Universiteit
Leiden
The Netherlands

Evaluating the Effectiveness of Multi-Striding on the Raspberry Pi 5 and Banana Pi F3.

Steffan Radojevic

Supervisors:
Miguel O. Blom & Rob V. van Nieuwpoort

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

29/08/2025

Abstract

Multi-striding defines a transformation that enables higher hardware prefetch utilization to tap into the underutilized memory bandwidth, improving memory-bound kernels. An initial evaluation on x86-64 micro-architectures yields results on-par with modern hand-optimized solutions. However, evaluations on alternative architectures with distinct hardware implementations, like ARM and RISC-V have not yet been carried out. In this thesis, we evaluate the effectiveness of multi-striding on two single-board computers, the Raspberry Pi 5 (ARM Cortex A76) and the Banana Pi BPI-F3 (RISC-V SpacemiT K1). We modify the original approach to map out larger spaces of striding configurations and adjust input sizes as a measure to reduce the number of address collisions, improving the accuracy of the measurements. These measurements show that configurations must enforce accesses to each cache line to be contained within a single loop iteration, i.e., preventing any cache line from being shared across iterations. For both architectures, we see that multi-striding improves throughput in our micro-benchmarks. We see an improvement of 1.03× for scalar writes from multi-striding when compared to the single-strided configurations, on the RISC-V SpacemiT K1. On the more sophisticated ARM Cortex-A76 we achieve up to 83% of the theoretical maximum bandwidth, and compared to the single-strided baselines, we find speedup factors of 1.51× for scalar loads, 1.47× for scalar stores and 1.54× for vector stores. Most notably, when compared to the ubiquitous `memset` from the standard C library, we achieve a speedup of 1.55× on the Raspberry Pi 5 using vector stores.

Contents

1 Introduction

Many fields in Computer Science, such as artificial intelligence and high-performance computing, perform memory-bound computations. Although computing power is increasing at a tremendous pace, improvements in memory have not been as rapid, thus affecting big data workloads [CDK+15]. Furthermore, in artificial intelligence, the performance bottleneck seems to be shifting towards memory due to being limited by the traffic between CPU and memory as seen in matrix-vector operations, which indicates the importance of bandwidth utilization [GYK+24].

Convolutions, Matrix Vector Multiplication, and other memory-bound kernels take advantage of the hardware prefetcher to retrieve data from memory into the cache. By detecting memory access patterns in memory-bound kernels, the hardware prefetcher can fetch data into the cache in advance, thus resulting in fewer cache misses and improving bandwidth utilization, and therefore, throughput.

A recent paper by Miguel O. Blom, Kristian F. D. Rietveld, and Rob V. van Nieuwpoort demonstrated that by transforming memory access patterns from single strides to multiple strides, we can improve the utilization of the hardware prefetcher [BRN25]. This novel idea, which involves transforming memory access patterns to multiple strides, is named in the paper as **multi-striding**. The paper shows that memory-bound kernels using multi-striding outperform single-strided memory-bound kernels for memory throughput up to a factor of 2.18x and also state-of-the-art kernels like 1.98x over OpenBLAS, 1.08x over Halide, 2.99x over Intel's MKL, and 1.87x over OpenCV. The compute kernel is transformed so that multiple contiguous sequences of memory accesses occur, or in other words, multiple strides occur in the access pattern. As a result, more cache lines are brought ahead of time into the cache, improving cache hits and effective memory bandwidth.

The experiments in the paper by Blom et. al., were conducted on x86-64 architectures. Hardware prefetchers differ between architectures, and therefore, further research is required to see the effects on different architectures like RISC-V and ARM, so that these architectures could potentially also benefit from multi-striding. In this thesis, we will evaluate the effectiveness of multi-striding on the ARM (AArch64) and RISC-V (RV64) architectures. This is done on two devices: a Raspberry Pi 5 equipped with an ARM Cortex-A76 processor, and a Banana Pi BPI-F3 equipped with a RISC-V SpacemiT K1 processor.

Section 3 provides the background information required for understanding this thesis. In Section 4, we will discuss which micro-kernels are implemented. We examine an example of an x86_64 memory access instruction implemented in the paper by Blom et. al., and explain the reason why this is not feasible for the ARM and RISC-V architectures. In Section 5, we show our approach of implementing and evaluating multi-striding for the ARM and RISC-V architectures. In Section 6.3, we discuss our experimental setup and the hardware and software specifications of the tested devices. Finally, in Section 8 and in Section 9, we discuss the results and evaluate whether multi-striding improves performance on the ARM and RISC-V devices.

2 Related Work

Many kernels have a low arithmetic intensity, and are therefore bound by memory bandwidth. Benchmarking this feature is crucial for the analysis of its scalability and suitability. By utilizing four simple kernels, the STREAM benchmark is a widely used tool for evaluating memory bandwidth [McC07]. Volokitin et. al. [VKK+23] measured the memory bandwidth on RISC-V micro-architectures using this benchmark. Other researchers implemented their own benchmark, such as arm-bench by Burth et. al. [BVS25], achieving throughput results closer to the limit of the hardware architecture. While these benchmarks are run on multiple devices, some studies focus on a single device, such as the Raspberry Pi 5, running many benchmarks to evaluate features beyond memory bandwidth [Lon24].

Since numerous kernels are memory-bound, increasing memory bandwidth throughput is essential for computational performance. Various studies investigate techniques and optimizations for increasing this throughput, such as the paper by Liu et. al. [LLT+23], which developed a new automatic vectorization technique achieving speedups of 1.20x over GCC[CITE5]. Pirova et. al [PVK+25], implemented four micro-kernels using band matrices, achieving speedups of 1.5x up to 10x compared to openBLAS for RISC-V micro-architectures, such as the Banana Pi BPI-F3. The paper by Li et. al. [LZRX25], applied a new hardware prefetcher selection algorithm named `Alecto`, outperforming other state-of-the-art RL-based selection algorithms.

Extensive research has been conducted on hardware prefetchers. For example, showing the effectiveness of hardware prefetching for B^+-Trees and binary search loads [MWR25]. A recent study by Ho et. al. [HFP+25], investigated the impact of hardware prefetchers on ARM-based high-end processors, reporting an extensive evaluation of hardware prefetchers on various kernels. They analyzed the memory access patterns, and evaluated these kernels using relevant prefetching techniques. Despite being the least accurate, the Next-Line prefetcher achieves the highest bandwidth utilization compared to other hardware prefetchers, such as the stride prefetcher. Additionally, the impact of prefetching aggressiveness is investigated. They reported that more aggressive prefetching for the Next-Line prefetcher, which increases useless prefetches, does not lead to performance degradation.

Most micro-architectures include multiple hardware prefetchers. Examples of hardware prefetchers are adjacent-line-prefetchers (automatically fetches neighboring cache lines), stream prefetchers (detecting linear sequential memory accesses and prefetches next cache lines), and stride prefetchers (which detect memory access patterns based on strides and predict future memory accesses). Some hardware prefetchers are characterized by certain features. Schlüter et. al. [SCH+23] identify these features, and the hardware prefetchers present in 19 different ARM and x86_64 micro-architectures, including those for the Cortex A-76. One important feature is the minimal stride in bytes and the number of cache lines prefetched into the cache when the hardware prefetcher is triggered. For the Cortex A-76, the minimal stride in bytes has to be 64 bytes, corresponding to one cache line. When this stride prefetcher is triggered, it brings between one and sixteen cache lines into the cache.

While several studies investigated performance optimizations, such as boosting hardware prefetcher utilization on the ARM and RISC-V architectures, none examined the effectiveness of multi-striding

by changing the memory access pattern on these architectures.

3 Background

In this section, we present the necessary background required to understand this thesis. First, we will give a brief explanation of hardware architectures, including registers and stacks. Second, we will provide information about the ARM and RISC-V architectures, as well as descriptions of all the instructions used in our approach. Finally, we will explain multi-striding and provide a brief description of address collisions.

3.1 Hardware Architectures

3.1.1 General-purpose registers and Stack

The general-purpose registers can be viewed as generic integer registers. The size of general-purpose registers depends on the processor architecture, which can be either 32-bit or 64-bit. Most processors have sixteen or thirty-two general-purpose registers, such as Intel processors, which typically have sixteen registers in their x86_64 architecture [1]. These registers are typically used for temporary storage of data during program execution. This saves time because intermediate data can be kept in the registers instead of repeatedly reading from or writing to memory (RAM). Some registers are unique, such as the zero registers (hardwired to zero). These special registers are not available for use. In addition, the ARM and RISC-V architectures include callee-saved and caller-saved registers. Callee-saved registers hold values that have to be preserved across function calls. Therefore, these values have to be restored before the functions return. Caller-saved register does not have to be preserved across function calls.

3.1.2 Floating-point registers

The floating-point registers (often referred to as FP registers) are special registers that typically hold single-precision floating point values (32-bit) or double-precision floating point values (64-bit). These floating-point registers contain representations floating point numbers, and therefore, these registers have a limited precision, and a maximum range they can accurately represent. In some architectures, such as ARM, the floating-point registers are combined with the vector/SIMD registers [2].

3.1.3 Vector registers

Vector registers are single instruction, multiple data (SIMD) registers used to hold and process multiple data elements simultaneously, such as integers or single-precision floating points. One vector instruction processes multiple data elements simultaneously, enabling SIMD processing. These vector registers are typically defined as an architectural extension, such as the "V" extension in the RISC-V architecture. It is important to note that, although vector registers are part of a micro-architecture,

[1] Microsoft, x64 Architecture, https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/x64-architecture

[2] ARM, Registers in AArch64 - general-purpose registers, https://developer.arm.com/documentation/102374/0102/Registers-in-AArch64---general-purpose-registers

not all micro-architectures support vector registers. Modern micro-architectures typically include vector registers, as they often contribute to significantly improved performance [KP02]. Most micro-architectures have between sixteen and thirty-two vector registers with a standard size (e.g., 128-bit, 256-bit). However, some vector extensions support scalable vectors, allowing the register size to exceed the fixed standard size, such as the scalable vector extensions (SVE) in the ARM architecture.

3.1.4 Stack

The number of available general-purpose registers is limited, thus we need a place to store values, including callee-saved registers, when they are not in use (i.e. *spilling)*. The stack is an allocated memory segment within main memory (RAM). The stack pointer register keeps track of the top of the stack. Data can be placed on top of the stack, and this data can also be removed from the stack. In most architectures, the stack grows towards lower memory addresses.

3.2 ARM

The ARM architecture is a RISC (Reduced Instruction Set Computer) based architecture licensed by ARM Ltd., and is not open-source. Some important characteristics of RISC-based architecture are: instructions are simple and fixed format, most instructions execute in one clock cycle, and fewer instructions and addressing modes are supported [Ale92]. Popular in mobile devices and embedded systems, where approximately 99% of mobile smartphones are powered by ARM [3], the ARM architecture in recent years also gained traction in High-Performance Computing [JTW+19]. In 2020, Apple Inc. introduced its first ARM-based processor for laptops, making a transition from the Intel x86 [4]. Different architecture versions are available (e.g., ARMv7-A, ARMv8-A), and each version operates in either AArch32 (32-bit) or AArch64 (64-bit) mode. For the purpose of this thesis, we will only discuss the AArch64 (64-bit) mode [5].

3.2.1 Registers, Calling Conventions and Vectors

General-purpose Registers The AArch64 architecture provides thirty-one 64-bit general-purpose registers [6]. Each general-purpose register can be used as either `x0-x30` (64-bit register), or as `w0-w30` (32-bit register). In Table 1, the Procedure Call Standard for the AArch64 architecture. Registers `x0-x7` are parameter and result registers. The `XR` register (`x8`) is an indirect result register that holds a memory address pointing to a value larger than the typical 64-bit general-purpose register. Registers `x9-x15` are corruptible registers, the called function can overwrite these registers without needing to restore. Register `x16` and `x17` are intra-procedure-call corruptible registers. Linkers use these registers to insert small pieces of code between the caller and the callee, for

[3] ARM, Consumer Technologies Smartphones, https://www.arm.com/markets/consumer-technologies/smartphones

[4] Apple, Apple unleashes M1, Press Release, November 10, 2020, https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/

[5] ARM, Learn the architecture - A64 Instruction Set Architecture Guide 1.2, https://developer.arm.com/documentation/102374/latest/

[6] Registers in AArch64 - general-purpose registers, https://developer.arm.com/documentation/102374/0102/Registers-in-AArch64---general-purpose-registers

example, for branch range extension. The PR register is the platform register. Registers x19-28 are callee-saved registers. The FP register (x29) is the frame pointer, and the LR register(x30) is the link register, for function calls. The zero register in the AArch64 is xzr and wzr, these register always reads zero and ignore writes. The register alias for the stack pointer is sp.

Register	Caller/callee-saved	Alias
x0-7	Caller-saved	-
x8	Caller-saved	XR
x9-15	Caller-saved	-
x16	Caller-saved	IP1
x17	Caller-saved	IP2
x18	Caller-saved	PR
x19-28	Callee-saved	-
x29	Callee-saved	FP
x30	Callee-saved	LR

Table 1: Calling convention for the general-purpose registers in the ARM architecture.
ARM, Procedure Call Standard,
https://developer.arm.com/documentation/102374/0102/Procedure-Call-Standard

Floating-point/Vector Registers The AArch64 architecture do not have dedicated floating-point registers, instead it has thirty-two registers used for both floating-point and vector operations. The registers are 128-bit. These registers can be accessed in several ways. Bx (byte) is 8 bits, Hx (half) is 16 bits, Sx is 32-bit, Dx is 64-bit, and Qx is 128-bit.

Advanced SIMD In the AArch64 architecture, there are two types of vector processing: Advanced SIMD, also known as NEON, and Scalable Vector Extension (SVE and SVE2) [7]. In this thesis, we will utilize the Advanced SIMD vector processing. Advanced SIMD uses the floating-point/vector registers.

3.2.2 Implemented Assembly Instructions

All the instruction encodings in the following paragraphs are described in the ARM reference manual [ARM24].

LDR - Load (immediate/SIMD&FP) There are numerous variants for this instruction (e.g., LDR register, LDR immediate, LDR literal). We will only discuss the immediate variant used in this thesis. The LDR (immediate/SIMD&FP) instruction has three indexing modes, Post-index, Pre-index, and Unsigned offset. In this thesis, we will only use the Post-index encoding. In Listing 1 the (immediate/SIMD&FP) Post-index encoding. The instruction reads a value from memory and stores it in the general-purpose register **Xt**. The Xt register can also be other register types, such as a 32-bit floating-point register (e.g., S0) or a 128-bit vector register (e.g., Q0). The <Xn|SP> is the 64-bit general-purpose base register, which either holds the address of the memory to be read

[7] ARM Data processing - vector and matrix data, https://developer.arm.com/documentation/102374/0102/Data-processing---vector-and-matrix-data

or the stack pointer. The `#<simm>` indicates that we use the Post-index encoding. For the AArch64 architecture, this immediate value must be a multiple of eight and fall within the range 0 to 32760. When using Post-index encoding, the immediate value is added to `<Xn|SP>` after the value is read from memory.

```
1 LDR <Xt>, [<Xn|SP>], #<simm>
```

Listing 1: Encoding for the 64-bit variant of the load instruction LDR (immediate) in the AAarch64 architecture.

`STR` - Store (immediate/SIMD&FP) In Listing 1 the `LDR (immediate/SIMD&FP)` Post-index encoding. The instruction stores the value of the general-purpose register `Xt` to memory. All encoding components in this instruction share the same definition as the `LDR (immediate/SIMD&FP)` instruction. Therefore, they will not be discussed here.

```
1 STR <Xt>, [<Xn|SP>], #<simm>
```

Listing 2: Encoding for the 64-bit variant of the store instruction STR (immediate) in the AAarch64 architecture.

`LD1` - Load multiple single-element structures There are four variants of this instruction (i.e., `LD1`,`LD2`, `LD3` and `LD4`), indicating how many single-element structures are loaded. We will focus on the `LD1` instruction used in this paper. This instruction has two indexing modes, No offset and Post-index modes. We will discuss the Post-index mode. The `LD1` instruction is part of the Advanced SIMD extension of the AArch64 architecture. In Listing **??**, the encoding is shown for the `LD1` (Multiple Structures) instruction. This instruction reads multiple single-element structures from memory and loads these elements into the vector register `Vt`. The `<T>` indicates the arrangement specifier (i.e., element type). We will use the `4S` arrangement specifier, representing four 32-bit floating-point values. The `[<Xn|SP>]` is the 64-bit general-purpose base register or stack pointer. This is the memory address from which the data will be read. At last, the `<imm>` represents the Post-index immediate value. This value is added to the `<Xn|SP>` general-purpose register after the elements are read from memory. For example, when stream data sequentially from memory with 128-bit vector registers, the immediate value is `#16`, since four 32-bit floating-points (i.e., 16 bytes) are read.

```
1 LD1 { <Vt>.<T> }, [<Xn|SP>], <imm>
```

Listing 3: Encoding one register (LD1), immediate offset variant of the vector load instruction LD1 (Multiple Structures) in the AAarch64 architecture.

`ST1` - Store multiple single-element structures There are numerous variants and encodings for this instruction, similar to the `LD1` instruction. The only difference between the `LD1` (Multiple Structures) and the `ST1` (Multiple Structures) instruction is that the `ST1` instruction stores the elements from vector `Vt` to the memory address in 64-bit general-purpose register `[<Xn|SP>]`. All the encoding components share the same definition. In Listing 4 the encoding for the `ST1` (Multiple Structures) instruction.

```
1 ST1 { <Vt>.<T> }, [<Xn|SP>], <imm>
```

Listing 4: Encoding for the 64-bit variant of the store instruction STR (immediate) in the AAarch64 architecture.

MOVZ - Move wide with zero The encoding of the MOVZ is shown in Listing 5. The <Xd> specifies the 64-bit general-purpose destination registers. The <#imm> encoding is a 16-bit value, ranging from 0 to 65535. The MOVZ instruction clears all the bits in the general-purpose register by zero, then the immediate value is loaded into the least significant bits of the general-purpose register <Xd>. If desired, an additional immediate value can be encoded in the instruction to shift the loaded immediate to the left using the #<shift> encoding.

```
1 MOVZ <Xd>, #<imm>{, LSL #<shift>}
```

Listing 5: Encoding for the 64-bit variant of the MOVZ instruction in the AAarch64 architecture.

MOVK - Move wide with zero The encoding of the MOVK is shown in Listing 6. The <Xd> specifies the 64-bit general-purpose destination registers. The <#imm> encoding is a 16-bit value, ranging from 0 to 65535. The MOVK instruction keeps all the bits in the general-purpose. The immediate value is loaded into the least significant bits of the general-purpose register <Xd>. If desired, an additional immediate value can be encoded in the instruction to shift the loaded immediate to the left using the #<shift> encoding.

```
1 MOVK <Xd>, #<imm>{, LSL #<shift>}
```

Listing 6: Encoding for the 64-bit variant of the MOVK instruction in the AAarch64 architecture.

3.3 RISC-V

The RISC-V architecture is an extension of the RISC-based architecture. A well-known feature of RISC-V is its open-source licensing model. The RISC-V development started in May 2010 at UC Berkeley [8]. The RISC-V architecture is relatively new compared to other architectures, with the vector processing specification only being released in 2021 [9]. Moreover, RISC-V is actively being explored in many areas, such as Automotive AI[10] and Embedded Edge Computing [OL23].

3.3.1 Registers, Calling Conventions and Vectors

Three register widths version are available, RV32 (32-bit), RV64 (64-bit) and the RV128 (128-bit) version [11]. For the purpose of this thesis, we will only discuss the RV64 (64-bit) mode.

[8] RISC-V, History of RISC-V, https://riscv.org/about/

[9] RISC-V, riscv-v-spec, https://github.com/riscvarchive/riscv-v-spec/releases/tag/v1.0

[10] RISC-V, RISC-V for Automotive AI Use Cases, https://riscv.org/wp-content/uploads/2025/04/RISC-V_AIOpportunitiesChallenges_042825.pdf

[11] WikiChip, Overview, https://en.wikichip.org/wiki/risc-v/registers

General-purpose Registers The RISC-V (RV64 variant) architecture defines thirty-two 64-bit general-purpose registers. Each general-purpose register can be used as `x0-x31`. In Table 2, a list of the assembler mnemonics and *Saver* (i.e., value saved across function calls). Register `x0` is hard-wired to zero, which means that store operations to this register have no effect, and it always reads a zero. The `ra` register is utilized for the return address. The `sp` register is used for the stack pointer, which keeps track of the top of the stack. The global pointer (i.e., `gp` register) points to the middle of the global data area (GDA) in memory. The `tp` register is the thread pointer, pointing to the thread-local storage. Both these registers are considered *persistent* registers: their values must not be modified during runtime. The `t0-6` are temporary values. These registers may be used freely within a function. The `a0-7` contain the function arguments, used to pass function arguments, with the `a0` and `a1` registers also functioning as registers for return values. The `s0-11` are callee-saved registers, with the addition of `s0` serving as the frame pointer `fp`.

Register	Caller/callee-saved	ABI Name
x0	-	zero
x1	Caller-saved	ra
x2	Callee-saved	sp
x3	-	gp
x4	-	tp
x5-7	Caller-saved	t0-2
x8	Callee-saved	s0/fp
x9	Callee-saved	s1
x10-17	Caller-saved	a0-7
x18-27	Callee-saved	s2-11
x28-31	Caller-saved	t3-6

Table 2: Calling convention for the general-purpose registers in RISC-V architecture.
RISC-V, Calling Convention, https://riscv.org/wp-content/uploads/2024/12/riscv-calling.pdf

Floating-point Registers The floating-point registers are part of the "F" extensions in the RISC-V architecture. The registers are a fixed set size of 32-bit. The registers are defined as `f0-31`. Some floating-point registers are callee-saved. However, in this thesis, only `f0` register is used. Therefore, the remaining register will not be discussed. The `f0` register is caller-saved.

Vector Registers The vector registers are part of the "V" extensions in the RISC-V architecture. This extension adds thirty-two vector registers to the RISC-V architecture, defined as `v0-v31`. The vector registers have a fixed set of bits, denoted as `VLEN`. In this thesis, the vector register width (`VLEN`) is 256-bits. Every time vectors are used in RISC-V, the vector configuration must be set using the *Vector Configuration Setting*. In this thesis, we will only discuss the vector configurations presented in Listing 7.

```
1 vsetivli rd, imm, vsew, vmul
```

Listing 7: Vector Configuration Setting instruction for the "V" vector extension in RISC-V.

The `vsetivli` has many configurations and arguments, we will only discuss those relevant to this thesis. The `imm` value indicates the desired number of elements processed. The **vsew**, or Vector

Selected Element Width, defines the size of the element that will be processed. For example, `e32` indicates 32-bit elements. The `vmul` is the vector multiplier. Vector registers can be grouped and one instruction can process multiple vector registers. For example, `m1` means each vector corresponds to one full vector register (256-bits) and no vector register grouping is used. Based on these setting, the destination register `rd` will hold the total number of elements that can be processed. If this value is set to `x0` (zero register), the result will not be stored.

3.3.2 Implemented Assembly Instructions

All the instruction encodings in the following paragraphs are described in the RISC-V reference manual [WA19].

`flw` - Single-precision Load The `flw` instruction load a 32-bit single-precision floating point from memory to the floating-point destination register `rd`. The encoding of the `flw` instruction is shown in Listing 8. The `rs1` is a general-purpose register holding the memory address. Optionally, an `offset` can be provided, which will be added to the memory address contained `rs1`.

```
1 flw  rd, offset(rs1)
```

Listing 8: Encoding for the flw load instruction in the "F" extension on the RISC-V architecture.

`fsw` - Single-precision Store The `fsw` instruction stores a 32-bit single-precision floating point from the floating-point destination register `rd` to memory address stored on general-purpose register `rs1`. The encoding of the `fsw` instruction is shown in Listing 9. Similarly to `flw`, an `offset` can be provided, which will be added to the memory address contained `rs1`.

```
1 fsw  rd, offset(rs1)
```

Listing 9: Encoding for the fsw store instruction in the "F" extension on RISC-V architecture.

`vle32.v` - Vector Load The instruction encoding for the `vle32.v` vector instruction is shown in Listing 10. The `vle32.v` instruction loads a 256-bit vector from memory to the vector register encoded in `vd`. The general-purpose register `rs1` hold the memory address. An offset can be added to the `rs1` register. However, in this thesis, we will not use this because immediate offset value is too limited for practical use, and therefore we will not discuss this encoding. The number thirty-two in the `vle32.v` encoding specifies that 32-bit elements (single-precision floating points) are read.

```
1 vle32.v vd, (rs1)
```

Listing 10: Encoding for the vle32.v store instruction in the "V" extension on the RISC-V architecture.

`vse32.v` - Vector Store The instruction encoding for the `vse32.v` vector instruction is shown in Listing 11. The `vle32.v` instruction stores a 256-bit vector from memory to the vector register encoded in `vd`. All encoding components in this instruction share the same definition as the `vle32.v` instruction.

```
vse32.v vd, (rs1)
```

Listing 11: Encoding for the vse32.v store instruction in the "V" extension on the RISC-V architecture.

3.4 Multi-striding

In this section, we will discuss the transformation of memory access patterns in the compute kernels, defined as multi-striding. The paper by Blom et. al. [BRN25], transformed a single-strided memory access pattern into multiple contiguous sequences of memory addresses, in other words, strides. We hope that the hardware stride prefetcher detects these strides at multiple positions in our memory-bound kernel, resulting in more prefetching of data from memory into the cache, expecting an increase in cache hits, and therefore memory throughput.

An example of multi-striding is shown in Figure 1, representing the start of two stride unrolls. In this example, we sequentially access array elements starting at index *0* and at index $\frac{N}{2}$. In the case of four strides, we start at index *0*, $\frac{N}{4}$, $\frac{N}{2}$, and $\frac{3}{4}N$. This same concept can also be applied to other powers of two, such as eight strides.

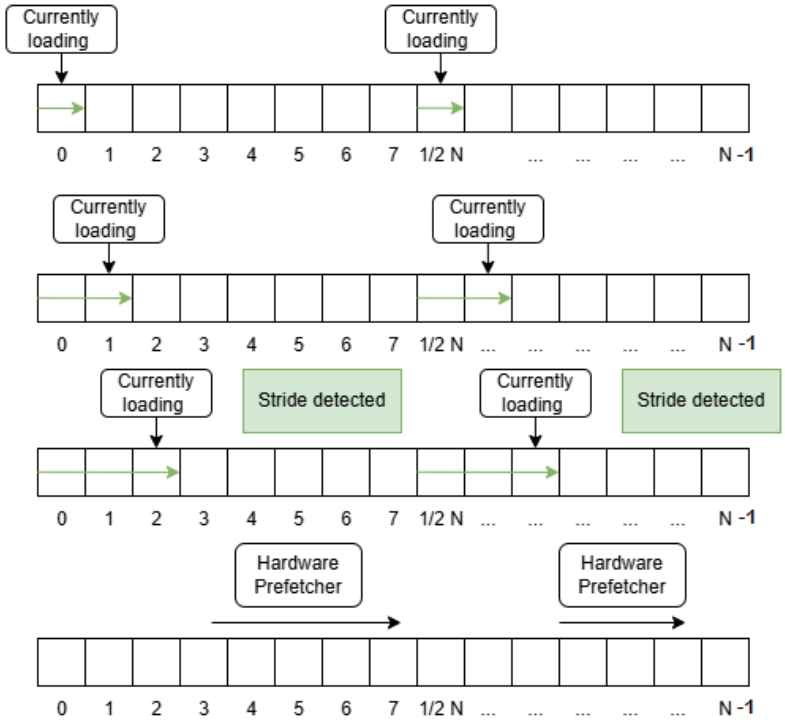

Figure 1: A schematic view of the memory access patterns sequentially accessing array elements at two strides.

3.4.1 Stride-unrolls and Portion-unrolls

The stride-unroll number indicates the number of distinct starting positions for memory accesses. The portion-unroll number indicates the number of memory accesses executed within this stride-unroll. To illustrate this concept, we will present one of the multi-striding implementations presented in the paper by Blom et. al. [BRN25]. A total of 32 loop bodies is unrolled in various ways, as defined by the striding configuration, e.g., (1, 32), (2, 16), ..., (32, 1). Keeping the number of total unrolls as an invariant also causes the number of executed instructions (including branches) and code size to remain equal, eliminating these as sources of variations in the measurements. While this approach is able to demonstrate the presence of effects originating from multi-striding, ongoing research showed that a minimum number of portion unrolls is required to trigger hardware prefetchers to establish and enforce a pattern. In addition, the configuration space of this setup is very limited and allows only very few specific configurations to be evaluated. Therefore, in this thesis, we will diverge from this approach and explore a larger, more complete set of configurations, while being aware of the consequences given varying code sizes and loop overhead. We hypothesize these should not have a significant impact on detecting the effects of multi-striding, while on the other hand, they become slightly noticeable.

3.5 Address Collisions

Blom et al. [BRN25] describe the occurrence of concurrent memory accesses competing for the same cache sets. Specifically, caches are divided into cache sets, each containing a fixed number of cache lines, corresponding to the “way” or “associativity” of the cache. Each memory address maps to a specific cache set, and thus, when multiple addresses that map to the same set are in use, they may conflict when this number exceeds the set size. The authors reason that, for example, with an 8-way associative cache where each cache set contains 8 lines, a 10-strided configuration, with specifically aligned memory accesses, will experience a performance penalty due to 10 accesses consistently competing for the same 8 slots of the cache set. We generalize the notion of performance degradation due to conflicting addresses to include all resources and will refer to this as ”address collisions”.

4 Methodology

In this section, we discuss three main topics. First, we give an abstract description of the micro-kernels implemented to evaluate the effectiveness of multi-striding. Second, we demonstrate how memory access in the multi-strided memory patterns is produced. Third, we give a brief explanation of how multi-striding is implemented on the x86_64 architecture by Blom et. al. [BRN25], and why this setup cannot be achieved on the ARM and RISC-V architectures.

4.1 Micro-kernels

We implemented two micro-kernels, and these micro-kernels will be tested on the ARM and RISC-V architectures. One kernel for load accesses, and one kernel for store accesses, in this thesis referred to as the `readKernel` and `writeKernel`, respectively. The objective of the micro-kernels is to measure the throughput of memory accesses among striding configurations.

4.2 Addressing mode

Memory access patterns typically consist of a single stride. We transform these memory access patterns along multiple strides. Stride unroll zero corresponds to the start of the base of the memory address. The remaining stride unrolls represents displacements from the base memory address, each beginning at its own stride. The Formula (1) is used to determine the memory address of each stride unroll. The array size represented in bytes is denoted as B. This value is divided by the total number of stride unrolls n. By multiplying this value by the stride unroll index i, we obtain the offset corresponding to Stride_i. Adding the base memory address B results in the memory address for Stride_i.

$$\begin{aligned} A &= \text{Array Size In Bytes} \\ B &= \text{Base Memory Address} \\ \forall_{0 \leq i < n}\ (\text{Stride}_i) &= \frac{A \cdot i}{n} + B \end{aligned} \tag{1}$$

Each stride-unroll consists of one or more portion-unrolls. Portion-unroll values correspond to the number of memory accesses performed within a single loop. Each extra portion unroll produces n additional memory accesses, with n denoting the number of stride unrolls. All memory access corresponds to a stride unroll and a portion unroll. The formula for calculating the memory addresses for each memory access within the loop is given in formula (2). The memory access address is denoted as `Memory Access Address (i, j)`. The variable i represents the stride unroll number, and j represents the portion unroll number within this stride unroll. Multiplying the memory access size by the portion unroll index yields the displacement within a stride unroll. By adding this value to the memory address for Stride_i, calculated in formula (1), we obtain the memory address for a memory access for a given stride and portion unroll.

$$\begin{aligned} j &= \text{Portion Unroll Index} \\ i &= \text{Stride Unroll Index} \\ \text{Memory Access Address}\ (i, j) &= j \times \text{Memory Access Size}\ + \text{Stride}_i \end{aligned} \tag{2}$$

The example provided below demonstrates how the memory address for a memory access corresponding to a stride unroll i and portion unroll j is calculated.

Example 4.1 *We want to calculate the Memory Access Address (2, 6). The total number of stride-unrolls n is 10. The base address B is 0x200. The array size in bytes A is 400 bytes. The memory access size is 4 bytes. This results in calculation in demonstrated in* (3)*:*

$$\begin{aligned} \text{Stride}_2 &= \frac{400 \times 2}{10} + 0x200 = \text{0x250}\ (80 = \text{0x50 in hexadecimal}) \\ \text{Memory Access Address}(2, 6) &= 6 \times 4 + 0x250 = 0x268\ (24 = \text{0x18 in hecadecimal}) \end{aligned} \tag{3}$$

4.3 x86_64 and CISC-based architectures

The paper by Blom et. al. evaluated the effect of multi-striding on the x86_64 architecture. Each memory access corresponds to portion-unroll, and each stride-unroll corresponds to an additional

concurrent data stream added to the access pattern. These memory accesses within these stride and portion unrolls are represented by an offset. A fragment of the vector read kernel implemented on x86_64 by Blom et. al. is given in Listing 12. The `vmovaps` is an aligned AVX2 vector load. It loads data from memory into vector register $\texttt{ymm}_i$, from a base address in `rax` with a displacement (offset). This example demonstrates two stride unrolls, and four portion unrolls. The offset corresponds to the stride and portion unrolls. This instruction is possible because the x86_64 is based on the CISC (Complex Instruction Set Computer) architecture. Therefore, instructions can exceed 64 bits in size, allowing large immediate values (i.e., 252645120) to be encoded within a single instruction, such as the displacements encoded within the memory access instructions.

```
    # stride 1
    vmovaps   (%rax), %ymm0     # portion 1
    vmovaps   32(%rax), %ymm1   # portion 2
    vmovaps   64(%rax), %ymm2   # portion 3
    vmovaps   96(%rax), %ymm3   # portion 4

    # stride 2
    vmovaps   252645120(%rax), %ymm4    # portion 1
    vmovaps   252645152(%rax), %ymm5    # portion 2
    vmovaps   252645184(%rax), %ymm6    # portion 3
    vmovaps   252645216(%rax), %ymm7    # portion 4
```

Listing 12: A fragment of the vector read kernel implemented on x86_64 architecture, implemented by Blom et. al. `vmovaps` is an AVX2 vector load instruction, loading data into vector register $\texttt{ymm}_i$.

Architectures based on the RISC design (e.g., ARM and RISC-V) are constrained by a fixed instruction size, defined by the architecture. Therefore, an instruction consists of either 32-bits or 64-bits. The displacement required can not be encoded within a single instruction. Therefore, we are limited in applying this addressing mode. We need to utilize a different addressing mode to implement similar kernels for evaluating multi-striding on the ARM and RISC-V micro-architectures.

5 Approach

In this section, we will describe our approach for implementing multi-striding on the ARM and RISC-V micro-architecture. We generate the assembly files using Python scripts. The evaluation and validation of these assembly files consist of several steps. First, the kernels operate on arrays in memory. To reduce address collision and ensure that each striding configuration fits, we reshape the array size. Next, we discuss the code generation. This includes the scripts that generate the assembly files, how to load large immediate values into the registers, and how the addressing modes are implemented on the ARM and RISC-V architectures. Last, we describe the post-execution steps, the calculation of throughput, and the validation of the micro-kernels. All these steps are implemented within a framework. This framework is open source and available on GitHub [12].

[12] Multi-striding Framework for the ARM Raspberry Pi 5 and RISC-V Banana Pi BPI-F3, https://github.com/steffanradojevic/MFRB

5.1 Reshaping Input Size

Before array initialization, the dimensions of the input sizes are determined. We aim to reduce address collisions by multiplying the selected array size by $\frac{16}{17}$. By doing so, we aim to avoid cache lines being mapped to the same cache set. Additionally, to not be left with remainders of our array that need to be processed separately, we ensure the dimensions in the directions of the stride and portion unrolls are multiples of their respective step sizes. In Formula (4), we demonstrate how we fit an input size to a striding configuration. First, we determine the memory access size in bytes and for which striding configuration we have to reshape the array size. Second, we calculate the number of elements processed within one loop. Finally, we establish the new array size by performing floor division of the old array size by E_{loop}, and then multiplying the result by E_{loop}.

$$\begin{aligned} \mathrm{M}_{bytes} &= \text{Memory Access Size in Bytes} \\ \mathrm{S} &= \text{Stride Unroll} \\ \mathrm{P} &= \text{Portion Unroll} \\ \mathrm{E}_{loop} &= \mathrm{M}_{bytes} \times \mathrm{S} \times \mathrm{P} \\ \text{New Array Size} &= \left\lfloor \frac{\text{Old Array Size}}{\mathrm{E}_{loop}} \right\rfloor \times \mathrm{E}_{loop} \end{aligned} \tag{4}$$

5.2 Code generation

The assembly codes are generated using Python scripts. These scripts create classes where instance variables can be configured to define our parameter space. Our parameter space includes the striding configuration, vector register length (in bits), selected element width (in bits), and the type of micro-kernel. Within the generated assembly kernels, we initialize the assembly prologue, the loop body, and the function's return. In the following sections, we will discuss these components in detail.

5.2.1 Loading Large Immediate Values into Registers

In the generation of assembly files, we must work with large immediate values (e.g., storing the loop bound). In Section 4.3, we explained that the ARM and RISC-V architectures are constrained by a fixed instruction size, hence why we cannot load large immediate values. Thus, a method is required for both architectures to load these large values. This section provides a brief explanation of how these instructions are generated for both architectures.

ARM In Section 3.2.2, we briefly discussed the two instructions `MOVZ` and `MOVK`. These two instructions are utilized to load large immediate values in the ARM architecture. Using bit masks, we extract the 16-bits segments of each equally divided part of the immediate value. The first 16-bit segment is loaded using the `MOVZ` instruction, ensuring all other bits are set to zero. The other 16-bit segments are loaded using the `MOVK` instructions, along with their appropriate left shift. An example of loading a large offset in the ARM architecture using this method is given in Listing 13. In this example, we want to load the immediate value 1768515840. The bit representation of this value is given on the right-hand side of the first row. The `MOVZ` loads in the first sixteen bits of the immediate value, while setting all other bits to zero. The second instruction, `MOVK`, loads bits

[32:16] of the immediate value into the appropriate position. The remaining two instructions have no effect.

```
// immediate value = 1768515840                    [0110100101101001 0110100100000000]
    movz  x14, #26880                               [                 0110100100000000]
    movk  x14, #26985, LSL #16                      [0110100101101001                  ]
    movk  x14, #0, LSL #32
    movk  x14, #0, LSL #48
```

Listing 13: Instruction sequence for loading 64-bit immediate values using four instructions on the ARM architecture.

RISC-V For loading large immediate values, we use two instructions; `addi`, loading an immediate value into a general-purpose register, and `lui`, loading an immediate value into bits [32:12]. By using bit-masks, we split the 32-bit immediate value into two segments: the upper 20 bits and the lower 12 bits. The upper 20-bits are loaded using the `lui` instruction. The lower 12-bits are loaded using the `addi` instruction. Since the `addi` instruction adds a 12-bit sign-extended immediate value, any large immediate value where the twelfth bit is set must be encoded in the `addi` instruction as a negative number to ensure that the twelfth bit remains set in the register. An example loading an immediate value using this method in RISC-V is given in Listing 14. In this example, we want to load the immediate value 1768515840 into register `t4`. Using bitmasks, we split the value into two segments. The first instruction `addi` loads one of the segments, representing bits [11:0], into the appropriate position. The second instruction `lui` loads the other segment, representing bits [32:12], into the correct position.

```
# immediate value = 1768515840                    [01101001011010010111 100100000000]
    addi t4, x0, -1792                              [                     100100000000]
    lui t4, 431767                                  [01101001011010010111             ]
```

Listing 14: Instruction sequence for loading a 32-bit immediate value using two instructions on the RISC-V architecture.

5.2.2 Assembly Prologue Generation

Callee-saved registers To maximize register availability, the callee-saved registers are stored on the stack at the beginning of the generated assembly files. Both architectures and individual processors differ in which set of registers must be saved on the stack. The ARM processor allows us to freely use 31 registers when storing all the callee-saved registers onto the stack. For the RISC-V register, we have a total of 30 available registers when storing all callee-saved registers onto the stack.

Loop Bound We store the loop counter in a register, which we increment and compare to the value of the loop bound. Thus, one register is reserved for the loop bound, and one register is reserved for the loop counter. To generate the immediate value, we use formula (5). To calculate

the loop bound value, we divide the array size in elements by the array elements accessed within each loop.

$$\text{Loop Bound Value} = \frac{\text{Array Size}}{\text{Elements Accessed per Loop}} \tag{5}$$

Loading stride memory addresses Besides placing callee-saved registers on the stack, and storing the loop bound and loop increment into the designated registers, we will also store all the generated stride memory addresses into registers.

5.2.3 Loop Design for Strided Memory Access

The loop consists of two components: memory accesses, which correspond to the striding configuration dimensions, and the update of the loop counter and loop bound. In the following paragraphs, we will discuss how the addressing modes are implemented in the memory access instructions for ARM and RISC-V, respectively.

Generating Memory Accesses Listing 15 illustrates the starting points of a generated loop in ARM using scalar loads, without portion-unrolls. This loop consists of two strides, therefore two registers (`x3` and `x4`) are reserved to hold the memory address of the starting point of the stride. At the start of execution, the stride memory address `x3` contains the start address of the allocated data. Register `x4` contains the memory address at exactly half of the allocated data. The following two paragraphs describes the generation of portion-unrolls within these strides for both architectures.

```
1 .Loop:
2     // stride 1
3     ldr s0,[x3] // portion 1
4
5     // stride 2
6     ldr s4,[x4] // portion 1
```

Listing 15: Two stride-unrolls using the two registers that hold the stride memory addresses on the ARM architecture. Additionally, these memory accesses correspond to the first portion-unroll.

ARM For all memory accesses on the ARM architecture, a post-indexed encoding exists, which we will utilize to advance to the next portion-unroll. The immediate value of the post-indexed encoding corresponds to the size of the memory access. Because we use post-indexed encoding, the registers holding the starting point of the stride are automatically updated, and by the end of the portion-unroll, this register correctly points to the next stride memory address. An example of two stride-unrolls, four portion-unrolls, and a memory access of 4 bytes is given in Listing 16. This example consists of two strides. In each portion unroll (memory access), we increment the register holding the memory address of the stride by four, thus the register holds the memory address of the next portion unroll.

```
    // stride 1
    ldr s0,[x3], #4 // portion 1
    ldr s1,[x3], #4 // portion 2
    ldr s2,[x3], #4 // portion 3
    ldr s3,[x3], #4 // portion 4

    // stride 2
    ldr s4,[x4], #4 // portion 1
    ldr s5,[x4], #4 // portion 2
    ldr s6,[x4], #4 // portion 3
    ldr s7,[x4], #4 // portion 4
```

Listing 16: A generated loop using post-indexed encoding for two stride-unrolls, four portion-unrolls and a memory access size of 4 bytes on the ARM architecture.

RISC-V For the scalar memory accesses in the RISC-V architecture, no instruction encoding exists for incrementing the register within a single instruction. Therefore, we use an offset encoded within a single instruction, which is added to the base registers. An example for scalar stores using two stride-unroll, four portion-unroll, with a memory access size of 4 bytes is illustrated in Listing 17. This example consists of two strides. In each memory access, we add an offset representing the portion unroll to the stride memory address. This way, we access the correct memory in our allocated data.

```
    # stride 1
    flw f0, 0(x4)  # portion 1
    flw f0, 4(x4)  # portion 2
    flw f0, 8(x4)  # portion 3
    flw f0, 12(x4) # portion 4

    # stride 2
    flw f0, 0(x5)  # portion 1
    flw f0, 4(x5)  # portion 2
    flw f0, 8(x5)  # portion 3
    flw f0, 12(x5) # portion 4

    addi x4, x4, 16
    addi x5, x5, 16
```

Listing 17: A generated loop using base address + offset, for two stride-unrolls, four portion-unrolls and a memory access size of 4 bytes on the RISC-V architecture

Since the registers holding the stride memory addresses are not incremented directly within the instruction, we need to increment the register at the end of the loop. We use the following formula for incrementing the registers for scalar memory accesses in RISC-V:

$$\text{New Value Register} = (\text{Old Value Register}) + ((\text{Total Portion-unrolls}) \times (\text{Memory Access Size}))$$

The vector memory accesses in RISC-V do not support an offset encoding that can add or increment a register within a single instruction. Therefore, to implement vector memory accesses, the registers holding the stride memory address are incremented after each memory access. By adding an extra instruction to increment the register after each memory access, we eliminate the need to increment the registers at the end of the loop. An example for a memory access of 32 bytes, two stride-unrolls, and two-portion unrolls is shown in Listing 18. In this example, the two stride memory addresses are in `x4` and `x5`. Both registers are incremented by a single instruction after each memory access. By using this implementation, we do not have to increment the registers after all memory accesses have been executed.

```
    # stride 1
    vse32.v v0, (x4)  # portion 1
    addi x4, x4, 32
    vse32.v v0, (x4)  # portion 2
    addi x4, x4, 32

    # stride 2
    vse32.v v0, (x5)  # portion 1
    addi x5, x5, 32
    vse32.v v0, (x5)  # portion 2
    addi x5, x5, 32
```

Listing 18: A generated loop using an extra instruction to advance to the next portion, for two stride-unrolls, four portion-unrolls and a memory access size of 32 bytes on the RISC-V architecture

5.2.4 Maximum Striding Configuration

When abstractly outlining the addressing modes in Section 4.2, we noted that there is a limit to the number of stride-unrolls and portion-unrolls that can be generated with the implementation described above on the ARM and RISC-V architectures. In the next two paragraphs, we will briefly discuss the limits of these configurations.

Stride-unrolls Each stride-unroll memory address is stored in a register. Consequently, the limit on the number of stride-unrolls corresponds to the number of available registers in the architecture. Our implementation requires reserving two registers, one for storing the loop bound, and one for the loop counter. The formula (6) defines the maximum number of stride unrolls utilizing our addressing mode for the ARM and RISC-V architectures.

$$\text{Maximum Number of Stride-unrolls} = \text{Maximum Number of Available Registers} - 2 \tag{6}$$

Portion-unrolls Portion-unrolls do not rely on dedicated registers, but instead make use of encoded offsets, or an extra increment instruction in the case of RISC-V vector memory accesses. Therefore, there is no limit to the maximum number of portion-unrolls, provided that it fits within the allocated array size, alongside the stride-unrolls.

5.3 Throughput and Validation

Throughput Before calling the micro-kernels, a timer is started. Once execution is completed, the elapsed time is used to compute the throughput. The throughput is measured in gigabytes per second (GB/s). Formula (7) describes how the throughput is computed:

$$\text{Throughput (GB/s)} = \frac{\text{Array Size in Bytes}}{\text{Elapsed time in Seconds} \times 10^9} \tag{7}$$

Validation Addressing modes between load and store memory accesses are identical. Therefore, a validation code is implemented for the `writeKernel`, which thus covers both micro-kernels. The `writeKernel` writes the loop counter into the array. After execution of the `writeKernel`, the content of the written array can be verified to ensure that all indices hold the correct values.

6 Experimental Setup

In this section, we discuss the experimental setup for the ARM and RISC-V devices. First, we will explain all the instructions implemented in the micro-kernels. Second, we will define the tested devices and present their hardware and software specifications. Finally, we will discuss our experimental method.

6.1 Micro-kernels

We evaluate two micro-kernels. A `readKernel`, performing load memory accesses, and a `writeKernel`, performing store memory accesses. All memory accesses operate on 32-bit single-precision floating point values, equivalent to four bytes per element. In Table 3 an overview of all the instructions utilized in the `readKernel` and `writeKernel` for the ARM micro-architecture. Descriptions of these instructions are provided in Section 3.2.2. For the `readKernel`, we implemented three instructions. The scalar load instruction `LDR` and the two vector load instructions `LDR` and `LD1`. Subsequently, for the `writeKernel`, we implemented the scalar store instruction `STR` and the two vector instructions `STR` and `ST1`.

Instruction	Micro-Kernel	Instruction Type	Memory Access Size
`LDR`	Load	Scalar	4 bytes
`STR`	Store	Scalar	4 bytes
`LDR`	Load	Vector	16 bytes
`LD1`	Load	Vector	16 bytes
`STR`	Store	Vector	16 bytes
`ST1`	Store	Vector	16 bytes

Table 3: Overview of the instructions implemented in the micro-kernels for evaluating multi-striding on the ARM micro-architecture.

An overview of the implemented instructions utilized in the micro-kernels for the RISC-V micro-architecture is illustrated in Table 4. A description of these instructions is given in Section 3.3.2. Four instructions are implemented. Two instructions for the `readKernel`, the scalar load `flw` and

the vector load `vle32.v` instruction. The `writeKernel` is implemented with a scalar store `fsw` and vector store `vse32.v` instruction.

Instruction	Micro-Kernel	Instruction Type	Memory Access Size
`flw`	Load	Scalar	4 bytes
`fsw`	Store	Scalar	4 bytes
`vle32.v`	Load	Vector	32 bytes
`vse32.v`	Store	Vector	32 bytes

Table 4: Overview of the instructions implemented in the micro-kernels for evaluating multi-striding on the RISC-V micro-architecture.

6.2 Hardware and Software Specifications

In our setup, we use two devices. The device for the ARM micro-architecture is a Raspberry Pi 5, equipped with a Cortex-A76 (ARM) processor. The theoretical maximum bandwidth is 17.1 GB/s. Each core has a 128KB L1 cache, 512KB L2 cached, and a shared 2MB L3 cache. The Raspberry Pi 5 supports vector operations via the Advanced SIMD (NEON) extension. The RISC-V micro-architecture is a Banana Pi BPI-F3 equipped with a Spacemit(R) X60 (RISC-V) processor. The theoretical maximum bandwidth is 10.6 GB/s. Each core has a 64KB L1 cache, and a 1MB shared L2 cache. The Banana Pi BPI-F3 supports vector operations via the RVV 1.0 standard extension (i.e., "V" Extension). An overview of the hardware and architecture specifications of these devices is given in Table 6.2.

Table 5: Hardware and architecture specifications for the ARM and RISC-V device.

Component	ARM[1]	RISC-V[2]
Device Name	Raspberry Pi 5 Model B Rev 1.0	Banana Pi BPI-F3 SpacemiT K1
Architecture	AArch64	RV64
CPU Model	Cortex-A76	SpacemiT® X60
CPU max MHz	2400MHz	1600MHz
RAM Size	$\sim$ 8 GB	$\sim$ 16 GB
Memory Bandwidth	17.1 GB/s	10.6 GB/s
Cache Size L1, L2, L3	128KB, 512KB, 2MB (shared)	64 KB, 1MB (shared)
Cache Line Size	64 B	64 B
SIMD Support	Advanced SIMD (Neon)	RVV 1.0 standard
Vector Length	128-bit	256-bit

[1] ARM Hardware Specifications, https://www.cpu-monkey.com/en/compare_cpu-raspberry_pi_5_b_broadcom_bcm2712-vs-raspberry_pi_4_b_broadcom_bcm2711
[2] RISC-V Hardware Specifications, https://docs.banana-pi.org/en/BPI-F3/SpacemiT_K1

The Raspberry Pi 5 operates under the Debian GNU 12 operating system. To enable vector operations, the target architecture flag must be specified. For the Raspberry Pi 5, we used the compiler flag `-march=armv8.2-a`. To compile the C++ programs, we use the g++ version 12.2.0. The assembly file generation is done in Python, and we use Python version 3.11.2. The Banana Pi BPI-F3 operates under the Bianbu 2.2 operating system. To enable vector operations, we use

the compiler flag `-march=rv64gcv zba zbb zbs`. We use g++ version 13.2.0 and Python version 3.12.3. For both micro-architectures, we applied the optimization flag `-O0`, in order to enforce that the generated assembly micro-kernel remains unchanged. Additionally, we will use the `vsetivli x0, 8, e32, m1` configuration for the RISC-V vector kernels. An overview of all the software specifications for the Raspberry Pi 5 and Banana Pi BPI-F3 is shown in Table 6.

Table 6: Software specifications for the ARM and RISC-V device.

Software	**ARM**	**RISC-V**
Operating System	Debian GNU/Linux 12 (bookworm)	Bianbu 2.2
Target Architecture Flag	-march=armv8.2-a	-march=rv64gcv_zba_zbb_zbs
Compiler Flags	-O0	-O0
g++ Version	g++ (Debian) 12.2.0	g++ (Bianbu) 13.2.0
C++ Version	C++17	C++17
Python Version	Python 3.11.2	Python 3.12.3

6.3 Experimental Method

Our configuration space uses striding configurations consisting of 1 up to and including 20 stride unrolls, and 1 up to and including 32 portion unrolls. Furthermore, we aim to prevent measuring effects from address collisions, as described by Blom et. al. [cite] by choosing allocated memory with sizes that do not line up with powers of two. As our initial experiment showed signs of outliers, we implemented an additional countermeasure, where we use ten different sizes for our measurements in our configuration space. These sizes are 23529411, 23537182, ..., up to and including 23599350. These sizes are 7771 elements apart from one another, then adjusted to be divisible by the step sizes made by the striding configuration. The base array size is roughly 94.1 MB. This exceeds the total private cache capacity (including the shared cache) of the ARM device by a factor of about 36, and the RISC-V device by a factor of about 88. Each experiment is configured using a striding configuration and array size, which performs 7 function invocations upon each run. From these measurements, the first 2 are discarded as these act as warm-up runs, and we compute the mean throughput for the remaining 5 measurements. The program is invoked 5 times, and for each measurement, an average is obtained. The maximum throughput of these 5 measurements is taken, each representing the throughput of a given configuration.

7 Results

In this section we show and discuss the evaluation of our `readKernel` and `writeKernel` micro-benchmarks on the ARM and RISC-V micro-architectures. The results of all the micro-kernels for both architectures are available on GitHub [13]. In this thesis, all the results are represented using heatmaps, where we find the number of stride unrolls on the x-axes and the number of portion unrolls on the y-axes. Each cell contains the throughput in GB/s, where higher values are better, corresponding to the striding configuration as indicated by the x- and y-axis, and is

[13] Multi-striding Framework for the ARM Raspberry Pi 5 and RISC-V Banana Pi BPI-F3, https://github.com/steffanradojevic/MFRB

additionally colored according to a gradient to represent the magnitude of the throughput for this value. To evaluate if multi-striding is effective, we compare the best performing single-strided baseline configuration to the best performing multi-strided configurations.

7.1 Overview

An overview of the results for the ARM micro-architecture is given in Figure 2, showing the performance for the best performing baseline configuration compared to the best multi-strided configuration. On the y-axis, the throughput achieved by the best performing baseline and multi-strided configurations. Multi-striding configuration outperforms baseline configuration for all micro-kernels, except the vector load kernels. We find speedup factors of 1.51x for the scalar loads, 1.47× for scalar stores, and 1.54× for vector stores.

In Section 7.2, the results of all striding configurations are represented as a heatmap, for all the micro-kernels on the ARM micro-architecture. Portion unroll values do not significantly influence throughput for the scalar store kernel. In contrast to the store micro-kernels, one apparent pattern emerges. A prerequisite for multi-striding to be effective, portion unroll values for the scalar store have to be divisible by sixteen. For vector stores, portion unroll values have to be divisible by four. A key condition is that the total memory access size within a stride unroll equals one cache line. The best multi-strided micro-kernel is the `writeKernel Vectors ST1`. The optimal striding configuration is stride-unroll seven, paired with portion-unroll thirty-two. This configuration yields a throughput of 14.157 GB/s. The theoretical maximum bandwidth is 17.1 GB/s, the achieved performance achieved 83% of this maximum bandwidth.

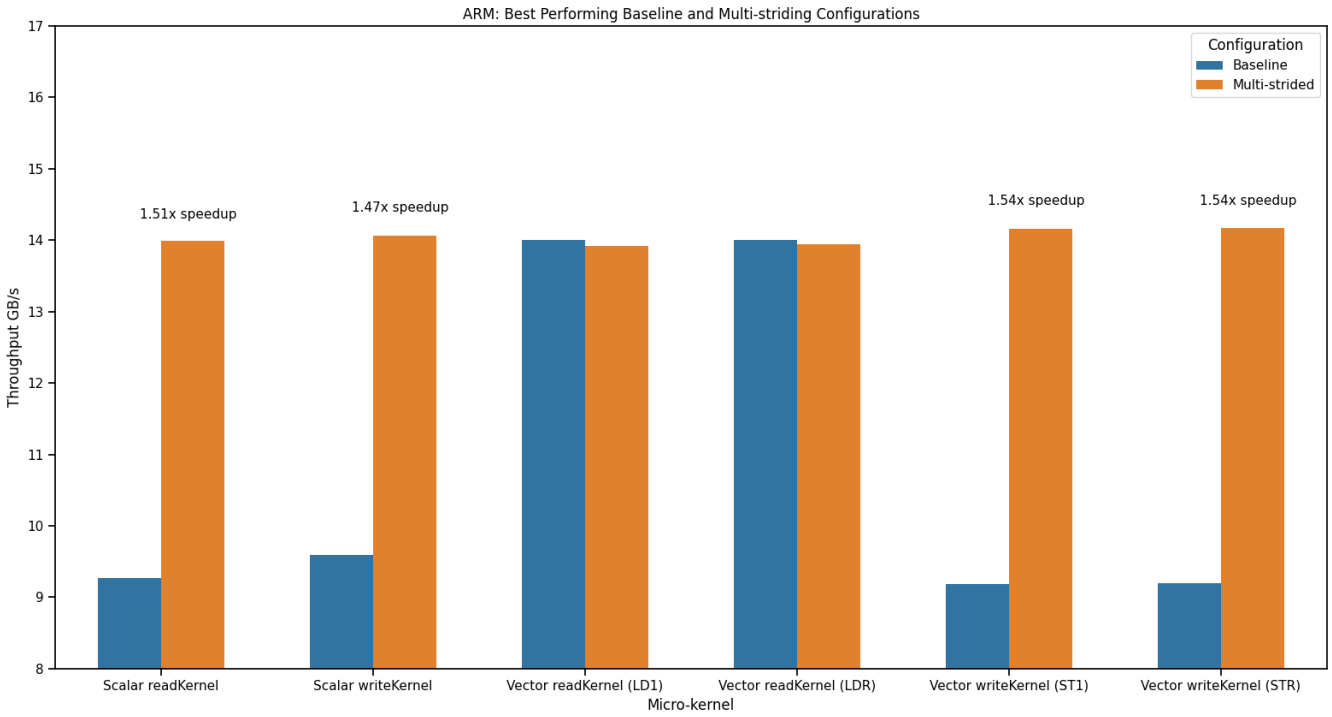

Figure 2: Comparison of the best single-strided baseline configurations to the best multi-strided configuration for all the implemented micro-kernels on the ARM Cortex-A76.

An overview of the results for the RISC-V micro architecture is given in Figure 3. Multi-striding achieved a higher throughput compared to baseline configuration for the scalar store kernel, achieving a speedup of 1.03x. For all other micro-kernels, baseline configuration performed better than multi-strided configurations. In Section 7.3, the results of all striding configurations are represented as a heatmap, for all the micro-kernels on the RISC-V micro-architecture. A similar pattern emerges for the store micro-kernels. A prerequisite for multi-striding to be effective, or to achieve reasonable performance, portion unroll values for the scalar stores have to be divisible by sixteen. For the vector stores, values have to be divisible by two. Therefore, the total memory access size within stride unrolls equals one cache line. The optimal multi-strided configuration for the scalar store kernel is stride-unroll sixteen, paired with portion-unroll thirty-two, achieving a throughput of 4.172 GB/s. For the Banana Pi BPI-F3, the theoretical maximum bandwidth is 10.6 GB/s. Therefore, the achieved performance for this micro-kernel achieved 39% of the maximum bandwidth. One speculation for the significantly lower performance of the scalar micro-kernels compared to the vector micro-kernels, may be attributed to the bottleneck in the front-end, such as the instruction decoder. Scalar operations can access one element per instruction, while a vector operation accesses eight elements per instruction, reducing the total instructions issued for the memory accesses by a factor of eight.

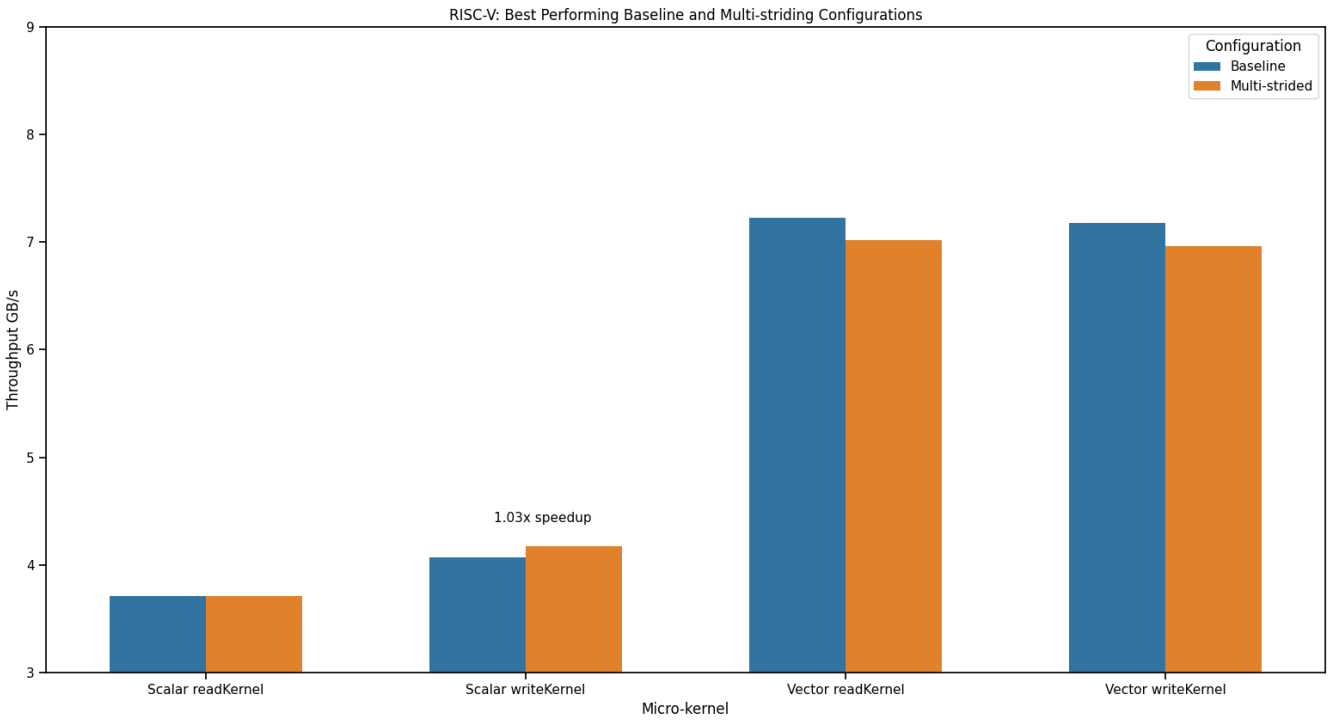

Figure 3: Comparison of the best single-strided baseline configurations to the best multi-strided configuration for all the implemented micro-kernels on the RISC-V SpacemiT K1.

7.2 ARM

ARM readKernel Scalars LDR In Figure 4 we show the results of the readKernel benchmark with scalars and the LDR instruction on the ARM Cortex-A76. Depending on the exact striding and memory size configuration, an array size between 23529411 and 23599350 single-precision floating points is used. Multi-strided configurations up to and including ten strides outperform all single-strided baseline configurations and generally outperform all configurations past ten strides. Based on the best performing single-strided $((1, 19))$, yielding a throughput of 9.271 GB/s, and multi-strided $((3, 3)$ configurations achieving a throughput of 13.996 GB/s, we find a maximum speedup factor of $1.51\times$. This significant improvement in throughput entirely relies on transforming the access pattern to make more efficient use of the available memory bandwidth through better hardware prefetch utilization, which lines up with Blom et. al. [BRN25]. In addition, increasing the number of portion unrolls past thresholds specific to each number of stride unrolls will decrease performance. For example, performance for the 6-strided configuration decreases when eleven or more portion unrolls are used. Also, it is noteworthy that using two, instead of one, portion unrolls or stride unrolls already greatly benefits throughput compared to a single portion or stride unroll. Furthermore, we observe powers-of-two related striding configurations to behave slightly differently. More specifically, for stride unrolls, this results in a decreased throughput, most possibly due to the address collisions discussed in Blom et. al. [BRN25].

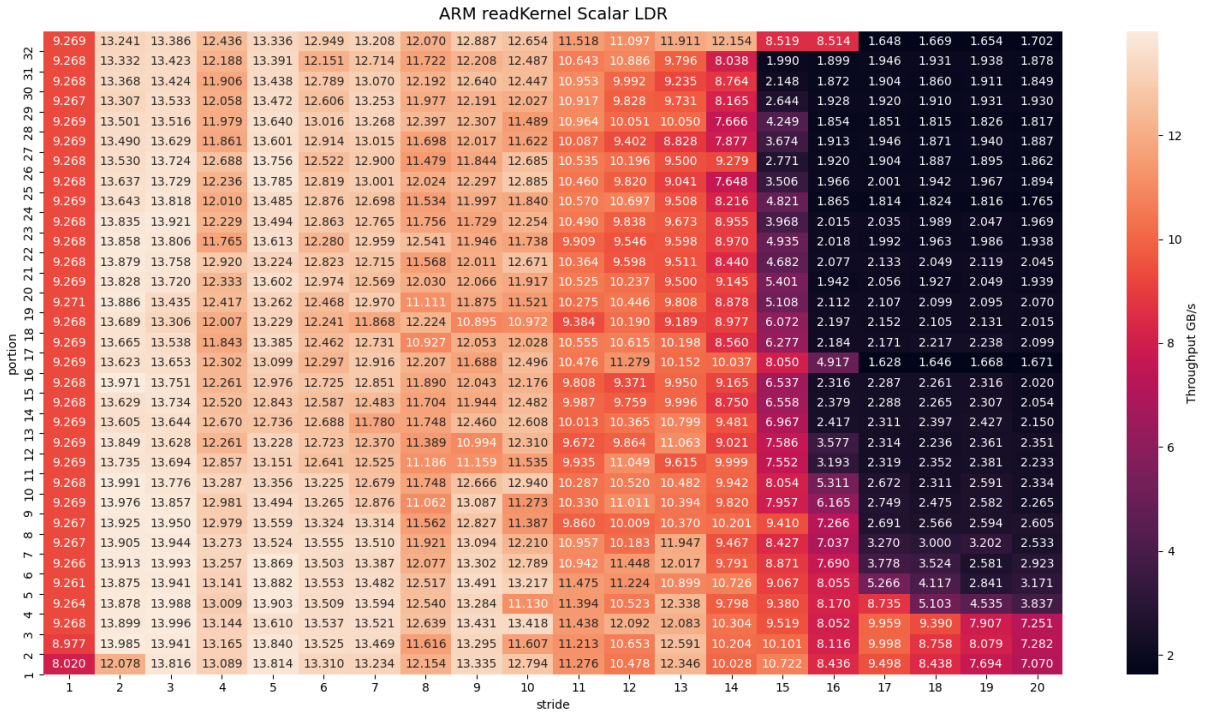

Figure 4: Throughput (GB/s) per striding configuration for `readKernel` using scalar registers with single-precision floating points and instruction `LDR` for the ARM Cortex-A76.

ARM writeKernel Scalars STR In Figure 5 the results of the experiment. Based on the best performing single-strided ((1, 26)), yielding a throughput of 9.596 GB/s, and multi-strided ((6, 16) configurations achieving a throughput of 14.068 GB/s, we find a maximum speedup factor of 1.47×. Portion unrolls divisible by 16 demonstrates an increased throughput, presumably due to filling up until the exact size of a cache line. This increase in throughput is most likely due to the address collisions discussed in Blom et. al. [BRN25]. Portion unroll configurations 16 and 32 correspond to filling up exactly one and two cache lines each iteration, respectively, and therefore perform best. This extends to other configurations that align periodically with the cache line size, where the periodicity of this occurrence determines how much of this benefit is reflected in our results.

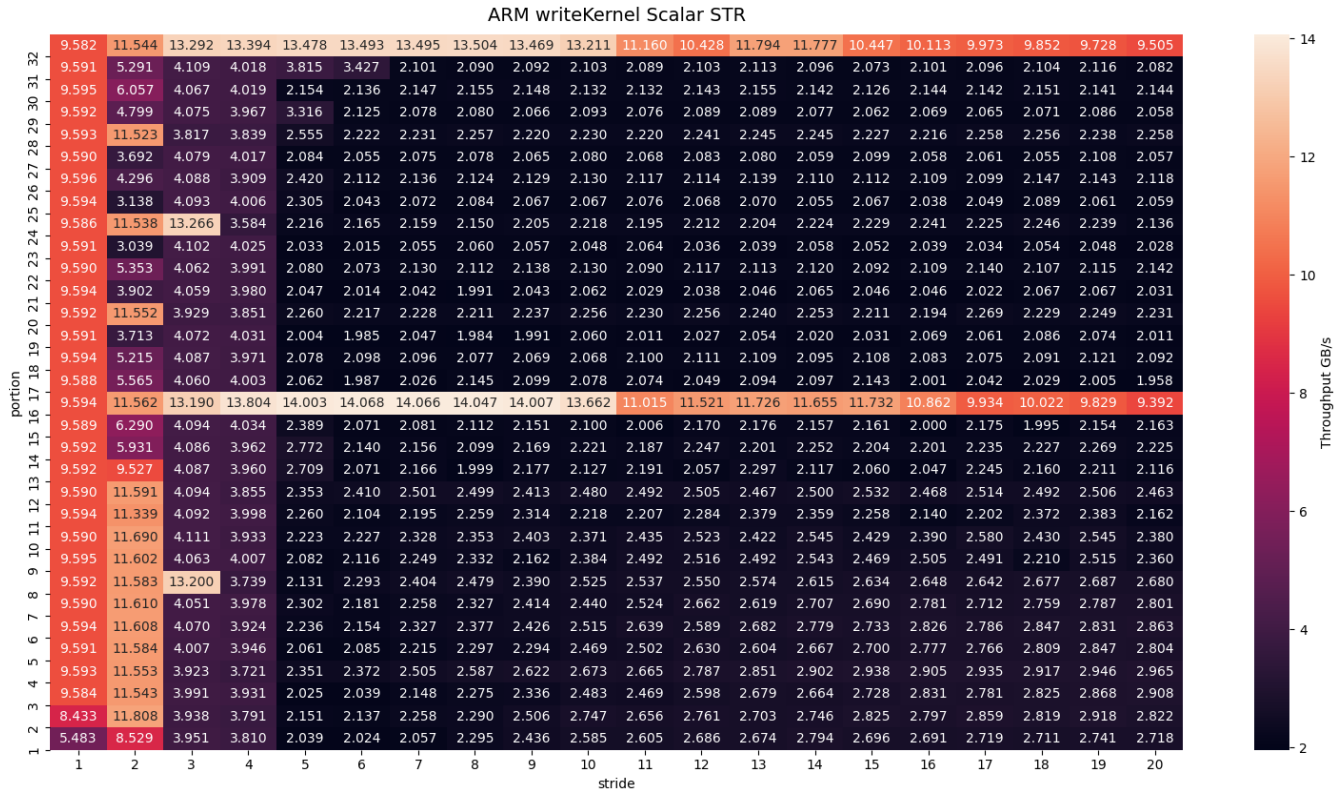

Figure 5: Throughput (GB/s) per striding configuration for `writeKernel` using scalar registers with single-precision floating points and instruction `STR` for the ARM Cortex-A76.

readKernel Vectors LD1 In Figure 6 the results of the experiment. Based on the best performing single-strided ((1, 2)), yielding a throughput of 14.007 GB/s, and multi-strided ((2, 16) configurations achieving a throughput of 13.932 GB/s, we find no multi-strided speedup compared to single-strided baseline.

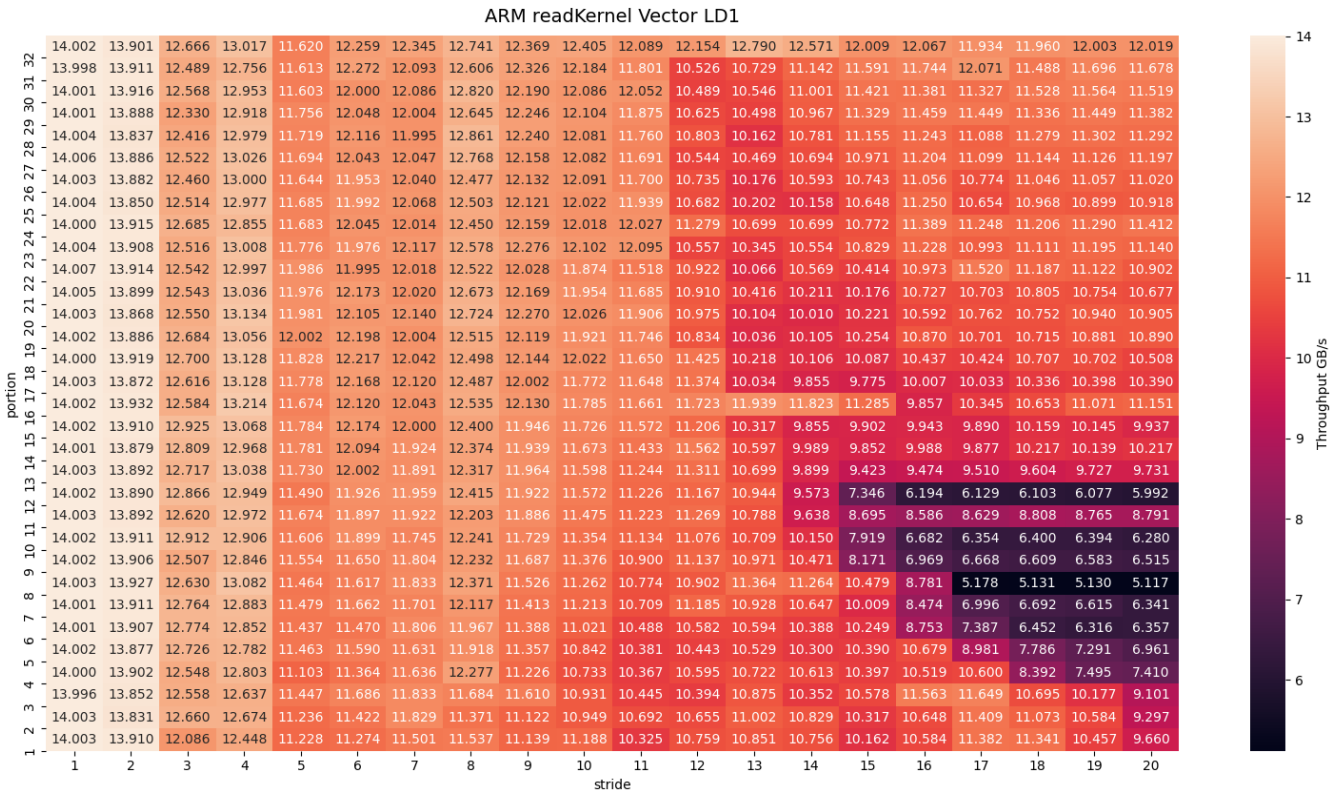

Figure 6: Throughput (GB/s) per striding configuration for `readKernel` using vector registers with four single-precision floating points and instruction `LD1` for the ARM Cortex-A76.

readKernel Vectors LDR In Figure 7 the results of the experiment. Based on the best performing single-strided ((1, 10)), yielding a throughput of 14.008 GB/s, and multi-strided ((2, 30) configurations achieving a throughput of 13.924 GB/s, we find no multi-strided speedup compared to single-strided baseline.

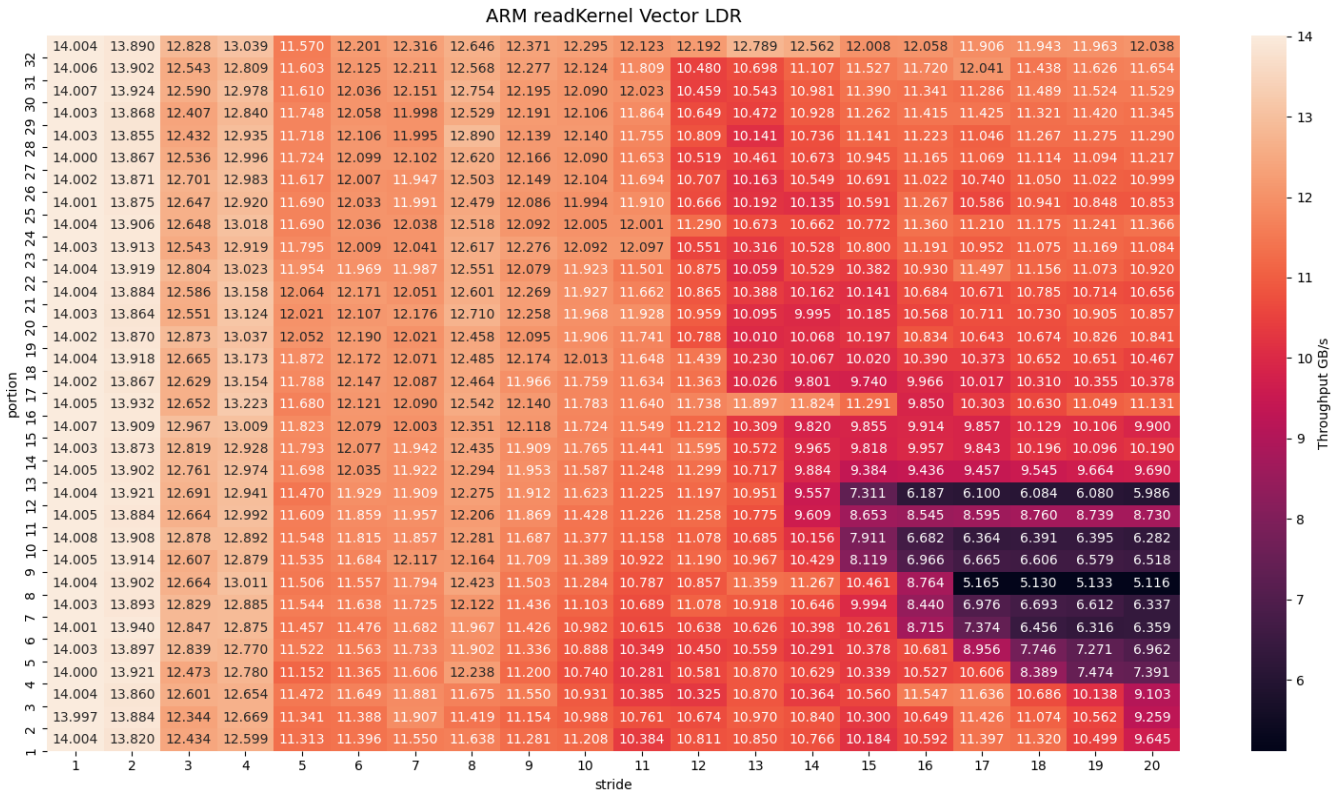

Figure 7: Throughput (GB/s) per striding configuration for `readKernel` using vector registers with four single-precision floating points and instruction `LDR` for the ARM Cortex-A76.

writeKernel Vectors ST1 In Figure 8 the results of the experiment. Based on the best performing single-strided ((1, 30)), yielding a throughput of 9.180 GB/s, and multi-strided ((7, 32) configurations achieving a throughput of 14.157 GB/s, we find a maximum speedup factor of 1.54×. Similarly to all other store micro-kernels, striding configurations must align periodically with the cache line size for multi-striding to be effective.

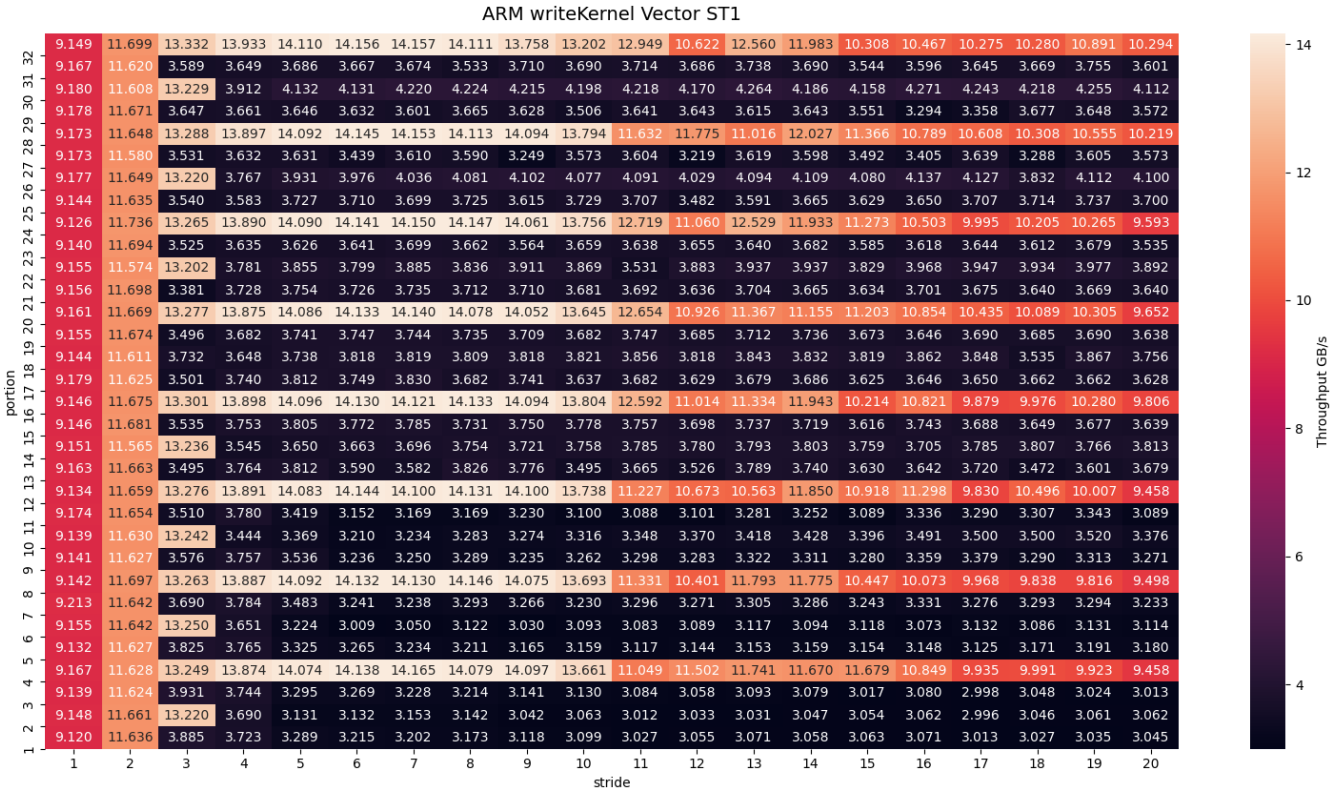

Figure 8: Throughput (GB/s) per striding configuration for `writeKernel` using vector registers with four single-precision floating points and instruction `ST1` for the ARM Cortex-A76.

writeKernel Vectors STR In Figure 9 the results of the experiment. Based on the best performing single-strided ((1, 31)), yielding a throughput of 9.201 GB/s, and multi-strided ((7, 8) configurations achieving a throughput of 14.165 GB/s, we find a maximum speedup factor of 1.54×. Similar to all other store micro-kernels, striding configurations must align periodically with the cache line size for multi-striding to be effective.

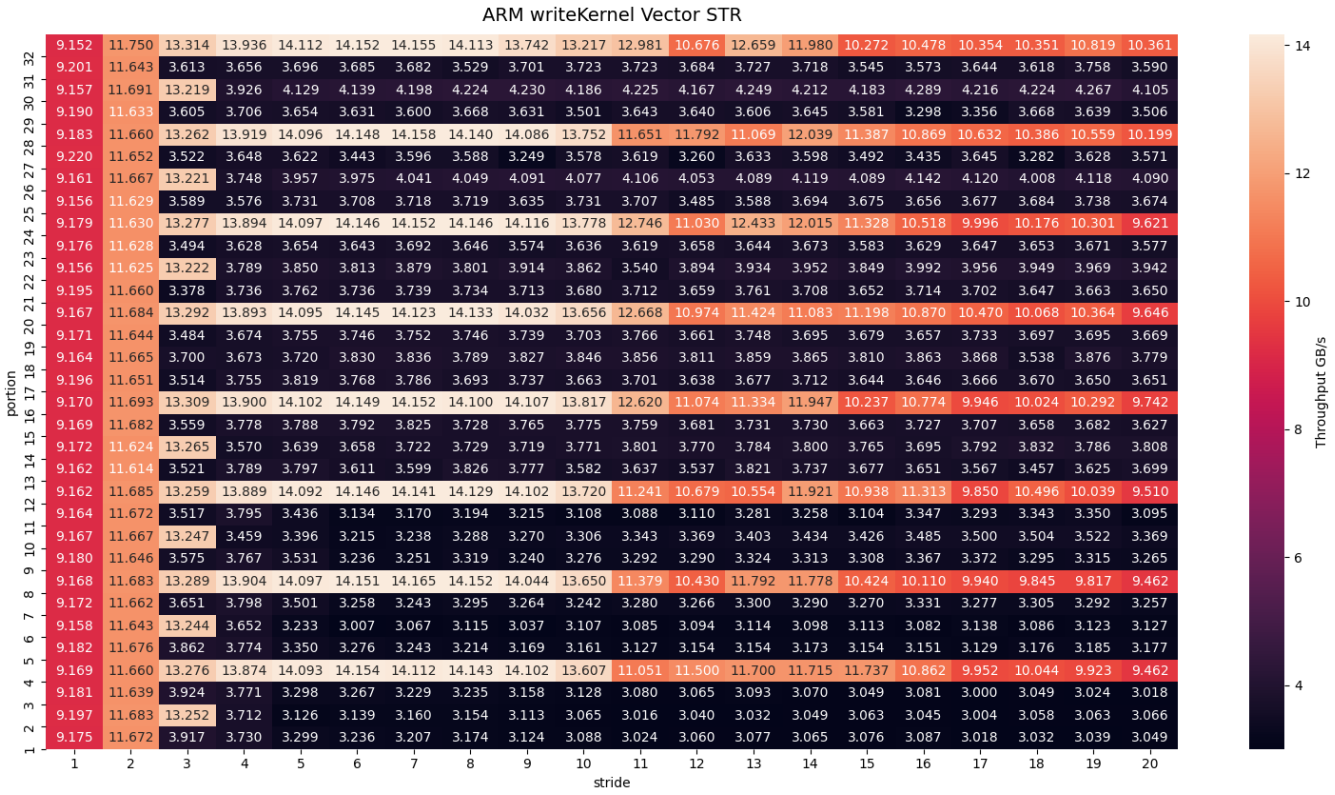

Figure 9: Throughput (GB/s) per striding configuration for `writeKernel` using vector registers with four single-precision floating points and instruction `LDR` for the ARM Cortex-A76.

7.3 RISC-V

In Figure 10 we show the results of the readKernel benchmark with scalars and the flw instruction on the RISC-V SpacemiT K1. Depending on the exact striding and memory size configuration, an array size between 23529411 and 23599350 single-precision floating points is used. Based on the best performing single-strided $((1, 28))$, yielding a throughput of 3.707 GB/s, and multi-strided $((2, 33)$ configurations achieving a throughput of 3.710 GB/s, we find a maximum speedup factor of 1.008×. Low portion-unrolls (i.e., one and two) appear to significantly impact the performance of stride-unroll one, more so than higher neighboring stride-unroll values (e.g., two, three, four). For high stride-unroll values, paired with portion-unroll values divisible by sixteen, the performance is slightly worse than other portion-unroll configurations.

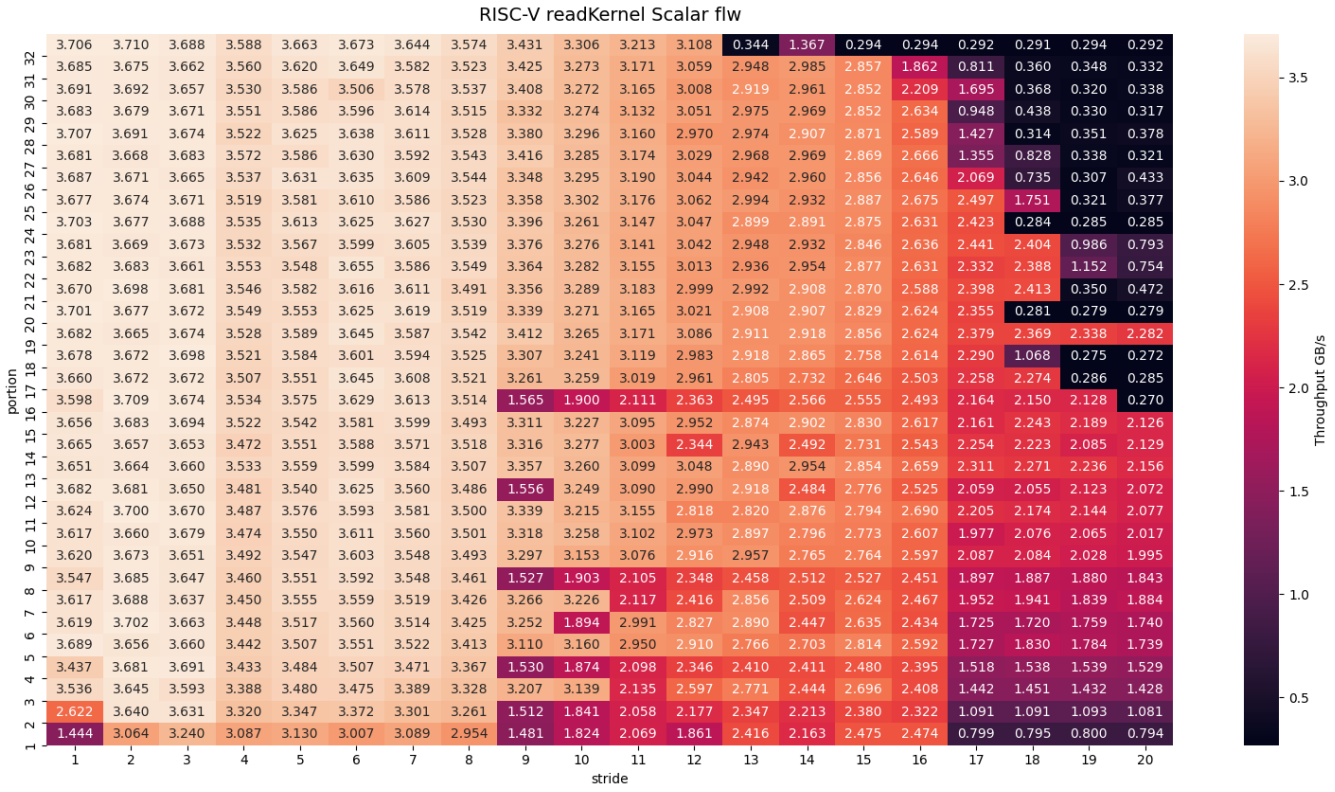

Figure 10: Throughput (GB/s) per striding configuration for `readKernel` using scalar registers with single-precision floating points and instruction `flw` for the RISC-V SpacemiT K1.

writeKernel Scalars In Figure 11 the results of the experiment. Based on the best performing single-strided $((1, 32))$, yielding a throughput of 4.072 GB/s, and multi-strided $((16, 32)$ configurations achieving a throughput of 4.172 GB/s, we find a maximum speedup factor of 1.03×. Similarly to the ARM `writeKernels`, a prerequisite for multi-striding to be effective is that the total memory access size within one stride unroll aligns with the cache line size.

Figure 11: Throughput (GB/s) per striding configuration for `writeKernel` using scalar registers with single-precision floating points and instruction `fsw` for the RISC-V SpacemiT K1.

readKernel Vectors In Figure 12 the results of the experiment. Based on the best performing single-strided ((1, 2)), yielding a throughput of 7.222 GB/s, and multi-strided ((2, 28) configurations achieving a throughput of 7.014 GB/s, we find no multi-strided speedup compared to single-strided baseline.

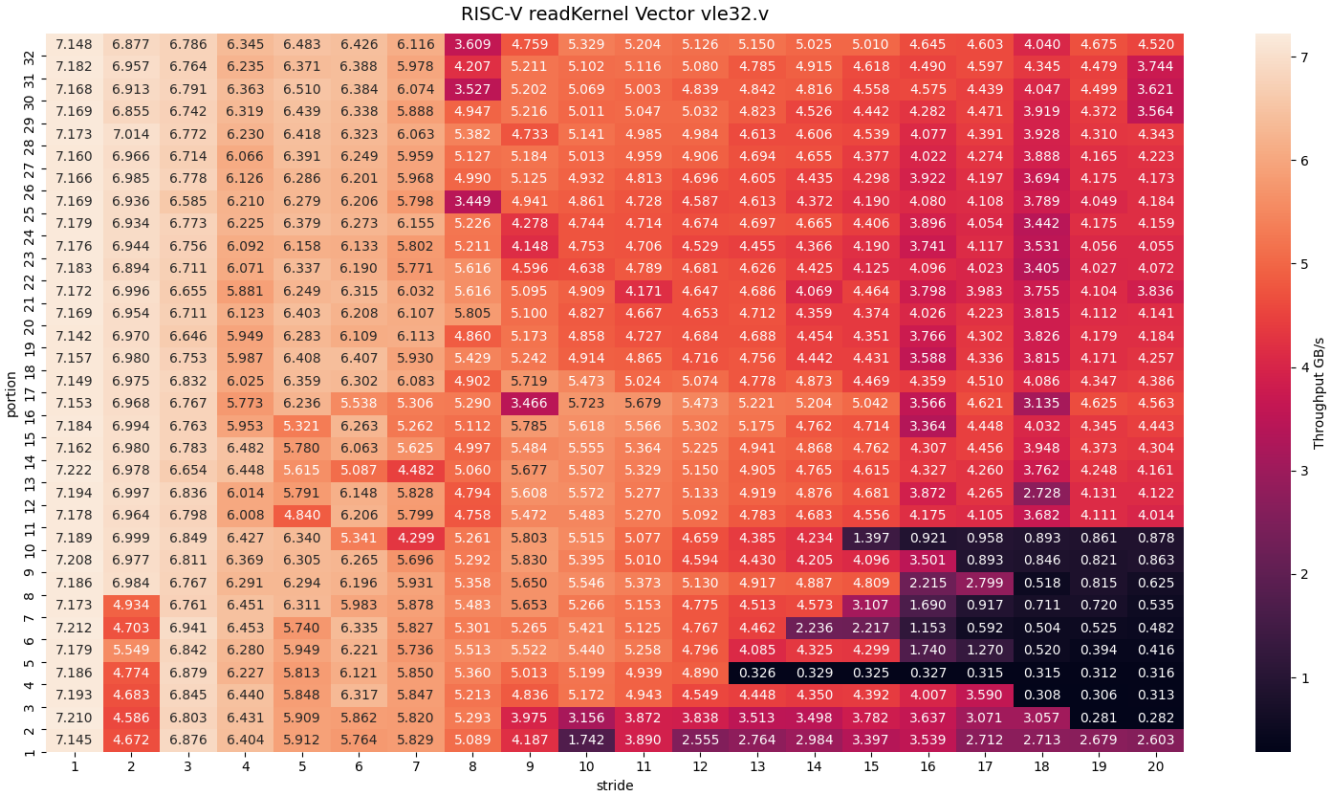

Figure 12: Throughput (GB/s) per striding configuration for `readKernel` using vector registers with eight single-precision floating points and instruction `vse32.v` for the RISC-V SpacemiT K1.

writeKernel Vectors In Figure 13 the results of the experiment. Based on the best performing single-strided ($(1, 12)$), yielding a throughput of 7.180 GB/s, and multi-strided ($(2, 14)$ configurations achieving a throughput of 6.962 GB/s, we find a maximum speedup factor of 1.03×. Similarly to the other store micro-kernels, a prerequisite for multi-striding to be effective is that the total memory access size within one stride unroll aligns with the cache line size.

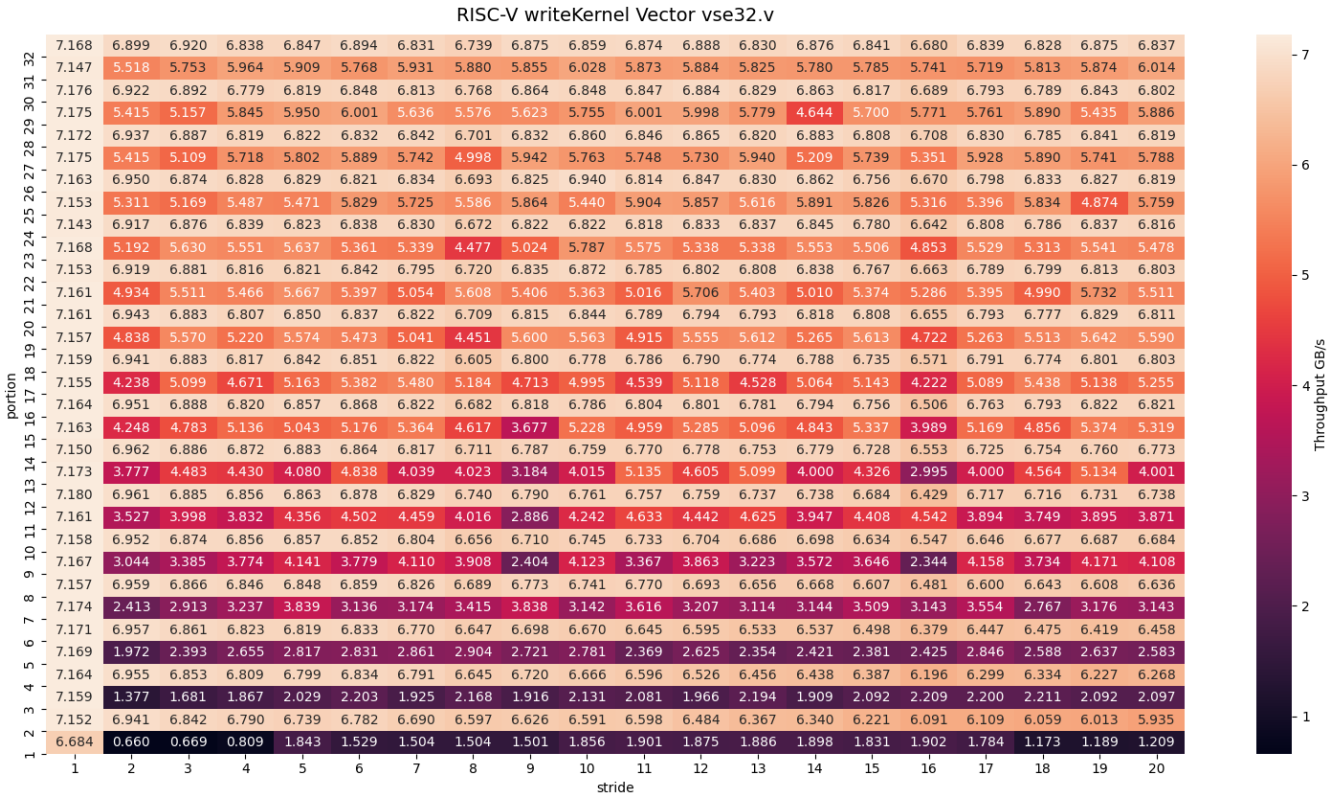

Figure 13: Throughput (GB/s) per striding configuration for `writeKernel` using vector registers with eight single-precision floating points and instruction `vse32.v` for the RISC-V SpacemiT K1.

7.4 Performance Comparison with memset

The Standard C Library function memset[14] writes a constant byte to a memory area. The implemented store kernels in this thesis are equivalent in functionality. Therefore, we will compare the throughput of the optimal multi-strided configuration for the best performing store kernel on the ARM Cortex-A76 with that of memset. As with all other striding configurations, for memset, we take the maximum throughput aggregated over the ten array sizes conducted in the experiments. This results in a throughput of 9.149 GB/s. With the best multi-strided configuration $((7, 8))$ for the vector `writeKernel` STR, we achieve a throughput of 14.165 GB/s, resulting in a speedup factor of 1.55x over memset.

8 Discussion

The experiments are conducted on ten arrays with varying sizes. In Section 6.3, we hypothesized that certain array sizes may lead to address collisions for some striding configurations. This effect occurs on the ARM Cortex-A76. In Appendix A.1, an example of a conducted experiment for the scalar load memory accesses on a single array size for the ARM Cortex-A76. It can be observed that some stride-unrolls perform worse than our baseline configurations, which performed better in

[14]Standard C Library, memset, https://man7.org/linux/man-pages/man3/memset.3.html

the conducted experiments. In Appendix A.2, the worst performing array size is selected for each striding configuration for the scalar `writeKernel` on the ARM Cortex-A76. In this case, the baseline configuration outperforms all the striding configurations. Appendix A.3 shows a heatmap illustrating the difference in throughput between the best and worst performing arrays, in GB/s for the vector `writeKernel` on the ARM Cortex-A76. The single-strided baseline configuration shows almost no differences in throughput between array sizes, indicating that selecting the appropriate array size for reducing address collisions has minimal impact on the performance for these configurations. The multi-strided configurations, which significantly improve performance in the conducted experiments, show large differences in performance between array sizes. We hypothesize that these multi-strided configurations may be more prone to address collisions, resulting in performance degradation. This highlights the importance of selecting varying array sizes to avoid cache collisions and obtain more accurate measurements.

Moreover, the RISC-V SpacemiT does not seem to be affected by varying array sizes. In Appendix A.4, a heatmap of the scalar read micro-kernel, representing the differences in throughput between the best and worst performing array size. Except for a few striding configurations, the differences in throughput across varying array sizes are almost identical.

Additionally, some stride-unroll values appear to achieve the highest performance, these results might be skewed in their favor due to the fact that one of the ten tested array sizes was the smallest. Our initial experiments demonstrated that smaller array sizes achieve higher throughput than larger array sizes. and therefore result in a higher throughput. Therefore, we cannot conclude with certainty which striding configurations achieve the highest throughput, because the throughput shown in the heatmaps may originate from the smallest array size.

9 Conclusion

In this thesis, we evaluated the effectiveness of multi-striding for the Raspberry Pi 5, equipped with an ARM Cortex-A76, and the Banana Pi BPI-F3, equipped with an RISC-V SpacemiT K1. For the ARM Cortex-A76, multi-strided configurations significantly improve throughput compared to single-strided baseline configurations. We find speedup factors of 1.51x for the scalar loads, 1.47× for scalar stores, and 1.54× for vector stores. Additionally, we showed that the vector `writeKernel` achieved a speedup factor of 1.55x over the Standard C Library function memset. Moreover, we demonstrated the importance of input sizes for the ARM Cortex-A76, hypothesizing that address collisions significantly reduce throughput for striding configurations.

For the RISC-V SpacemiT K1, multi-striding for the scalar `writeKernel` achieves a speedup of 1.03x over single-strided baseline configurations. The striding configurations seems not sensitive to varying array sizes, making the selection of input sizes conducted in the experiments less critical. For all the store micro-kernels on both the ARM and RISC-V micro-architectures, the total memory access size within stride unrolls must align with cache line sizes to maximize throughput for striding configurations.

10 Future Work

This thesis evaluated multi-striding on the ARM Cortex A-76 and RISC-V SpacemiT K1, and we are interested in how it affects other ARM and RISC-V micro-architecture, such as the Apple Silicon processors. Furthermore, many embedded processors lack vector processing support, and therefore rely on scalar memory accesses. Multi-striding has been shown to significantly increase performance for scalar load and store memory accesses on the Cortex A-76 micro-architecture. Further research is required to determine how multi-striding for scalar memory accesses affects performance on these embedded processors. Moreover, the results for all the store micro-kernels for both micro-architectures showed that the total memory access size must align with the cache line size. The `readKernel` on the ARM Cortex-A76 is not affected by this. Additional study is necessary to understand the underlying mechanics.

Additionally, another one of our speculations is that some array sizes cause address collisions for certain striding configurations, and therefore reduce performance. Further study is needed to investigate how varying array sizes influence multi-strided performance, and whether our speculation on cache conflicts is responsible. The paper by Sato. et. al. [SE17], implemented a cache-line conflict simulator revealing cache-line conflicts during execution, and demonstrated that cache-line conflict misses can be avoided by padding the array. We also require a profiling tool for identifying address collisions for multi-strided configurations, and generate the appropriate padding for the array sizes in order to reduce address collisions.

Furthermore, we are interested in how the best multi-strided configurations on the ARM Cortex-76 compare against `memcpy`[15] and the `STREAM` [McC07] benchmark. Moreover, we are interested in the performance of multi-striding in vector operations, such as vector-matrix multiplications, and would like to compare this to OpenBLAS[16].

References

[Ale92] Samuel O. Aletan. An overview of risc architecture. In *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing: Technological Challenges of the 1990's*, SAC '92, page 11–20, New York, NY, USA, 1992. Association for Computing Machinery.

[ARM24] ARM Limited. *Arm® Architecture Reference Manual for A-profile architecture*, 2024.

[BRN25] Miguel Blom, Kristian Rietveld, and Rob Nieuwpoort. Multi-strided access patterns to boost hardware prefetching. pages 204–215, 05 2025.

[BVS25] Cyrill Burth, Markus Velten, and Robert Schöne. *Introducing the Arm-Membench Throughput Benchmark*, page 99–112. Springer Nature Switzerland, 2025.

[CDK+15] Russell Clapp, Martin Dimitrov, Karthik Kumar, Vish Viswanathan, and Thomas Willhalm. Quantifying the Performance Impact of Memory Latency and Bandwidth for Big

[15] Standard C Library, memset, https://man7.org/linux/man-pages/man3/memcpy.3.html
[16] openBLAS, An optimized BLAS library, http://www.openmathlib.org/OpenBLAS/

Data Workloads. In *2015 IEEE International Symposium on Workload Characterization*, pages 213–224, 2015.

[GYK+24] Amir Gholami, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W. Mahoney, and Kurt Keutzer. AI and Memory Wall. *IEEE Micro*, 44(3):33–39, 2024.

[HFP+25] Nam Ho, Carlos Falquez, Antoni Portero, Estela Suarez, and Dirk Pleiter. Memory prefetching evaluation of scientific applications on a modern hpc arm-based processor. *IEEE Access*, 13:85898–85926, 2025.

[JTW+19] Adrian Jackson, Andrew Turner, Michèle Weiland, Nick Johnson, Olly Perks, and Mark Parsons. Evaluating the arm ecosystem for high performance computing. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, PASC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[KP02] C. Kozyrakis and D. Patterson. Vector vs. superscalar and vliw architectures for embedded multimedia benchmarks. In *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35). Proceedings.*, pages 283–293, 2002.

[LLT+23] Bangtian Liu, Avery Laird, Wai Hung Tsang, Bardia Mahjour, and Maryam Mehri Dehnavi. Combining run-time checks and compile-time analysis to improve control flow auto-vectorization. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, PACT '22, page 439–450, New York, NY, USA, 2023. Association for Computing Machinery.

[Lon24] Roy Longbottom. Raspberry pi 5 benchmarks and stress tests, 01 2024.

[LZRX25] Mengming Li, Qijun Zhang, Yongqing Ren, and Zhiyao Xie. Integrating Prefetcher Selection with Dynamic Request Allocation Improves Prefetching Efficiency . In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 204–216, Los Alamitos, CA, USA, March 2025. IEEE Computer Society.

[McC07] John D. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, Charlottesville, Virginia, 1991-2007. A continually updated technical report. http://www.cs.virginia.edu/stream/.

[MWR25] Fabian Mahling, Marcel Weisgut, and Tilmann Rabl. Fetch me if you can: Evaluating cpu cache prefetching and its reliability on high latency memory. In *Proceedings of the 21st International Workshop on Data Management on New Hardware*, DaMoN '25, New York, NY, USA, 2025. Association for Computing Machinery.

[OL23] Hyun Woo Oh and Seung Eun Lee. The design of optimized risc processor for edge artificial intelligence based on custom instruction set extension. *IEEE Access*, 11:49409–49421, 2023.

[PVK+25] Anna Pirova, Anastasia Vodeneeva, Konstantin Kovalev, Alexander Ustinov, Evgeniy Kozinov, Alexey Liniov, Valentin Volokitin, and Iosif Meyerov. Performance optimization of blas algorithms with band matrices for risc-v processors. *Future Generation Computer Systems*, 174:107936, 06 2025.

[SCH+23] Till Schlüter, Amit Choudhari, Lorenz Hetterich, Leon Trampert, Hamed Nemati, Ahmad Ibrahim, Michael Schwarz, Christian Rossow, and Nils Ole Tippenhauer. Fetchbench: Systematic identification and characterization of proprietary prefetchers. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 975–989, New York, NY, USA, 2023. Association for Computing Machinery.

[SE17] Yukinori Sato and Toshio Endo. An accurate simulator of cache-line conflicts to exploit the underlying cache performance. pages 119–133, 08 2017.

[VKK+23] Valentin Volokitin, Evgeny Kozinov, Valentina Kustikova, Alexey Liniov, and Iosif Meyerov. *Case Study for Running Memory-Bound Kernels on RISC-V CPUs*, page 51–65. Springer Nature Switzerland, 2023.

[WA19] Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191214-*, 2019. Draft version.

A Appendix

A.1 ARM readKernel Scalar One Array Size

In Figure 14, a heatmap is shown representing the throughput in an adapted experiment for the scalar load micro-kernel on the ARM Cortex-A76. Instead of evaluating the performance on ten arrays, varying in size, the performance is evaluated on one array size. This demonstrates that evaluating on a single array size can bias the results in favor of striding configurations that pairs well to that array size. In this example, the selected array size appear to pair well with low stride-unroll values that are odd.

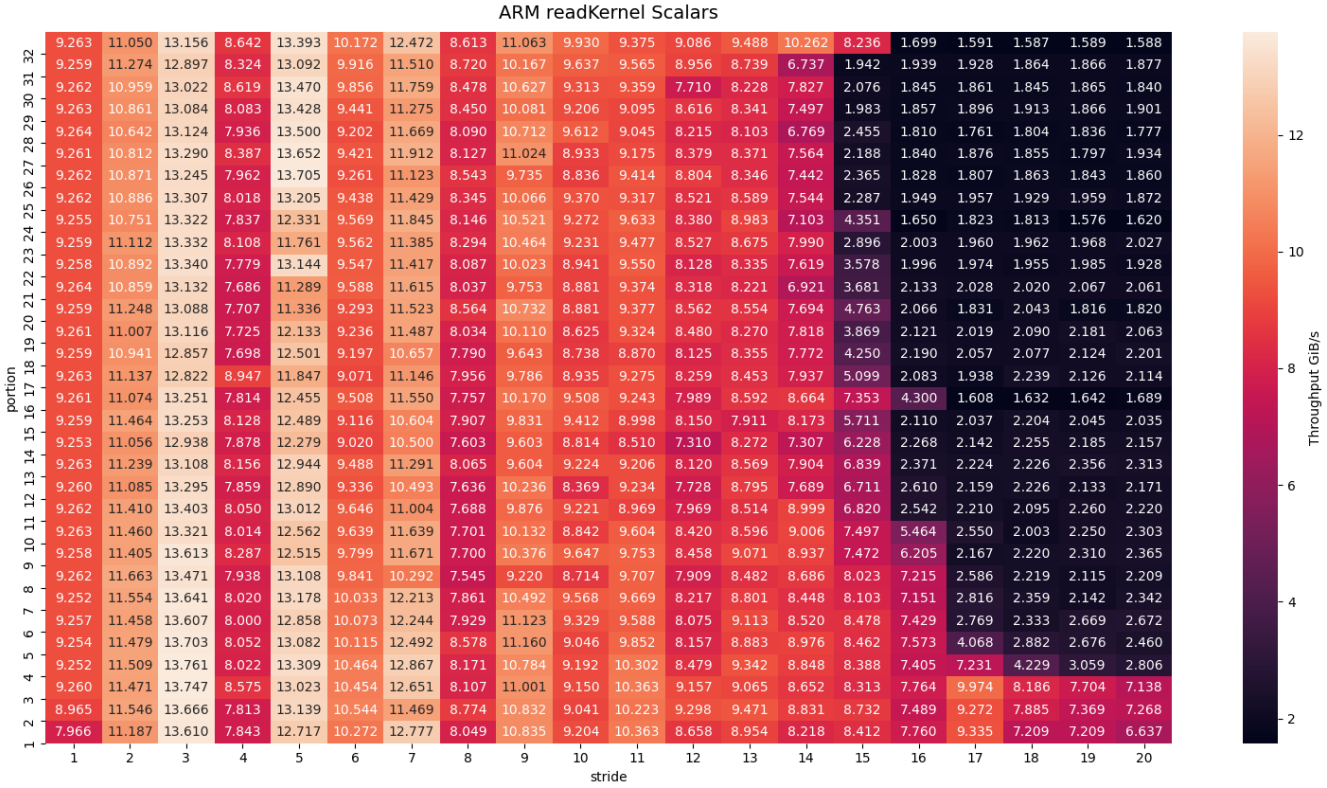

Figure 14: Throughput (GB/s) of a single array size for the scalar STR `writeKernel` on the ARM Cortex-A76.

A.2 ARM writeKernel Scalar Worst Performing Array

In Figure 15, a heatmap is shown representing the throughput for the scalar store micro-kernel on the ARM Cortex-A76. For each striding configuration, the throughput is reported of the worst performing array size. These results further support our hypothesis about the impact address collisions on performance, as discussed in Section **??**.

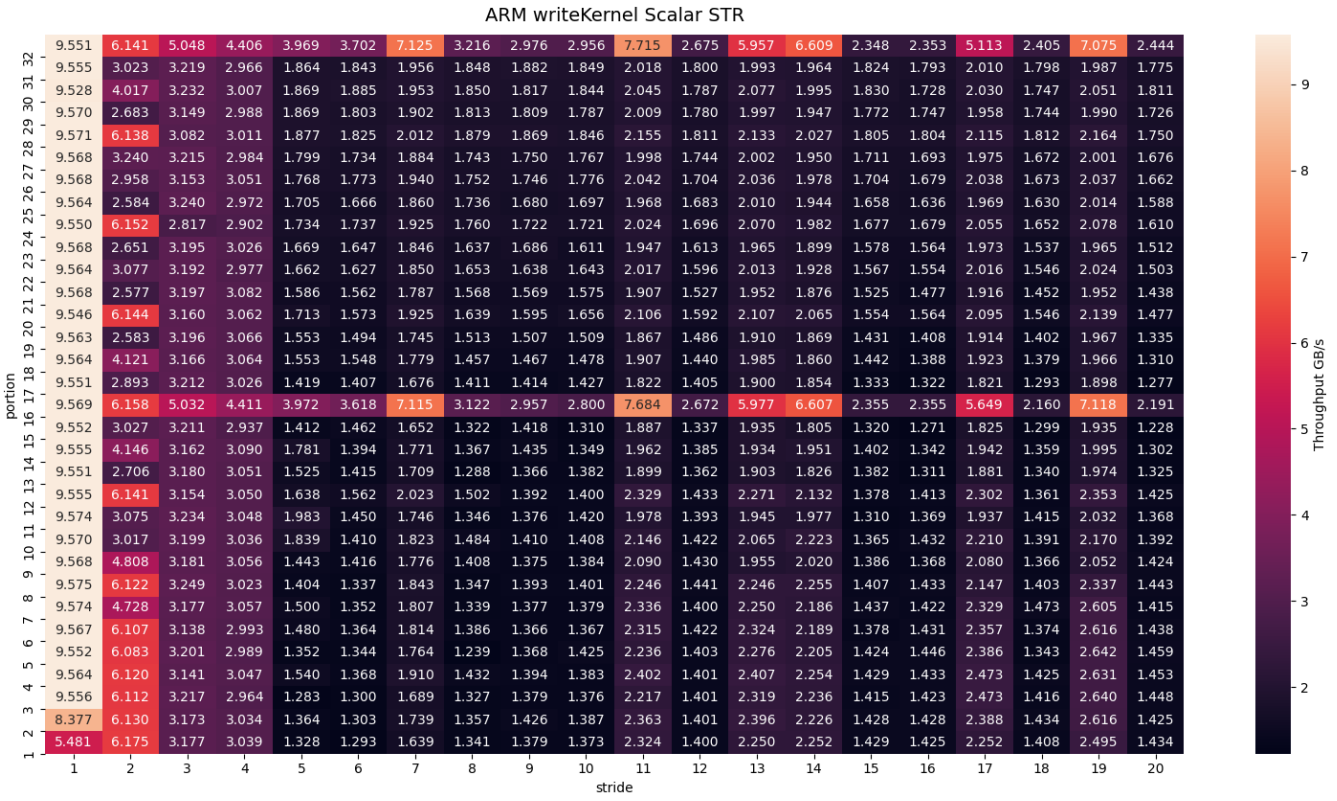

Figure 15: Throughput (GB/s) of the worst performing array size for each striding configuration, for the scalar STR `writeKernel` on the ARM Cortex-A76.

A.3 ARM writeKernel Vector Difference Best and Worst Performing Array

In Figure 16, a heatmap is shown reporting the differences in throughput for the best and worst performing array size for vector stores for the `STR` instruction, on the ARM Cortex-A76. Similarly to the scalar stores, these results clearly shows the impact of selecting the appropriate array size for each striding configurations.

Figure 16: Throughput (GB/s) difference of the best performing and worst performing array size for each striding configuration, for the vector ST1 `readKernel` on the ARM Cortex-A76.

A.4 RISC-V readKernel Scalar Difference Best and Worst Performing Array

In Figure 17, the results of the differences in throughput between the best and worst performing array for the scalar load micro-kernel, on the RISC-V SpacemiT K1. Compared to the ARM Cortex-A76, throughput across varying array sizes appear to remain equal.

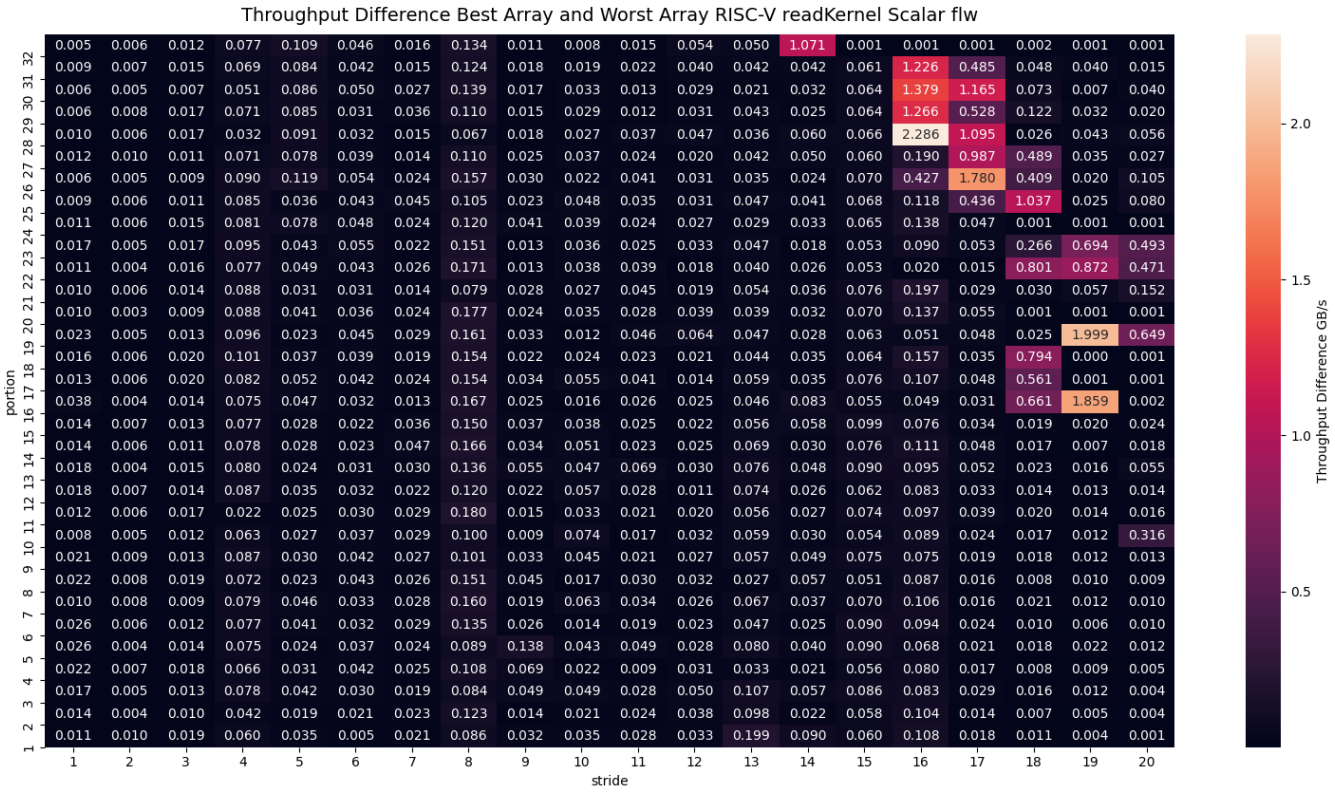

Figure 17: Throughput (GB/s) difference of the best performing and worst performing array size for each striding configuration, for the scalar `readKernel` on the RISC-V SpacemiT K1.

A.5 ARM Assembly

In Listing 19, a generated assembly file for the ARM architecture. The assembly file starts with the assembly prologue, then the loop body performing all the memory accesses, and then the return of the function.

```
    .text
    .global readKernel // ARM

readKernel:  // arr_length=500001660 | in bytes=2000006640

// put registers on stack
    sub sp, sp, #128
    stp x19, x20, [sp, #16]
    stp x21, x22, [sp, #32]
    stp x23, x24, [sp, #48]
    stp x25, x26, [sp, #64]
    stp x27, x28, [sp, #80]
    stp x29, x30, [sp, #96]

// loop counter and loop bound
    movz x1, #0

// loop bound=6250020
    movz x2, #24100
    movk x2, #95, LSL #16
    movk x2, #0, LSL #32
    movk x2, #0, LSL #48

// stride 1 | bytes offset: 0
    movz x3, #0
    movk x3, #0, LSL #16
    movk x3, #0, LSL #32
    movk x3, #0, LSL #48
    add x3, x3, x0

// stride 2 | bytes offset: 400001328
    movz x4, #35120
    movk x4, #6103, LSL #16
    movk x4, #0, LSL #32
    movk x4, #0, LSL #48
    add x4, x4, x0

// stride 3 | bytes offset: 800002656
    movz x5, #4704
    movk x5, #12207, LSL #16
    movk x5, #0, LSL #32
    movk x5, #0, LSL #48
    add x5, x5, x0
```

```
44
45 // stride 4 | bytes offset: 1200003984
46     movz x6, #39824
47     movk x6, #18310, LSL #16
48     movk x6, #0, LSL #32
49     movk x6, #0, LSL #48
50     add x6, x6, x0
51
52 // stride 5 | bytes offset: 1600005312
53     movz x7, #9408
54     movk x7, #24414, LSL #16
55     movk x7, #0, LSL #32
56     movk x7, #0, LSL #48
57     add x7, x7, x0
58
59 .Loop:
60
61     // stride 1
62     ldr q0, [x3], #16 //array id: 0 bytes=0
63     ldr q1, [x3], #16 //array id: 4 bytes=16
64     ldr q2, [x3], #16 //array id: 8 bytes=32
65     ldr q3, [x3], #16 //array id: 12 bytes=48
66
67     // stride 2
68     ldr q4, [x4], #16 //array id: 100000332 bytes=400001328
69     ldr q5, [x4], #16 //array id: 100000336 bytes=400001344
70     ldr q6, [x4], #16 //array id: 100000340 bytes=400001360
71     ldr q7, [x4], #16 //array id: 100000344 bytes=400001376
72
73     // stride 3
74     ldr q8, [x5], #16 //array id: 200000664 bytes=800002656
75     ldr q9, [x5], #16 //array id: 200000668 bytes=800002672
76     ldr q10, [x5], #16 //array id: 200000672 bytes=800002688
77     ldr q11, [x5], #16 //array id: 200000676 bytes=800002704
78
79     // stride 4
80     ldr q12, [x6], #16 //array id: 300000996 bytes=1200003984
81     ldr q13, [x6], #16 //array id: 300001000 bytes=1200004000
82     ldr q14, [x6], #16 //array id: 300001004 bytes=1200004016
83     ldr q15, [x6], #16 //array id: 300001008 bytes=1200004032
84
85     // stride 5
86     ldr q16, [x7], #16 //array id: 400001328 bytes=1600005312
87     ldr q17, [x7], #16 //array id: 400001332 bytes=1600005328
88     ldr q18, [x7], #16 //array id: 400001336 bytes=1600005344
89     ldr q19, [x7], #16 //array id: 400001340 bytes=1600005360
90
91 // loop control
```

```
92      add x1, x1, #1
93      cmp x1, x2
94      BNE .Loop
95
96  .Eind:
97
98  // put registers on stack
99      ldp x19, x20, [sp, #16]
100     ldp x21, x22, [sp, #32]
101     ldp x23, x24, [sp, #48]
102     ldp x25, x26, [sp, #64]
103     ldp x27, x28, [sp, #80]
104     ldp x29, x30, [sp, #96]
105     add sp, sp, #128
106     ret
```

Listing 19: Generated multi-striding assembly file for the vector `readKernel` on the RISC-V architecture. In this assmebly file we use the LDR instruction.

A.6 RISC-V Assembly

In Listing 20, a generated assembly file for the ARM architecture. The assembly file starts with the assembly prologue, then the loop body performing all the memory accesses, and then the return of the function.

```
1       .text
2       .global microKernel # Scalar
3
4   microKernel:  # arr_length=505248 | in bytes=2020992
5
6   # put registers on stack
7       addi sp, sp, -128
8       sd x1, 0(sp)
9       sd x3, 8(sp)
10      sd x4, 16(sp)
11      sd x8, 24(sp)
12      sd x9, 32(sp)
13      sd x10, 40(sp)
14      sd x18, 48(sp)
15      sd x19, 56(sp)
16      sd x20, 64(sp)
17      sd x21, 72(sp)
18      sd x22, 80(sp)
19      sd x23, 88(sp)
20      sd x24, 96(sp)
21      sd x25, 104(sp)
22      sd x26, 112(sp)
23      sd x27, 120(sp)
```

```
24
25 # loop counter and loop bound
26     addi x3, x0, 0
27
28 # loop bound=5263
29     lui x1, 1
30     addi x1, x1, 1167
31
32 # stride 1 | bytes offset: 0
33     lui x4, 0
34     addi x4, x4, 0
35     add x4, x4, a0
36
37 # stride 2 | bytes offset: 673664
38     lui x5, 164
39     addi x5, x5, 1920
40     add x5, x5, x4
41
42 # stride 3 | bytes offset: 1347328
43     lui x6, 329
44     addi x6, x6, -256
45     add x6, x6, x4
46
47     vsetivli x0, 8, e32, m1
48
49 .Loop:
50     fcvt.s.w f0, x3
51     vfmv.v.f v0, f0
52
53     # stride 1
54     vse32.v v0, (x4)  #array id: 0 bytes=0
55     addi x4, x4, 32
56     vse32.v v0, (x4)  #array id: 8 bytes=32
57     addi x4, x4, 32
58     vse32.v v0, (x4)  #array id: 16 bytes=64
59     addi x4, x4, 32
60     vse32.v v0, (x4)  #array id: 24 bytes=96
61     addi x4, x4, 32
62
63     # stride 2
64     vse32.v v0, (x5)  #array id: 0 bytes=0
65     addi x5, x5, 32
66     vse32.v v0, (x5)  #array id: 8 bytes=32
67     addi x5, x5, 32
68     vse32.v v0, (x5)  #array id: 16 bytes=64
69     addi x5, x5, 32
70     vse32.v v0, (x5)  #array id: 24 bytes=96
71     addi x5, x5, 32
```

```
72
73      # stride 3
74      vse32.v v0, (x6)  #array id: 0 bytes=0
75      addi x6, x6, 32
76      vse32.v v0, (x6)  #array id: 8 bytes=32
77      addi x6, x6, 32
78      vse32.v v0, (x6)  #array id: 16 bytes=64
79      addi x6, x6, 32
80      vse32.v v0, (x6)  #array id: 24 bytes=96
81      addi x6, x6, 32
82
83  # loop control
84      addi x3, x3, 1
85      bne x3, x1, .Loop
86
87  .Eind:
88      ld x1, 0(sp)
89      ld x3, 8(sp)
90      ld x4, 16(sp)
91      ld x8, 24(sp)
92      ld x9, 32(sp)
93      ld x10, 40(sp)
94      ld x18, 48(sp)
95      ld x19, 56(sp)
96      ld x20, 64(sp)
97      ld x21, 72(sp)
98      ld x22, 80(sp)
99      ld x23, 88(sp)
100     ld x24, 96(sp)
101     ld x25, 104(sp)
102     ld x26, 112(sp)
103     ld x27, 120(sp)
104     addi sp, sp, 128
105     ret
```

Listing 20: Generated multi-striding assembly file for the vector `writeKernel` on the RISC-V architecture.