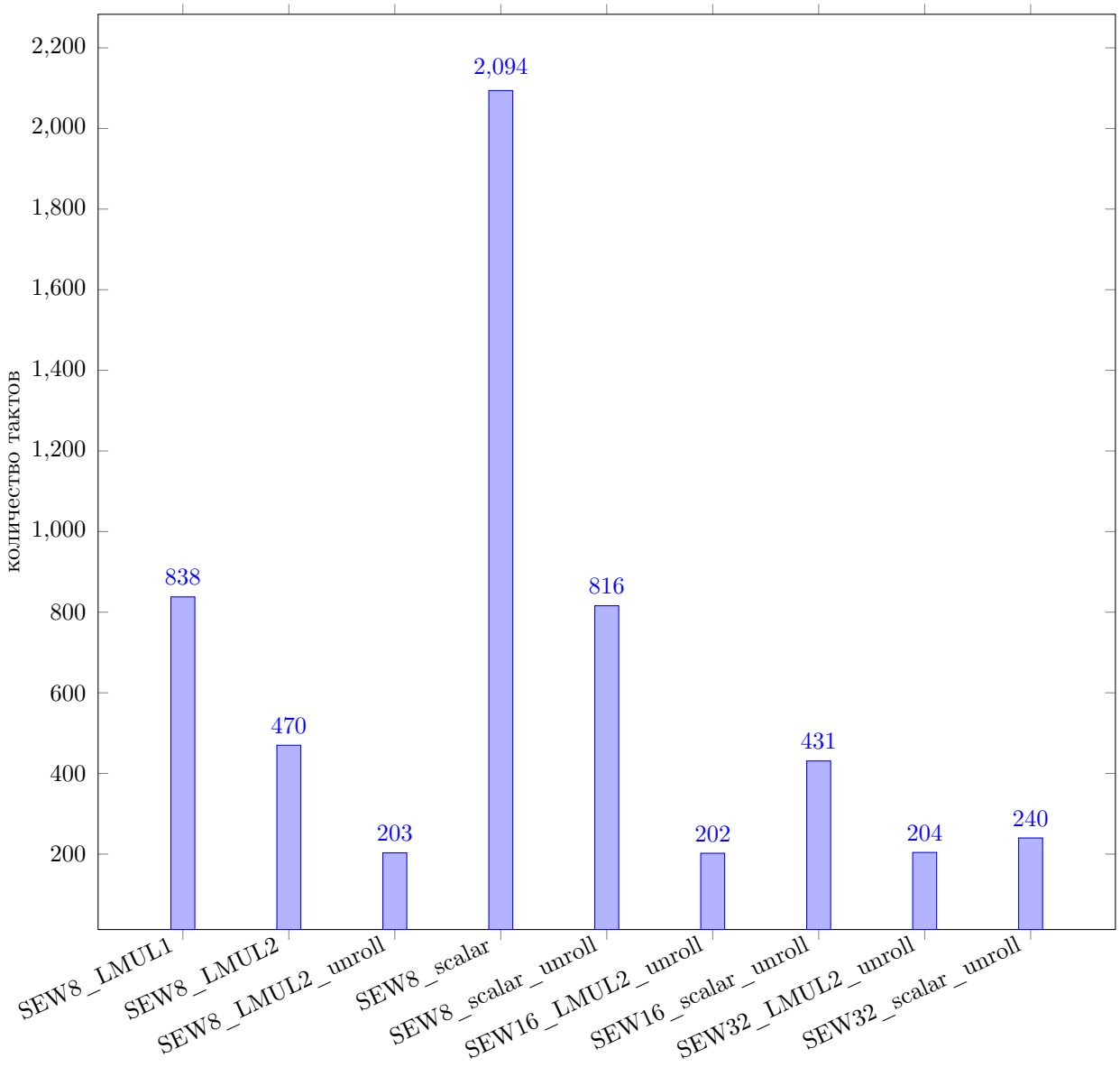
- векторные инструкции загрузки/выгрузки содержат проблемы с кэшами, из-за чего доступ к памяти становится значительно дороже;

- некоторые векторные инструкции являются барьерами, что мешает пайплайну процессора.

По тем же причинам можно видеть отставание по производительности векторных реализаций функции memset (1024 байта):



В предыдущих двух случаях векторизованные функции показали плохие результаты из-за доступа к памяти. Но также необходимо посмотреть на работу векторного арифметико-логического устройства. Для этого был реализован бенчмарк, который измерял производительность векторных и скалярных сложений без доступа к памяти (1024 байта):



В данных тестах использовалась раскрутка цикла на 4 элемента. При этом раскрутка цикла улучшала производительность почти прямо пропорционально как в векторных, так и в скалярных бенчмарках. Кроме того, с увеличением ширины элемента уменьшалась разница между скалярными и векторными версиями. Это ожидаемо, так как максимальная длина векторного регистра фиксирована. При увеличении ширины элемента количество обрабатываемых элементов за векторную инструкцию уменьшается. При этом распараллеливание сложений с помощью векторных инструкций может давать четырехкратное преимущество.

Таким образом, разработчики аппаратного и программного обеспе-

чения могут понимать, на что следует обращать внимание при разработке и улучшении своих продуктов.

Заключение

В рамках данной работы были достигнуты следующие результаты:

- выполнен обзор векторного расширения RISC-V, выявлены его отличительные черты по сравнению с “упакованным” векторным расширением AVX512;
- на основе утилит LLVM разработано приложение для тестирования автовекторизации компилятора clang под архитектуру RISC-V;
- разработано приложение для автоматического измерения производительности векторизованных алгоритмов на FPGA;
- поставлены эксперименты на основе разработанных приложений
  - проведено тестирование автовекторизации с помощью разработанного приложения и выявлены сценарии, при которых алгоритмы не векторизуются компилятором clang;
  - проведены эксперименты с помощью разработанного приложения на процессоре SCR9, выявлены ошибки и недочеты в производительности векторных инструкций на данном процессоре.

Список литературы

[1] Adit Neil, Sampson Adrian. Performance Left on the Table: An Evaluation of Compiler Autovectorization for RISC-V. — 2022. — URL: https://neiladit.com/papers/Performance_Left_on_the_Table_An_Evaluation_of_Compiler_Autovectorization_for_RISC-V.pdf (дата обращения: 2023-04-29).

[2] Arm Developer. Introduction to SVE. — 2022. — URL: https://developer.arm.com/documentation/102476/0100/Introducing-SVE (дата обращения: 2022-12-04).

[3] Charith Mendis Saman Amarasinghe. goSLP: Globally Optimized Superword Level Parallelism Framework. — 2018. — URL: https://groups.csail.mit.edu/commit/papers/2018/oopsla-goslp.pdf (дата обращения: 2023-04-29).

[4] Charith Mendis Cambridge Yang Yewen Pu Saman Amarasinghe Michael Carbin. Compiler Auto-Vectorization with Imitation Learning. — 2018.

[5] Colin Schmidt Albert Ou Yunsup Lee Krste Asanović. Vector Extension Proposal. — 2015. — URL: https://riscv.org/wp-content/uploads/2015/06/riscv-vector-workshop-june2015.pdf (дата обращения: 2022-12-06).

[6] Engheim Erik. Advantages of RISC-V vector processing over x86 style SIMD. — 2022. — URL: https://itnext.io/advantages-of-risc-v-vector-processing-over-x86-simd-1b72f3a3e8
(дата обращения: 2022-12-04).

[7] Huimin Li Nele Mentens, Picek Stjepan. Maximizing the Potential of Custom RISC-V Vector Extensions for Speeding up SHA-3 Hash Functions. — 2022. — URL: https://eprint.iacr.org/2022/868.pdf (дата обращения: 2022-12-04).

[8] Intel® 64 and IA-32 Architectures Optimization Reference Manual. — 2023. — P. 5–20–5–22.

[9] Jing Ge Feng Ye Ping He, Tao Qiu Ming. Evaluation of Compilers' Capability of Automatic Vectorization Based on Source Code Analysis. — 2021. — URL: https://www.hindawi.com/journals/sp/2021/3264624/ (дата обращения: 2022-12-09).

[10] Matteo Perotti Matheus Cavalcante Nils Wistoff Renzo Andri Lukas Cavigelli Luca Benini. A "New Ara" for Vector Computing: An Open Source Highly Efficient RISC-V V 1.0 Vector Processor Design. — 2022. — URL: https://arxiv.org/pdf/2210.08882.pdf (дата обращения: 2023-04-29).

[11] Patterson David, Waterman Andrew. SIMD Instructions Considered Harmful. — 2017. — URL: https://www.sigarch.org/simd-instructions-considered-harmful (дата обращения: 2022-12-06).

[12] RISC-V Assembly Programmer's Manual. — 2022. — URL: https://github.com/riscv-non-isa/riscv-asm-manual/blob/master/riscv-asm.md (дата обращения: 2022-12-04).

[13] RISC-V ISA Manual. — 2019. — URL: https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf (дата обращения: 2022-12-04).

[14] The RISC-V Instruction Set Manual Volume II: Privileged Architecture. — 2021. — URL: https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf (дата обращения: 2023-05-18).

[15] RISC-V "V" Vector Extension Specification. — 2022. — URL: https://github.com/riscv/riscv-v-spec/blob/master/v-spec.adoc (дата обращения: 2022-12-04).

[16] S. Pircher J. Geier A. Zeh, Mueller-Gritschneder D. Exploring the RISC-V Vector Extension for the Classic McEliece Post-Quantum Cryptosystem. — 2021. — URL: https://ieeexplore.ieee.org/document/9424273 (дата обращения: 2022-12-04).

[17] Shahla Gul Noman Aftab Arfa Rani. A Comparison between RISC and CISC Microprocessor Architectures. — 2016. — URL: https://core.ac.uk/download/pdf/235196731.pdf (дата обращения: 2023-04-28).