CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Performance Evaluation of SYCL on RISC-V Vector Architectures

An in depth analysis of different SYCL implementations optimization efficiency on RISC-V Vector Architectures

Master's thesis in Computer science and engineering

Manfred Hästmark, Isak Hansson

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's thesis 2025

Performance Evaluation of SYCL on RISC-V Vector Architectures

An in depth analysis of different SYCL implementations optimization efficiency on RISC-V Vector Architectures

Manfred Hästmark, Isak Hansson

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Performance Evaluation of SYCL on RISC-V Vector Architectures
An in depth analysis of different SYCL implementations optimization efficiency on RISC-V Vector Architectures
Manfred Hästmark, Isak Hansson

Supervisor: Hari Abram, Computer Science and Engineering
Examiner: Miquel Pericas, Computer Science and Engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2025

Performance Evaluation of SYCL on RISC-V Vector Architectures
An in depth analysis of different SYCL implementations optimization efficiency on RISC-V Vector Architectures
Manfred Hästmark, Isak Hansson
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This paper explores the vectorization capabilities of different SYCL compilers, targeting processors with RISC-V Vector Extension, RVV. The primary aim is to evaluate the readiness of various SYCL backends of the AdaptiveCpp SYCL compiler, namely the OpenMP JIT, OpenMP AOT and OpenCL backends. We detail how each approach compiles to RISC-V assembly code, examining how vector instructions emerge or fail to do so across different these different compilation methods. A major part of the work involves discussing how SYCL backends can be configured or adapted to target RVV through toolchains such as LLVM, while also pointing out limitations in support. To support our investigations, QEMU, a general purpose emulator, was used to compile and run RISC-V binaries, and RAVE, a QEMU extension, was used to log instructions executed. These tools were essential in validating whether vector instructions are generated, but also to validate that they are executed. The results show that support for RVV across the ecosystem remains fragmented. Portable Computing Language does not support RVV, nor does it aim to. OneAPI construction kit shows signs of RVV support as vector instructions appear in assembly, however RAVE traces confirm that actual execution does not occur. LLVM offers solid RVV support, but using it with AdaptiveCpp requires modifications to enable full vectorization. In conclusion, although support for RISC-V vectorization is progressing, the broader SYCL ecosystem and its heterogeneous compilation pathways remain underdeveloped, requiring significant manual effort and tooling adaptation to achieve practical performance portability.

Keywords: SYCL, RISC-V, Vector Processing, SIMD, AdaptiveCpp, QEMU, OpenMP, OpenCL, LLVM, Super Computing.

Acknowledgements

We sincerely thank Miquel Pericas for his excellent leadership and willingness to guide and help us get in contact with relevant contacts in the field. Furthermore, we thank Hari Abram for his excellent supervision guiding us on a weekly basis, making sure that we keep our work relevant for the research question.

Manfred Hästmark, Isak Hansson, Gothenburg, 2025-06-23

Contents

List of Figures

List of Tables

1

Introduction

Recently, the European Union have been investing funds into SYCL related projects, such as DARE and SYCLOPS [1], [2]. SYCL is an abstractions layer for heterogenous computing using C++, developed by the Khronos group. It aims at abstracting hardware accelerators such as GPU's, FPGA's and more behind a common interface [3]. In these projects, both AdaptiveCpp and Intel's oneAPI compiler have been big focus points [4], [5], these are SYCL compilers aimed at implementing Khronos's SYCL abstraction layer.

Meanwhile, at the Barcelona Supercomputing Center, one of Europe's largest, they are researching a new RISC-V based chip focused on using vectorization as an accelerator [6]. The aim is for the chip to be used in future European supercomputers. What's special about the chip is the RISC-V's instruction set architecture (ISA) and the use of its Vector Extension (RVV) 1.0, which can compute multiple operations at once, similar to a GPU, but with single instructions only as part of the CPU.

Due to these recent developments, an interesting research question becomes how SYCL can utilize the recent developments in RISC-V. More specifically, how well the SYCL abstractions layer can be used to utilize the vectorization capabilities of RISC-V.

1.1 Aim

This master thesis aims at exploring the vectorization capabilities of SYCL compilers and their compilation methods in relation to RVV. The focus is on assessing how effectively these methods can generate vectorized code and if there are any significant differences between them. The goal is to identify the current strengths, limitations, and weaknesses in SYCL support for RISC-V vector processing.

1.2 Limitations

The original goal of this thesis was to evaluate multiple SYCL compilers across various backends. However, this task had to be scaled down due to limitations in compiling for the RISC-V architecture and setting up emulation environments. This meant that only a select of compilation methods of AdaptiveCpp were analyzed instead.

1.3 Background

The background serves to prove the fundamental assumptions on which the relevance of this topic is based. It will start by covering current trends in heterogeneity in supercomputers, followed by an introduction of the RISC-V architecture and its potential benefits in future computing. Furthermore, the SYCL programming model is presented, along with arguments for why the SYCL programming model may be the solution to adopt and use future heterogeneous computing systems. Lastly, a more in depth view is given on RISC-V with an explanation of what makes RISC-V vectorization special and why it is beneficial as a means for performance increase.

1.3.1 Trends in heterogeneous computing

In 1965 Dr. Gordon E. Moore [7] published his famous paper "The future of integrated electronics", and thereby coined Moore's law [7]. Moore's law states that the transistor density of IC's doubles every 18 months, something which has been a driving factor in the semiconductor industry in the past decades. The industry has managed to keep pace with this growth, though some suggest that Moores law has evolved into more of a metaphor for technological innovation than a literal prediction of transistor density.

Moreover, as predicted already back in 1989, the dynamic power dissipation would eventually come to limit the CPU clock frequency, leading to more research in architectural components. Subsequently, in 2005, the multicore architecture was introduced [8]. Furthermore, *accelerators*, specialized at solving problems of a specific nature, were also introduced to combat, and keep the growth of computing constant. One such example is the introduction of GPU's, as a means to solve highly parallelizable problems. These types of accelerators will serve as one of the means to keep the constant growth of computing, along with the introduction of stacked integration approaches for integrated circuits. This is essential to keep the growth of computing steady, as two-dimensional scaling is expected to stop by 2031 [8]. This has led to the coining of a new mantra:

"The number of bits produced in an integrated package can cost effectively double every 2 years" - Paolo Gargini 2020 [8]

The quote seeks to expand on Moore's law about transistor doubling, to include innovational techniques as a means to achieve the same results as the doubling of transistor density has been able to in the past. This trend can also be observed by analyzing current supercomputers in the top 500 list [9], illustrated in Table 1.1, which shows that current supercomputers use heterogenous approaches to achieve high performance.

Table 1.1: Excerpts from top 500 list illustrating heterogeneous capabilities of supercomputers

Computer	CPU	CPU-capabilities	ISA
El Capitan	AMD 4th Gen EPYC 24C 1.8GHz	AVX-512, 24-cores [10]	x86
Frontier	AMD 3rd Generation EPYC 64C 2GHz	AVX-512, 64-cores [10]	x86
Aurora	Xeon CPU Max 9470 52C 2.4GHz	AVX-512, 52-cores [11]	x86
Fugaku	FUJITSU A64FX	SVE 128/256/512-bit, 52-cores [12]	ArmV8.2+SVE

1.3.2 Introduction to RISC-V

Reduced Instruction Set Computing (RISC) emerged in the early 1980s as a response to the increasing complexity of Complex Instruction Set Computers (CISC) [13]. Patterson and Ditzel [13] argues for cost-effective designs through simple architectures, emphasizing efficient resource utilization and faster design processes. RISC provides a smaller set of instructions that can execute at higher frequencies, removing the support for infrequent, synthesizable instructions.

RISC-V is a result of several generations of research projects at the University of California, Berkeley [14]–[17]. The original instruction set architecture (ISA) for the RISC-V processor was published in 2011 along with a 28nm FD-SOI chip [18]. The goals of the ISA were to be a realistic yet open architecture that captures the essential commercial features [19]. It should support both 32-bit and 64-bit address space variants, accommodates highly parallel multicore or many core systems (including heterogeneous processors). Asanovi and Patterson [20] claims open source ISAs to have several key advantages over closed source, mainly a reduced barrier of entry, increased cost efficiency and greater competition.

Over the years Berkeley continued developing the ISA of RISC-V. The project was later transferred to the RISC-V Foundation [18]. However, due to the geopolitical landscape of the US, RISC-V Foundation was later relocated to Switzerland and renamed to RISC-V International in 2019.

What distinguishes the RISC-V architecture from traditional architectures like x86 is its vector processor. The x86 architecture uses fixed-width vector implementations, which are typically limited to 256-bit or 512-bit vector registers (e.g., AVX-512). As of September 19th 2021, the RISC-V Foundation introduced version 1.0 of its Vector Extension [21]. RISC-V adopts a more flexible and scalable approach similar to ARM's scalable vector extensions (SVE) [22], which support variable-length vector registers ranging between 128 and 2048 bits. However, RISC-V extends this scalability even further, allowing vector lengths to range between 128 bits to a maximum of 65,536 bits.

Two upcoming European projects named SYCLOPS and DARE [1], [2] aim to use RISC-V in high-performance applications. The European Processor Initiative (EPI) is a major European effort to develop competitive supercomputing capabilities, positioning Europe alongside global leaders such as the United States and China. It focuses on designing high-performance, energy-efficient processors to support applications in AI, scientific computing, and autonomous systems [23]. The EPI

is currently developing a heterogeneous chip capable of delivering one exaflopone quintillion floating-point operations per second. This chip integrates RISC-V vector tiles, specialized deep learning and stencil accelerators, and variable-precision floating-point cores to balance peak performance with energy efficiency [6]. The RISC-V vector unit in the chip, named Avispado, supports vectors up to 256 double-precision floating-point elementsequivalent to 16,384-bit wide vectors.

1.3.3 Utilizing heterogeneous hardware and the growth of SYCL

There are multiple ways of using hardware accelerators, and the type of hardware accelerator affects the implementation. For example, programming Nvidia GPUs requires the use of CUDA, a programming model which interfaces with their GPUs using the C++ programming language [24]. However, certain features, such as vectorization, are natively supported by most modern C++ compilers. For example, the LLVM backend used by the Clang compiler provides vectorization support across all supported instruction sets [25].

The fragmentation in programming interfaces for hardware accelerators led the Khronos group to release their first specification of the SYCL programming interface for C++ in 2014 [3]. The purpose of the interface is to abstract heterogeneous computing, enabling more flexible integration of accelerators in software while also acting as a catalyst for the future evolution of the C++ language.

In the scope of computing, the SYCL programming model can be considered quite new. However, it has already gained traction as an HPC computing interface. For example, Aurora, MareNostrum and Dawn, listed on the top 500 list [9] uses Intel's oneAPI compiler. Furthermore, the European Union is investing in projects such as SYCLOPS and Dare [1], [2], which focus on the development of SYCL compilers in conjunction with RISC-V. Currently, there exist five implementations of the interface, listed below.

- **Intel oneAPI** - Full support for SYCL 2020 interface [4].
- **AdaptiveCpp** - Independent, community-driven modern platform for C++-based heterogeneous programming [5].
- **triSYCL** - A research project to experiment with the specification of the SYCL standard [26].
- **neoSYCL** - A SYCL implementation for SX-Aurora TSUBASA [27].
- **SimSYCL** - Single-threaded, library implementation of SYCL 2020 to test SYCL applications against simulated hardware [28].

Furthermore, Intel's oneAPI compiler and AdaptiveCpp are the most active projects on GitHub, with triSYCL being the third most developed compiler with SYCL support [4], [5], [26]. In this master thesis, a closer analysis is performed using the AdaptiveCpp compiler.

1.4 Related work

Research on the SYCL programming language has been conducted across various hardware platforms, with the most prominent being GPUs from vendors such as NVIDIA, Intel, and AMD, as well as x86 CPUs [29]–[32]. SYCL research commonly tests portability and performance, for instance, Reguly [30] evaluated the effectiveness and portability of SYCL across CPUs and GPUs using multiple compilers. Reguly highlights the effectiveness of SYCL on GPU hardware and adds certain CPU limitations. Likewise, L. Crisci et al. [29] created the SYCL-Bench 2020 benchmark suite [33] for testing key SYCL features, like unified shared memory (USM and reduction kernels. They performed their tests on AMD, Intel and NVIDIA GPUs, providing valuable insights on GPU performance and portability.

AdaptiveCpp is a SYCL compiler which integrates multiple backends, including OpenMP and OpenCL. The compiler was rigorously benchmarked utilizing both CPU and GPU in the paper by W. Lin, T. Deakin and S. McIntosh-Smith [31]. They showed that AdaptiveCpps performance is competitive with other SYCL implementations and native programming solutions like CUDA. However, most evaluations of SYCL has been conducted using the x86 hardware platform, with limited testing of the newer upcoming architectures like RISC-V.

There has been limited research on the RISC-V architecture, including RVV 1.0. RVV offers a scalable and versatile model for hardware acceleration, still its integration with high-level programming models like SYCL continues to be underexplored. This thesis aims to address that gap by analyzing AdaptiveCpps SYCL compiler and its different backends capability to generate vectorized code for RVV. This paper will look into assessing its performance utlilizting both OpenMP and OpenCL, highlighting current strengths and limitations in SYCLs support for RISC-V vector processing.

2

Theory

The theory chapter contains key concepts required to get a complete picture of RISC-V and SYCL compilers. It will begin with an introduction to the internal architecture and distinctive characteristics of RISC-V processors. This is followed by showcasing SYCL and its programming abstraction layer in combination with describing the compilation methods of SYCL. Lastly, the theory will cover emulation software and benchmarking utilities, and what makes it possible to run RISC-V on an x86 machine.

2.1 RISC-V Profiles

RISC-V profiles are a growing set of standard hardware extensions, an example being the floating-point extension [34]. The goal with profiles is to give the ability to selectively optimize hardware while providing a shorthand for describing the ISA standards which the profile follows. Each profile is built on the base ISA with the additions of other capabilities. For example, the RV32E profile targets low-power embedded systems with a minimal core, while the RV64GC profile includes integer, multiplication/division, atomic, floating-point, and compressed instruction sets, making it suitable for operative systems.

2.2 ABI

An Application Binary Interface (ABI) defines the low-level rules that govern how software components interact at the machine-code level [35]. It specifies details such as how function parameters are passed, how return values are handled, and which registers are used and preserved. It also covers stack organization, memory representation, the naming, and linking of compiled symbols. Unlike an API, which operates at the source-code level, an ABI ensures binary compatibility between separately compiled modules, such as between a program and a library or between user code and the operating system.

In the context of RISC-V, the ABI plays a critical role in determining how integer and floating-point values are passed during function calls [35]. RISC-V supports multiple ABI variants, including lp64 (64-bit with no floating-point), lp64f (with single-precision float), and lp64d (with full double-precision support). These variants control if the compiler should use hardware floating-point instructions or calls to

software-emulated helper functions like *__adddf3* and *__muldf3* [36]. For systems with a hardware FPU, using the lp64d ABI ensures efficient use of instructions like *fadd.d* and *fmul.d*, while lp64 results in slower, software-based implementations.

2.3 Vector processors

Vector processing is a concept of applying a single assembly instruction to an entire array of values simultaneously. Hence, single instruction, multiple data (SIMD). SIMD is common in both consumer processors and supercomputers as a general-purpose hardware accelerator, improving for example vector or matrix arithmetics without the need to offload work to external devices.

Vector architectures are designed to maximize parallelism by leveraging memory locality and pipelining. It does this by using a multitude of vector components and functionalities.

- Vector Registers, a way to efficiently store values in arrays.
- Vector Functional Units to execute arithmetic on said values.
- Vector Length Registers to express the size of the vector.
- Vector Load & Stores for specialized segmented manipulation of memory.

The speed-up achieved through vectorization is determined by the fraction of a program that can be vectorized and the speed of vector execution relative to scalar execution [37]. This relationship can be expressed using the following equation (2.1).

$$S = \frac{r}{(1-f) * r + f} \tag{2.1}$$

Where S is the overall speed-up due to vectorization, r is the speed-up factor of the vectorized portion relative to scalar execution, and f is the fraction of the program that can be vectorized.

2.4 SYCL

SYCL is a royalty-free abstraction layer for heterogenous computing in C++. It provides a single-source programming model, and is developed by the Khronos Group [3]. With SYCL, developers can target CPUs, GPUs, FPGAs, and other accelerators without having to change the source code.

The main execution model of SYCL is based on five SYCL constructs; queues, buffers, accessors, unified shared memory (USM) and kernels.

- Queues handle execution across devices and allows for dispatching kernels and memory operations.
- Buffers represent shared memory objects.

- Accessors is responsible for handling memory dependencies and correct data flow.
- Kernels are used to wrap larger operations, which can be defined using lambda functions, or explicit kernel classes.
- Unified shared memory (USM) allows for distinct memory allocation where the user can choose to allocate the memory on host, device or shared.

SYCL's execution utilizes a hierarchical structure across devices and divides execution into three levels [38]. The host program controls execution by managing memory transfers and queuing commands into SYCL queues. The SYCL runtime schedules kernels, controlling execution order by resolving dependencies. At the device level, kernels execute on individual work-items. The work items are grouped into multiple work-groups to enable optimal shared memory usage.

SYCL queues are key to kernel execution and memory management, determining the schedule on devices [38]. Queues can operate in in-order mode, executing commands sequentially as submitted. They can also be executed in out-of-order mode, which optimizes execution based on dependencies, improving performance. Additionally, SYCL supports multiple queues targeting different devices, enabling multi-device execution for efficient resource utilization.

2.5 Just-in-time compilation

Just-in-time (JIT) compilation is a technique that compiles code during runtime instead of traditional ahead-of-time (AOT) compilation. It utilizes the benefits of both interpreted execution and AOT compilation, allowing for program optimization during runtime. It does this by compiling the source code into intermediate representation (IR) and interpreting it during runtime. This allows for runtime optimizations of hot paths, such as inlining functions, loop unrolling and branch predictions. Optimized code is executed and cached for later use. Reaching optimal speed might take several optimization passes.

JIT compilation plays an important role in optimizing code for heterogeneous computing. SYCL builds upon a generic compiler which converts C++ code into IR. The IR is then compiled to different kernels dynamically during runtime, allowing for execution on multiple target devices.

2.6 SYCL Compilers

Different SYCL compilers each implement their compilation flow in different ways. The SYCL standard supports both JIT and AOT compilation and it also allows for parsing the source code one time only or multiple times per compilation. This means that different SYCL compilers such as AdaptiveCpp and oneAPI have vastly different implementation structures. In this section, AdaptiveCpp's different compilation flows for different targets are covered.

2.6.1 AdaptiveCpp

AdaptiveCpp implements multiple different compilation flows depending on target and also what compilation method is used. It has a library only compiler flow for AOT compilation with OpenMP, and it also has a JIT compilation flow which supports OpenMP JIT and OpenCL JIT depending on the target. Furthermore, it can also target AMD and NVIDIA GPU's directly [5].

AdaptiveCpp implements a single-pass compiler flow, which is unique for this compiler while writing. The main advantage with the single pass compiler pass are shortened compile times, especially when compiling against multiple targets [39].

AdaptiveCpp consists of four different parts, the interface, runtime, compiler, and the glue [5], each with a different purpose.

- The interface provide SYCL's functions and classes which the user interacts with. Along with this, it also implements kernel libraries with device specific implementations.
- The runtime implements the task management and decides what backend the different tasks should be run on. The different backends/devices are loaded dynamically during runtime and is using a Python script which wraps the different compilers for the different backends with a common interface.
- The compiler is managed by a Python script named acpp, which offers a consistent interface to the compilers of the various backends.
- The glue handles the kernel launchers and requires a unique implementation for each backend.

The main focus of AdaptiveCpp is their generic target which is a (JIT) implementation of the SYCL interface [5]. This target works by first translating the SYCL kernels into generic unoptimized llvm intermediate representation (IR), which is one step above assembly language in the LLVM compiler toolchain [40]. This IR is then optimized in multiple stages, applying different types of optimizations and specilizations in each stage. In the first stage, the kernel is outlined in the IR. Next, various specializations are applied, followed by the processing of reflection queries. Finally, information obtained through JIT is used to further optimize the kernel. Lastly, the IR is optimized for the specific backend that is targeted, and then some last reflections are resolved which mostly come from AdaptiveCpp builtins [5].

2.6.2 Compilation methods

Compilation methods (or backends) serve as the underlying execution platforms that allow SYCL to interface with different hardware architectures. The choice of backend determines how SYCL kernels are compiled, scheduled, and executed on the target hardware. Key backends include OpenMP, OpenCL, CUDA, HIP, and Level Zero, each catering to different hardware and use cases. In our case, we are using RISC-V as both the host and device, utilizing its RVV for parallel execution, this requires a backend which can efficiently vectorize for RISC-V hardware.

The OpenMP backend is most commonly used for CPU execution and translates the parallel SYCL constructs into OpenMP parallel regions utilizing OpenMPs thread-based execution model, enabling efficient exploitation of multicore CPU architectures through dynamic scheduling and workload distribution.

The OpenCL backend allows SYCL to execute workloads on OpenCL-compatible devices such as CPUs, GPUs, FPGAs, and other hardware accelerators. A key component of this backend is the Installable Client Driver (ICD) loader [41], which allows multiple OpenCL drivers to coexist on a system. The ICD loader dynamically selects the appropriate OpenCL implementation at runtime, enabling flexibility in targeting different hardware without modifying the SYCL application.

2.7 Emulation

Qemu (Quick Emulator) is an open-source machine emulator and virtualizer that supports a wide range of hardware architectures, including RISC-V support [42]. It enables developers and researchers to experiment with different instruction sets and system configurations without the need for actual physical hardware. Qemu has several operating modes, each designed to suit a specific use cases.

System emulation mode, emulates an entire hardware platform including the CPU, memory and various I/O devices [42]. This allows the possibility to run a complete operating system designed for a different architecture than that of the host machine. System emulation is valuable in contexts such as kernel development, operating system-level debugging, and firmware testing, where having a complete and controlled emulated environment is crucial.

In contrast, user-mode emulation allows Qemu to run individual programs compiled for a different architecture than the host [42]. Rather than simulating a whole system, this mode focuses on translating and executing guest binaries. System calls made by the guest application are intercepted and translated into corresponding native calls on the host system. This is useful for tasks such as benchmarking, since logs of executed instructions can be collected without the overhead of simulating a full operating system.

2.7.1 RAVE and Paraver

RAVE is a plugin for qemu-riscv64 user mode emulation [43], and the plugin enables logging of all instructions executed. More specifically, all vector instructions are logged at an instruction level granularity, while regular scalar instructions are only counted and not categorized. The recorded instructions are output in a plain text file in Paraver format. The logs can then be analyzed using a program called Paraver [44], provided by the Barcelona Supercomputing Center. Furthermore, RAVE provides a set of configurations for Paraver which enables quick analysis of the amount of different instructions executed and also timelines of when the different instructions were executed. These tools enable analysis of the amount of vector instructions executed and also how many instructions were executed in total for the kernel.

However, it does not allow for analyzing the execution time.

3

Methods

This chapter aims to give an overview of the methods used to retrieve the results, as well as covering the benchmarks used. Furthermore, reasoning is given as to why a particular method was chosen and what other options exists.

To begin, there exists multiple ways of evaluating the performance of vectorization against RISC-V, and each have their pros and cons. In the case of this project, the main goal is to compare the vectorization capabilities of different SYCL backends against RISC-V. The emulator chosen was the RAVE [43] emulator which could provide the number of vector instructions executed and with what vector length as well. The data, in combination with manual analysis of compiler output, would be sufficient to compare vectorization efficiency of different compilers.

3.1 Emulation method

In order to analyze the vectorization capabilities of SYCL-compilers on RISC-V, a method that could quantify the amount of vectorization optimizations performed by these compilers was needed. This quantification can be performed at varying levels of granularity, each offering a different degree of precision, often involving a trade-off with simulation or run-time performance. Since the aim was to analyze the vectorization capabilities of SYCL compilers, the RAVE [43] emulator, which provided each vector instruction executed along with number of scalar instructions, was chosen. The RAVE emulator is a plugin for Qemu user mode emulation, which also meant that the emulations were quick to run.

Furthermore, Qemu user mode emulation required an environment to passed to the emulator for the binaries that were run in this thesis, which meant that this had to be developed. This is because Qemu can run statically linked binaries without needing any environment variables being set. However, when JIT-compiling binaries with AdaptiveCpp or oneAPI, its impossible to compile statically. On top of this, a compiler is needed at runtime to compile the generated immediate representation into binary code that can be run. Once the binary has been JIT-optimized, it caches an executable form, removing the need for a compiler at runtime. Furthermore, Qemu user mode emulation provides the *-L* flag, which sets the root directory for where the binary should look for dynamically linked libraries. These features allows for JIT binaries to execute inside of RAVE.

The solution to JIT compilations was to setup an emulated RISC-V operating system using Qemu, in order to have an environment to compile and run the RISC-V binaries on. Debian's Trixie release branch [45] was used as it had a release for RISC-V, and the aptitude package manager meade it easy to install dependencies. However, any Linux distribution with a kernel with RVV support would work.

Another issue to run the compiled binaries in RAVE. Initially, RAVE was compiled on the same RISC-V system running Debian. However, when running RAVE in an emulator, it was not able to capture any of the executed instructions, and thus, did not provide any data. However, when running RAVE on the host x86 system with a test binary, it was able to capture the instructions executed. Therefore, some way of running the RISC-V JIT binaries in RAVE on the host operating system had to be figured out.

The solution to this issue was to utilize a shared folder between the emulated Debian RISC-V system and the host system. This shared folder could then be accessed by RAVE in order to give access to shared libraries needed by the JIT binary. More specifically, the shared library generated by the JIT compilation in the JIT-cache was also put in the shared folder, such that the JIT binary could be run and optimized on the emulated system, and then ran in RAVE without having to do the JIT optimizations. The Qemu image for the emulated RISC-V system was also mounted to a folder, and passed via the *-L* flag to RAVE in order for Qemu to find all shared libraries that it had access to inside the Qemu Debian RISC-V system. An illustration of the simulation environment can be seen in figure 3.1.

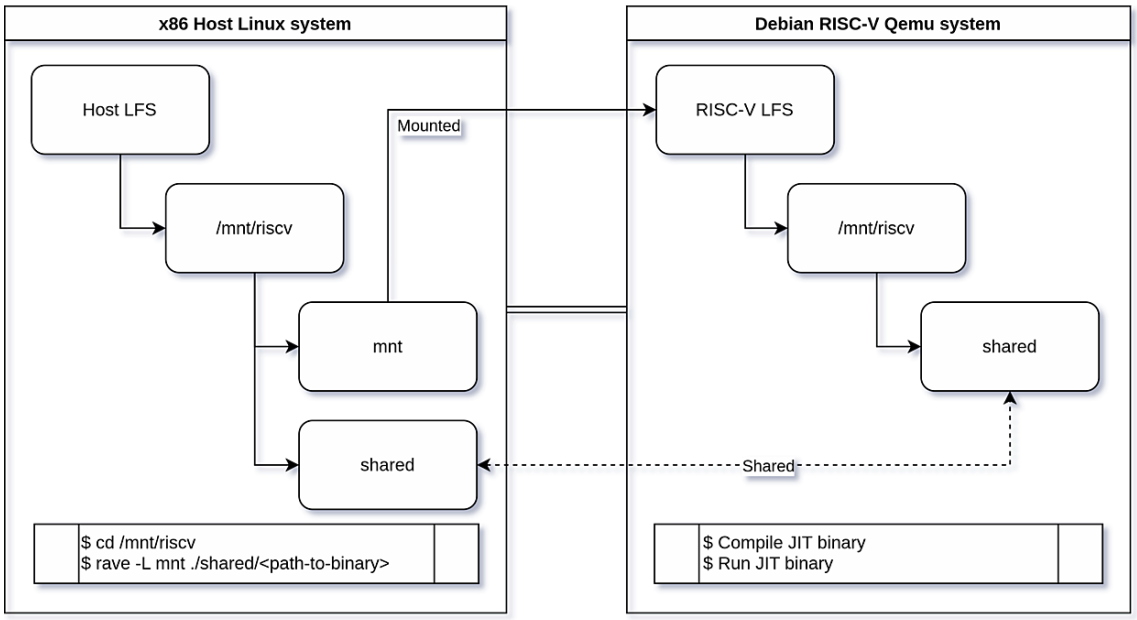

Figure 3.1: Concept for architecture for evaluation platform

3.2 Analyzing and running AdaptiveCpp with OpenMP on RISC-V

As mentioned in section 2.6.2, AdaptiveCpp implements multiple different backends, were two of them utilize OpenMP and LLVM to either JIT or AOT compile depending on the AdaptiveCpp target chosen. In order to analyze the vectorization capabilites of these backends, some compile flags and modifications has to be either passed or added in AdaptiveCpp.

Since both of the OpenMP backends of AdaptiveCpp rely on LLVM for compilation to binary code, the appropiate flags have to be passed to LLVM such that it can optimize the binary to the best of its abilities. The first flag that is passed is the *-O3* flag, which is the highest optimization level that LLVM recommends using [46]. Furthermore, LLVM also requires knowledge of which hardware to optimize for, and since there is no *-native* flag for RISC-V in LLVM yet, the *-march=rv64gcv_<vlen>b* has to be used instead. When compiling with AOT, this flag can be passed directly as a command line argument to the AdaptiveCpp compiler. However, when compiling with the JIT backend, the AdaptiveCpp source code has to be modified since AdaptiveCpp only adds the *-native* flag (if supported on the ISA). This can be done by editing manually setting the *HOST_CPU_FLAG* in the file *src/compiler/llvm-to-backend/CMakeLists.txt*.

Furthermore, in order to aid in the analysis of compiler optimizations, and especially for vectorization, LLVM provides a few different flags. These are the *Rpass-analysis=loop-vectorize -fsave-optimization-record* in order to emit a report of what optimization were applied by LLVM to the source/IR file, and also *-S -g -emit-llvm* flags in order to emit assembly or intermediate representation (IR) source code with debugging comments. In order to analyze the AOT compilations, the easiset way is to extract the kernel into a separate source file and compile it as a shared library by passing the *-shared*, this makes the output files a bit smaller. Furthermore, two passes have to be made, one where the *-emit-llvm* flag is passed and one without, this gives both assembly and IR source code output.

In order to pass the same flags to the JIT compiler, the source code for AdaptiveCpp has to be modified further as the flags passed as command line arguments are not provided to the JIT compiler. This can be achieved by adding the same flags to the *Invocation* vector on line number 147 in *src/compiler/llvm-to-backend/host/LLVMToHost.cpp*. In order to also get the IR for the kernel, the environment variable *ACPP_S2_DUMP_IR_ALL* has to be set to 1 in order for AdaptiveCpp to emit the IR.

3.3 OpenCL oneAPI Construction Kit

OneAPI Construction Kit, part of the UXL Foundation, is a framework providing open standards such as OpenCL. This Kit enables SYCL offloading to custom

hardware, and in our case, RISC-V. We found compiling was easiest using the latest development branch, however note that across all branches and releases tested, the removal of the *-Werror* flag was necessary due to persistent compilation warnings, regardless of the LLVM version. The project was configured with LLVM 19, the only version supported on the main branch at the time of writing, and *CA_RISCV_DEMO_MODE* was enabled to allow disassembly output from SYCL kernels [47]. The complete set of required apt packages, as well as the build invocation used for the toolchain, is documented in Appendix A.2 for reproducibility.

Vectorization behavior is configured using the *CA_RISCV_VF=<SVA1-64>* environment variable. Here, S activates scalable vectorization based on RVV, V limits vectorization only, and A allows the Vecz, a vectorizing component, to determine the vectorization factor. Setting this variable to a value between 1 and 64 directly controls the vector width. Additionally, the *CA_RISCV_VLEN_BITS_MIN* flag sets the minimum reported minimum VLEN bits, which may also override the VLEN if a Hardware Abstraction Layer (HAL) supports it, and should only be used if the actual VLEN used in the device is updated. Assembly dumps were generated by exporting the environment variable *CA_RISCV_DUMP_ASM=1* before invoking the runtime.

3.4 OpenCL PoCL

The Portable Computing Language (PoCL) is an open-source implementation of the OpenCL standard designed to enable portable performance across a wide range of hardware targets. PoCL currently supports a variety of targets including x86, ARM and RISC-V CPUs, NVIDIA GPUs through libCUDA, Intel GPUs. PoCL can be used with or without an ICD loader, making it flexible for both standard and minimal environments. When operating without an ICD loader, PoCL can be compiled and linked directly using $pkg-config\ --libs\ --cflags\ pocl$, simplifying integration into experimental toolchains.

PoCL was built on the latest development branch using clang-19 and with *HOST_CPU_TARGET_ABI* set to lp64d to ensure the correct target ABI. We encountered ABI mismatch issues during compilation when targeting generic-rv64, specifically linker errors indicating that double-float modules could not be linked with soft-float modules. To resolve this, we selected a more specific CPU target, SiFive X280.

To inspect the intermediate code generated during kernel compilation, we used *POCL_DEBUG=all* and *POCL_LEAVE_KERNEL_COMPILER_TEMP_FILES=1*. This preserves the intermediate LLVM bitcode. This bitcode can then be compiled to RISC-V assembly using llc, giving us a view of the generated instructions. The path of the bitcode is dumped during runtime which can be seen in listing 3.1.

Listing 3.1: Pocl IR location

```
| LLVM | Cached compiled SPIRV binary not found, generating
   SPIR IR to /tmp/my-pocl-cache/AM/.../app.bc
```

The full CMake invocation along with the list of required Debian packages for building this configuration is provided in Appendix A.3 to ensure reproducibility.

3.5 Executing OpenCL

To compile AdaptiveCpp with support for the OpenCL backend we used their latest release 24.10.0 with Clang 19 as host compiler and explicitly enabled *WITH_OPENCL_BACKEND*. Note that clang-19 breaks OpenMP support which is fine since we are looking to run OpenCL as the backend. The OpenCL headers and ICD loader were provided by the oneAPI Construction Kit. Note that the system must be set up to properly expose the OpenCL device through the ICD loader. By placing a .icd file pointing to the vendor library in */etc/OpenCL/vendors/*, after which availability can be verified using clinfo. AdaptiveCpp was invoked with *acpp -O3 app.cpp -o out* to build the SYCL application. The full list of required apt packages as well as the complete cmake invocation used to build AdaptiveCpp in this configuration is provided in Appendix A.4.

To run binaries in RAVE without triggering JIT execution, the binaries must be run twice. The first run intentionally causes JIT compilation to fail, which creates a JIT-folder, see listing **lst:OpenCLJITFail**. This folder is then used to copy the Qemu systems JIT binaries for the second. This can be seen in listing 3.3.

Listing 3.2: Failed JIT compilation on host system

```
[AdaptiveCpp Info] LLVMToSpirv: Invoking ... llvm-spirv ...
[AdaptiveCpp Error] from ... jit.hpp:320 @ compile(): jit::
   compile:
Encountered errors: 0: LLVMToSpirv: llvm-spirv invocation
   failed with exit code -1
```

Listing 3.3: Copy over the JIT binaries

```
jit-cache/apps $ ls
app-12006407574750455966  app-8183243966643012215
rm -rf app-8183243966643012215/*
cp -r app-12006407574750455966/* app-8183243966643012215
```

3.6 RVV support

In order to emulate RVV, both RAVE and the Qemu system emulation needs to support it. One of the modification applied by the RAVE plugin, is a modification for the Qemu user mode emulation which enables wider vector width and RVV. The same modification can be applied to the source code of Qemu, which can then be

compiled from source in order to get a Qemu system emulation which supports RVV. The modification and configuration command for Qemu can be seen in listing 3.4 and 3.5.

Listing 3.4: Qemu modification for RVV support

```
diff --git a/target/riscv/cpu.h b/target/riscv/cpu.h
index 7de19b4183..54f8eecf78 100644
--- a/target/riscv/cpu.h
+++ b/target/riscv/cpu.h
@@ -169,7 +169,7 @@ #include "debug.h"

-#define RV_VLEN_MAX 1024
+#define RV_VLEN_MAX (256*64)
 #define RV_MAX_MHPMEVENTS 32
 #define RV_MAX_MHPMCOUNTERS 32
```

Listing 3.5: Qemu build command used

```
./configure --enable-system \
        --target-list=riscv64-linux-user,riscv64-softmmu \
        --enable-linux-user \
        --enable-slirp
```

Furthermore, a Linux kernel which supports RVV is required. In this project, a Linux kernel of version 6.12.25 was used, which at the time of this project was the default in Debians trixie release branch [45]. In order to verify the vector width, the RVV support can be tested with the C-code provided in Appendix A.1, which queries for the vector width used by the system.

3.7 Benchmarks

The purpose of the benchmarks were to have a set of SYCL kernels of different kinds in order to identify what SYCL constructs were more prone to be vectorized and if there was any difference in how the different compiler backends would optimize the different kernels.

Some criteria for the benchmarks were that they should use either SYCL unified shared memory or SYCL buffer accessor model, as this is quite a big topic in SYCL. Furthermore, the kernels had to be parallelizable to be vectorizable, and it would be beneficial to have some different granularities of parallelism to test.

Two different benchmarks were found, one called Sycl-bench [33], an open-source benchmark suite, and another open-source suite called microSYCL [48], created at Chalmers by H. Abram. The sycl-bench benchmark suite consists of multiple directories with different types of benchmarks of varying complexity. The benchmarks that were chosen were the polybench benchmarks, since these provided a lot of different suites that all consisted of SYCL's parallel for. These should in theory be vectorizable and should illustrate if SYCL compilers vectorize or not.

Additionally, the benchmarks were also slightly modified in order to start and stop the tracing by RAVE, so that the logs would capture only the SYCL kernel being run. This modification can be seen in listing 3.6, which is an excerpt from the gemv benchmark.

Listing 3.6: RAVE stop start traces

```
rave_restart_trace(); // Reset the tracing
auto event = args.device_queue.submit([&](handler& cgh) {
  auto A = A_buf.get_access<access::mode::read>(cgh);
  auto B = B_buf.get_access<access::mode::read>(cgh);
  auto x = x_buf.get_access<access::mode::read>(cgh);
  auto y = y_buf.get_access<access::mode::read_write>(cgh);
  auto t = tmp_buf.get_access<access::mode::read_write>(cgh);
  cgh.parallel_for<Gesummv>(y.get_range(),
                            [=, N_=size]
                            (item<1> item) {
    const auto i = item[0];
    for (size_t j = 0; j < N_; j++) {
      t[item] += A[{i, j}] * x[j];
      y[item] += B[{i, j}] * x[j];
    }
    y[item] = ALPHA * t[item] + BETA * y[item];
  });
});
event.wait();
rave_stop_trace(); // Stop tracing
```

A common characteristic of the SYCL-Bench benchmarks in the PolyBench folder is that they each implement an algorithm using *sycl::parallel_for*, consistently relying on regular ranges rather than *sycl::nd_range*. Additionally, these benchmarks employ the buffer-accessor memory model, as opposed to Unified Shared Memory (USM).

This was in contrast to microSYCL, which used both *sycl::nd_range* and also regular range. Furthermore, microSYCL also had benchmarks with buffer accessor models and also USM. Thus, microSYCL extended the variety of SYCL constructs that could be tested. However, the drawback with microSYCL was that it did not implement as many different algorithms as Sycl-bench.

Both benchmark suites shared the capability to measure kernel execution times and provided interfaces for adjusting the problem size. The build files for both benchmarks also had to be modified to specify the RISC-V architecture and what vector width was being used, this was done by adding the *-march=rv64gcv_zvl4096b* flag, where the *v* in *gcv* stands for vector extension, and the number after *zvl* corresponds to the vector width in bits.

4

Results

This section contains a collection of software benchmarks structured to test the behavior of SYCL kernels under different backend configurations. It begins with a minimal for-loop kernel meant to directly tests vectorization patterns in isolation. This initial case exists as a baseline for RVV code generation observation. Following this, the tests split into combinations where access patterns and execution configurations are systematically tested. These include combinations of Unified Shared Memory and traditional buffer accessors paired with both nd_range and range dispatch modes. Each configuration is compiled under OpenMP JIT, OpenMP AOT, PoCL and oneAPI Construction Kit, then run on a QEMU-emulated RISC-V system. Instruction traces are captured with RAVE and manually inspected for RVV usage and scheduling behavior. The complete set of assembly instructions from all kernels are included in Appendix B. Test cases which failed, crashed at runtime or produced incorrect outputs are documented in the final part of this section together with relevant debug logs and trace output.

4.1 Simple for-loop

To start, a simple for-loop program was created in order to check that the different compiler backends would vectorize against RISC-V. The benchmarks consisted of three arrays allocated using USM on the host device. The kernel used *sycl::range* to iterate the arrays, and would take two of the arrays, add them together and then store them in the third array. The kernel used floating point numbers and the problem size was 4096. The source code for the kernel can be seen in listing 4.1.

Listing 4.1: Simple for-loop

```
1 // sres, sarr1. sarr2 USM allocated
2 queue.submit([&](sycl::handler& cgh) {
3     cgh.parallel_for(sycl::range<1>{PROBLEM_SIZE},
4                      [=](sycl::id<1> idx) {
5         sres[idx[0]] = sarr1[idx[0]] + sarr2[idx[0]];
6     });
7 });
```

The kernel was compiled using three backends of the AdaptiveCpp compiler, these were, OpenMP AOT, OpenMP JIT and OpenCL with the oneAPI construction kit

implementation. The vectorization results can be seen in table 4.1.

Table 4.1: Sime for-loop vectorization results using different AdaptiveCpp backends

Backend	vfadd	vector width (bytes)	vectorized operations (%)
OpenMP AOT	16	1024	100
OpenMP JIT	16	1024	100
OpenCL oneAPI*	-	-	-
OpenCL PoCL*	-	-	-

* Keeps recompiling JIT in RAVE

4.2 Unified shared memory with sycl::range

The following tests evaluates the relationship between USM and sycl range. The tests utilize the GEMM kernel from the microSYCL benchmark suite. The kernel was run using *./main-acpp-<target> –gemm -i 1*. The matrix size that was used was 32x32 and using double-precision floats. The source code for the kernel can be seen in listling 4.2.

Listing 4.2: Gemm kernel using USM and sycl::range

```
// m1, v1, v2 allocated on host with USM
Q.submit([&](sycl::handler &cgh) {
  cgh.parallel_for<>(sycl::range<1>(global), [=](sycl::item
     <1> it) {
    const int i = it.get_id(0);
    const int N = it.get_range(0);
    TYPE temp = 0.0;
    for (size_t k = 0; k < N; k++) {
      temp += m1[i * N + k] * v1[k];
    }
    v2[i] = temp;
  });
});
Q.wait();
```

The resulting vectorization metrics from running the kernel with three backends of AdaptiveCpp can be seen in table 4.2, where *vfmacc* is a multiply and accumulate vector instruction and *vfredosum* does a vectorized sum reduction.

Table 4.2: Vectorization metrics of GEMM kernel with 32x32 matrices using doubles implemented using USM and range

Backend	vfmacc	vector width	vfredusum	vector width	vectorized flops (%)
OpenMP AOT	0	0	0	0	0
OpenMP JIT*	-	-	-	-	-
OpenCL oneAPI*	-	-	-	-	-
OpenCL PoCL*	-	-	-	-	-

* Keeps recompiling JIT in RAVE

4.3 Buffer accessors memory with sycl::range

To evaluate the interaction between buffer accessors and range, a GEMM kernel from the microSYCL benchmark was used. The test uses three 32x32 matrices using double-precsicion floats created using buffer allocations. The kernel was run using *./main-acpp-<target> –gemm -i 2*. The source code for the kernel can be seen in listing 4.3.

Listing 4.3: GEMM kernel using buffer accessors with sycl::range

```
/// v1_buff, v2_buff and m1_buff allocated with buffers
Q.submit([&](sycl::handler &cgh) {
  auto v1_acc = v1_buff.get_access<sycl::access::mode::read>(
     cgh);
  auto v2_acc = v2_buff.get_access<sycl::access::mode::
     read_write>(cgh);
  auto m1_acc = m1_buff.get_access<sycl::access::mode::read>(
     cgh);

  cgh.parallel_for<>(sycl::range<1>(global), [=](sycl::item
     <1> it) {
    const int i = it.get_id(0);

    const int N = it.get_range(0);
    TYPE temp = 0.0;
    for (size_t k = 0; k < N; k++) {
      temp += m1_acc[i * N + k] * v1_acc[k];
    }
    v2_acc[i] = temp;
  });
});
```

The resulting vectorization metrics from running the kernel with three backends of AdaptiveCpp can be seen in table 4.3.

Table 4.3: Vectorization metrics of GEMM kernel with 32x32 matrices using doubles implemented using the buffer accessor memory model and range

Backend	vfmacc	vector width (bytes)	vfredusum	vector width (bytes)	vectorized flops (%)
OpenMP AOT	256	256	256	256	100
OpenMP JIT*	-	-	-	-	-
OpenCL oneAPI	0	0	0	0	0
OpenCL PoCL**	-	-	-	-	-

* No vectorization below 128x128 size matrices
** Keeps recompiling JIT in RAVE

4.4 Unified shared memory with sycl::nd_range

One of the tests conducted was how the backends would vectorize a kernel using unified shared memory in combination with *sycl::nd_range*. The kernel that was chosen was the GEMM kernel from the microSYCL benchmark suite. This kernel could be run with *./main-acpp-<target> –gemm -i 3*. The matrix size that was used was 32x32 and the number type was double, furthermore, the blocksize could be changed, which affected the assembly, but did not affect the final vectorization instructions being run. The source code for the kernel can be seen in listing 4.4.

Listing 4.4: GEMM kernel using USM with sycl::nd_range

```
// m1, m2, m3 allocated on host with USM
Q.submit([&](sycl::handler& cgh){
    cgh.parallel_for< >(sycl::range<2>(global1),
                        [=](sycl::item<2>it) {
        auto i = it.get_id(0);
        auto j = it.get_id(1);
        TYPE temp = 0.0;
        for (size_t k = 0; k < N; k++) {
            temp += m2[i*N+k]*m1[k*N+j];
        }
        m3[i*N+j] = temp;
    });
});
```

The resulting vectorization metrics from running the kernel with three backends of AdaptiveCpp can be seen in table 4.4, where *vfmacc* is a multiply and accumulate vector instruction and *vfredosum* does a vectorized sum reduction.

Table 4.4: Vectorization metrics of GEMM kernel with 32x32 matrices using doubles implemented using USM and nd_range

Backend	vfmacc	vector width (bytes)	vfredusum	vector width (bytes)	vectorized flops (%)
OpenMP AOT	256	256	256	256	100
OpenMP JIT	256	256	256	256	100
OpenCL oneAPI*	-	-	-	-	-
OpenCL PoCL*	-	-	-	-	-

* Keeps recompiling JIT in RAVE

4.5 Buffer accessors memory model with sycl::nd_range

For the combination of buffer accessors with nd_range, a GEMM kernel from the microSYCL benchmarks was chosen. This benchmark allocated three matrices of size 32x32, containing double precision floating point numbers using buffers. The benchmark then uses accessors to access the buffers inside the kernel. Furthermore, the block size of the nd_range could be changed, however, there was no difference in the vectorization traces when changing it. The kernel can be seen in listing 4.5.

Listing 4.5: GEMM kernel using buffer accessors with sycl::nd_range

```
/// m1b, m2b and m3b allocated with buffers
Q.submit([&](sycl::handler& c){
    auto m1=m1b.get_access<sycl::access::mode::read>(c);
    auto m2=m2b.get_access<sycl::access::mode::read>(c);
    auto m3=m3b.get_access<sycl::access::mode::read_write>(c);
    cgh.parallel_for(sycl::nd_range<2>(global1,local1),
                     [=](sycl::nd_item<2>it){
          auto i = it.get_global_id(0);
          auto j = it.get_global_id(1);
          TYPE temp = 0.0;
          for (size_t k = 0; k < N; k++)
          {
              temp += m2_acc[i*N+k]*m1_acc[k*N+j];
          }
          m3_acc[i*N+j] = temp;
    });
});
```

The resulting vectorization metrics from running the kernel with three backends of AdaptiveCpp can be seen in table 4.5.

Table 4.5: Vectorization metrics of GEMM kernel with 32x32 matrices using doubles implemented using the buffer accessor memory model and nd_range

Backend	vfmacc	vector width (bytes)	vfredusum	vector width (bytes)	vectorized flops (%)
OpenMP AOT	1024	256	1024	256	100
OpenMP JIT*	-	-	-	-	-
OpenCL oneAPI	0	0	0	0	0
OpenCL PoCL*	-	-	-	-	-

*Keeps recompiling JIT in RAVE

4.6 Benchmark issues

This section will briefly show errors encountered upon during testing of SYCL backends.

OneAPI Construction Kit consistently failed at runtime when using USM in both QEMU and RAVE. Every attempt to run even the simplest kernel resulted in identical USM host allocation errors. The runtime output showed repeated messages which can be seen in listing 4.6.

Listing 4.6: USM host allocation failed using oneAPI construction kit

```
[AdaptiveCpp Error] from AdaptiveCpp-24.10.0/.../
   ocl_allocator.cpp:54 @raw_allocate_optimized_host():
   ocl_allocator: USM host allocation failed (error code = CL
   :-59)
```

Running PoCL inside RAVE also proved unsuccessful. All runs failed during program run time with consistent which can be seen in listing 4.7.

Listing 4.7: PoCL failed to compile inside RAVE

```
[AdaptiveCpp Error] from /AdaptiveCpp/.../ocl_code_object.cpp
   :76 @ ocl_executable_object(): ocl_code_object: Building
   CL program failed. Build log: Device cpu failed to build
   the program (error code = CL:-11)
```

When using OpenMP with JIT compilation, AdaptiveCpp would choose to rerun the JIT compilation every time in RAVE, even if the emulation was setup the same in RAVE and the Qemu system emulation with Debian. This caused a problem since the simulation method relied on the JIT optimized binary being cached before running the executable in RAVE.

5

Discussion

This chapter aims to provide a deeper analysis of the results, with the goal of understanding the behavior of the different backends, as well as identifying their respective differences and advantages. To achieve this, both the generated assembly code and the Paraver traces will be examined to assess the runtime behavior of the kernel.

5.1 OpenMP AOT

The results from running OpenMP AOT on the four different combinations of USM, buffer accessors and nd_range/range was that it was able to vectorize all the combinations.

Firstly, a comparison of the assemblies between using USM and buffer accessors memory model was done. There was no difference in the vectorization part of the assembly code, and the assemblies looked identical when comparing the two *sycl::range* based benchmarks, and the two *sycl::nd_range* benchmarks. However, the loop depth was one less with *sycl::range*, showing that one loop had been optimized away. There was also still many calls to other shared libraries and functions within the assembly left which had been inlined in the JIT-compiled binaries.

5.2 OpenMP JIT

There were multiple benchmarks which were not possible to run in RAVE when running OpenMP with JIT-compilation. It was therefor only possibly to retrieve the Paraver traces for the simple for-loop using USM and sycl::range and the GEMM kernel using USM and sycl::nd_range. The issue with the emulation method was that AdaptiveCpp would recompile the JIT binary, even if it had already been optimized and cached, for which the reason is unknown.

This meant that the main analysis has to be done by analyzing the assembly code and optimization reports emitted by clang. In order to emit these reports and assembly instead of the regular shared library from the JIT compilation stage, the modifications described in section 3.2 were applied.

Firsly, the simple for-loop is analyzed in order to understand the steps that are taken in order to vectorize the loop. In the compiler report C.1 it is stated that a vectorization optimization was applied to the IR during compilation. This can be seen in the assembly code B.2, were two versions of the loop can be found, one which is vectorized and one which is scalar. The assembly then contains a check to see if the CPU has wide enough vectors, and otherwise falls back to the scalar version of the loop.

This same analysis method was conducted for all of the four benchmarks, which did reveal some interesting findings. First of, all of the benchmarks B.3, B.4 contained vector assembly instructions for vectorizing the operations executed within the innermost for-loop, furthermore, the assembly code when using buffers compared to USM was identical, showing that the memory model did not affect the final runtime. This is further confirmed when looking at the compiler optimization reports, as they are completely identical to each other. Furthermore, the final IR output by AdaptiveCpp is completely identical to each other as well between the two different memory models.

The main difference observed between the benchmarks was between *sycl::range* and *sycl::nd_range*. Firstly, completely different optimizers were applied to the IR by LLVM, the benchmarks using *sycl::range* had one of the loops unrolled and the innermost loop vectorized. Meanwhile, the benchmarks using *sycl::nd_range* had the innermost loop vectorized as well, however, using the SLP vectorizer to combine stores into vector instructions instead. This can also be seen in the assembly were the *sycl::range* B.3 benchmarks make use of a strided vector store instruction inside of a loop to store back all the computed values, while the *sycl::nd_range* B.4 benchmarks make use of one vector instruction to store all values. The reason for this optimization is most likely due to the final IR output by AdaptiveCpp where there is a loop in the *sycl::range* benchmark and a completely unrolled loop in the *sycl::nd_range* benchmarks. The reason for a full loop being deleted with *sycl::range* is hard to analyze in the IR, but it could be due to the loops upper bound being frozen at the entry of the function, which was not the case for the *sycl::nd_range* benchmarks.

5.3 OneAPI Construction Kit

OneAPI's results were inconsistent across the tests, failing in three out of the five configurations. Even in the two configurations that passed, no vectorization was performed. Tests involving USM consistently failed in three of the five configurations, producing runtime memory allocation exceptions 4.6. This suggests that LLVM 19 and OneAPIs USM are not fully supported in a QEMU-based emulation environment.

Switching to buffer-based accessors allowed two tests (4.3 using *sycl::nd_range* and 4.5 using range) to complete execution. Note that increasing the problem size above the vector length *RAVE_VLEN=128 ... .main-acpp-generic -s 200 –gemm -i 2* would cause the program to crash. Despite both tests producing valid vectorized code with instructions such as *vl2re64.v*, *vlse64.v*, *vfmul.vv*, and *vfredosum.vs*, the

runtime profiling via RAVE showed that no vector instructions were executed. This discrepancy between JIT code generation and runtime behavior might be due to conditions during execution, such as not being able to read vector width or loop evaluation not preferring vectorization. Notably, even with the *CA_RISCV_VF=V CA_RISCV_VLEN_BITS_MIN=2048* environment variable set, vectorization did not occur. This suggests that the issue is not simply related to the reported VLEN. In both test 4.3 and 4.5, the presence of structured vector loop followed by scalar fallback paths is evident which can be seen in listing 5.1.

Listing 5.1: OneAPI's loop output showing both vector and scalar instructions

```
... // Loop selection
fmv.d.x fa5, zero
vsetvli a0, zero, e64, m1, ta, ma
vmv.s.x v8, zero
...
bgeu    a6, a0, Vector loop
li      s10, 0
fmv.d   fa4, fa5
j       Scalar loop

... // Vector Loop
vsetvli s7, zero, e64, m2, ta, ma
vl2re64.v v12, (a5)
vlse64.v v14, (a2), a3
vfmul.vv v12, v12, v14
vfredosum.vs v10, v12, v10

... // Scalar fallback
fld fa3, 0(s0)
fld fa2, 0(a0)
fmul.d fa3, fa3, fa2
fadd.d fa4, fa4, fa3
```

The assembly suggests that the compiler generated vector loops but failed to trigger them at runtime. This could point to the *bgeu a6, a0* instruction which contains one of the function arguments passed by AdaptiveCpp. This presumably calculates if vectorization is worthwhile.

The vector configuration in both 4.3 and 4.5 matches what would be expected from loop auto-vectorization. The use of *vsetvli* with e64, m2 shows the intent to use 64-bit wide doubles across two registers per element group, which should have been ideal for the matrix-vector dot product pattern tested. The fact that the compiler is generating this code indicates it recognizes the potential for vectorization in the loop. However, the runtime behavior suggests that the vector length calculation is not being correctly used, possibly due to issues in how the loop trip count or memory access patterns are handled at execution time within the Qemu environment.

5.4 PoCL

None of the five PoCL tests completed when run under RAVE. All executions crashed early during runtime setup. The issue stems from PoCL attempting to recompile the JIT kernel during execution, a process that does not function correctly within the RAVE runtime. This made it impossible to analyze dynamic instruction traces or instruction-level RVV behavior using RAVE for PoCL.

All runs executed perfectly inside the Qemu emulator which meant we could retrieve assembly corresponding to each configuration. Across all tests, including range, *nd_range*, and both USM and buffer accessor memory models, no RVV vector instructions were generated. This somewhat aligned with expectations since PoCL does not yet officially support RVV code generation. As such, the LLVM backend defaults to scalar code generation paths. All inner loops were simplified to scalar loads, floating point multiplies, and adds, and relied on soft-float helper functions like *__muldf3* and *__adddf3* to perform arithmetic. These calls were all visible in the assembly. It's worth noting that PoCL was compiled with the *HOST_CPU_TARGET_ABI=lp64d* flag, and the SiFive X280 CPU has an FPU, but PoCL still used soft floats. Given that the SiFive X280 has an FPU, it should be using the *mul.d* instruction for double-precision floating-point multiplication instead of the *__muldf3* software function.

Despite the lack of RVV usage, PoCL's code generation was stable and consistent. The structure of the generated loops across all configurations showed no divergence. All performed unrolled accumulation using standard scalar loop constructs. Notably, each kernel was compiled to a form using explicit memory addressing with computed offsets and the common double-precision floating point multiply-accumulate sequence mediated through software floats. This indicates that PoCL's OpenCL backend is functioning in a purely scalar mode with no active vectorization even in the presence of vectorizable loops.

5.5 Simulation Strategy

The emulation and testing strategy relied heavily on QEMU system emulation and the custom tool RAVE. QEMU for RISC-V proved to be quite unstable, especially when used with RISC-V Ubuntu, which frequently resulted in boot failures and dropped us into initramfs recovery. This called for a switch to Debian trixie, which proved more stable. A containerized RISC-V base image might have offered more stability and isolated system configurations. However, there were several challenges that restricted its use. RAVE cannot provide traces when running within a RISC-V system. Additionally, setting hardware specifications like RV64GCV on a container is not possible. While compiling binaries within a container for execution on the host would be possible, JIT compilation would still necessitate an emulator, introducing further complications due to the need to migrate system packages required by the JIT runtime. These issues ultimately ruled out the usage of containers.

Compile times were identified as a significant bottleneck. Because builds were exe-

cuted on an emulated, non-native processor, even minimal builds required substantial time often extending into several hours. As a result, full toolchain builds, such as compiling the entirety of LLVM, were deemed unfeasible. Instead, the builds of SYCL compilers and backends were based on prebuilt packages provided by the aptitude package manager. This limitation could not be addressed by containerization, as the cost of emulation remained unchanged. Additionally, the extended compile times restricted the range of compiler versions that could be practically tested. Compatibility issues across versions were encountered frequently, with only a small subset of configurations producing usable binaries. Numerous LLVM versions failed at various stages, often without clear patterns. It is suspected that alternative versions of LLVM may have offered better support for features like USM, though this could not be verified due to the compilation difficulties.

RAVE added a completely different layer of complexity. While invaluable for instruction-level tracing, it imposed harsh limitations on how binaries could be executed. Any test that involved JIT compilation would cause a runtime error if executed inside RAVE due to its incompatibility with dynamic code generation on the host, and the inability to change compiler paths in already compiled JIT binaries by AdaptiveCpp. This forced us to precompile and cache JIT binaries inside the emulator, which still would not solve the problem always, as AdaptiveCpp would sometimes decide to recompile the JIT binaries no matter what in RAVE. This meant that only a subset of the benchmarks that were intended to be run in rave could be run, which meant we had to rely on analyzing assembly, compiler optimization reports and intermediate representation and IR instead.

5.6 Benchmarks

The inital idea was to try a multitude of different benchmarks from both microSYCL and sycl-bench, however, after retrieving results from the benchmarks in microSYCL which tried different combinations of ranges and memory models, we realized that it would be more interesting to analyze a few benchmarks more in depth, especially since RAVE would only work for some benchmarks. This meant that the focus was shifted to analyzing compiler reports, assembly and intermediate representation, instead of just looking at instructions counts from RAVE. This meant that there could potentially be ways of structuring kernels that would be more or less beneficial for vectorization, that may have been missed due to the small selection of benchmarks.

While the limited selection of benchmarks may seem like a shortcoming in the results and analysis, this was deemed more valuable as it would provide an analysis with a greater depth. Furthermore, this focus allowed us to be less reliant on the results from RAVE, and to also be able statically analyze different compiler outputs manually.

The benchmarks were chosen due to their simplicity and since they covered all combinations of memory models and ranges. Furthermore, when compiler outputs were analyzed, the benchmarks were stripped down to only contain the kernel, in

order to make compiler reports and assembly code smaller and easier to analyze.

6

Conclusion & Future Work

This chapter will discuss future work. The current implementation and evaluation demonstrates the viability of using SYCL to target RISC-V RVV. However, there are several limitations and unexplored areas remaining. The following section outline possible directions to expand and refine the work.

6.1 Future Work

Future work includes extending our test coverage toward more generalized and modern SYCL constructs. Existing SYCL testing frameworks typically rely on outdated buffer-based models which are not aligned with the latest USM-centric workflows. This reveals a broader issue that there is no established standard for SYCL conformance testing, especially not one that supports vector hardware targets like RVV. A test suite tailored for RVV-capable backends using USM and experimental features would help push both compiler and hardware vendors toward more robust support. Alongside this, test granularity should move beyond correctness alone and encompass fine-grained performance behaviors across varying vector widths and memory access patterns.

We are currently limited to emulation-based evaluation due to a lack of available RVV hardware that supports full-length 64k-bit vectors. Although this is technically allowed in the specification, no silicon is capable of such. As a result, our work depends heavily on QEMU system-mode emulation and the RAVE tool for instruction-level inspection. The upcoming Avispado supercomputer aims to implement 16k-bit RVV vectors which will allow meaningful hardware validation in the near future. Until then, performance extrapolation based on software models remains speculative. Our use of Banana Pi hardware, while useful for testing software correctness on actual silicon, only supports 256-bit RVV. This width is comparable to AVX256 on x86 and may offer similar performance scaling behavior. However, we note that many optimizations targeting wide vector lengths will not be observable on this platform.

Since OpenCL binaries could not be executed within RAVE, we were restricted to analyzing raw machine code without runtime behavioral data. This restricts deeper insight into dynamic scheduling, memory aliasing effects or instruction selection strategies that the JIT backends reveal more clearly. A custom OpenCL runtime that integrates with RAVE or similar instruction-logging tools would provide a better window into runtime behavior.

RVV is still in an early stage across the tool chain landscape, and it will take time before stable and efficient implementations appear across platforms. All SYCL stacks does not generate ideal code, often due to gaps in how vector length agnostic constructs are handled or missing support in the backend. The wide range of RISC-V variants adds another layer of complexity, making broad and effective targeting difficult. Work is ongoing to improve consistency in the RISC-V specification to better tune backends for RVV patterns and make low-level hardware features usable through clean SYCL interfaces. These efforts will be key in making the platform practical for real-world use.

6.2 Conclusion

In conclusion, the support for RISC-V is still in early stages, and multiple compilers and backends are still in development. Furthermore, tooling support such as emulators for tracing instructions such as RAVE are still under development and behaves inconsistently. Our emulation setup was also not ideal since it did not allow for running all benchmarks in RAVE, however, it was still complemented well with manual analysis of assembly code, IR, and compiler reports. We can conclude that AdaptiveCpp is able to translate SYCL kernels into intermediate representation which LLVM and some OpenCL implementations are able to vectorize. This means that the vectorization capabilities at this stage of AdaptiveCpp are most reliant on the vectorization support of OpenCL implementations and LLVM. At this stage, PoCL themselves recognize that they have not yet implemented vectorization support for RISC-V, and oneAPI construction kit does not claim full support for RISC-V either. Furthermore, in order to utilize LLVM's vectorization capabilities on RISC-V, the AdaptiveCpp compiler has to be modified due to LLVM not supporting the *-native* flag on RISC-V yet.

Bibliography

[1] *SYCLOPS - Democratizing AI Acceleration Using Open Standards*. [Online]. Available: https://www.syclops.org/updates/2023/05/10/launch-of-the-new-horizon-europe-project-syclops.

[2] M. Traviss, *DARE project puts Europe on the map for chip development*, en-GB, Nov. 2024. [Online]. Available: https://www.innovationnewsnetwork.com/dare-project-puts-europe-on-the-map-for-chip-development/53502/.

[3] *SYCL - C++ Single-source Heterogeneous Programming for Acceleration Offload — khronos.org*, https://www.khronos.org/sycl/, [Accessed 27-01-2025].

[4] *GitHub - intel/llvm: Intel staging area for llvm.org contribution. Home for Intel LLVM-based projects. — github.com*, https://github.com/intel/llvm, [Accessed 27-01-2025].

[5] *GitHub - AdaptiveCpp/AdaptiveCpp: Implementation of SYCL and C++ standard parallelism for CPUs and GPUs from all vendors: The independent, community-driven compiler for C++-based heterogeneous programming models. Lets applications adapt themselves to all the hardware in the system - even at runtime! — github.com*, https://github.com/AdaptiveCpp/AdaptiveCpp, [Accessed 27-01-2025].

[6] Dec. 2023. [Online]. Available: https://www.european-processor-initiative.eu/accelerator/.

[7] R. Schaller, "Moore's law: Past, present and future," *IEEE Spectrum*, vol. 34, no. 6, pp. 52–59, 1997. DOI: 10.1109/6.591665.

[8] IEEE International Roadmap for Devices and Systems, *International Roadmap for Devices and Systems 2023 Edition*, Accessed: 2025-01-23, 2023. [Online]. Available: https://irds.ieee.org/editions/2023.

[9] *Home - | TOP500 — top500.org*, https://top500.org/, [Accessed 27-01-2025].

[10] AMD, *4TH GEN AMD EPYC PROCESSOR ARCHITECTURE*, https://www.amd.com/en/products/processors/server/epyc/4th-generation-architecture.html, [Accessed 27-01-2025].

[11] *Intelő Xeonő CPU Max 9470 Processor (105M Cache, 2.00 GHz) - Product Specifications | Intel — intel.com*, https://www.intel.com/content/www/us/en/products/sku/232594/intel-xeon-cpu-max-9470-processor-105m-cache-2-00-ghz/specifications.html, [Accessed 27-01-2025].

[12] Fujitsu, *Fujitsu.com*, https://www.fujitsu.com/downloads/JP/jsuper/a64fx/a64fx_datasheet.pdf, [Accessed 27-01-2025], 2020.

[13] D. A. Patterson and D. R. Ditzel, "The case for the reduced instruction set computer," *SIGARCH Comput. Archit. News*, vol. 8, no. 6, pp. 25–33, Oct. 1980, ISSN: 0163-5964. DOI: 10.1145/641914.641917. [Online]. Available: https://doi.org/10.1145/641914.641917.

[14] C. H. Séquin and D. A. Patterson, "Design and implementation of risc i," Tech. Rep. UCB/CSD-82-106, Oct. 1982. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/1982/5449.html.

[15] R. Sherburne, M. Katevenis, D. Patterson, and C. Sequin, "A 32-bit nmos microprocessor with a large register file," Tech. Rep. 5, 1984, pp. 682–689. DOI: 10.1109/JSSC.1984.1052208.

[16] A. D. Samples, M. Klein, and P. Foley, "Soar architecture," Tech. Rep. UCB/CSD-85-226, 1985. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/1985/5940.html.

[17] M. D. Hill, S. J. Eggers, J. R. Larus, *et al.*, "Spur: A vlsi multiprocessor workstation," Tech. Rep. UCB/CSD-86-273, Dec. 1986. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/1986/6083.html.

[18] *About risc-v international — riscv.org*, Accessed 27-01-2025. [Online]. Available: https://riscv.org/about/#history.

[19] D. A. P. Andrew Waterman Yunsup Lee, "The risc-v instruction set manual, volume i: Base user-level isa," Tech. Rep., May 2011. [Online]. Available: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-62.pdf.

[20] K. Asanovi and D. A. Patterson, "Instruction sets should be free: The case for risc-v," Tech. Rep. UCB/EECS-2014-146, Aug. 2014. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html.

[21] Riscvarchive, *Risc-v "v" vector extension version 1.0*, Sep. 20, 2021. [Online]. Available: https://github.com/riscvarchive/riscv-v-spec/releases/tag/v1.0.

[22] N. Stephens, S. Biles, M. Boettcher, *et al.*, "The arm scalable vector extension," *IEEE Micro*, vol. 37, no. 2, pp. 26–39, Mar. 2017. DOI: 10.1109/mm.2017.35.

[23] Aug. 2019. [Online]. Available: https://riscv.org/ecosystem-news/2019/08/how-the-european-processor-initiative-is-leveraging-risc-v-for-the-future-of-supercomputing/.

[24] Nvidia, *CUDA Toolkit - Free Tools and Training — developer.nvidia.com*, https://developer.nvidia.com/cuda-toolkit, [Accessed 27-01-2025].

[25] *Auto-Vectorization in LLVM — LLVM 20.0.0git documentation — llvm.org*, https://llvm.org/docs/Vectorizers.html, [Accessed 27-01-2025].

[26] *GitHub - triSYCL/triSYCL: Generic system-wide modern C++ for heterogeneous platforms with SYCL from Khronos Group — github.com*, https://github.com/triSYCL/triSYCL, [Accessed 27-01-2025].

[27] [Accessed 29-04-2025]. [Online]. Available: https://github.com/Tohoku-University-Takizawa-Lab/neoSYCL.

[28] [Accessed 29-04-2025]. [Online]. Available: https://github.com/celerity/SimSYCL.

[29] L. Crisci, L. Carpentieri, P. Thoman, A. Alpay, V. Heuveline, and B. Cosenza, "Sycl-bench 2020: Benchmarking sycl 2020 on amd, intel, and nvidia gpus,"

in *Proceedings of the 12th International Workshop on OpenCL and SYCL*, ser. IWOCL '24, Chicago, IL, USA: Association for Computing Machinery, 2024, ISBN: 9798400717901. DOI: 10.1145/3648115.3648120. [Online]. Available: https://doi.org/10.1145/3648115.3648120.

[30] I. Z. Reguly, *Evaluating the performance portability of sycl across cpus and gpus on bandwidth-bound applications*, 2023. arXiv: 2309.10075 [cs.PF]. [Online]. Available: https://arxiv.org/abs/2309.10075.

[31] W.-C. Lin, T. Deakin, and S. McIntosh-Smith, "On measuring the maturity of sycl implementations by tracking historical performance improvements," in *Proceedings of the 9th International Workshop on OpenCL*, ser. IWOCL '21, Munich, Germany: Association for Computing Machinery, 2021, ISBN: 9781450390330. DOI: 10.1145/3456669.3456701. [Online]. Available: https://doi.org/10.1145/3456669.3456701.

[32] A. Alpay and V. Heuveline, "Adaptivecpp stdpar: C++ standard parallelism integrated into a sycl compiler," in *Proceedings of the 12th International Workshop on OpenCL and SYCL*, ser. IWOCL '24, Chicago, IL, USA: Association for Computing Machinery, 2024, ISBN: 9798400717901. DOI: 10.1145/3648115.3648117. [Online]. Available: https://doi.org/10.1145/3648115.3648117.

[33] L. Crisci, L. Carpentieri, P. Thoman, A. Alpay, V. Heuveline, and B. Cosenza, "Sycl-bench 2020: Benchmarking sycl 2020 on amd, intel, and nvidia gpus," in *Proceedings of the 12th International Workshop on OpenCL and SYCL*, ser. IWOCL '24, Chicago, IL, USA: Association for Computing Machinery, 2024, ISBN: 9798400717901. DOI: 10.1145/3648115.3648120. [Online]. Available: https://doi.org/10.1145/3648115.3648120.

[34] Riscv, *Riscv-profiles/src/profiles.adoc at main ů riscv/riscv-profiles*, [Accessed 024-04-2025]. [Online]. Available: https://github.com/riscv/riscv-profiles/blob/main/src/profiles.adoc.

[35] [Accessed 20-05-2025]. [Online]. Available: https://d3s.mff.cuni.cz/files/teaching/nswi200/202324/doc/riscv-abi.pdf.

[36] [Accessed 20-05-2025]. [Online]. Available: https://gcc.gnu.org/onlinedocs/gccint/Soft-float-library-routines.html.

[37] [Online]. Available: https://cvw.cac.cornell.edu/vector/performance/performance-amdahl.

[38] [Accessed 24-04-2025]. [Online]. Available: https://registry.khronos.org/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf.

[39] A. Alpay and V. Heuveline, "One pass to bind them: The first single-pass sycl compiler with unified code representation across backends," in *Proceedings of the 2023 International Workshop on OpenCL*, ser. IWOCL '23, Cambridge, United Kingdom: Association for Computing Machinery, 2023, ISBN: 9798400707452. DOI: 10.1145/3585341.3585351. [Online]. Available: https://doi.org/10.1145/3585341.3585351.

[40] *LLVM Language Reference Manual &x2014; LLVM 21.0.0git documentation — llvm.org*, https://llvm.org/docs/LangRef.html, [Accessed 19-05-2025].

[41] KhronosGroup, *Khronosgroup/opencl-icd-loader: The opencl icd loader project.* [Online]. Available: https://github.com/KhronosGroup/OpenCL-ICD-Loader.

[42] [Accessed 24-04-2025]. [Online]. Available: https://www.qemu.org/docs/master/.

[43] *Pablo Vizcaino / RAVE ů GitLab — repo.hca.bsc.es*, https://repo.hca.bsc.es/gitlab/pvizcaino/rave, [Accessed 04-02-2025].

[44] Bsc-Performance-Tools, *Bsc-performance-tools/wxparaver: Wxparaver is a trace-based visualization and analysis tool designed to study quantitative detailed metrics and obtain qualitative knowledge of the performance of applications, libraries, processors and whole architectures.* [Accessed 20-05-2025]. [Online]. Available: https://github.com/bsc-performance-tools/wxparaver.

[45] Debian, [Accessed 20-05-2025]. [Online]. Available: https://wiki.debian.org/DebianTrixie.

[46] *Proposal Raised To Deprecate "-Ofast" For The LLVM/Clang Compiler — phoronix.com*, https://www.phoronix.com/news/LLVM-Might-Deprecate-Ofast, [Accessed 20-05-2025].

[47] Codeplay, [Accessed 20-05-2025]. [Online]. Available: https://developer.codeplay.com/products/oneapi/construction-kit/4.0.0/guides/modules/riscv.

[48] H. Abram, *Hariabram/microsycl: Sycl micro-benchmarks repository*, [Accessed 20-05-2025]. [Online]. Available: https://github.com/HariAbram/microSYCL.

A

Build Scripts

Listing A.1: RVV support source code

```
#include <stdio.h>
#include <riscv_vector.h>

int main() {
  // Query max VL for 8-bit elements, LMUL=1
  size_t max_vl = __riscv_vsetvlmax_e8m1();
  printf("Maximum Vector Length (VL) in elements: %zu\n",
     max_vl);

  size_t vlen = max_vl * 8;
  printf("Available Vector Width (VLEN): %zu bits\n", vlen);
  return 0;
}
$ gcc -march=rv64gcv -mabi=lp64d -o vlen vlen.c
$ ./vlen
Maximum Vector Length (VL) in elements: 512
Available Vector Width (VLEN): 4096 bits
```

Listing A.2: OneAPI Construction Kit build

```
apt install -y python3-dev libpython3-dev build-essential ocl
   -icd-libopencl1 cmake git pkg-config make ninja-build ocl-
   icd-libopencl1 ocl-icd-dev ocl-icd-opencl-dev llvm-19
   libhwloc-dev zlib1g zlib1g-dev dialog apt-utils llvm-spirv
   -19 libxml2-dev glslang-tools libzstd-dev libclang-19-dev
   clang-19 libclang-cpp19-dev libclang-cpp19 llvm-19-dev
   libpolly-19-dev

https://github.com/uxlfoundation/oneapi-construction-kit.git
   \\
Commit hash 904eb4fd97d39fa50479753e34dd1706a8614eda

cmake -BBuild \
  -DCA_LLVM_INSTALL_DIR=/usr/lib/llvm-19 \
  -DCMAKE_C_COMPILER=clang-19 \
  -DCMAKE_CXX_COMPILER=clang++-19 \
  -DCA_RISCV_ENABLED=ON \
```

```
  -DCA_MUX_TARGETS_TO_ENABLE="riscv" \
  -DCA_ENABLE_HOST_IMAGE_SUPPORT=OFF \
  -DCA_CL_ENABLE_ICD_LOADER=ON \
  -DCA_RISCV_DEMO_MODE=true \
  -DCMAKE_BUILD_TYPE=Debug
```

Listing A.3: PoCL build

```
sudo apt install -y python3-dev libpython3-dev build-
   essential cmake git pkg-config make ninja-build libhwloc-
   dev zlib1g zlib1g-dev clinfo dialog apt-utils libxml2-dev
   clang-19 llvm-19 libclang-19-dev libclang-cpp19-dev
   libclang-cpp19 llvm-19-dev llvm-spirv-19

https://github.com/pocl/pocl.git\\
Commit hash 79a68a048eef0766e7dacd798b4ac17bb073c568

cmake -BBuild \
  -DCMAKE_C_COMPILER=clang-19 \
  -DCMAKE_CXX_COMPILER=clang++-19 \
  -DCMAKE_INSTALL_PREFIX=$(pwd)/install \
  -DHOST_CPU_TARGET_ABI=lp64d \
  -DLLC_HOST_CPU=sifive-x280
```

Listing A.4: AdaptiveCpp build with OpenCL

```
apt install libomp-19 libclang-19 clang-19 libboost-fiber

https://github.com/AdaptiveCpp/AdaptiveCpp/releases/tag/v24
   .10.0

cmake -BBuild \
   -DCMAKE_C_COMPILER=clang-19 \
   -DCMAKE_CXX_COMPILER=clang++-19 \
   -DACPP_COMPILER_FEATURE_PROFILE=full \
   -DWITH_OPENCL_BACKEND=ON \
   -DOpenCL_LIBRARY=/path/to/libOpenCL.so \
   -DOpenCL_INCLUDE_DIR=/path/to/oneapi-construction-kit/
      include
```

B

Assembly

B.1 OpenMP AOT

Listing B.1: Simple for-loop

```
.Ltmp2515:
      1     .loc    7 30 41 is_stmt 1              # main.
         cpp:30:41
      2     vl2re64.v   v8, (s1)
      3     .loc    7 30 57 is_stmt 0              # main.
         cpp:30:57
      4     vl2re64.v   v10, (a3)
      5     .loc    7 30 55                        # main.
         cpp:30:55
      6     vfadd.vv    v8, v8, v10
      7     .loc    7 30 39                        # main.
         cpp:30:39
      8     vs2r.v  v8, (a2)
      9     sub a5, a5, a4
     10     add s1, s1, s0
     11     add a3, a3, s0
     12     add a2, a2, s0
     13     bnez    a5, .LBB50_9
```

B.2 OpenMP JIT

Listing B.2: Simple for loop using USM and sycl::range

```
        ld              a5, 24(a0)
        csrr        a0, vlenb
        srli        a0, a0, 2
        li              a6, 128
        slli        a5, a5, 7
        bgeu        a6, a0, .LBB0_2
        li              a4, 0
        j               .LBB0_7
.LBB0_2:    # vector.ph
        addiw       a1, a0, -1
        andi        t0, a1, 128
        xori        a4, t0, 128
        vsetvli     a1, zero, e64, m2, ta, ma
        vid.v       v8
        mv              a1, a4
.LBB0_3:    # vector.body
        vor.vx      v10, v8, a5
        vmsltu.vx       v0, v10, a3
        vsll.vi     v10, v10, 3
        vluxei64.v      v12, (t1), v10, v0.t
        vluxei64.v      v14, (a2), v10, v0.t
        vfadd.vv        v12, v12, v14
        vsoxei64.v      v12, (a7), v10, v0.t
        sub             a1, a1, a0
        vadd.vx     v8, v8, a0
        bnez        a1, .LBB0_3
        bnez        t0, .LBB0_7
.LBB0_5:    # original.exit
        ret
.LBB0_6:    # exit
        addi        a4, a4, 1
        beq             a4, a6, .LBB0_5
.LBB0_7:
        or              a0, a4, a5
        bgeu        a0, a3, .LBB0_6
        slli    a0, a0, 3
        add     a1, t1, a0
        fld     fa5, 0(a1)
        add     a1, a2, a0
        fld     fa4, 0(a1)
        fadd.d  fa5, fa5, fa4
        add     a0, a0, a7
        fsd     fa5, 0(a0)
        j       .LBB0_6
```

Listing B.3: GEMM kernel using USM and Buffer accessors (same assembly) and sycl::range

```
# %bb.0:                                # %entry
        addi    sp, sp, -48
        sd      s0, 40(sp)                      # 8-byte
           Folded Spill
        sd      s1, 32(sp)                      # 8-byte
           Folded Spill
        sd      s2, 24(sp)                      # 8-byte
           Folded Spill
        sd      s3, 16(sp)                      # 8-byte
           Folded Spill
        sd      s4, 8(sp)                       # 8-byte
           Folded Spill
        ld      s2, 24(a0)
        ld      a0, 32(a0)
        ld      a2, 0(a1)
        ld      a3, 24(a1)
        ld      a4, 32(a1)
        ld      a5, 40(a1)
        ld      s3, 0(a2)
        ld      t0, 0(a3)
        ld      a2, 0(a4)
        ld      t3, 0(a5)
        slli    s2, s2, 7
        mul     t2, s3, a0
        sltu    a0, a0, a2
        andi    a0, a0, 1
        beqz    s3, .LBB0_13
# %bb.1:                                # %header.x.subcfg.0b
   .i.us.preheader
        beqz    a0, .LBB0_19
# %bb.2:                                # %header.x.subcfg.0b
   .i.us.preheader44
        ld      a0, 8(a1)
        ld      a1, 16(a1)
        ld      a6, 0(a0)
        li      a3, 0
        ld      t1, 0(a1)
        slli    a7, t2, 3
        add     a7, a7, a6
        csrr    a1, vlenb
        srli    a2, a1, 3
        slli    a0, a1, 1
        srli    a1, a1, 2
        mul     a4, s3, a2
        slli    a4, a4, 4
        slli    a2, s3, 3
        fmv.d.x fa5, zero
```

```
41          addi    a5, a1, -1
42          and     t4, s3, a5
43          sub     t5, s3, t4
44          li      t6, 127
45          j       .LBB0_5
46  .LBB0_3:                                # exit.i_crit_edge.us
47                                          # in Loop: Header=
                                               BB0_5 Depth=1
48          add     s4, s4, t2
49          slli    s4, s4, 3
50          add     s4, s4, t0
51          fsd     fa4, 0(s4)
52  .LBB0_4:                                # exit.i.us
53                                          # in Loop: Header=
                                               BB0_5 Depth=1
54          addi    a3, s0, 1
55          beq     s0, t6, .LBB0_19
56  .LBB0_5:                                # %header.x.subcfg.0b
       .i.us
57                                          # =>This Loop Header:
                                               Depth=1
58                                          #     Child Loop
                                               BB0_9 Depth 2
59                                          #     Child Loop
                                               BB0_12 Depth 2
60          mv      s0, a3
61          or      s4, a3, s2
62          bgeu    s4, t3, .LBB0_4
63  # %bb.6:                                # %.preheader.i.us.
       preheader
64                                          #   in Loop: Header=
                                               BB0_5 Depth=1
65          bgeu    s3, a1, .LBB0_8
66  # %bb.7:                                #   in Loop: Header=
       BB0_5 Depth=1
67          li      s1, 0
68          fmv.d   fa4, fa5
69          j       .LBB0_11
70  .LBB0_8:                                # %vector.ph
71                                          #   in Loop: Header=
                                               BB0_5 Depth=1
72          slli    a3, s4, 3
73          add     a3, a3, t1
74          vsetvli a5, zero, e64, m2, ta, ma
75          mv      a5, t5
76          mv      s1, a7
77          fmv.d   fa4, fa5
78  .LBB0_9:                                # %vector.body
79                                          #   Parent Loop BB0_5
```

```
                                                    Depth=1
80                                                # =>  This Inner Loop
                                                    Header: Depth=2
81          vl2re64.v       v8, (s1)
82          vlse64.v        v10, (a3), a2
83          vfmul.vv        v8, v8, v10
84          vfmv.s.f        v10, fa4
85          vfredosum.vs    v8, v8, v10
86          vfmv.f.s        fa4, v8
87          add     s1, s1, a0
88          sub     a5, a5, a1
89          add     a3, a3, a4
90          bnez    a5, .LBB0_9
91  # %bb.10:                                      # %middle.block
92                                                #   in Loop: Header=
                                                    BB0_5 Depth=1
93          mv      s1, t5
94          beqz    t4, .LBB0_3
95  .LBB0_11:                                     # %.preheader.i.us.
    preheader42
96                                                #   in Loop: Header=
                                                    BB0_5 Depth=1
97          sub     a3, s3, s1
98          mul     a5, s3, s1
99          add     a5, a5, s4
100         slli    a5, a5, 3
101         add     a5, a5, t1
102         add     s1, s1, t2
103         slli    s1, s1, 3
104         add     s1, s1, a6
105 .LBB0_12:                                     # %.preheader.i.us
106                                               #   Parent Loop BB0_5
                                                    Depth=1
107                                               # =>  This Inner Loop
                                                    Header: Depth=2
108         fld     fa3, 0(s1)
109         fld     fa2, 0(a5)
110         fmadd.d fa4, fa3, fa2, fa4
111         addi    a3, a3, -1
112         add     a5, a5, a2
113         addi    s1, s1, 8
114         bnez    a3, .LBB0_12
115         j       .LBB0_3
116 .LBB0_13:                                     # %header.x.subcfg.0b
    .i.preheader
117         beqz    a0, .LBB0_19
118 # %bb.14:                                     # %header.x.subcfg.0b
    .i.preheader14
119         csrr    a2, vlenb
```

```
120         srli    a2, a2, 2
121         li      a0, 128
122         bgeu    a0, a2, .LBB0_16
123 # %bb.15:
124         li      a1, 0
125         j       .LBB0_21
126 .LBB0_16:                               # %vector.ph22
127         addiw   a1, a2, -1
128         andi    a3, a1, 128
129         xori    a1, a3, 128
130         vsetvli a4, zero, e64, m2, ta, ma
131         vid.v   v8
132         vmv.v.i v10, 0
133         mv      a4, a1
134 .LBB0_17:                               # %vector.body27
135                                         # =>This Inner Loop
                                               Header: Depth=1
136         vor.vx  v12, v8, s2
137         vmsltu.vx       v0, v12, t3
138         vadd.vx v12, v12, t2
139         vsll.vi v12, v12, 3
140         vsoxei64.v      v10, (t0), v12, v0.t
141         sub     a4, a4, a2
142         vadd.vx v8, v8, a2
143         bnez    a4, .LBB0_17
144 # %bb.18:                               # %middle.block19
145         bnez    a3, .LBB0_21
146 .LBB0_19:                               # original.exit
147         ld      s0, 40(sp)                      # 8-byte
              Folded Reload
148         ld      s1, 32(sp)                      # 8-byte
              Folded Reload
149         ld      s2, 24(sp)                      # 8-byte
              Folded Reload
150         ld      s3, 16(sp)                      # 8-byte
              Folded Reload
151         ld      s4, 8(sp)                       # 8-byte
              Folded Reload
152         addi    sp, sp, 48
153         ret
```

Listing B.4: GEMM kernel using USM and Buffer accessors (same assembly) and sycl::nd_range

```
1 # %bb.0:                                # %entry
2         addi    sp, sp, -48
3         sd      s0, 40(sp)                      # 8-byte
            Folded Spill
4         sd      s1, 32(sp)                      # 8-byte
            Folded Spill
```

```
5         sd      s2, 24(sp)                      # 8-byte
             Folded Spill
6         sd      s3, 16(sp)                      # 8-byte
             Folded Spill
7         sd      s4, 8(sp)                       # 8-byte
             Folded Spill
8         sd      s5, 0(sp)                       # 8-byte
             Folded Spill
9         ld      a2, 0(a1)
10        ld      a3, 24(a1)
11        ld      t1, 24(a0)
12        ld      s2, 0(a2)
13        ld      a7, 0(a3)
14        slli    t1, t1, 5
15        beqz    s2, .LBB0_12
16# %bb.1:                                # %header.y.subcfg.0b
     .i.us.preheader
17        ld      a2, 8(a1)
18        ld      a1, 16(a1)
19        ld      a6, 32(a0)
20        li      a0, 0
21        ld      t0, 0(a2)
22        ld      t4, 0(a1)
23        slli    a6, a6, 5
24        csrr    s5, vlenb
25        srli    a1, s5, 3
26        srli    a5, s5, 2
27        slli    a4, s2, 3
28        slli    s5, s5, 1
29        mul     s0, s2, a1
30        slli    s0, s0, 4
31        fmv.d.x fa5, zero
32        li      t2, 31
33        j       .LBB0_3
34.LBB0_2:                                # %latch.y.subcfg.0b.
     i.split.us.us
35                                        #   in Loop: Header=
                                             BB0_3 Depth=1
36        addi    a0, t3, 1
37        beq     t3, t2, .LBB0_13
38.LBB0_3:                                # %header.y.subcfg.0b
     .i.us
39                                        # =>This Loop Header:
                                             Depth=1
40                                        #     Child Loop
                                             BB0_5 Depth 2
41                                        #       Child Loop
                                             BB0_8 Depth 3
42                                        #       Child Loop
```

```
                                                    BB0_11 Depth 3
        li      a1, 0
        mv      t3, a0
        or      a0, a0, a6
        mul     t6, a0, s2
        mul     t5, a4, a0
        add     t5, t5, t0
        j       .LBB0_5
.LBB0_4:                                # exit.i_crit_edge.us
   .us
                                        # in Loop: Header=
                                           BB0_5 Depth=2
        add     s3, s3, t6
        slli    s3, s3, 3
        add     s3, s3, a7
        fsd     fa4, 0(s3)
        addi    a1, s4, 1
        beq     s4, t2, .LBB0_2
.LBB0_5:                                # %header.x.subcfg.0b
   .i.us.us
                                        #   Parent Loop BB0_3
                                           Depth=1
                                        # =>  This Loop
                                           Header: Depth=2
                                        #       Child Loop
                                           BB0_8 Depth 3
                                        #       Child Loop
                                           BB0_11 Depth 3
        mv      s4, a1
        or      s3, a1, t1
        bgeu    s2, a5, .LBB0_7
# %bb.6:                                #   in Loop: Header=
   BB0_5 Depth=2
        li      a1, 0
        fmv.d   fa4, fa5
        j       .LBB0_10
.LBB0_7:                                # %vector.ph
                                        #   in Loop: Header=
                                           BB0_5 Depth=2
        addi    a0, a5, -1
        and     a2, s2, a0
        sub     a1, s2, a2
        slli    a3, s3, 3
        add     a3, a3, t4
        vsetvli a0, zero, e64, m2, ta, ma
        mv      s1, a1
        mv      a0, t5
        fmv.d   fa4, fa5
.LBB0_8:                                # %vector.body
```

```
82                                                  #   Parent Loop BB0_3
                                                        Depth=1
83                                                  #     Parent Loop
                                                      BB0_5 Depth=2
84                                                  # =>    This Inner
                                                      Loop Header: Depth
                                                      =3
85          vl2re64.v       v8, (a0)
86          vlse64.v        v10, (a3), a4
87          vfmul.vv        v8, v8, v10
88          vfmv.s.f        v10, fa4
89          vfredosum.vs    v8, v8, v10
90          vfmv.f.s        fa4, v8
91          add     a0, a0, s5
92          sub     s1, s1, a5
93          add     a3, a3, s0
94          bnez    s1, .LBB0_8
95  # %bb.9:                                        # %middle.block
96                                                  #   in Loop: Header=
                                                      BB0_5 Depth=2
97          beqz    a2, .LBB0_4
98  .LBB0_10:                                       # %scalar.ph.
      preheader
99                                                  #   in Loop: Header=
                                                      BB0_5 Depth=2
100         sub     a2, s2, a1
101         mul     a0, s2, a1
102         add     a0, a0, s3
103         slli    a0, a0, 3
104         add     a3, t4, a0
105         add     a1, a1, t6
106         slli    a1, a1, 3
107         add     a1, a1, t0
108 .LBB0_11:                                       # %scalar.ph
109                                                 #   Parent Loop BB0_3
                                                        Depth=1
110                                                 #     Parent Loop
                                                      BB0_5 Depth=2
111                                                 # =>    This Inner
                                                      Loop Header: Depth
                                                      =3
112         fld     fa3, 0(a1)
113         fld     fa2, 0(a3)
114         fmadd.d fa4, fa3, fa2, fa4
115         addi    a2, a2, -1
116         add     a3, a3, a4
117         addi    a1, a1, 8
118         bnez    a2, .LBB0_11
119         j       .LBB0_4
```

```
120 .LBB0_12:                                   # original.exit.
    loopexit
121         slli    t1, t1, 3
122         add     a7, a7, t1
123         li      a0, 32
124         vsetvli zero, a0, e64, m1, ta, ma
125         vmv.v.i v8, 0
126         vse64.v v8, (a7)
127 .LBB0_13:                                   # original.exit
128         ld      s0, 40(sp)                          # 8-byte
            Folded Reload
129         ld      s1, 32(sp)                          # 8-byte
            Folded Reload
130         ld      s2, 24(sp)                          # 8-byte
            Folded Reload
131         ld      s3, 16(sp)                          # 8-byte
            Folded Reload
132         ld      s4, 8(sp)                           # 8-byte
            Folded Reload
133         ld      s5, 0(sp)                           # 8-byte
            Folded Reload
134         addi    sp, sp, 48
135         ret
```

B.3 OneAPI

Listing B.5: Simple for-loop

```
1 N/A
```

Listing B.6: Unified shared memory with sycl::range

```
1 N/A
```

Listing B.7: Buffer accessors memory with sycl::range

```
1  .LBB3_7:    # Prepares for scalar or vector processing.
2      or    s4, s11, a3
3      bgeu    s4, s9, .LBB3_6
4      slli    s4, s4, 3
5      bgeu    s3, s1, .LBB3_10
6      li    s5, 0
7      fmv.d    fa4, fa5
8      j    .LBB3_13
9  .LBB3_10:   # Initializes for vector floating-point
     computation.
10     addi    a0, s1, -1
11     and    s6, s3, a0
12     sub    s5, s3, s6
```

```
13      add     a0, t5, s4
14      mv      s0, t3
15      mv      a4, s5
16      vmv2r.v     v10, v8
17      vsetvli     a2, zero, e64, m2, ta, ma
18  .LBB3_11:   # Main loop for vector floating-point operations.
19      vl2re64.v     v12, (s0)
20      vlse64.v    v14, (a0), a5
21      vfmul.vv    v12, v12, v14
22      vfredosum.vs     v10, v12, v10
23      sub     a4, a4, s1
24      add     s0, s0, s7
25      add     a0, a0, a1
26      bnez     a4, .LBB3_11
27      vfmv.f.s     fa4, v10
28      beqz     s6, .LBB3_5
29  .LBB3_13:   # Initializes for scalar floating-point
        computation.
30      sub     a4, s3, s5
31      mul     a0, a5, s5
32      add     a0, a0, s4
33      add     a0, a0, t5
34      slli     s0, s5, 3
35      add     s0, s0, t6
36  .LBB3_14:   # Main loop for scalar floating-point operations.
37      fld     fa3, 0(s0)
38      fld     fa2, 0(a0)
39      fmul.d     fa3, fa3, fa2
40      fadd.d     fa4, fa4, fa3
41      addi     a4, a4, -1
42      add     a0, a0, a5
43      addi     s0, s0, 8
44      bnez     a4, .LBB3_14
45      j     .LBB3_5
```

Listing B.8: Unified shared memory with sycl::nd_range

```
1   N/A
```

Listing B.9: Buffer accessors memory model with sycl::nd_range

```
1   .LBB3_3:    # Prepares for scalar or vector processing.
2           mv      t0, s10
3           mv      a7, s0
4           ld      a1, 16(sp)
5           or      s6, a1, s9
6           mul     s11, s6, a6
7           slli    s11, s11, 3
8           add     s8, ra, s11
9           bgeu    a6, a0, .LBB3_5
```

```
10          li      s10, 0
11          fmv.d   fa4, fa5
12          j       .LBB3_8
13  .LBB3_5:  # Remainder handling setup
14      addi a1, a0, -1
15      and s0, a6, a1
16      sub s10, a6, s0
17      ld a2, 0(sp)
18      mv a5, s8
19      mv a1, s10
20      vmv2r.v v10, v8
21      vsetvli s7, zero, e64, m2, ta, ma
22  .LBB3_6:  # Vector loop
23      vl2re64.v v12, (a5)
24      vlse64.v v14, (a2), a3
25      vfmul.vv v12, v12, v14
26      vfredosum.vs v10, v12, v10
27      sub a1, a1, a0
28      add a5, a5, s1
29      add a2, a2, a4
30      bnez a1, .LBB3_6
31      vfmv.f.s fa4, v10
32      beqz s0, .LBB3_10
33  .LBB3_8:  # Scalar loop setup
34      sub a5, a6, s10
35      mul s0, a3, s10
36      ld a1, 8(sp)
37      add s0, s0, a1
38      mul a1, a3, s6
39      slli s10, s10, 3
40      add a1, a1, s10
41      add a2, ra, a1
42  .LBB3_9:  # Scalar loop
43      fld fa3, 0(a2)
44      fld fa2, 0(s0)
45      fmul.d fa3, fa3, fa2
46      fadd.d fa4, fa4, fa3
47      addi a5, a5, -1
48      add s0, s0, a3
49      addi a2, a2, 8
50      bnez a5, .LBB3_9
51  .LBB3_10: # Store result
52      ld a1, 136(sp)
53      add s11, s11, a1
54      ld a1, 24(sp)
55      add a1, a1, s11
56      fsd fa4, 0(a1)
```

B.4 PoCL

Listing B.10: Simple for-loop

```
# %bb.1:                                # %.critedge.critedge.i.i
        slli    s0, a0, 2
        add     s4, s4, s0
        lw      a0, 0(s4)
        add     s3, s3, s0
        lw      a1, 0(s3)
        call    __addsf3
        add     s0, s2, s0
        sw      a0, 0(s0)
```

Listing B.11: Unified shared memory with sycl::range

```
# %bb.1:
        slli    s4, s4, 4
        or      s4, s5, s4
        bgeu    s4, s1, .LBB0_8
# %bb.2:                                # %.critedge.
   preheader.i.i
        beqz    s0, .LBB0_6
# %bb.3:                                # %.lr.ph.i.i
        mv      a0, s3
        mv      a1, s0
        call    __muldi3
        mv      s1, a0
        li      s2, 0
        slli    a0, s4, 3
        add     s8, s8, a0
        slli    s5, s0, 3
        slli    a0, s1, 3
        add     s7, s7, a0
.LBB0_4:                                # =>This Inner Loop
   Header: Depth=1
        ld      a1, 0(s7)
        ld      a0, 0(s8)
        call    __muldf3
        mv      s3, a0
        mv      a0, s2
        call    __extendsfdf2
        mv      a1, a0
        mv      a0, s3
        call    __adddf3
        call    __truncdfsf2
        mv      s2, a0
        addi    s0, s0, -1
        add     s8, s8, s5
        addi    s7, s7, 8
        bnez    s0, .LBB0_4
# %bb.5:                                # loopexit.i.i
        mv      a0, s2
        call    __extendsfdf2
        j       .LBB0_7
.LBB0_6:
        li      s1, 0
        li      a0, 0
.LBB0_7:                                # exit.i.i
        add     s1, s1, s4
        slli    s1, s1, 3
        add     s1, s6, s1
        sd      a0, 0(s1)
```

Listing B.12: Buffer accessors memory with sycl::range

```
# %bb.1:
        slli    s4, s4, 4
        or      s4, s5, s4
        bgeu    s4, s1, .LBB0_7
# %bb.2:                                # %.critedge.
   preheader.i.i
        beqz    s0, .LBB0_5
# %bb.3:                                # %.lr.ph.i.i
        mv      a0, s3
        mv      a1, s0
        call    __muldi3
        mv      s1, a0
        li      s2, 0
        slli    a0, s4, 3
        add     s8, s8, a0
        slli    s3, s0, 3
        slli    a0, s1, 3
        add     s7, s7, a0
.LBB0_4:                                # %.critedge.i.i
                                        # =>This Inner Loop
                                           Header: Depth=1
        ld      a1, 0(s7)
        ld      a0, 0(s8)
        call    __muldf3
        mv      a1, s2
        call    __adddf3
        mv      s2, a0
        addi    s0, s0, -1
        add     s8, s8, s3
        addi    s7, s7, 8
        bnez    s0, .LBB0_4
        j       .LBB0_6
.LBB0_5:
        li      s1, 0
        li      s2, 0
.LBB0_6:                                # exit.i.i
        add     s1, s1, s4
        slli    s1, s1, 3
        add     s1, s6, s1
        sd      s2, 0(s1)
```

Listing B.13: Unified shared memory with sycl::nd_range

```
# %bb.1:                                # %.lr.ph.i.i
        slli    s1, s1, 4
        or      a0, s2, s1
        mv      a1, s0
        call    __muldi3
        mv      s1, a0
        li      s2, 0
        slli    a0, s3, 3
        add     s6, s6, a0
        slli    s7, s0, 3
        slli    a0, s1, 3
        add     s5, s5, a0
.LBB0_2:                                # =>This Inner Loop
   Header: Depth=1
        ld      a1, 0(s5)
        ld      a0, 0(s6)
        call    __muldf3
        mv      a1, s2
        call    __adddf3
        mv      s2, a0
        addi    s0, s0, -1
        add     s6, s6, s7
        addi    s5, s5, 8
        bnez    s0, .LBB0_2
        j       .LBB0_4
.LBB0_3:
        li      s1, 0
        li      s2, 0
```

Listing B.14: Buffer accessors memory model with sycl::nd_range

```
# %bb.1:                                # %.lr.ph.i.i
        slli    s1, s1, 4
        or      a0, s2, s1
        mv      a1, s0
        call    __muldi3
        mv      s1, a0
        li      s2, 0
        slli    a0, s3, 3
        add     s6, s6, a0
        slli    s7, s0, 3
        slli    a0, s1, 3
        add     s5, s5, a0
.LBB0_2:                                # =>This Inner Loop
   Header: Depth=1
        ld      a1, 0(s5)
        ld      a0, 0(s6)
        call    __muldf3
        mv      a1, s2
        call    __adddf3
        mv      s2, a0
        addi    s0, s0, -1
        add     s6, s6, s7
        addi    s5, s5, 8
        bnez    s0, .LBB0_2
        j       .LBB0_4
.LBB0_3:
        li      s1, 0
        li      s2, 0
```

C

Compiler reports

C.1 OpenMP JIT

Listing C.1: Simple for loop using USM and sycl::range

```
--- !Passed
Pass:     loop-vectorize
Name:     Vectorized
Function: 'hipsycl_glue_sscp_dispatch_basic_parallel_for'
Args:
  - String:          'vectorized loop (vectorization width:'
  - VectorizationFactor: vscale x 2
  - String:          ', interleaved count: '
  - InterleaveCount: '1'
  - String:          ')'
```