TÉCNICO LISBOA

uve

Scalable vector streaming

Unlimited Vector Extension with data streaming support

João Mário Ribeiro Domingos

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisors: Prof. Pedro Filipe Zeferino Aidos Tomás
Prof. Nuno Filipe Valentim Roma

Examination Committee

Chairperson: Prof. Teresa Maria Sá Ferreira Vazão Vasques
Supervisor: Prof. Pedro Filipe Zeferino Aidos Tomás
Member of the Committee: Prof. Leonel Augusto Pires Seabra de Sousa

October 2020

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

This document marks the end of a long but enticing journey, and as with any journey, one would not get through alone. I'm eternally grateful to my parents and sister, who have always been there and carried me on their shoulders throughout life's bumps and falls. A very special thank you to Mariana, who stood by my side and supported me throughout every decision and commitment.

Of course, this academic journey would not be the same without Prof. Pedro Tomás and Prof. Nuno Roma. Without their ideas, advice, constant presence and extensive experience, this work would not even be possible. Thank you for trusting me with the opportunity of making UVE real and allowing me to learn so much with you.

As no journey is complete without friends, I have a special acknowledgement to Miguel Pinho. Who patiently listened to my internal debates and was always available to help. Furthermore, i also want to thank Simão Sarmento and Valter Mário who, undoubtedly where the best working partners, great friends and crucial to this journey. Additionally, João Santos and Gonçalo Pereira thank you for always being ready to play some pool.

I'm incredibly grateful to everyone I met at JUNITEC, where I had the opportunity to explore the practical engineering part of me and to have a glimpse of the entrepreneurship world.

This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) under projects UIDB/50021/2020 and PTDC/EEI-HAC/30485/2017, and by funds from the European Union Horizon 2020 research and innovation programme under grant agreement No. 826647.

Abstract

Unlimited vector extension (UVE) is a novel instruction set architecture extension that takes streaming and SIMD processing together into the modern computing scenario. It aims to overcome the shortcomings of state-of-the-art scalable vector extensions by adding data streaming as a way to simultaneously reduce the overheads associated with loop control and memory access indexations, and memory access latency. This is achieved through a set of instructions which are able to pre-configure the loop memory access pattern(s), attaining accurate and timely data prefetching on predictable access patterns, such as in multidimensional arrays or on indirect memory access patterns. Each of the configured data streams is associated with a general vector register, which is then used to interface with the streams. In particular, iterating over the stream is simply achieved by reading/writing to the corresponding input/output stream, as the data is instantly consumed/produced. To evaluate the proposed UVE, a first gem5 implementation was made on an out-of-order processor model, based on the ARM Cortex-A76, thus taking into consideration typical speculative and out-of-order execution found in high-performance computing processors. The evaluation was carried out on a set of representative kernels, by assessing the number of executed instructions, its impact on the memory bus and its overall performance. Compared with state-of-the-art solutions, such as the upcoming ARM Scalable Vector Extension (SVE), results show that the proposed solution attains performance speedups between 2x and 4x.

Keywords

ISA SIMD Extensions, Scalable Vector Processing, Stream Computing, General-purpose Processors

Resumo

Unlimited Vector Extension é uma arquitectura de conjuntos de instruções que junta os conceitos de transmissão de dados em série (fluxo) com o paradigma de processamento instrução única em múltiplos dados. Este trabalho tenta melhorar o atual estado da arte das extensões vectoriais escaláveis através da adição do conceito de transmissão de dados em série, permitindo assim reduzir as instruções de controlo de loop e acesso à memória despensáveis e melhorar a latencia de acesso à memoria. Através da pré-configuração do acesso à memoria em fluxo é possivel realizar o pré-carregamento de dados da memoria com absoluta precisão e excelente temporização, mesmo em acessos multidimensionais, complexos e indirectos. A transmissão de dados para o processador é conseguida realizada através de um conjunto de registos vectoriais genéricos, servindo de interface para a transmissão do fluxo de dados. Em particular, a iteração do fluxo de dados é executada através da leitura/escrita do registo vectorial associado, sendo os dados automaticamente consumidos/produzidos para a transmissão em série associada. A avaliação foi realizada utilizando o simulador gem5, onde o modelo de um processador fora de ordem, basedo no Cortex-A76 da ARM, foi modificado e extendido para suportar a extensão vectorial. A avaliação teve em conta um conjuto de aplicações representativas, tendo obtido melhorias de performance entre 2 e 4 vezes, relativamente à extensão ARM Scalable Vector Extension (SVE).

Palavras Chave

Extensão SIMD, Processamento Vectorial Escalável, Computação em Fluxo de Dados, Processadores de Uso Geral

Contents

List of Figures

List of Tables

Listings

Acronyms

ALU	Arithmetic and Logic Unit
AMPM	Access Map Pattern Matching
BOP	Best-Offset Prefetcher
CPU	Central Processing Unit
DLP	Data-Level Parallelism
FP	Floating-Point
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
GPP	General Purpose Processor
HPC	High-Performance Computing
ILP	Instruction-Level Parallelism
ISA	Instruction Set Architecture
MEMCPY	Memory copy. Procedure that copies a source zone of memory to a destination one.
SAXPY	Computation of the product of the Real Value A with each element of the Real matrix X added to the respective element of the Real matrix Y
SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data
SVE	Scalable Vector Extension
RVV	RISC-V "V" Vector Extension
UVE	Unlimited Vector Extension
VL	Vector-Length

1 Introduction

1.1 Motivation

In computing, there is a constant race to obtain the most performance out of a processor. This competition has been shifting from raw performance increases to more energy-efficient processors, while still increasing the processing power. This movement removed the main focus from Instruction-Level Parallelism (ILP) only and shifted it to a combination of instruction, task and Data-Level Parallelism (DLP) that make out the architecture paradigm of modern high-performance processors.

While instruction-level and multicore parallelism are easily noticed in modern processors, the DLP is hidden in single-instruction multiple-data units. These units are made of wide Arithmetic and Logic Units (ALUs) that allow for multiple data elements to be processed at the same time, proportionally decreasing the computation time, or increasing the processing throughput. In contrast, the scalar nature of a processing core allows only for one instruction to be processed in one arithmetic and logic unit at once. While processing more data in one cycle is a clear benefit; Memory access bandwidth, complex memory patterns, higher latency SIMD instructions, and other constraints limit the promised potential.

Specific applications, like matrix computations, image processing, audio processing are ideal for being accelerated with SIMD units, due to their array-like memory and processing organization and large datasets. Also, the SIMD units can only process a fixed data size at each cycle, e.g. Arm NEON is limited 128-bits wide [1], and x86 AVX to 256-bits [2]; these are refered to as fixed Vector-Length (VL) extensions. Taking into account the current processor registers being 64-bits, it places the mentioned SIMD ISAs in 2 to 4 times the processing width. Considering application overhead and processor architecture constraints, the increase in processing width is not linearly proportional to a theoretical speed-up.

By having a fixed vector-length, any modification of the length requires a new instruction set to be defined, and therefore new code needs to be written, compiled and deployed. This quickly leads to application incompatibilities, additional development time and system downtime. Fixed vector-length extensions have followed a recent trend where the vector register size rough doubles every five years [2, 8, 9]. This trend is well observed in the SIMD extension for the x86 ISA, where sizes have gone from 64-bit (MMX), to 128-bit (SSE) and 256-bit (AVX), currently staying at 512-bit with AVX-512 [2, 8–10]

Such a problem can be solved by creating an instruction set that does not rely on a fixed vector size. As such, ARM introduced Arm SVE that limits the vector length to 2048-bits, but does not require the full length to be implemented in any processor [7]. Consequently, the same software is compatible with multiple implemented lengths up to 2048-bits. Which in its turn, allows High-Performance Computing (HPC) targeted processors to perform better, while allowing lower-power targeted processors to maintain their energy and resource efficiencies. However, it is not without its own problems and limitations.

On the one hand, SIMD extensions work with continuously growing data vectors in order to increase their performance potential. On the other hand, the novel processor architectures have seen a competition for power and resources in the various processor sections. This means that the budget for wasted power is reduced and the SIMD extensions may not be able to grow further. Also, to provide such large vector processing units with data creates a new challenge where the memory access mechanisms also need improvement and optimisation. One possible solution is relying on data prefetchers, which allow for indexation prediction and thus increased memory access performance and lower latency [11–13]. However, as data prefetching may wield performance improvements, it is not as efficient in shorter memory accesses and in irregular ones. Moreover, the maximum prefetching accuracy is not easily achievable. Another solution that can improve further is data streaming [14–16]. Configuring the memory access pattern in software and fetching the data in the background, will lead to perfect prefetching accuracy and remove the memory access procedures from the core. Data streaming presents itself as an opportunity to further improve the memory access latency and throughput, as well as removing indexation and access instructions from the core [16–19].

In this thesis, streaming is merged with scalable SIMD extensions to create a novel and still unexplored opportunity. While the SVE extension and the RISC-V counterpart RVV [6] are quite new, they are based on common load-store mechanisms and did not contemplate streaming as a possible solution. With streaming being a different take on memory accesses, it is expected to bring new performance aspects as well as new difficulties. This work aims to create a proof-of-concept where streaming poses as a new path forward in modern computer architectures.

1.2 Objectives

Considering the future computational needs of High-Performance Computing applications, scalable vectorial extensions have been pondered. The scalable nature of such extensions allow the CPU implementations to adapt their vector length to the targeted applications. Moreover, with a flexible vector-length it is possible to optimize the resulting performance, power and energy efficiency requirements.

Considering their scalable essence, implementations could easily support up to 2048-bit vectors that represents huge and fast memory accesses in order to keep the vectorial units active. Making use of memory accesses pre-configurarion and decloupled memory access, data streaming is an opportunity to improve the overall performance in memory transactions, thus representing an enhanced application performance.

To validate this proposition and their impact in application performance, this work is cleary defined by two objectives. Being the first, the definition of a streaming scalable vectorial instruction set architecture extension, featuring an agnostic vector length programing model based on the open-source RISC-V ISA architecture. And the second, the evaluation of the extension performance in a representative out-of-order CPU model.

1.3 Contributions

The Unlimited Vector Extension (UVE) is the pilar of this thesis, combining novel scalable SIMD processing with the data streaming paradigm, the main contribution of this thesis is proving that data streaming in a vectorial extension is a real opportunity to increase the SIMD processing performance. In detail, contributions of this work are:

- The creation of a streaming tailored scalable SIMD extension, which can exploit increased performance in memory access and increased processing performance.
- A set of architectural modifications to a representative out-of-order Central Processing Unit (CPU) model that bring support for streaming mechanisms and the UVE extension.
- The proposal of mechanisms that allow streaming to work seamlessly with a representative processing core model and memory hierarchy.
- An implementation of the CPU model on the Gem5 simulator, that can be easily tuned to target any common processor model, and extended or modified for future iterations of UVE.
- A minimal implementation of UVE in the compiler toolchain, which allows for simpler application development and porting.
- An estimation of the performance improvements of a reference streaming scalable vectorial extension implementation.

1.4 Outline

This document proceeds with a background and literature review in chapter 2. chapter 2 introduces the necessary concepts and backgrounds in computer architectures that are necessary to follow the work that was developed in this thesis. After, chapter 3 delves into the core concepts of data streaming and stream representations; this chapter prepares the reader for an in-depth discussion of streaming and streams in the subsequent sections. At this point, Part I develops on the instruction set architecture. Being followed by the supporting microarchitecture in Part II. Each document part is composed of a design, discussion and implementation chapter; and a chapter where the results are presented and discussed. The document ends with chapter 8, where a conclusion is taken on the objectives, learnings, achievements and the future opportunities of this work.

This work groups four chapter in two parts. In the first part, the higher level ISA definition is given, where UVE is designed, defined and implemented (chapter 4). At the end of part one, the potential of the UVE instruction set is examined and compared with the state-of-the-art scalable extension (chapter 5). In the second part, a supporting microarchitecture that implements streaming is defined and implemented (chapter 6). A battery of applications is then simulated with the microarchitecture and compared with an identical microarchitecture model using the state-of-the-art scalable extension (chapter 7).

2 Background & Related Work

This chapter gives an overview of the single instruction multiple data technologies. These technologies are the ones employed by modern ISAs, such as x86 [1] and Arm [2]. These will serve as a knowledge base for the developed work. Additionally, the state-of-the-art scalable vector extensions, SVE and RVV are explored as an improvement over the earlier fixed vector-length extensions. After, we will address methods for describing memory accesses, which are fundamental to the description of memory streams. Additionally, the concept of memory streams is further detailed. Last but not least, it is carried out the description of an already existing streaming ISA extension technology.

2.1 SIMD: Architectures & Extensions

To set the tone, let us start by establishing that the objective of all SIMD ISA extensions is to allow the CPU to explore DLP, thus providing a solution that is a compromise, in terms of resources and processing performance, between using external specialized accelerators (e.g., Field Programmable Gate Array (FPGA), Graphics Processing Unit (GPU)) and the typical scalar execution. However, due to the proximity with the CPU, there are critical advantages from the low communication overhead and the shared processing pipeline. Overall, SIMD processing is a discussable solution, with considerations of energy and space, as well as processing performance.

Taking this into consideration, the following subsections detail the Arm NEON and x86 SSX and AVX extensions. These present two distinct approaches that somewhat reflect the respective base ISA. As a side note, RISC-V also defined a fixed-length SIMD extension. However, since its been deprecated, it will not be refered here.

2.1.1 SIMD in RISC: Arm NEON

The NEON extension is ARM's leading solution in accelerating media content processing. This extension is composed of 32 registers that are 64-bits wide. These registers are not shared with the Arm core registers but are an extension to the floating-point extension (VFPv2) registers. To give more flexibility to the processing; NEON defines groups of instructions operating on vectors stored in 64-bit registers (double-word (D)) and 128-bit register (quad-word (Q)). The instructions defined in NEON perform the following operations [1,20]:

- data processing (additions, multiplications, among others)
- memory accesses (load, store)

[1] x86 is used to represent the ia-32e ISA family. The x86 term is kept as it is more common and widespread.

[2] Any references to the Arm ISA are relative to the ISA version Armv8. On contrast, ARM, with all letters uppercase, refers to Arm Holdings, the company.

- data moving between NEON register and Arm core registers (general purpose)

- datatype conversion (width extension, integer to floating-point conversion)

Figure 2.1 shows an example of data processing using NEON. Here, the used registers are 128-bit wide and divided into 8 lanes. The addition operation executes by summing each register (Q1 and Q2) lane and storing the result in the Q0 register.

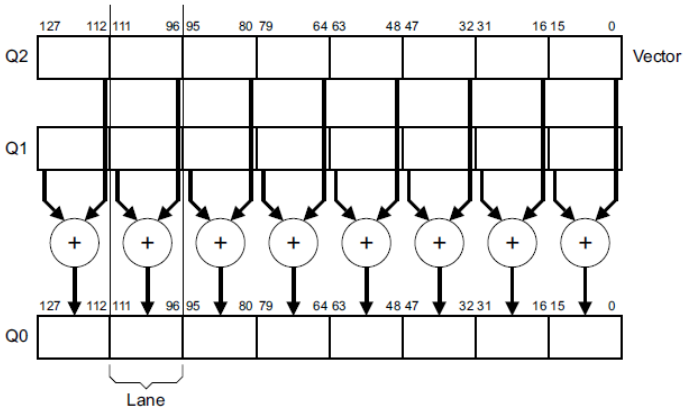


Figure 2.1: NEON 8-way 16-bit add operation [1].

NEON sets support for 8-bit, 16-bit, 32-bit, and 64-bit signed and unsigned integer elements. For floating-point elements, only 32-bit single-precision and 16-bit half-precision are available. There is also support for 8-bit and 16-bit polynomials, mainly used for carry-less multiplications. Additionally, some data processing operations can make use of different sized input and output registers. Thus, results can have higher precision (long multiplications) or lower precision (narrowing) in the output. NEON also defines specific instructions to address the narrowing and widening of the datatype. The NEON instructions operate only in register-register mode, following the principle of the Reduced Instruction Set Computer (RISC) mindset [1].

However, NEON is not withouth downsides, it is limited when it comes to complex memory accesses, where a scatter-gather mechanism is essential but not present. Additonally, there is no per-lane execution control mechanisms, which considerably limits the vectorizable applications.

2.1.2 SIMD in CISC: x86 SSE/AVX

In the x86 side, currently the most used and available SIMD extensions are SSE (Streaming SIMD Extensions) and AVX (Advanced Vector eXtensions), with the latest version supporting up to 512-bit registers.

Nevertheless, we should remember that Intel started SIMD extensions with the MMX extension. From MMX to AVX, the SIMD extensions improved multiple times, taking around 20 years of technology improvements. From these improvements, some honourable mentions follow [2,8,9,21–23]:

1) MMX 94': The MMX extension allowed programmers to execute instructions on multiple data elements. It makes use of MMX technology registers (standard SIMD vector registers), packing 64-bits.

2) SSE 99': SSE extension allowed SIMD operations to use four floating-point single-precision elements packed in 128-bit vector registers. All saved on a 1024-bit register bank.

3) SSE4 06': SSE4 provided 128-bit integers (using SIMD engine) and double-precision floating-point. The vector register size used is still 128-bit. The total number of SIMD extension instructions is around 350. Also, at this particular time, applications could already benefit from wider vector-registers.

4) AVX 08': AVX introduces 256-bit vector registers, all fitted in a 16x256-bit register bank. It also added four operand and three operand SIMD instructions, with support for single and double-precision floating-point operations. AVX shares the register bank with the previous SSE extensions (see Figure 2.2). A further revision of AVX, named AVX2, added support for integer SIMD processing.

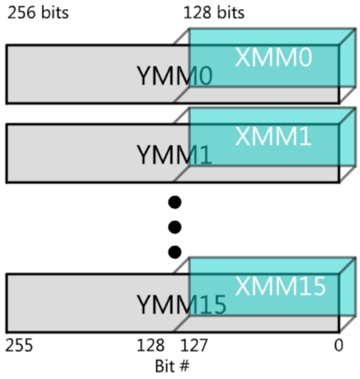


Figure 2.2: x86 AVX register bank structure [2].

5) AVX-512 15': AVX comprises 512-bit vector registers.

While the most-recent x86 SIMD extension (AVX-512) supports 512-bit vector registers, NEON still stands at 128-bit. With consideration to the vector-width, AVX has more processing capability, additonally, as it is a CISC ISA, it support a wider range of generic and specialized instructions. In contrast, NEON is limited to narrower range of instructions, being the main constraint the lack of scatter-gather support. Even so, in terms of energy efficiency, having a narrower vector pipeline and a more curated set of instructions gives NEON an advantage over AVX.

2.1.3 Discussion

To better understand the effect of a SIMD approach on the programmer's perspective, a simple program will be used, consisting of the well known SAXPY (Listing 2.1) loop code.

To accelerate the loop, consider the use of 128-bit SIMD registers. Hence, these can pack four single-precision floating-point elements (32-bit). Also, we use a pseudo assembly that loosely relates to the Arm + NEON extension dialect in the assembly code. The choice of assembly dialect was majorly influenced by the additional complexity of the x86 + AVX instruction set.

Listing 2.1: SAXPY C code listing.

```c
#define n 34
void saxpy(float *X, float *Y, float A, int n){
    for( int i = 0; i < n; i++){
        Y[i] = A*X[i] + Y[i];
    }
}
```

The standard approach to SIMD vectorization is first to unroll the application loop. By taking into account that the packing factor is 4, the loop unrolls four times. Additionally, as we have four results in each iteration, it is necessary to skip three out of four scalar iterations. As so, Listing 2.2 shows the result. What Listing 2.2 also shows is that in some special conditions, we have to process edge cases without SIMD. These aroused from the loop iterations count (34) not being a multiple of our packing factor (4). Thus, the creation of an epilogue or prologue is necessary to solve the edge cases. Some other strategies could be employed, such as forming a mirrored or zero padding around the loop boundaries [24, pp. 222-230] [25].

Listing 2.2: SAXPY C unrolled loop listing.

```c
for( int i = 0; i < n; i+=4){
    Y[i] = A*X[i] + Y[i];
    Y[i+1] = A*X[i+1] + Y[i+1];
    Y[i+2] = A*X[i+2] + Y[i+2];
    Y[i+3] = A*X[i+3] + Y[i+3];
}
Y[n-1] = A*X[n-1] + Y[n-1];
Y[n] = A*X[n] + Y[n];
```

Sequentially, the following step is to condense the four unrolls in a SIMD operation (see Listing 2.3).

Listing 2.3: SAXPY SIMD-parallel loop listing.

```c
for( int i = 0; i < n; i+=4){
    SIMD_LOAD(vY, Y, i)
    SIMD_LOAD(vX, X, i)
    vY = SIMD_ADD(SIMD_MUL(vX,vA), vY)
    SIMD_STORE(Y, vY, i)
}
Y[n-1] = A*X[n-1] + Y[n-1];
Y[n] = A*X[n] + Y[n];
```

We can now write it to assembly, as in Listing 2.4. Here, we observe that the loop control and memory address calculation instructions are present inside the loop and undesirably occupy a considerable portion of the code. Additionally, the edge cases are a concern that needs special attention, as they exist in any traditional SIMD processing, sometimes leading to considerable overhead (particullarly for a low number of iterations). While in cases with a low number of total iterations the overhead may be negligible, sometimes the overhead is repeated multiple times due to being inside an outer loop, which gives rise to a significant negative impact on application performance.

Listing 2.4: SAXPY loop assembly code listing for SIMD processing. Pseudo ISA based on NEON instruction set.

```
V.DUP_32 V0, R4 ;Loading value of A (R4) into all elements of V0 (vector register) with elements of size 32-bits

Loop:
    ;R2 contains X address, R1 is the indexer i
    VLD     V1, [R2+R1]    ;Loads 4 elements of 32-bits of the array X to V1
    VMUL_SP V1, V0, V1
    VLD     V2, [R3+R1]    ;R3 contains Y address
    VADD_SP V2, V2, V1
    VST     [R3+R1], V2   ;Stores the result of the 4 parallel operations
    ADD      R1, 16     ;Increments i by 4 floating point elements(4-bit addressing)
    SUB      R5, 4      ;R5 contains N (number of iterations)
    B.NP    Loop       ;Repeats until N < 0

LD       R12, [R2+R1]      ;Case Y[n-1]
MUL_SP   R12, R4, R12
LD       R13, [R3+R1]
ADD_SP   R13, R12, R13
ST       [R3+R1], R13

ADD      R1, R1, 1

LD       R12, [R2+R1]    ;Case Y[n]
MUL_SP   R12, R4, R12
LD       R13, [R3+R1]
ADD_SP   R13, R12, R13
ST       [R3+1], R13
```

2.1.4 SIMD Extensions Summary

In a more architecture-related overview, one of the advantages that the SIMD technologies bring is the ability to exploit DLP inside the CPUs. By exploiting DLP, the CPU can reach higher levels of performance in some applications characterized by massive levels of data parallelism. Additionally, a SIMD architecture executes a lower number of loop iterations when compared with a Single Instruction Single Data (SISD) architecture, as shown in Listing 2.3, and therefore, there will be, ideally, less clogging in the functional units, and fewer accesses to the instructions memory. Also, there are advantages concerning the loop control instructions, as there will be fewer loop iterations, and less speculative execution. Hence, due to the fewer loop instructions, there is a lower impact in wrong branch predictions, leading to more power-efficient CPUs.

As with every technology, there are also some caveats. Traditional SIMD is not an exception, and there are some downsides to it as well. First, the limited vector width is a significant concern. A static vector width, specified in the ISA definition forces the implementations to conform with it. Second, having a specific vector length limits the achievable performance in some applications that could profit from being executed in larger vector widths [26]. As an example, the implemented vectors can be too wide for some applications (leading to larger epilogues/prologues) and too narrow for others (leading to an excess overhead in memory access indexing and loop control). This leads to significant implementation inefficiency and ultimately to performance boundaries constrained by the vector length. Additionally, SIMD technology also has drawbacks related to memory access. In simple, linear memory accesses, the SIMD architecture can easily fetch all the data. However, more complicated memory access patterns can lower the performance of the architecture. Admittedly, we can make use of scatter-gather mechanisms to access intricate patterns of data. By taking AVX2 as example, a gather load works by using a pre-loaded vector with indexes that is used to index the memory accesses of the load destination vector. This means that the AVX2 scatter-gather mechanism needs to access memory twice per fetched data element, and the vector that indexes the memory access must be indexed ahead of the data access. Leading to inevitably lower performance. In general, scatter-gather mechanisms need to be deeply intertwined with the cache memory controller, which is an added dificulty in implementing such mechanisms. Specifically, NEON and the first version AVX do not support scatter-gather instructions.

Finally, by making an example out of x86 SIMD extensions progression (see Figure 2.3), we realize that there is a continuous progression in vector register width. This progression has been driven by technological progress, involving video media processing and the continuous uprising of machine learning applications. However, continuing to create more extensions as the vector lenght increases is unsustainable. In detail, by creating a new extension, all the code must be recompiled, effectivelly rendering obsolete all non open-source software packages. Also, to guarantee that old binaries (not recompiled) are still executable by newer machines, the old ISA extensions must be part of the implementation, posing a significant pressure on both the decoding and execution stages of new processor implementations. In conclusion, the drawbacks of maintaining code may seriously outmatch the benefits of using broader vector registers.

Even so, the path of improvement is a never-ending one. Hence, the objective is to devise a new extension that can benefit from broader registers while avoiding the previously described drawbacks.

2.2 Scalable Vectorial Extensions

The scalable vector extensions are the most recent state-of-the-art in vectorial processing. The vectorial SIMD architecture has as main objective of mitigating the disadvantages of the SIMD architecture, with the premise that registers have to be scalable at the implementation level. The first question that should come to mind is how much this will improve on the non-scalable SIMD architectures. First, scalability means we can compile once and run in different vector width implementations. Second, if the

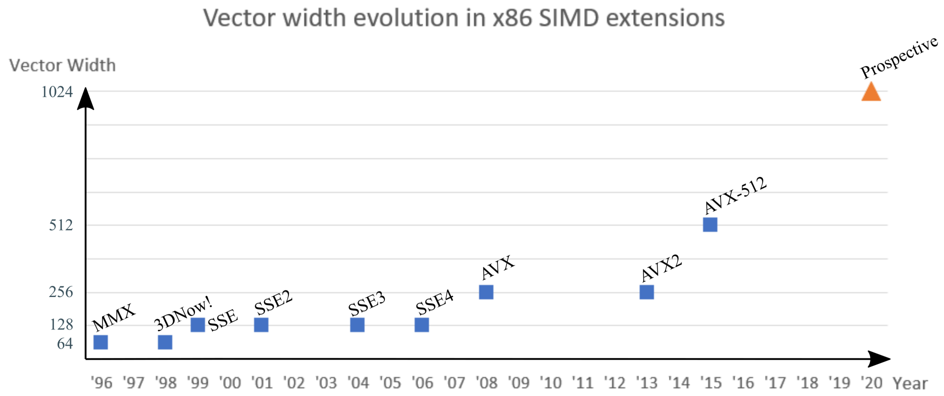


Figure 2.3: Evolution of vector width in x86 SIMD extensions. The rightmost orange mark is a prospective that represents the trend in the usage of wider vector registers [8, 9, 22].

implementation can scale well, we can eliminate the edge-cases. By using a vector of the size of a scalar register, there will be no edge cases. Hence, by scaling that same example from scalar to a more wide register, we will remain with no edge cases.

Currently, two scalable vector extensions are worth mentioning: RVV and SVE.

2.2.1 RISC-V "V" Vector Extension

Instruction set architecture The RISC-V vector extension follows the simplicity and genericity of the RISC-V base ISA, so the constraints in terms of maximum and mininum vector width are very loose, not limiting any implementation to a particular configuration. Hence, the definition of the extension states that a single vector element must have size ELEN (element length, unspecified), and that the size of a vector register is VLEN ($\geq$ ELEN)(vector-length), both constrained to powers of two. The total number of register is 32.

Additionally, the available integer and fixed-point data types are all inherited from the implemented base ISA [27]. Finally, the implemented Floating-Point (FP) extensions define the FP data types [27]. To guarantee fine-grained control over the execution, a set of 8 predicate registers is defined, which allow for finer control over the execution by setting an execution control mask on top of a vector register.

By using specific control status registers (CSRs), one can control the configuration of each vector as it executes in run-time, namely regarding the vector length and the vector data length and data type. The vector configuration depends fully on the "vsetvli" instruction.

```
vsetvli   "configured size",  "size",  #element size,  [#grouping factor]
```

The configuration starts with requesting a vector size (in elements) and an element size. Then, the total requested size (element size * vector size) is compared with the implemented vector-length, the minimum of both is used to configure all the vectors and written do the destination operand ("configured size") [6]. The illustration in Figure 2.4 is included to clarify the behavior of "vsetvli" and the respective effects in the registers.

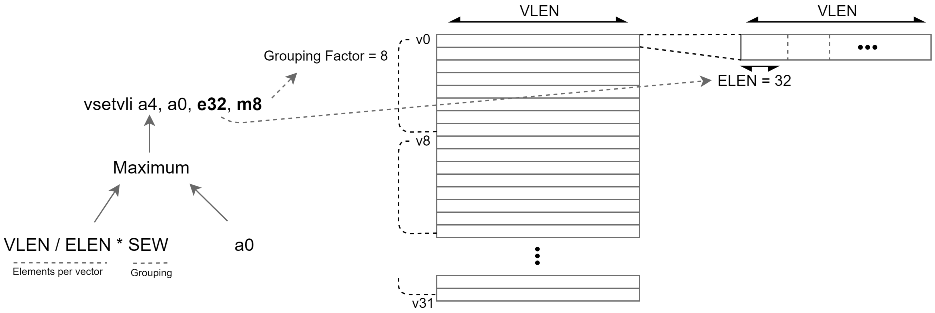


Figure 2.4: Illustration of "vsetvli" instruction behavior, with calculation of the configured size and effects on the vectorial registers.

Moreover, the operand "grouping factor" can be used to group a series of consecutive registers, being particularly useful when the total requested size is larger than the implemented vector size. In detail, a grouping factor of 4 will group registers 0 to 3 into register 0 and so on for the remaining vector registers.

Application example To materialize this in an application example, the assembly code in Listing 2.5, represents the SAXPY application ported to RVV, given before in Listing 2.1.

Listing 2.5: SAXPY loop assembly code listing for RVV scalable vector processing. RISC-V with V extension [6].

```
; register arguments:
;     a0      n
;     fa0     A
;     a1      X
;     a2      Y

saxpy:
vsetvli a4, a0, e32, m8 ; Ask for n elements of size 32b and group 8 vector registers
vlw.v v0, (a1)          ; Load data from X to v{0-7}
sub a0, a0, a4          ; Decrement n with the number of processed elements
slli a4, a4, 2
add a1, a1, a4
vlw.v v8, (a2)          ; Load data from Y to v{8-15}
vfmacc.vf v8, fa0, v0   ; Floating-point multiply and acumulate
vsw.v v8, (a2)          ; Store data
add a2, a2, a4
bnez a0, saxpy
ret
```

The RVV SAXPY application shows the high dependency on the "vsetvli" instruction which is the core point of the loop iteration. In detail, the application requests to "vsetvli" the total remaining elements of the loop, then the actual number of processable elements in the iteration is subtracted to the total remaining elements, the loop ends when there are no remaining elements. As note, in the case where the number of elements is smaller than the total vector-length (considering the grouped registers), there will be only one iteration. The internal part of the loop brings nothing new, there is address calculation, memory access and computation.

2.2.2 Arm "SVE" Scalable Vector Extension

On the SVE side, the vector registers do not follow the width flexibility of RVV. In this case, the vector registers are limited from 128-bits to 2048-bits, in multiples of 128-bits. Also, the allowed element widths range from 8-bit to 64-bit. The total number of registers is 32, the same as in NEON. The SVE registers extend on top of the NEON registers. [7].

Alongside the scalable registers, SVE introduces a set of predicate registers that behave has execution masks, like in the RVV architecture. In specific, the SVE predicate registers are 16 in total; however, only eight are usable on arithmetic and memory instructions. ARM states that this balance mitigates the register pressure observed in other architectures [7, 28].

The SAXPY function (see Listing 2.1) will be reused to explain the differences between SVE and RVV. The matching assembly for SAXPY in SVE is in Listing 2.6. The SVE assembly the instructions "whilelt" and "incs" are in the core of the scalability.

Listing 2.6: SAXPY loop assembly code listing for scalable architecture processing in Arm SVE [7].

```

mov x4, #0                ;Initialize iteration counter (i) in r4
whilelt p0.d, x4, x3      ;Set predicate p0 for each r4 element that is lower than
    n
ld1rd z0.d, p0/z, [x2]   ;Broadcast value A to v0, with predicate zeroing
.loop:
ld1d z1.d, p0/z, [x8, x4, lsl #2] ;Loads to v1 the values pointed by X[i] with a
    stride of 32-bits elements
ld1d z2.d, p0/z, [x9, x4, lsl #2] ;v2:=Y[i]
fmla z2.d, p0/m, z1.d, z0.d       ;Fused multiply-accumulate Y[i]=A*X[i]+Y[i],
    with predicate merging
st1d z2.d, p0, [x1, x4, lsl #2]   ;Store v2 to memory pointed by Y[i]
incs x4                 ;Increment i based on the vector size and SP (VL/32)
.latch:
whilelt p0.d, r4, r3
b.any.loop              ;Loop again if all of the predicate elements are 0
```

On the one hand, the "whilelt" instruction will populate a predicate register that will condition the loop iteration. This instruction matches the following format:

while**lt** "predicate", "start value", "comparison value"

In simple terms, the predicate is filled with ones while the incremented start value is less than the comparison value. Figure 2.5 gives more detail in the behavior of "whilelt". Additionally, to loop into the next iterations, the branch instruction uses the AArch64 NZCV condition code flags [29]. The "whilelt" instruction sets the condition codes that are later used in the branch [7, 30].

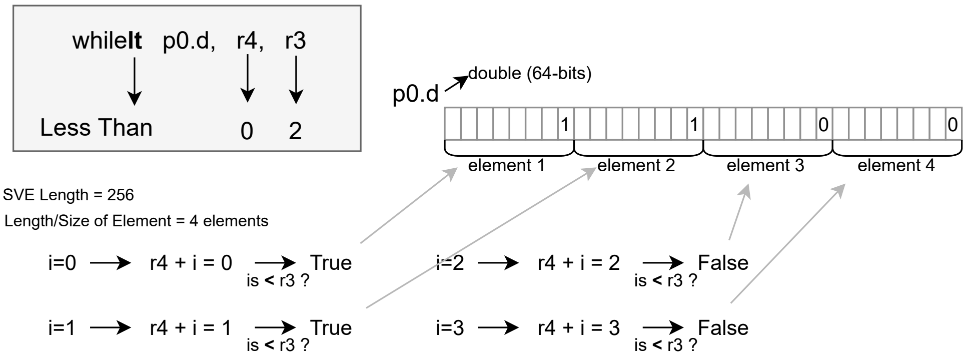


Figure 2.5: Illustration of "whilelt" instruction behavior.

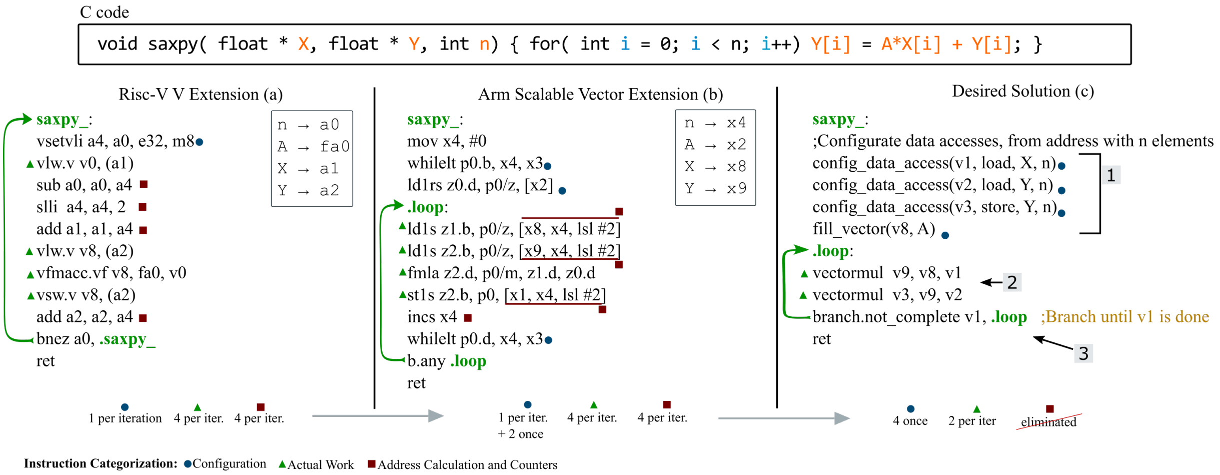
On the other hand, the "incs" instruction iterates the start value by the vector size, simplifiing the memory addressing calculations and removing more unnecessary iteration counters [7, 30]. To directly compare the scalability approach between RVV and SVE, we can compare the "setvli" instruction directly to the "whilelt" instruction. While RVV configures the vector register behavior, the SVE contrasts by setting a predicate register and controlling the iterations through it. In sum, both the approaches are entirely valid. However, both could still achieve more simplicity.

2.2.3 Discussion

When comparing to the SIMD extensions discussed in section 2.1 with the scalable solutions in RVV and SVE presented in section 2.2, the later shows significant benefits, particularly related with the handling of loop edge cases and with the automatic scaling of the vector length. However, to understand their caveats, let us analyze the RVV and SVE codes in Listings 2.5 and 2.6) as well as Figure 2.6, which presents the different implementations side-by-side to make it easier to interpret and analyze. For interpretation and analysis simplification, both RVV and SVE assembly codes are shown in Figure 2.6.

But first, let us discuss the primary purpose of vectorization. As already mentioned, SIMD architectures lean on DLP to increase computation performance, which is ultimately achieved by processing data in parallel and consequently reducing the total loop iterations. For instance, having fewer iterations will schedule fewer instructions and diminish the speculation cost, which is more prominent in SIMD processing. Additionally, having fewer loop iterations is usually beneficial, up to a point where overhead instructions are not removed or even added. Furthermore, works on decoupled access-execute architectures that physically separate memory-accessing and memory-consuming instruction promote the decoupling of the mechanisms of memory access from the processing ones. Outrider [19], is one decoupled access-execute architecture that shows high tolerancy to memory access latency while using a low complexity in-order microarchitecture. In specific, Outrider takes leverage of the decoupling to reduce the latency. Moreover, DeSC (Decoupled Supply-Compute) [31] is another work based on the decoupled access-execute architecture that fully decouples the processing from the program control, DeSC shows an average of 2.04x speedup over a baseline out-of-order single-chip multiprocessor. In sum, there are energy efficiency, area and performance advantages in separating the memory access and the processing part, thus reducing latency and diminishing the undesirable overheads. For this same reason, the comparison between scalable SIMD extensions targets the aforementioned undesirable overheads directly.

For this same reason, the comparison between scalable SIMD extensions targets the undesirable overheads directly. By observing in Figure 2.6 a) and b), it is concluded that there are more instructions than the minimum necessary inside the loops; in other words, the only instructions that contribute to data processing are the memory accessors (load/store) and the multiply and accumulate instructions. The memory address calculation and loop control are not responsible for any data processing; hence



C code

```
void saxpy( float * X, float * Y, int n) { for( int i = 0; i < n; i++) Y[i] = A*X[i] + Y[i]; }
```

Risc-V V Extension (a)

```
saxpy_:
  vsetvli a4, a0, e32, m8●
▲ vlw.v v0, (a1)
  sub a0, a0, a4 ■
  slli  a4, a4, 2 ■
  add a1, a1, a4 ■
▲ vlw.v v8, (a2)
▲ vfmacc.vf v8, fa0, v0
▲ vsw.v v8, (a2)
  add a2, a2, a4 ■
  bnez a0, .saxpy_
  ret
```

```
n → a0
A → fa0
X → a1
Y → a2
```

Arm Scalable Vector Extension (b)

```
saxpy_:
 mov x4, #0
 whilelt p0.b, x4, x3●
 ld1rs z0.d, p0/z, [x2] ●
.loop:
▲ld1s z1.b, p0/z, [x8, x4, lsl #2] ■
▲ld1s z2.b, p0/z, [x9, x4, lsl #2] ■
▲fmla z2.d, p0/m, z1.d, z0.d
▲st1s z2.b, p0, [x1, x4, lsl #2] ■
 incs x4 ■
 whilelt p0.d, x4, x3●
 b.any .loop
 ret
```

```
n → x4
A → x2
X → x8
Y → x9
```

Desired Solution (c)

```
saxpy_:
;Configurate data accesses, from address with n elements
config_data_access(v1, load, X, n)●
config_data_access(v2, load, Y, n) ●
config_data_access(v3, store, Y, n)●
fill_vector(v8, A) ●
.loop:
▲ vectormul  v9, v8, v1
▲ vectormul  v3, v9, v2
branch.not_complete v1, .loop   ;Branch until v1 is done
ret
```

Instruction Categorization: ●Configuration ▲Actual Work ■Address Calculation and Counters

Figure 2.6: Side by side code complexity comparison. At the top, the C SAXPY source code (linear memory copy), from whom the following are based. In the left, the RISC-V V extension assembly. In the center, the Arm SVE extension assembly. And in the right, a possible extension loosely based on RISC-V. Finally, at the bottom, the progression of the number of looped instructions is shown. The rightmost assembly, shows that a complete removal of looping unnecessary instructions is possible.

can be labelled as extraneous. As contrast, the branch instruction is not considered in any of these categorizations, as it is the core instruction that creates the loop. Concisely, it's reportable that more than 50% of the loop instructions are overhead. Importantly, as stated by Kuck et al. [32] and reaffirmed by Knijnenburg et al. [33], a computer program spends the majority of computation time in a small portion of code, namely, in the loops. As it is, the computation paradigm as changed significantly, but still, data processing times remain concentrated inside the computational kernels. Taking back to the example of the SAXPY kernel, to exacerbate the subject, the example shown here is rather simple. In contrast, cascaded loops would add further overhead in loop control and address calculations.

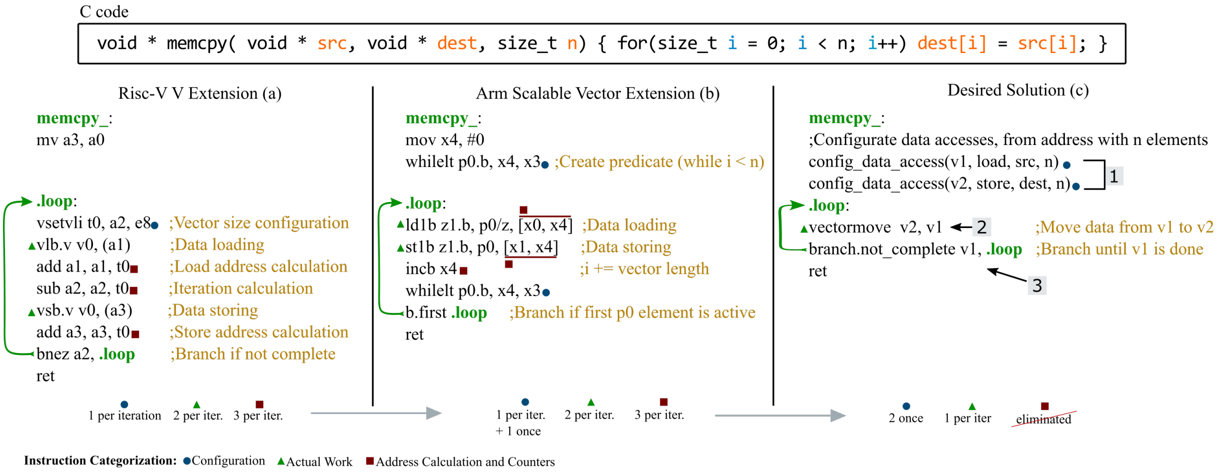
With this in mind, Figure 2.6 c) details a solution to remove overhead entirely. The pseudo-code presented can be split into the following 3 phases: Configuration, Execution, Control.

1 In the first phase, the data is configured, and three data accesses are coupled with three register vectors.

2 Subsequently, the data is processed, provided that the registers already have the necessary data.

3 Finally, the execution process repeats until all of the configured input is processed.

In addition to the SAXPY kernel example, in Figure 2.7 another example is deptícted, where a MEMCPY kernel is compared between RVV and SVE, in which an according result is obtained.



C code

```
void * memcpy( void * src, void * dest, size_t n) { for(size_t i = 0; i < n; i++) dest[i] = src[i]; }
```

Risc-V V Extension (a)

```
memcpy_:
mv a3, a0

.loop:
 vsetvli t0, a2, e8●  ;Vector size configuration
▲vlb.v v0, (a1)       ;Data loading
 add a1, a1, t0■      ;Load address calculation
 sub a2, a2, t0■      ;Iteration calculation
▲vsb.v v0, (a3)       ;Data storing
 add a3, a3, t0■      ;Store address calculation
 bnez a2, .loop       ;Branch if not complete
 ret
```

Arm Scalable Vector Extension (b)

```
memcpy_:
mov x4, #0
whilelt p0.b, x4, x3● ;Create predicate (while i < n)

.loop:
▲ld1b z1.b, p0/z, [x0, x4]■ ;Data loading
▲st1b z1.b, p0, [x1, x4]■   ;Data storing
 incb x4■                   ;i += vector length
 whilelt p0.b, x4, x3●
 b.first .loop      ;Branch if first p0 element is active
 ret
```

Desired Solution (c)

```
memcpy_:
;Configurate data accesses, from address with n elements
config_data_access(v1, load, src, n) ●
config_data_access(v2, store, dest, n)●
.loop:
▲vectormove  v2, v1                     ;Move data from v1 to v2
 branch.not_complete v1, .loop          ;Branch until v1 is done
 ret
```

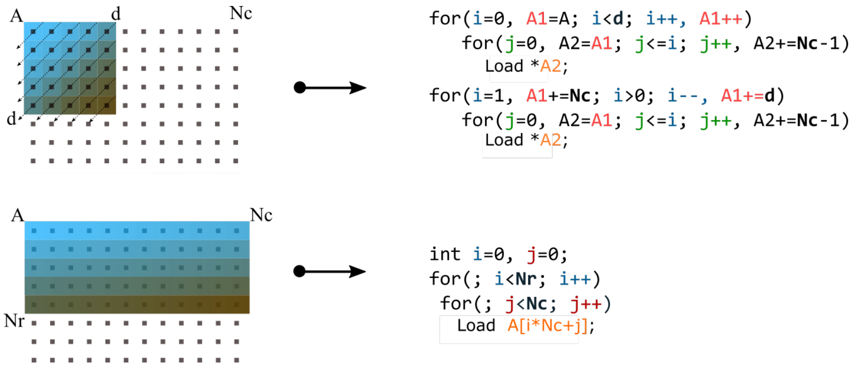
Figure 2.7: Side by side code complexity comparison. At the top, the C MEMCPY source code (linear memory copy), from whom the following are based. In the left, the RISC-V V extension assembly. In the center, the SVE extension assembly. And in the right, a possible extension loosely based on RISC-V. Finally, at the bottom, the progression of the number of looped instructions is shown. The rightmost assembly, shows that a complete removal of looping unnecessary instructions is possible. The RISC-V V example was gathered from [6]. Arm SVE example is based on [30, Listing 1.1].

In summary, by opposing to the SVE and RVV approaches, it is clear that this proposal is more straightforward and achieves the objective of removing overhead inside the loop. With the presented pseudo solution, it remains to develop further and to materialize into a closing solution. To be able to do that, the following section focuses on that by defining an adequate form to represent memory access patterns and in addressing the concept of data streaming.

2.3 Streaming Paradigm

To be able to apply streaming mechanisms to this work, methods of representing the memory access pattern are fundamental. Not only is it crucial to be able to express memory accesses, as it is essential to do it efficiently and effectively. In standard ISAs, describing memory accesses is achieved by directly describing the pattern with a set of standard instructions, before effectively issuing the load/store. However, other solutions can be devised that attain similar solution with a lower overhead.

As an example Figure 2.8 compares the visual and C language representation of two memory access patterns. These two examples show that while a easily visually represented memory access pattern, as



```
for(i=0, A1=A; i<d; i++, A1++)
   for(j=0, A2=A1; j<=i; j++, A2+=Nc-1)
     Load *A2;
for(i=1, A1+=Nc; i>0; i--, A1+=d)
   for(j=0, A2=A1; j<=i; j++, A2+=Nc-1)
     Load *A2;
```

```
int i=0, j=0;
for(; i<Nr; i++)
 for(; j<Nc; j++)
   Load A[i*Nc+j];
```

Figure 2.8: Visual and Loop memory access representations side by side. At the top, an uncommon memory access that loads a square in diagonal pattern. At the bottom, a common rectangular access.

the diagonal pattern, is not easily converted to a simple C for loop; in contrast, the rectangular example is easily representable in both C language and visually. As this chapter unfolds, an alternative method to represent memory accesses will be devised. In detail the for loop structure is the base of such pattern description method.

2.3.1 Memory Access Pattern Descriptors

Describing memory access patterns allows doing more efficient memory accesses, particularly on occasions where the access pattern is complex (but still regular). As proof, recent works advocate that the usage of descriptors enables the compiler to organize memory transfers more efficiently by matching data movement to the capabilities of the underlying hardware [16]. Furthermore, in the context of GPUs, a technique for improving memory performance is sustained on the analysis and characterization of memory access patterns in loop bodies, which achieved significant speedups [34]. These works motivate the use of efficient memory access pattern descriptors by allowing for better organized memory accesses and increased performance.

A) Simple Pattern Description

To allow such a description of memory access patterns, we can start by defining a core component of the description. To do so, we can refer to the works of López-Lagunas et al. [35] where a stream description is achieved through defining a tuple of parameters (Start Address, Offset(), Stride, Span, Skip, Type, Element_Count), defined as follows :

- Start Address is the memory address of the first element of the data stream.

- Offset() represents a user-defined function that computes the displacement of the next stream record from the base address.

- Stride is the spacing between two consecutive stream record elements. The units for the spacing are in data elements.

- Span indicates how many elements are gathered into the stream record before the skip displacement is applied.
- Skip is the displacement in data elements that is applied between groups of Span elements.
- Type indicates how many bytes are in each data element. For example, 8-bit data is associated with a type value of one, 16-bit of data associates a value of two and so on.
- Element_Count indicates how many stream elements are in the data stream.

While this description provides a way to describe many memory access patterns, it is considerably complex while limiting the set of memory access patterns that can be efficiently described. Neves et al. works [3, 18, 36], proposed an alternative representation that makes use of a hierarchy of simpler descriptors. The representation methods were iterated throughout the authors works, the latest and most optimized version is detailed here. The authors show that by combining simpler descriptors in a multi-dimensional tree, it is possible to describe virtually any regular memory access pattern, independently of their complexity. The key idea is to provide a affine representation of the address sequence, for any n-dimensional pattern. Neves et. al propose that any address sequence can be represented with the following model:

$$y(X) = y_{base} + \sum_{k=0}^{dim_y} x_k \times stride_k, \ \textbf{with} \ x_k \in [\alpha_k, \beta_k], \ X = \{x_0 \ldots x_{dim_y}\} \tag{2.1}$$

Where each stream access $y(X)$ is described by the base address y_{base}, and dim_y pairs of sizes (x_k) and strides ($stride_k$). With this representation, it is possible to describe any pattern, however, more complex and irregular patterns would result in an unreasonable number of descriptors.

To create a memory access descripition, Neves et al. proposed the specification in Figure 2.9. This descriptor specification is composed of a context header, base descriptor and modifier chain parts. The base descriptor contains the stream identification, the base address, number of configured dimensions and the number of modifier chains. Particullarly, each dimension is described by the pairs size $xsize$ and stride $xstride$, in similarity with the representation model in Equation 2.1.

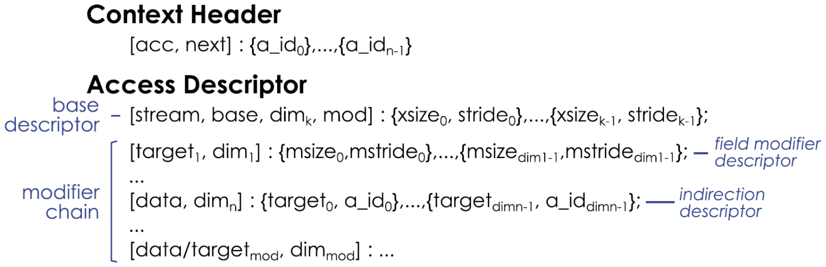
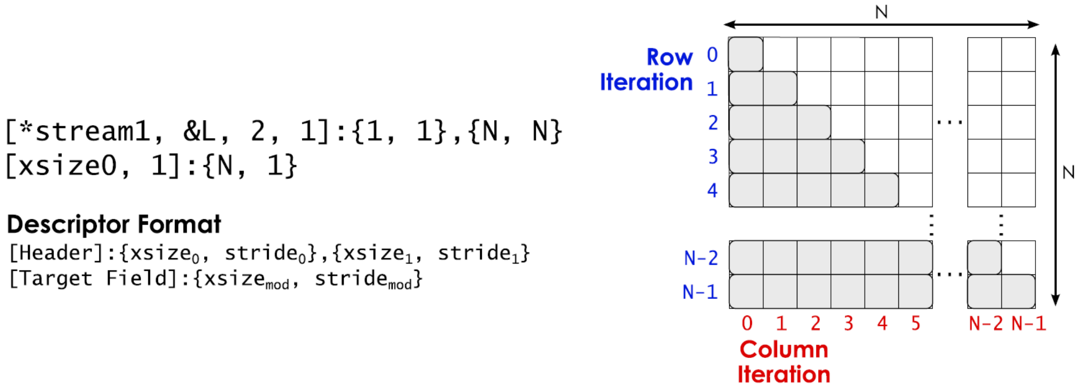

Context Header
[acc, next] : {a_id$_0$},...,{a_id$_{n-1}$}
Access Descriptor
base descriptor
[stream, base, dim$_k$, mod] : {xsize$_0$, stride$_0$},...,{xsize$_{k-1}$, stride$_{k-1}$};
modifier chain
[target$_1$, dim$_1$] : {msize$_0$,mstride$_0$},...,{msize$_{dim1-1}$,mstride$_{dim1-1}$};
field modifier descriptor
...
[data, dim$_n$] : {target$_0$, a_id$_0$},...,{target$_{dimn-1}$, a_id$_{dimn-1}$};
indirection descriptor
...
[data/target$_{mod}$, dim$_{mod}$] : ...

Figure 2.9: Descriptor specification proposed by Neves et al. [3].

While this is sufficient to describe any pattern by using N dimensions, this would lead to an enourmous usage of dimensions that would not be possible in any real implementation. However, Neves et al. proposes that a modifier chain can be used to add non-linear (e.g. dimension size changes) modifications to the base descriptor. The authors propose two types of modifiers: a field modifier descriptor and an indirection descriptor. In detail, the field modifier descriptor can be used to modify a field of the base descriptor, an example that better explains the usage of field modifiers is depicted in Figure 2.10. In this example, a 2-D pattern is configured with 1 (row, $xsize_0$) by N (columns, $xsize_1$) of size. The modifier will increase the size of the first dimension (the target $xsize_0$), the applied modification will increase $xsize_0$ by the modifier stride (1). This increment is repeated N (modifier size) times, and modifies at the same time and rate as the second dimension iterates.



```
[*stream1, &L, 2, 1]:{1, 1},{N, N}
[xsize0, 1]:{N, 1}
```

Descriptor Format

```
[Header]:{xsize0, stride0},{xsize1, stride1}
[Target Field]:{xsizemod, stridemod}
```

Figure 2.10: Example of a triangular matrix access represented using the base descriptor and one field modifier. Retrieved from [3].

Furthermore, the indirection descriptors allow for diferent descriptions to be linked through a data-indexation dependency e.g. $B[A[i]]$. Finally, the context header is used for stream management, and does not affect the representations.

Neves et al. work shows that a multiple memory access patterns can be represented by relying on the presented descriptor specification, Figure 2.11 depicts some of the representable patterns.

In addition to this work, other solutions that employed descriptors were proposed. Specifically, S. Paiagua et al. proposed a data streaming accelerator framework that made use of stream descriptors, being able to increase the available memory access bandwidth by relying on streams [37].

2.3.2 Scalar Stream-Based ISA

Although streaming is not a widely used concept in the CPU Instruction Set Architecture, there is one work that employs streaming to address the inefficiency in memory accesses. In fact, works such as Imagine Stream Processor [38], RSVP [39], Q100 [40], Stream-dataflow acceleration [41], VEAL [42], CoRAM++ [43] employ stream abstractions but none target general-purpose out-of-order cores and the associated speculation mechanisms. Besides, there is also the Memory Access Dataflow [44], a

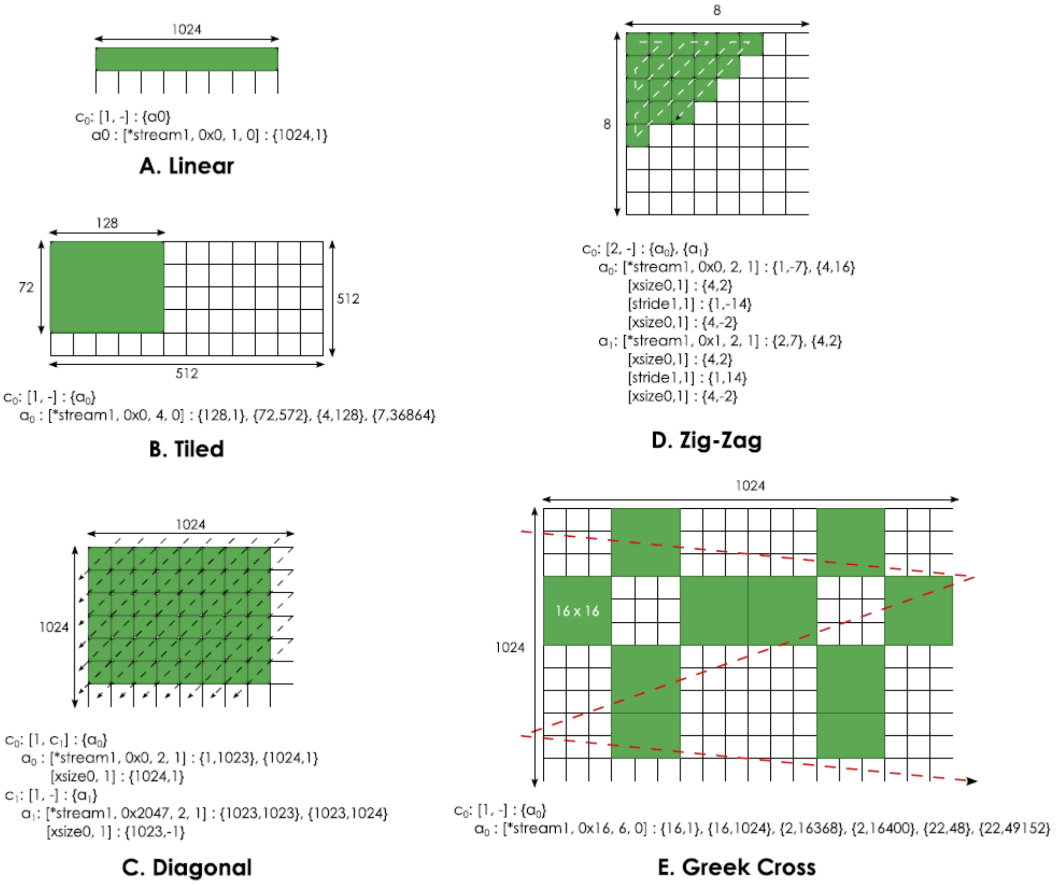

1024
c0: [1, -] : {a0}
a0 : [*stream1, 0x0, 1, 0] : {1024,1}
A. Linear
128
72
512
512
c0: [1, -] : {a0}
a0 : [*stream1, 0x0, 4, 0] : {128,1}, {72,572}, {4,128}, {7,36864}
B. Tiled
8
8
c0: [2, -] : {a0}, {a1}
a0: [*stream1, 0x0, 2, 1] : {1,-7}, {4,16}
[xsize0,1] : {4,2}
[stride1,1] : {1,-14}
[xsize0,1] : {4,-2}
a1: [*stream1, 0x1, 2, 1] : {2,7}, {4,2}
[xsize0,1] : {4,2}
[stride1,1] : {1,14}
[xsize0,1] : {4,-2}
D. Zig-Zag
1024
1024
c0: [1, c1] : {a0}
a0 : [*stream1, 0x0, 2, 1] : {1,1023}, {1024,1}
[xsize0, 1] : {1024,1}
c1: [1, -] : {a1}
a1: [*stream1, 0x2047, 2, 1] : {1023,1023}, {1023,1024}
[xsize0, 1] : {1023,-1}
C. Diagonal
1024
1024
16 x 16
c0: [1, -] : {a0}
a0 : [*stream1, 0x16, 6, 0] : {16,1}, {16,1024}, {2,16368}, {2,16400}, {22,48}, {22,49152}
E. Greek Cross

Figure 2.11: Multiple examples of representable pattern using Neves et al. proposed pattern description methods [3].

reconfigurable front-end/memory-fetch engine for accelerators and SIMD units, which is not an extension to the ISA. It is an in-core stream mechanism that is not part of the ISA, and it directly interferes with the core capabilities (e.g., control of the core power).

Additionally, Wang et al. [4] proposed a stream-based memory access specialization for general purpose processors. The authors identified the structure of the memory access patterns in the Cortex-Suite [45] and SPEC CPU 2017 [46] benchmark suites, realizing that much improvement is extractable by employing streams. Figure 2.12, presents the main results of the authors' work, showing that, on avaregar, more than half of dynamic accesses across both benchmark suites can be considered as streams, with the simplest affine streams being the most common.

Additionally, Wang et al. leverage the concept of streaming to remove some of the overhead from the loop body [4]. By decoupling the memory access from the CPU core, a limited form of decoupled access execute architecture is achieved, which is able to hide some of the latency involved in the memory access process [17, 47]. In terms of results, the work achieved speedups in the order of 1.67x in comparison with a baseline OOO core. While the baseline OOO core provided with a stride prefetcher achieves 1.22x in speedup. It is noteworthy that this work makes use of streams with scalar registers and does

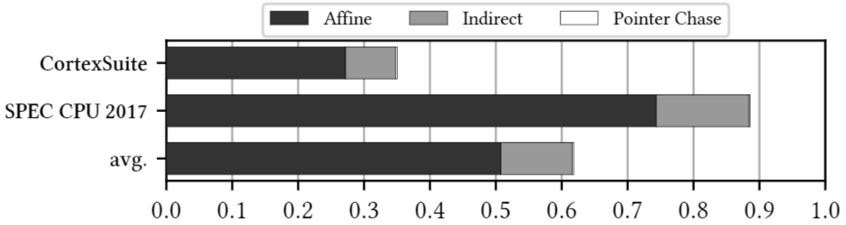


Figure 2.12: Memory access pattern decomposition percentage in CortexSuite and SPEC CPU 2017 benchmark suites [4].

not consider vectorial or SIMD architecture extensions.

In this work, Wang et al. identified the following three opportunities are possible to attain [4]:

- Stream-based prefetching: With the knowledge of the access patterns and the relation to the control flow, it is possible to infer the timing of the prefetching based on how far ahead the prefetcher is related to the core execution. Thus leading to more effective prefetching.
- Stream-decoupling: By following the principle of the decoupled access execute [47], a streaming engine can generate memory requests, removing the logic from the core and allowing the core to focus on the processing instructions. There are several potential advantages of doing so, including diminishing the latency of memory accesses, coalescing the memory accesses from related streams, and even reducing the amount of cache pollution.
- Specialized cache policies: Various cache policies could take advantage of the knowledge of the memory access pattern. Wang focuses on bypassing the cache hierarchy based on each stream footprint. To be able to explore these opportunities, Wang et al. proposed Stream-ISA extensions and potential microarchitecture support.

The ISA extensions focus on allowing the configuration and control of the streams. In specific, a **stream_cfg** instruction enables the configuration of new streams. This instruction is responsible for defining a stream through the following parameters: type, pattern, dependencies, and starting address. Additionally, a **stream_step** iterates the stream, doing so iterates all the dependent streams. Consequently, a register can be read multiple times before a step instruction changes the state of the streams. Finally, a **stream_end** instruction deallocates the corresponding streams, terminating the access. This work also implemented compiler support to the ISA extensions. The compiler support is separated into three phases: recognizing stream candidates, then selecting the qualified candidates, and finally generate the code.

Additionally, the proposed set of microarchitecture extensions contains changes to the core and the creation of a stream engine. From these extensions the following sum up the architecture changes:

- Inside the core, the changes include an iteration map that maintains the count of each stream iterations. In specific, the iteration map is controlled by the "stream_cfg" and "stream_step" instructions.

- The stream engine is responsible for generating the memory accesses and the management of each stream.
- A set of queues is included to allow decoupling between the stream engine and the core pipeline.

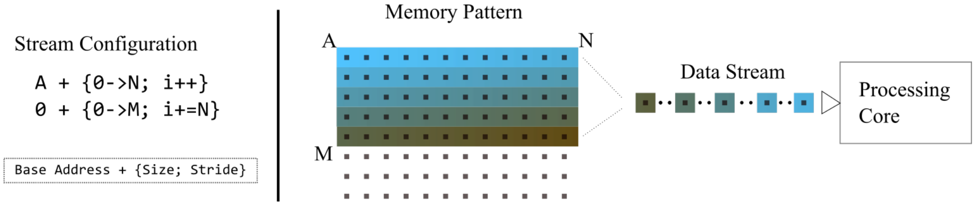
2.4 Summary

As stated throughout this chapter, the modern vectorial extensions (RVV and SVE) leave space for improvement, both in terms of latency, as in removing instructions that do not contribute to the data processing. It was not yet merged the concepts of SIMD execution with memory access decoupling. By doing so, it is believed that it is possible to decrease the processing times inside the CPU. Additionally, it is possible to increase the number of previously vectorizable codes, as the memory addressing is executed outside the instruction window. Finally, the work of Wang et al. [4] shows that this is a promising research path.

3 Pattern Description and Stream Fundamentals

Taking into consideration the state of the art in SIMD instruction set extensions and streaming mechanisms, it is clear that new scalable streaming vector extension should be devised and explored to improve the performance of modern high-perfomance computing systems. However, before defining the instruction set extension proposed in this thesis, it is first necessary to define the concept of a stream, in the context of this thesis, and the associated mechanisms to allow manipulating (by loading and storing) data.

A stream is a continuous flow of data, typically one having a constant or predictable pattern at the moment it starts executing. Hence, while the pattern, length or even data-type may not be known at compile-time, it must be defined through a set of variables which value can be computed when the stream is marked to start executing. Additionally, the streams are guaranteed to process (load/store) sequentially and in-order. A basic example of a streaming access to a 2-D, matrix-like, memory pattern is shown in Figure 3.1.



```
A + {0->N; i++}
0 + {0->M; i+=N}
```

```
Base Address + {Size; Stride}
```

Figure 3.1: Example of a simple 2-D memory access with streaming. In the left, the stream configuration with two dimensions. On the right, the representation of the pattern in memory, alongside with the delivery of data to the processing core.

To represent a stream of data, the concept of a descriptor is herein used. Hence, as in Nuno Neves, et al [36] a descriptor must describe a pattern of memory accesses including starting address, word-length (in bits) and size. However, additional fields can be added to represent more complex patterns (e.g., 1-D, 2-D, 3-D, etc), such as multidimensional strides. Descriptors can also be made hierarchal or dependent to allow representing more complex memory access patterns (e.g., diagonal, hexagonal).

However, there is a natural balance between the complexity of a single descriptor and its capacity to easily represent different patterns. Hence, while simple descriptors require more complex hierarchical structures to achieve complex memory access pattern representations, more complex descriptors use more space, which must be encoded into a limited set of instructions.

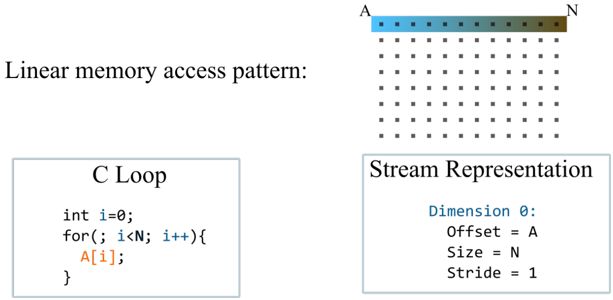
Hence, this work adopted a similar solution to Nuno Neves, et al [36], although simplifying some structures to more closely resemble the structure of typical for loops.

To explain the adopted memory access representations. this chapter starts by presenting simple (linear) memory access patterns, before introducing descriptors modifiers that allow representing more complex structures.

3.1 Introduction to linear patterns - Dimensions

In section 2.3, different techniques to describe memory access patterns were presented, each with different complexities and allowing to cover different application scenarios. However, while on application-specific systems complex and specially tailored solutions can be used, for general-purpose computing, the used techniques must target the general use case and represent a good balance between pattern complexity and coverage. Hence, this chapter focus on tailoring these techniques towards integration in an arbitrary ISA extension. To do so, it is worthy to recall that one single linear dimension can be described by a set of three parameters, namely **offset**, **size**, **stride**. Particularly, the **offset** parameter represents the base address in single dimension descriptions. However, for multidimensional descriptions the base address depends on the offsets and strides of all dimensions. Additionally, it is also necessary to know the data-type of the memory access, thus the parameter **data-type** is also present in the first dimension of any description. These parameters are what makes a descriptor, the minimal representation for a stream.

Figure 3.2 such pattern is represented. As it is clear, by using the three aforementioned parameters, it is trivial to represent such pattern. As it is expected, by adding more dimensions, it is simple to describe multi-dimensional memory accesses. To clarify, Figure 3.3a shows a simple 2-D memory patterns alongside the respective stream representation. As stated before, higher dimensionality can be represented by adding more dimensions, even so, in these examples only up to two dimensions are shown as it simplifies visualization. In detail, each additional dimension can be configured by adding one descriptor on top, using three parameters, offset, size and stride. Any 2-D description based on two linear descriptors is represented by the following function:



```
int i=0;
for(; i<N; i++){
  A[i];
}
```

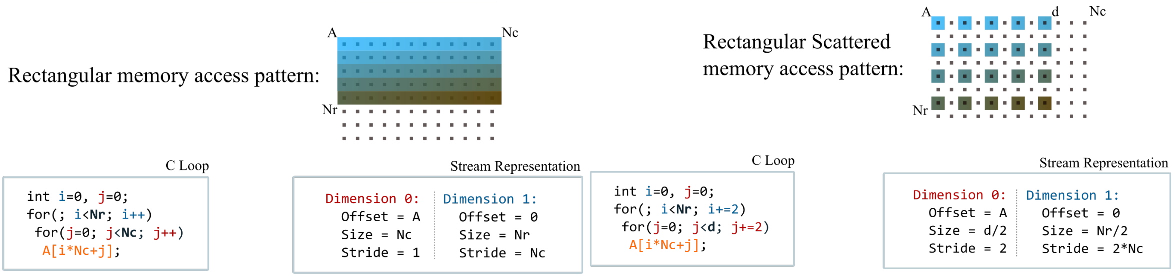
```
Dimension 0:
  Offset = A
  Size = N
  Stride = 1
```

Figure 3.2: Linear memory access. In top, a visual representation of the pattern. In the bottom left, a C loop representing the memory access. In the bottom right, the respective stream representation.

$$i \in \{0 \dots E_i\}, j \in \{0 \dots E_j\}; \qquad Y(i,j) = O_j + (O_i \times S_i) + (S_j \times j) + (S_i \times i); \tag{3.1}$$

Where, E_x, O_x and S_x represent, respectivelly, size, offset and stride for a given dimension x. Generally, the offset of dimension j is the base address of the description, however, the base address of any stream description is obtained from all the dimensions' offsets and strides (e.g. $BA = O_j + (S_i \times O_i)$).

To achieve more intricate memory patterns, it is possible to cascade up to 8 descriptors, the maximum number of descriptors will be discussed later, as it is constrained by the hardware implementation complexity. In order to support descriptions that need more than 8 descriptors would require one description to be coupled with another, however, it is highly unlikely that any application that would require such description complexity would highly benefit from vectorization.


Rectangular memory access pattern:
A
Nc
Nr
C Loop
Stream Representation
Rectangular Scattered memory access pattern:
A
d
Nc
Nr
C Loop
Stream Representation

```
int i=0, j=0;
for(; i<Nr; i++)
 for(j=0; j<Nc; j++)
  A[i*Nc+j];
```

```
Dimension 0:   | Dimension 1:
  Offset = A   |   Offset = 0
  Size = Nc    |   Size = Nr
  Stride = 1   |   Stride = Nc
```

```
int i=0, j=0;
for(; i<Nr; i+=2)
 for(j=0; j<d; j+=2)
  A[i*Nc+j];
```

```
Dimension 0:   | Dimension 1:
  Offset = A   |   Offset = 0
  Size = d/2   |   Size = Nr/2
  Stride = 2   |   Stride = 2*Nc
```

(a) Non-strided (unitary stride) pattern. In the top, a visual representation of the pattern. In the bottom left, a C loop representing the memory access. In the bottom right, the respective stream representation.

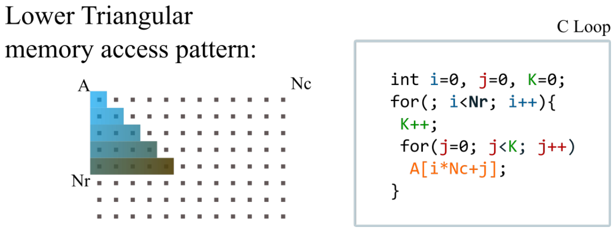
(b) Non-unitary strided pattern. In top, a visual representation of the pattern. In the bottom left, a C loop representing the memory access. In the bottom right, the respective stream representation.

Figure 3.3: Description of regular 2-D examples, with unitary and non-unitary strides. Alongside with the respective C language equivalent and a visual representation of the pattern.

Other more complex patterns can be represented by pooling only over a pattern of elements. As an example, Figure 3.3b pictures a possible rectangular access with non-unitary stride. At this respect the adopted memory access pattern description closely follows the solution adopted by Neves et al. [18]. However, due to encoding ISA restrictions an arbitrarly number of arguments cannot be used, as it would increase the number and complexity of the configuration instructions.

3.2 Complex accesses - Modifiers

To describe more complex patterns, a set of modifiers will be introduced, which allow manipulating each of the fields on a given n-D pattern. To illustrate this issue Figure 3.4. It is direct to infer that by relying solely on linear patterns, it is difficult, to define such access. In fact, by analyzing the behavior



```
int i=0, j=0, K=0;
for(; i<Nr; i++){
 K++;
 for(j=0; j<K; j++)
  A[i*Nc+j];
}
```

Figure 3.4: Lower triangular memory access. In the top, a visual representation of the pattern. In the right, a C loop representing the memory access.

of the corresponding C Loop, it is observed that for each outer loop iteration, the inner loop parameters are modified. The described behaviour can be achieved using a new descriptor named modifier, which is able to modify a parameter of a dimension descriptor.

As a starting point, the triangular access was decomposed into two parts: a bi-dimensional stream that is already describeable and a non-linear behaviour that is necessary to simplify the overall description, and depicted in Figure 3.5. To create a modifier with capabilities to modify the another dimension's properties, the following parameters are explicit candidates:

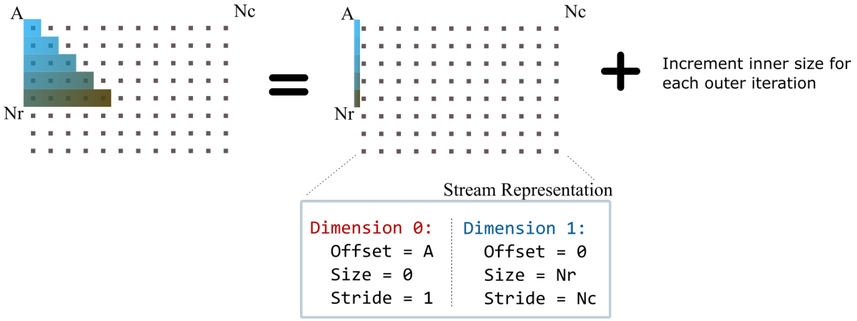
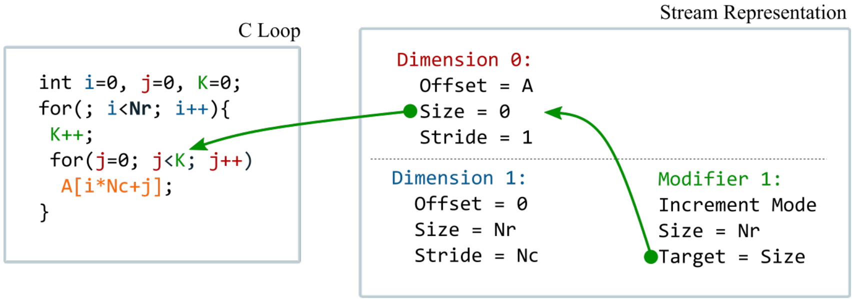


Figure 3.5: Lower triangular memory access deconstructed into a bi-dimensional access and a non-linear behavior. In the top, a visual representation of the pattern. In the bottom, a stream representation of the linear pattern.

- **Target**: The parameter to modify. One of offset, size, and stride.
- **Behavior**: The type of modification, one of increment and decrement. In the lower triangular example only the increment is needed.
- **Displacement**: The constant ammount of increment or decrement that can be applied - e.g. a displacement of two with increment behaviour will increment the target parameter by 2 in each modifier iteration.
- **Size**: The total number of iterations for which the target parameter is modified. In more irregular patterns, it can be of use to define a modifier that stops at a given iteration.

By relying solely upon these parameters, it is now possible to completely define the lower triangular pattern. To clarify, Figure 3.6 presents the final description that conveys the addition of a modifier, which for any outer loop iteration, will modify the size parameter of the inner loop. In specific, the modifier described here operates always with a non-changing configuration, hence the specific nomeculature being static modifiers.

Note that there is a first configuration pass on the outer dimensions before the start of the inner dimension (in agreement with a for loop mechanism), i.e., the modifier will be applied every time dimension 1 (outer) is iterated and, in the first time, it is applied right before the inner dimension iteration. Each modifier is coupled with a single dimension, which means that it is iterated at the same time as said dimension, however, the modifier will target, i.e. modify, the dimension immediately below. Additionally,



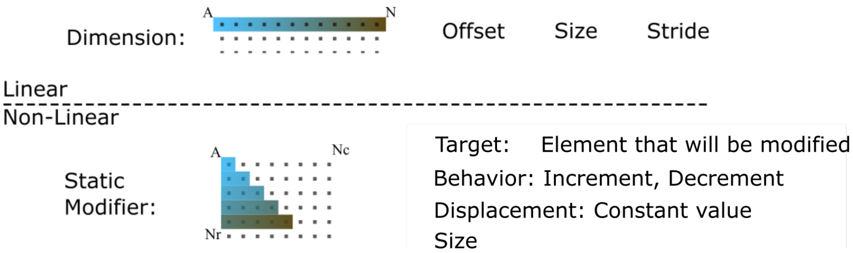
```
int i=0, j=0, K=0;
for(; i<Nr; i++){
 K++;
 for(j=0; j<K; j++)
  A[i*Nc+j];
}
```

```
Dimension 0:
  Offset = A
  Size = 0
  Stride = 1

Dimension 1:
  Offset = 0
  Size = Nr
  Stride = Nc

Modifier 1:
Increment Mode
Size = Nr
Target = Size
```

Figure 3.6: Lower triangular memory access fully described by the usage of dimensions (linear) and dimension modifiers (non-linear). In the left, the correspondig C loop. In the right, the stream representation of the memory pattern. This figure details the connections between the modifier and the respective target.

to constrain the maximum complexity of a stream description, only one modifier can be coupled with each dimension, meaning that the maximum sized description can be composed of 8 dimensions and 7 modifiers. The first dimension cannot have a modifier, as there is no dimension below.

To keep track of the ISA description capabilities, Figure 3.7 summarizes the set of introduced descriptors.

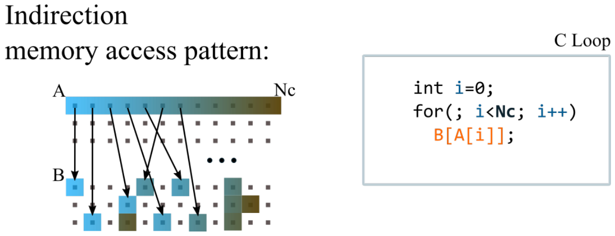


Dimension:
A
N
Offset
Size
Stride
Linear
Non-Linear
Static
Modifier:
A
Nc
Nr
Target: Element that will be modified
Behavior: Increment, Decrement
Displacement: Constant value
Size

Figure 3.7: Descriptors summary containing linear descriptors and static modifiers.

3.3 Indirection accesses - Indirection Modifiers

In processing kernels, it is common to find the use of indirect memory accesses, such as using one array to index another's data. Many applications, particularly those leading with sparse or graph data, are characterized by indirect memory access patterns, such as $B[A[i]]$ or $C[B[A[i]+k]]$ (see also the example in Figure 3.8). Consequently, this functionality must also be present in order to achieve comprehensive descriptors. Hence, by coupling both memory accesses together, it is simple to describe



```
int i=0;
for(; i<Nc; i++)
  B[A[i]];
```

Figure 3.8: Indirection memory access. In the left, a visual representation of the indirection pattern where the linear memory access starting in A produces pointers to index the memory access with offset B. In the right, a C loop representing the memory access.

the indirect patterns, since the access $B[A[i]]$ can be described as $*(B+A[i])$. Thus, the description needs to use the values of A[i] to modify the parameter offset of the pattern B dynamically. This can be achieved by relying on the previously defined static modifiers, only with the difference that the source value (displacement) is now dynamic. In particular, considering that the displacement value is now obtained from a stream, the incrementation and decrementation behaviours would limit the overall potential of the description. As a consequence, there five different behaviours in a dynamic modifier:

- Add: Adds the dynamic displacement to the target.
- Sub: Subtracts the displacement to the target.
- Inc: Adds the displacement to an incrementing counter from the previous iterations, then sums the value of the counter to the target.
- Dec: Analogous to the incrementing process, only it decrements.
- Set: Sets the target value to the value given by the origin stream.

In sum, a dynamic modifiers is composed of 4 parameters: **target**, **behaviour**, **origin stream** and **size**. Taking into account that the size may directly depend on the origin stream total size, this parameter is not always necessary and may be derived from the origin stream. Particularly, in the case where the origin stream description is composed of static or dynamic modifiers, it may be very difficult to calculate the origin stream size, in this case it is advisable to allow the size parameter to be derived.

By putting it all together into a pattern description, the result for a given pattern $B[A[i]]$ is depicted in Figure 3.9. As a note, the indirection modifier is executed the same number of times as stream A, and each time it iterates, the value offset of the stream B is summed to the value given by stream A. Additionally, by choice, the innermost dimension of the stream B was defined with size 1. However, that is not a requisite, and the dimension is open to different configurations.

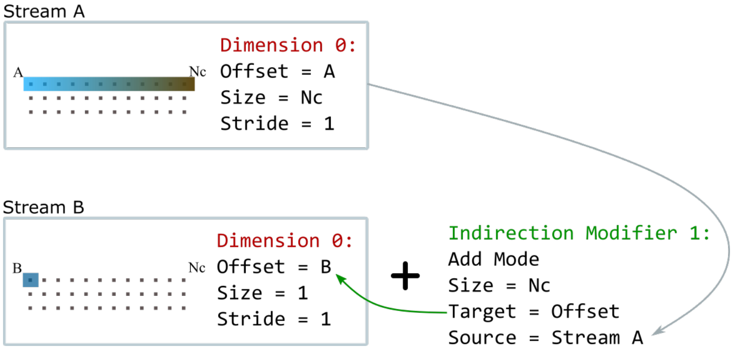
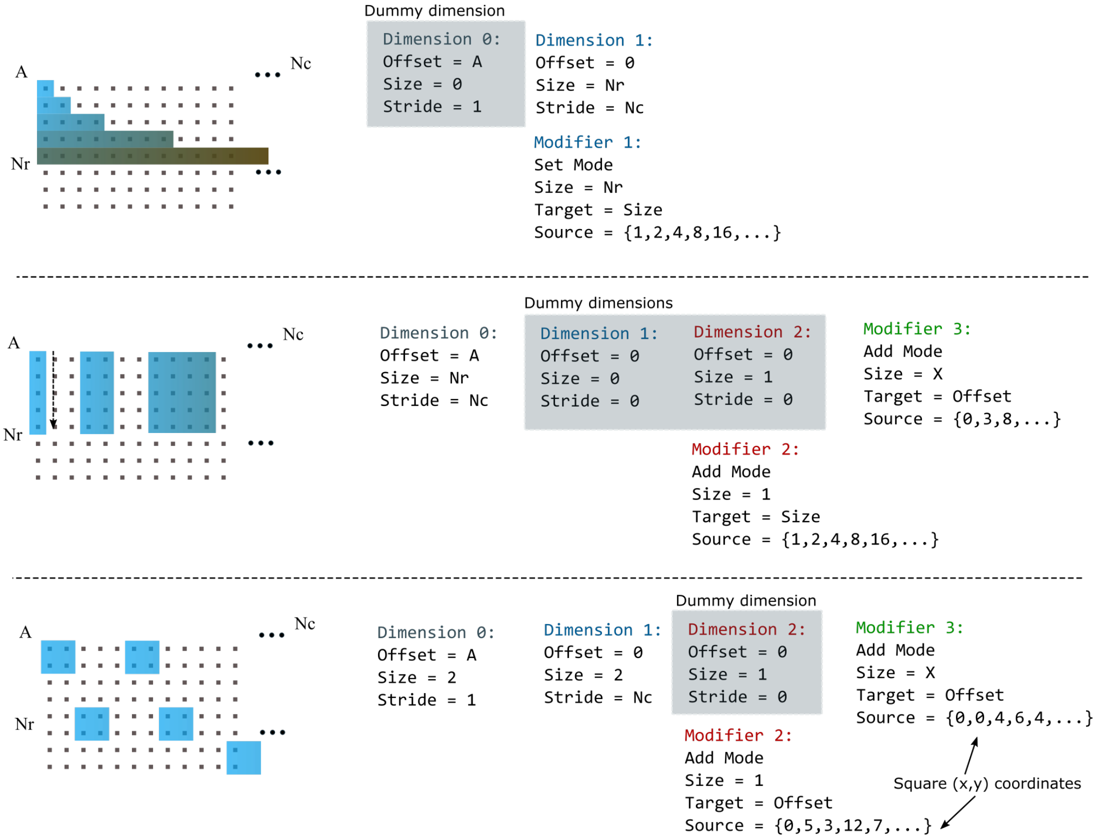


Figure 3.9: Indirection memory access separated into accesses A and B. In the top, the representation of he linear memory access with offset A. In the bottom, the linear memory access with offset B. The connection of the streams is done through the indirection modifier.

Finally, to showcase some possible descriptions, in Figure 3.10 some of the potential of dynamic modifiers is depicted. To simplify the representation, all streams assume a origin stream with infinite data



```
Dimension 0:      Dimension 1:
Offset = A        Offset = 0
Size = 0          Size = Nr
Stride = 1        Stride = Nc

                  Modifier 1:
                  Set Mode
                  Size = Nr
                  Target = Size
                  Source = {1,2,4,8,16,...}
```

```
Dimension 0:   Dimension 1:   Dimension 2:   Modifier 3:
Offset = A     Offset = 0     Offset = 0     Add Mode
Size = Nr      Size = 0       Size = 1       Size = X
Stride = Nc    Stride = 0     Stride = 0     Target = Offset
                                             Source = {0,3,8,...}

                              Modifier 2:
                              Add Mode
                              Size = 1
                              Target = Size
                              Source = {1,2,4,8,16,...}
```

```
Dimension 0:   Dimension 1:   Dimension 2:   Modifier 3:
Offset = A     Offset = 0     Offset = 0     Add Mode
Size = 2       Size = 2       Size = 1       Size = X
Stride = 1     Stride = Nc    Stride = 0     Target = Offset
                                             Source = {0,0,4,6,4,...}
                              Modifier 2:
                              Add Mode
                              Size = 1
                              Target = Offset
                              Source = {0,5,3,12,7,...}
```

Figure 3.10: A showcase of possible, yet very complex, memory access patterns. Each pattern is represented with the respective stream configuration. All the configurations specify the data in the not represented source streams. Values marked with X represent undefined but reasonable values. In the left, a representation of a linear pattern with increased size. In the middle, a linear 1x5 access that is modified in horizontal offset and horizontal size. In the right a 2x2 pattern that is indexed using a pair of (x,y) coordinates.

that is meaningful for the example at hand. To guarantee that an overview of the available descriptors was not lost, in Figure 3.11 it is presented the summary of the defined descriptors.

3.4 Multi-decriptor organization

Returning to the works of Neves et al., presented in section 2.3, it was proposed a descriptor organization in which dimensions were already linked by being in the same descriptor. In spite of its inherent flexibility, such a method is not adequate for this work. In the work of Neves et al [36], each descriptor can define up to N dimensions. However, it is not possible to describe N dimensions with a single instruction, as the encoding space is limited. Taking this into account, it would be more suitable that the placement of each descriptor was determined implicitly with the usage of multiple consecutive instructions. In the other hand, there is no need to use a descriptor organization so complete and flexible, as the majority of the patterns found in high-performance computing applications are regular, or can be represented by a set of more than one stream. Hence, this work defines a new descriptor organization that is more simple, although deliberately less capable.

The new organization method herein proposed is thus based on a list organization. In the such

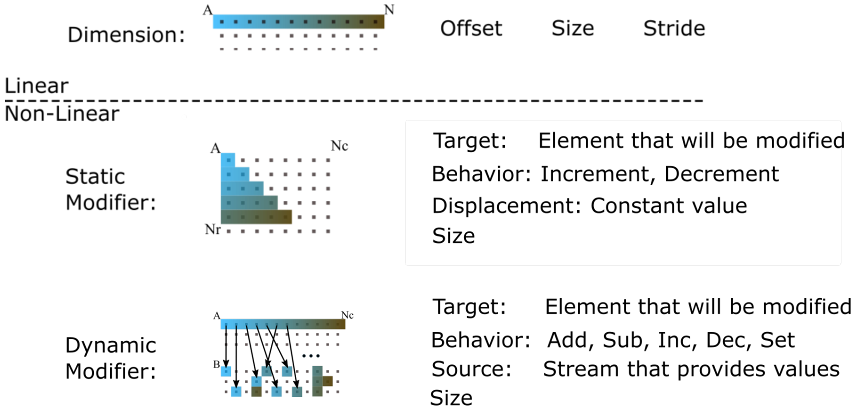

Dimension:
A
N
Offset
Size
Stride
Linear
Non-Linear
Static
Modifier:
A
Nc
Nr
Target: Element that will be modified
Behavior: Increment, Decrement
Displacement: Constant value
Size
Dynamic
Modifier:
A
Nc
B
Target: Element that will be modified
Behavior: Add, Sub, Inc, Dec, Set
Source: Stream that provides values
Size

Figure 3.11: Descriptors summary containing linear descriptors and dynamic modifiers.

organization, each list node can be composed of up to one dimension and one modifier. The first node (head node) corresponds to the lower level loop, i.e. the inner loop, while the tail node is relative to the outer loop. Additionally, the first node must only be composed by a dimension (no modifiers allowed). Additionally, any node that is composed of a modifier needs to have a lower level node with a dimension. To clarify, in Figure 3.12 a set of possible and not-possible lists is depicted. It should be noted that the

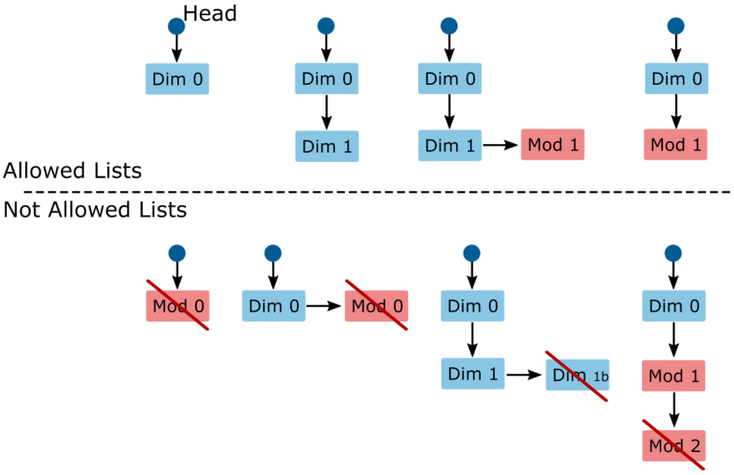


Figure 3.12: A set of possible combinations for the list nodes. On the top, the allowed combinations. On the bottom, the combinations that are not allowed to be used.

majority of these limitations are derived from the modifiers concept, in which a modifier must modify a target dimension. Hence, any modifier that has no lower-level dimension cannot be employed.

To allow for the descriptors to be ordered, an ordenation method is needed. Leveraging on the list concept, any node in a list is connected to at least one other node. Additionally, in a unidirectional list, any node connects to the following (with exception on the last). Consequently, if a dimension is inserted into an empty list, the following dimension will be implicitly connect to the first one. Hence, by filling a list with the consecutive dimensions will result in a list with the dimensions implicitly ordered and connected. In particular, the modifiers are linked to the dimensions, as such any modifier is only connected to the previously defined dimension. Undoubtedly, all instructions in a CPU start in order, so it is always possible to guarantee the descriptors order. Additionally, modifiers and dimensions must be distinguishable; fortunately, the format of each descriptor is different. Hence, by defining the descriptors and modifiers in order, it is simple to construct a tree that relates each node and its components.

In fact, it would be possible to link any ammount of modifiers to one arbitrary dimension, however, allowing only one modifier is not currently considered as an application porting limitation and simplifies possible implementations.

3.5 Summary

This chapter introduced the fundamental concepts of streaming and pattern description on top of which an instruction set will be devised.

The core element of a stream description is the descriptor, to describe streams with an higher level of memory access complexity, multiple descriptors can be chained into a complex description. Particularly, a descriptor can have three forms, a dimension, a static modifier, and a dynamic modifier. While a dimension describes a linear access with a specific size, stride and offset, the usage of a modifier allow for these parameters to be changed in a per-iteration basis and with resource to the data from other streams. Thus, by chaining up to 8 dimensions and 7 modifiers, it is possible to describe a vast assortment of HPC applications. While it is always possible to extend these capabilities, such as, allowing more descriptors, creating more complex modifiers or improving the descriptor chaining flexibility, we also need to consider the microarchitecture standpoint where more features and more complexity may correspond to challenging implementations that are not possible or viable.

Part I

Instruction Set Achitecture

4 Instruction Set Architecture Design

The process of designing the ISA extension is the most crucial part of the work, as the supporting microarchitectures will be affected by any mistake on the ISA definition. Consequently, the product of this chapter is based on state-of-the-art ISA architectures and extensions (Chapter 2). In the effort of creating a consistent ISA extension, the following set of requirements was defined before the ISA definition:

- RISC: The instruction set must have reduced size, but be comprehensive and generic.
- Scalability: The instruction set must be tailored for the scalability shown by the vector registers. Particularly, never constraining the maximum vector-length.
- Coherent: The extension should follow the principles of the base ISA.

Before it is possible to define an ISA extension, we must choose the base ISA for the extension. From the abundance of available instruction set architectures, only x86, Arm and RISC-V are viable candidates. First, the x86 ISA is the dominant ISA in the personal computer industry, as well as on datacenters and supercomputers. Secondly, the Arm ISA dominates the mobile device industry, and is now trying to make an important presence in datacenters. Finally, RISC-V is an open-source ISA that is very trending amongst computer technology industry leaders. Comparing RISC-V with x86 and Arm, one promptly concludes that due to the open-source nature of RISC-V, any future implementation would leverage from the absence of legal restrictions. Additionally, due to the more simple and less overloaded set of instructions, it is easier to develop work on top of RISC-V. Finally, RISC-V is very popular among academics, the target of this work. Additionally, previous studies compared side-by-side implementations of the Arm and RISC-V ISAs, by comparing, respectively, an ARM Cortex-A5 and a Rocket scalar core. Lee et al. [48] claim that the RISC-V implementation achieved a higher performance despite the lower usage of chip area. This final result cements the affirmation that RISC-V is, in fact, fit for realistic implementations. In summary, all the arguments point to RISC-V as being a suitable candidate for this extension base ISA.

Admittedly, the scalable vectorial extensions RVV and SVE (see section 2.2) are great candidates on which to apply the streaming concept of this work. However, at the time this work started, there were no available tools for SVE and RVV was still a minor draft. Also, none of them was created with streaming into consideration. Consequently, a new ISA extension was designed and is presented throughout this chapter.

At this respect, the architectural stated (register organization) is first presented, followed by the definition of the data processing and data transfer instructions. Subsequently, the stream configuration interface is detailed, which consists of the set of instructions that allow the configuration and management of streams. Finally, the encoding of the instruction set is outlined, followed by a summary of the chapter.

4.1 Architectural State

Since the extension is vectorial, it is necessary to define a set of vectorial registers. Moreover, some instructions will certainly require scalar registers (e.g. Loads and Stores). To cope with this, the considered scalar registers correspond to those defined by the base RISC-V ISA [27].

4.1.1 Vectorial Registers

By inspecting the state-of-art SIMD and vectorial architectures (Sections 2.1 and 2.2), it is clear that the number of vectorial registers is 32 in any state-of-the-art SIMD extension. Additionally, this number of registers has been used historically for the majority of the instruction sets. With 32 architectural registers, the encoding of an operand takes 5 bits, making of almost half of the encoding space for a three-operand instruction in a 32-bits encoding space. While it is possible to use 64 architectural registers (6 bits for encoding), there are no reports that the encoding limitations would be worthwhile for any possible performance increase, while considering a 32-bit encoding space. On the other hand, having only 16 registers could significantly limit the instruction set architecture capabilities. In detail, the RISC-V ISA reserves two opcodes in the 32-bit encoding space for custom intruction set extensions. As such, this extension will be limited to the two custom opcodes, and therefore, it is optimal to use 32 vector registers. In fact, this ISA could use a encoding space with 64-bits instructions, however, that would force any implementation to implement 64-bits instructions, which can be restrictive for future works.

From the 32 vector registers, the proposed ISA extension does not hardwire any register to any value to maximize the available registers (also following SVE and RVV standards). The vectorial registers are named from "u0" to "u31".

Configurable Length

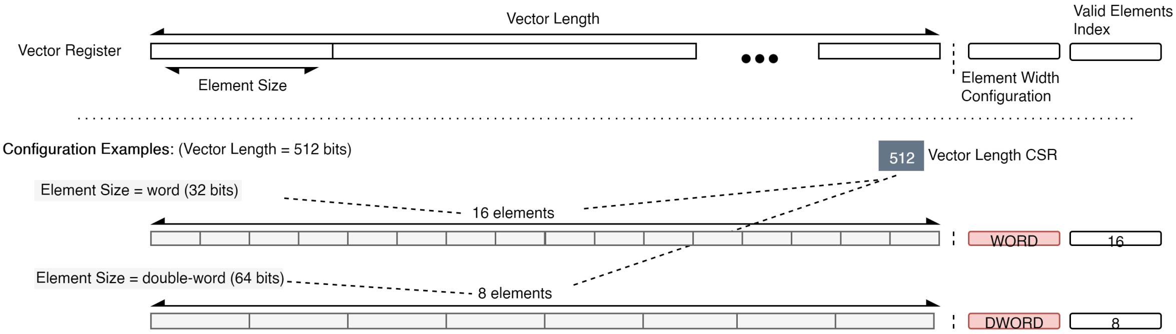
The length property of the registers is not limited in maximum size, in order to not affect scalability. Naturally, a minimum value is defined corresponding to the maximum width of the elements supported by the architecture (i.e., byte, short, int, float, double, etc). On the other hand, the maximum vector length must be a multiple of the minimum length. However, much like in SVE, the vector maximum length can be made available to the application by reading from a read-only CSR, and the working length can be configured through another CSR. **Configurable Width and Type**

Each vector register is partitioned in multiple vector elements. The available element widths are byte (8-bits), half-word (16-bits), word (32-bits), double-word (64-bits). Consequently, the minimum vector length is 64-bits. Additionally, the element width is configured independently for each vector register.

To support the configuration of the vector width, each vector register is extended with the necessary control bits. In a case where a vector is not completelly filled with data, e.g. edge cases and stream terminations, the vector register must hold the number of the valid (filled) bytes, specifically, this field is named valid index.

The data type (signed, unsigned, floating-point) could also be configured for each vector; however, following both SVE and RVV compute instructions specify the data-type [6, 7], the datatype is specified by the compute instruction.

To illustrate the different configurations of vector registers, two possible vector setups are depicted in Figure 4.1.



Vector Length
Valid Elements Index
Vector Register
Element Size
Element Width Configuration
Configuration Examples: (Vector Length = 512 bits)
512
Vector Length CSR
Element Size = word (32 bits)
16 elements
WORD
16
Element Size = double-word (64 bits)
8 elements
DWORD
8

Figure 4.1: Vector register architecture with configuration examples. At the top, a vector register diagram emphasizing the vector length, element size, width configuration, and valid index components. At the bottom, two examples with fixed vector length and two diferent element width configurations.

4.1.2 Streaming Registers

Each time any stream is configured, the data must be facilitated to the consuming instructions. Hence, the corresponding instructions must be capable of distinguishing the source of data (vector registers or stream). To solve this problem, two options were available:

- Explicit selection: Each instruction specifies the source of data through their encoding.
- Implicit selection: Each stream of data is associated with a specific vector register ("u0" to "u31"), and by reading/writting to such register is equivalent to reading/writting to the corresponding stream.

On the one hand, the major downside of the explicit selection is the need for additional bits in the instruction encoding, as well as the requiring of adding additional behaviours (e.g., by raising an exception) whenever reading/writting from an non-configured stream. On the other hand, explicit behaviour benefits from not needing to shadow the vector registers with the streams, virtually allowing for 32 vector register and 32 streams. Hence, the proposed ISA extension is based on an implicit selection, where there is no distinction between streaming registers and vectorial registers.

4.1.3 Predicate Registers

The predicate registers are necessary to allow lane execution control. In this extension, the predicate register file is composed of 16 predicate registers (p0-p15). However, only 8 (p0-p7) are useable in memory and arithmetic instructions. In detail, the remaining registers (p8-p15) can be used to configure the first 8, or to allow for context saving. This balance was validated by analyzing compiled and hand-optimized codes, having the benefit of mitigating the predicate register pressure [7, 28].

Using only eight predicates also takes up less encoding space for the actual instructions (3 bits, versus 4 bits with 16 registers). Also, the predicate register p0 is always hardwired to 1 (all valid lanes execute), removing the need to preconfigure the register in non-conditional loops.

4.2 Arithmetic Instructions and Predication Mechanisms

4.2.1 Instructions overview

The proposed ISA extension features a total of 26 integer, 15 floating-point and 19 memory (including streaming) instructions, to a total of 82 instructions (and around 450 variants), considering predicate manipulation and branching instructions. Since the majority of instructions is not much different from typical ISAs (apart from its encoding), in this section we focus on explaining only its principle. Consequently, only three different instructions are detailed. The first instruction is the add and the variant adde (add elements). Followed by the instruction sll (shift left logical). These instuctions and the respective variants are representative of all the instructions formats. All of the added arithmetic and logical instructions are based on similar instructions from the state-of-the-art SIMD and vectorial ISAs (sections 2.1 and 2.2). In fact, this extension is a work-in-progress and the instructions provided are close to the minimum necessary to provide a proof-of-concept.

A) Arithmetic Instruction Add - add, adde

There are two variants of the add instruction:

Vertical add The add variant follows an instruction format with two source operands, one destination operand and one predication operand. It executes the normal vertical add between the two source vector registers, writing the result to the destination vector register, as depicted in Figure 4.2. Additionally, the datatype of the instruction must also be defined. In this case, the instruction can operate on signed and unsigned values, as well as floating-point ones. The possible combinations of datatype and data width are represented in Table 4.1. In the RISC-V ISA there is no support for half-precision and byte-precision floating-point representations. When an invalid combination in Table 4.1 is used, an exception is raised accordingly. The datatype must encoded in the instruction bits. The support of floating-point processing is dependent on the implementation of the floating-point extension ("F" extension). The

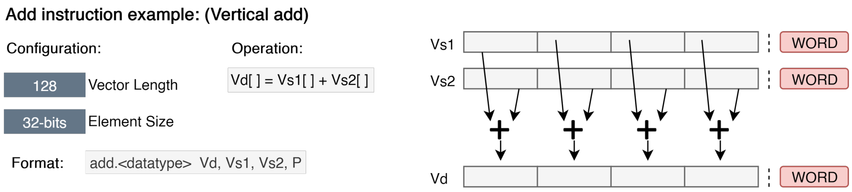


Figure 4.2: Vertical add variant operation. Each element of the first source register is added to the respective element of the second source register. The resulting value is saved in the respective element of the destination register. AThe example assumes that the vector register length is 128-bits and the element width is a word (32-bits).

extension presented here supports floating-point if the "F" extension is also implemented. Consequently, the available floating-point status flags are the ones described in the "F" extension [27].

Table 4.1: Possible combinations between datatype and data width for the add instruction. There are no available representations for half-precision and byte-precision floating-point.

		Datatype		
		unsigned	signed	floating-point
Width	B	✓	✓	—
	H	✓	✓	—
	W	✓	✓	✓
	D	✓	✓	✓

Horizontal add The adde (add elements) variant follows an instruction format with one source operand, one destination operand and one predication operand. It executes the reduction operation depicted in Figure 4.3,saving the result either on one scalar register or on the first element of a vector register (depending on the instruction variant).

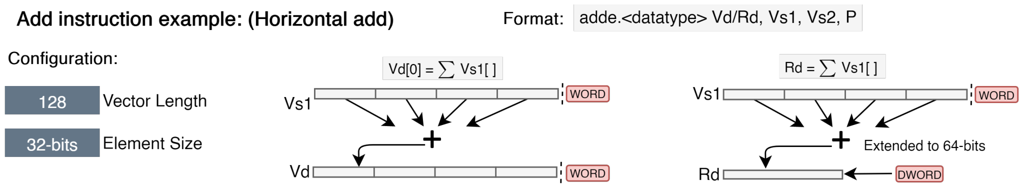


Figure 4.3: Horizontal add variant operation. All the elements of the source register are summed. The resulting value is saved in the first element of the destination vector register. The destination can be saved to a 64-bit scalar register if one is provided. The example assumes that the vector register length is 128-bits and the element width is a word (32-bits).

B) Logic Instruction Shift Left Logical - sll

There are two variants of the shift left logical instruction. The first variant uses the elements of a vector register as shift ammount source (**sll**). The second variant uses one scalar register as shift ammount source (**slls**).

sll - shift ammount from vector The sll (shift left logical) instruction follows an instruction format with two source operands, one destination operand and one predication operand. The source and destination operands must all be vectorial registers. The operation executed by this variant is depicted in Figure 4.4. In logical operations, the data is always interpreted as being unsigned. Therefore, there is no need to specify the datatype.

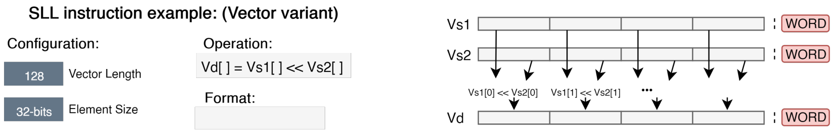


Figure 4.4: Shift left logical vector variant operation. Each element of the first source register is shifted by the ammount given by the scalar source register. The resulting value is saved in the respective element of the destination register. The example assumes that the vector register length is 128-bits and the element width is a word (32-bits).

slls - shift ammount from scalar The slls (shift logical left scalar) instruction follows an instruction format with two source operands, one destination operand and one predication operand. However, in this variant the second source operands is a scalar register. The operation executed by this variant is depicted in Figure 4.5.

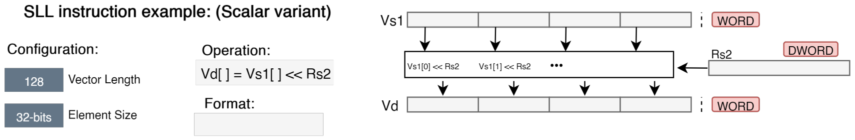


Figure 4.5: Shift left logical scalar variant operation. Each element of the first source register is shifted by the ammount given by the respective element of the second source register. The resulting value is saved in the respective element of the destination register. The example assumes that the vector register length is 128-bits and the element width is a word (32-bits).

4.2.2 Valid index and mismatched widths

The three different operands configuration are representative of all the arithmetic and logical instructions in the extension. Even so, we still need to discuss the influence of predication in the execution of the arithmetic and logical instructions. Some valid index and the width configurations, may lead to some unwanted problems and limitations, as presented next.

A) Valid index considerations

The first concern is based on the execution of instructions when there is less data in a vector than it's total size. When a instruction is executed with non-complete vectors, two problems can arise. First, the operation result may be left undefined in the elements respective to the invalid portion of the source vector. Secondly, the operation result may be undefined or an execption raised in the event of two source operands with different valid elements. For the first question, the valid index specifies the number of elements that are valid and the number of invalids. Consequently, the only elements to be calculated are the valid ones additionally, the valid index is copied to the destination register.

The second question comes with a more profound concern, having mismatched number of elements in both source operands would not normally occur in a well-structured code. Indeed, a programer error can lead to this, and consequently, it would be wise to launch an exception. However, unique and not expected cases could also trigger the discussed event. To give additional flexibility to the programming and execution, there will be no enforcement regarding mismatched valid indexes. To guarantee the coherence and validity of the instructions, the lower of the valid indexes will be used to constrain the execution and define the resulting valid elements. In particular, there is the hipothesis of using the higher of both valid indexes, where the elements that are valid in one operand but not in the other are merged to the destination. However, this behaviour could prove difficult to debug in an application, and therefore it is not adviseable.

B) Handling mismatched widths

The second concern is related to the handling of the instructions that have operands with mismatched widths. In this case, there is no chance of being an event expected by the programmer. Consequently, either a programmer error occurred, or possibly the processor is executing in deep speculation. In both cases, the current specification handles this issues by raising an illegal or undefined instruction exception. However, it could be better to create a specific exception for this case, as it can be difficult to understand why different widths, that are not part of the instruction enconding, would cause an illegal instruction exception.

4.2.3 Predication mechanisms

To maintain control over each element execution there needs to be predication mechanisms in-place, as described next.

A) Lane control

The core concept of predication is to control the execution in each execution lane. To specify how this works, let us use the instruction add in the vertical variant (Figure 4.2) as an example. The predicate register conditions the execution of each lane of the predicated instruction. In specific, when the predicate blocks the execution, the resulting value is not updated in the vector register. This type of predication is named in SVE as merging predication. In contrast to zeroing predication, where the result is set to 0. The latter is currently not implemented and is left for future versions.

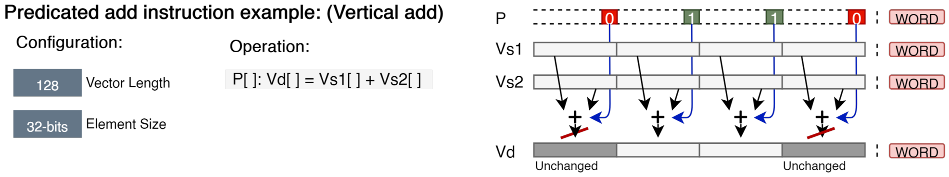


Figure 4.6: Example of predication, based on the vertical add instruction. Each element of the predicate register (P) will condition the execution of the operation. When the predicate does not allow the execution the resulting value is unchanged. The example assumes that the vector register length is 128-bits and the element width is a word (32-bits).

Table 4.2: Possible comparisons achieveable. By combining the instruction PNOT the number of available comparisons is doubled.

		Comparison		
		PEQ	PLT	PELT
Logical Modification	-	==	<	<=
	PNOT	! =	>=	>

B) Predicate configuration

To configure a predicate register, specific instructions are provided, which work by applying a condition test to the data (e.g. less or equal, greater than), such as: PELT (equal or less than), PEQ (equal) and PLT (less than). The instruction PNOT (negate) can be used in conjuction with any of PELT, PEQ or PLT to duplicate the total available comparisons (with only PELT, PEQ, PLT and PNOT), covered in Table 4.2.

To provide a more comprehensive set of comparisons, the instructions PAND (logical and) and POR (logical or) are also part of the instruction set. By using these to combine the values of two predicate registers, it is possible to cover the majority of regular comparisons.

In some situations it might be necessary to copy or exchange the predicate information, such as moving predicate data from the lower registers (p0-p7) to the higher registers (p8-p15). To support this, a PMOV (move) instruction and a PEXCH (exchange) are defined. In detail, the exchange instruction writes to two different registers in the same cycle, and thus can be considered a complex, almost CISC, instruction. However, any out-of-order CPU as well as an in-order with a renaming mechanim can execute this instruction in the rename stage, by swaping the rename target of both architectural registers. And thus, can be considered viable in the RISC paradigm. Additionally, a vector register, as well as a predicate register, can change in element size while maintaining part of the data. The instruction PCONV (convert) supports width conversion for the predicate registers. The details of such an instruction are detailed alongside the vector register convert instruction, in section 4.3.

To add even more capabilities to the predicate configuration instructions, each instruction can be predicated with a lower predicate register (p0-p7). To exemplify this and to cement the comparison instructions, Figure 4.7 provides an example where the instruction PLT is used to create a new predicate that is also dependent on the state of both source and destination predicates.

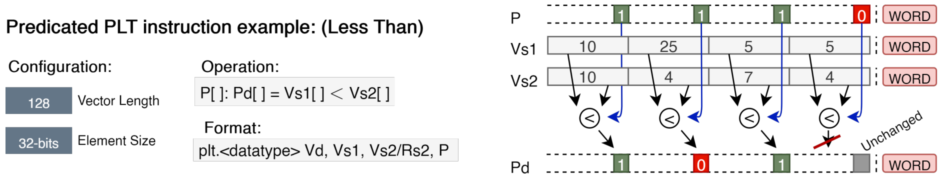


Figure 4.7: Example of predication in the creation of a predicate. Using the less than comparison between vectors to form a new predicate. Each element of the predicate register (P) will condition the execution of the operation. When the predicate does not allow the execution the resulting value is unchanged. The example assumes that the vector register length is 128-bits and the element width is a word (32-bits).

C) Iteration control

In addition to the control of the execution of each lane, a loop can leverage from ending when certain data-based conditions are met (e.g. end loop if a value is lower than some constant). To be able to end the loop or run some sub-routine based on the data values, two predicate-based branch instructions are defined, namelly: B.AND (Branch AND) and the B.OR (Branch OR). The B.AND instruction executes a jump when all the relevant elements of the predicate register are set to 1. In opposition, the B.OR instruction executes a jump when at least one element is set to 1. To achieve the reverse behaviours of NAND and NOR, the PNOT instruction must be combined with the B.AND and B.OR instructions, respectively. These instructions are always predicated with a lower predicate register (p0-p7). Figure 4.8 illustrates the behaviour of the B.AND instruction, where the operand .gotolabel is the jump target.

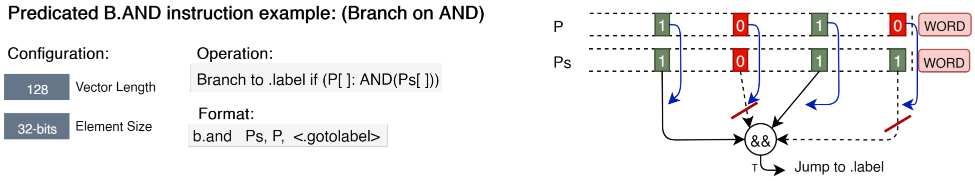

Predicated B.AND instruction example: (Branch on AND)
Configuration:
128
Vector Length
32-bits
Element Size
Operation:
Branch to .label if (P[ ]: AND(Ps[ ]))
Format:
b.and Ps, P, <.gotolabel>
P
Ps
1
0
1
0
1
0
1
1
WORD
WORD
&&
T
Jump to .label

Figure 4.8: Branch on AND (B.AND) instruction exection. The jump ocurs when all the valid elements of the predicate are true. Each element of the predicate register (P) will condition the execution of the operation. When the predicate does not allow the execution the respective value is not taken into account in the AND operation. The example assumes that the vector register length is 128-bits and the element width is a word (32-bits).

4.3 Data transfer instructions

Although this work is heavily based on applying the streaming concept to vector register architectures, some programs may still need to use common load/store mechanisms in order to transfer data to and from memory. To allow this, a set of load and store instructions are required. Additionally, there is a need to define instructions that allow data movement between registers (vector and scalar) and allow the conversion between data elements width.

4.3.1 Vector load and store

When loading or storing data to/from a vector register, it is not possible to infer at compile time how much data will be loaded. To avoid unauthorized data accesses, the load and store instructions are provided with a size operand. The size operand specifies the number of vector elements that will be transacted to or from memory. Also, it is possible to calculate this value based on the implemented vector-length, readable from a CSR. Additionally, when loading data, the width must be specified in the instruction. The store instruction uses the width configured in the vector register.

Even so, if the vector register is not wide enough to fit all the elements specified by size, it is important to allow the program to be able to maintain a coherent state. For this, one of the operands: size and base address, must be updated with the number of elements processed. Consequently, two variants of the instruction are provided: LOAD_A (STORE_A) and LOAD_S (STORE_S), where the address and size are updated, respectively. In the variant where the size is updated, the operand size is set to the actual number of processed elements. In the address updating variant, the address is incremented with the number of bytes processed. To clarify, in Figure 4.9 an example of the LOAD_S instruction is given. Notice that in the case where the requested elements are lower than the vector length, some invalid elements exist in the last portion of the vector. In this specific case, the valid index is different from the size of the vector (in elements). Here it is set to 3, the number of valid elements.

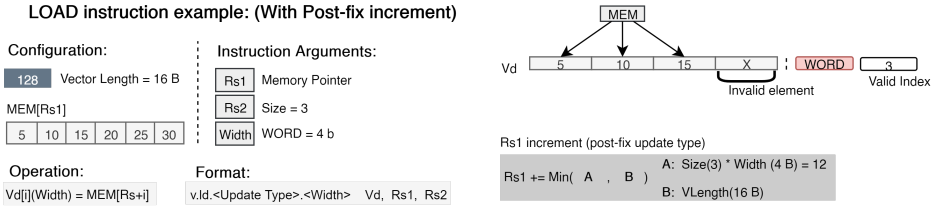


Figure 4.9: Load instruction example with address operand update. The element width, requested size and the memory address are defined by the instruction, via operands and encoding. The number of elements loaded is given by the minimum between the vector register size (in elements) and the requested size. Any non loaded element is invalid, and the vector register valid index is set acordingly. The example assumes that the vector register length is 128-bits.

4.3.2 Moving data between registers

Besides the memory transactions, it is necessary to allow the transaction of data between vector registers and between scalar and vector registers. The register transaction instructions are divided into three categories: Scalar-Vector, Vector-Vector and Width Conversion. While the first two are self-explanatory, the width conversion is a vector-vector transaction where the target register has a different element width.

A) Scalar to/from Vector transactions

Let us start by focusing on getting data from an to the scalar registers. Since a vector register can fit multiple scalar-sized elements, only one element can be transacted. The first question that can arise is which element, and how to index that element. Due to the unknown size of the vector registers, it may not be simple to index an element of the vector. Consequently, it is only wise to allow the first element to be populated, as it is the only guaranteed element to exist.

Vector-Scalar Move With this, a Vector to Scalar transaction will only move the first element of a vector register to the given scalar register. However, due to the scalar registers being 64-bits in width, the element data will be moved to the 64-bits register, with no conversions or extensions. The process of adequating the data to a different datatype must be done with the base ISA instructions.

Scalar-Vector Move When moving data from a scalar register to a vector register, the process is the same; any data conversion must be done with the base ISA instructions. In this case, the element width is specified in the instruction. As a note, if the data in the scalar register is represented with more than the requested element width, the data in the element will be different from the source.

Besides this, the data can also be replicated to all the vector register elements. A specific instruction (dup), allows the data to be replicated. This instruction uses predication, allowing more control to the replication process (e.g. replicate values with interleaving).

B) Vector to Vector transactions

Moving data between vector registers is a quite simple process. The source register contains the data and the element width specification, and the destination register will become a complete copy of the source register. This behavior is consistent both when either (source or destination) corresponds to a vector register, as well as when it corresponds to a stream.

The vector-vector move instruction contains two variants, a direct variant and a transpose variant, both allowing predication. The direct version copies the data directly to the corresponding destination element when the predicate allows the execution, whereas the transpose version swaps all elements. To exemplify, Figure 4.10 shows the relation between the source elements and destination elements.

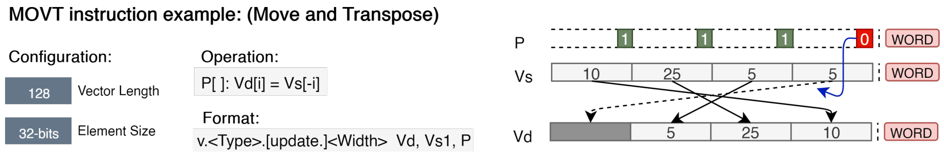


Figure 4.10: Move transpose instruction example. Each source element is copied to the simetric destination element. The predicate conditions the copy of the elements. In cases where the vector register has odd number of elements the central element is not swapped. The example assumes that the vector register length is 128-bits and elements with a width of 32-bits. The type parameter is set to "movt" for move and transpose, and "mov" for move.

Particullary, when the source operand is a stream, it might be important to be able to move the data without iterating over the stream. Effectively, this allows a streaming register to behave as a non-streaming register momentarily, bringing more flexibility into the vector/stream register paradigm. This behaviour is achieved by setting the update flag in the instruction that disables streaming on that instruction.

C) Width Conversion

The width conversion instructions are present in every modern ISA, including the RISC-V base ISA. In this extension, the possible width conversions and the respective type (narrowing, widening) are summarized in Table 4.3. In the cases where the is no need for conversion (same width), the vector move instruction should be used. Additionally, there is no conversion between different datatypes (e.g. float to unsigned word). In general, the datasets are already in the correct datatype and do not require conversion inside the vectorizeable processing kernel. Any dataype conversion must resort to the base ISA and standard extensions instructions.

Table 4.3: Allowed conversions and respective type. The direction of the conversion is represented by the symbols ↓ (narrowing), → (none), and ↑ (widenning). In the cases where the width is not changed, the usage of a regular vector move instruction is recommended.

Unsigned/Signed Src. \ Dest.	Byte	Half	Word	DWord
Byte	→	↑	↑	↑
Half	↓	→	↑	↑
Word	↓	↓	→	↑
DWord	↓	↓	↓	→

Floating-point Src. \ Dest.	Word	DWord
Word	→	↑
DWord	↓	→

Width conversion instructions generate a problem related with the conversion between widths when the vector registers have the same size, without losing any elements. To solve this issue, let us break the problem into two parts: narrowing and widening.

Element Widening The supported widths: Byte, Half, Word, and DWord have known powers of two size, respectively, 1, 2, 4, and 8. Taking this into account, a conversion from Byte to DWord will need eight times the size of the source register (see Figure 4.11). In other words, one register with byte configuration needs to be converted to eight vector registers.

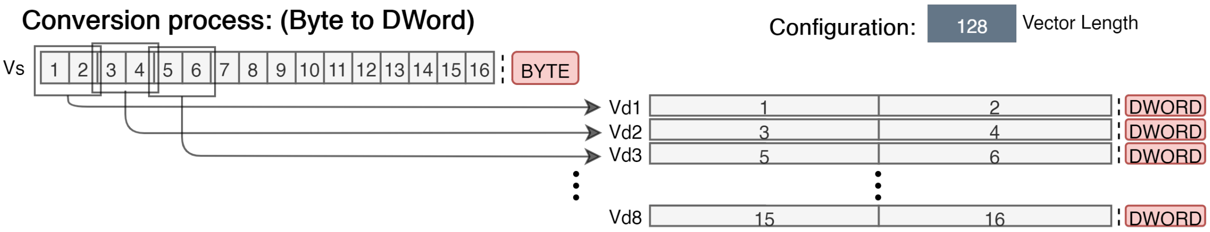


Figure 4.11: Conversion example with the initial element width of 8-bits (Byte) and the final element width of 64-bits (DWord). The contents of the source register occupy eight destination registers. The example assumes that the vector register length is 128-bits.

Although some ISA extensions make use of this behaviour, namely RVV and NEON, it should not be considered compatible with a genuine RISC instruction set. In addition, it is not a simple or elegant solution. To avoid this mechanism, it is possible to process the widening conversion in multiple iterations/instruction, instead of one heavy instruction.

To guarantee that each instruction knows what portion of the source vector it is converting, three options are on the table:

- Use an operand value to index the data transfer. This operand is updated with the first index for the next conversion.

- Use the instruction encoding to index the operation. Encode the portion of the vector that is to be converted.
- Shift the data of the source vector register after each conversion.

The first option requires an additional register to be used in the instruction and could lead to non-standard cases where the requested portion of the vector is not a real vector portion. The second option could lead to data swaps and consequently to the loss of coherency. Although it is a valid approach, the conversion process should not allow strange and non-standard behaviours.

For the last option, to better explain how the process of shifting the data works, the example in Figure 4.12 shows a conversion from the width Word to the width DWord.

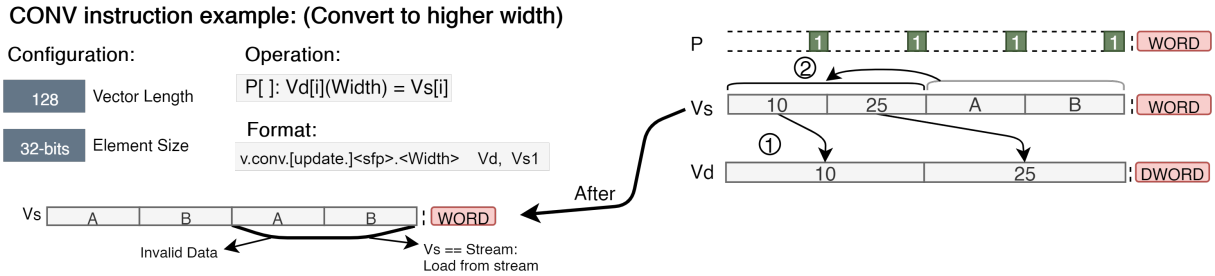


Figure 4.12: Shifted conversion example with the initial element width of 32-bits (Word) and the final element width of 64-bits (DWord). The contents of the source register are shifted after the conversion. The final block of data is invalid after the conversion; alternativelly, new data may be loaded from the associated stream. The example assumes that the vector register length is 128-bits.

In this case, the number of destination register space is the double of the source space. Therefore, to fully convert the source registers, two iterations are needed. In this example, the first step is to copy the left-most elements of the source register, and then, all the source elements are shifted to the left. Consequently, the ending state of the source register will contain two valid elements and the other two that can be either invalid (vector register), or two elements loaded from the stream, as shown in Figure 4.12. This is an initial decision that could lead to a negative impact in performance and still needs evaluation. In sum, by shifting the data, the final state of the source register allows for subsequential conversions that will iterate over the register contents automatically.

Element narrowing The element narrowing conversion works in a similar way to the widening. When a narrowing is executed, the number of source registers space is higher than the destination space. Consequently, the shifting must be done in the destination register to accommodate the data from multiple source registers. The narrowing process starts by shifting the left-most elements of the register to the right; therefore, a portion of the data will be discarded. Then, the source data is converted into the left-most position of the destination register. As before, it is possible to interact directly with the stream, where the converted data can be saved to the stream immediately, however, this is also needs further testing to access the performance impact.

Figure 4.13 shows a conversion example from word-sized elements to half-word ones.

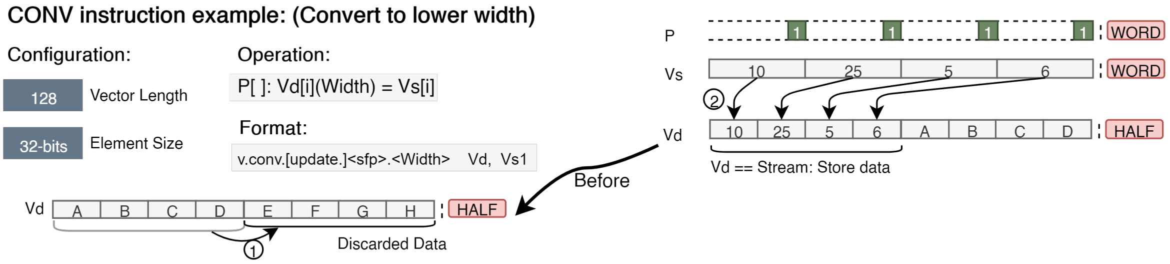


Figure 4.13: Shifted conversion example with the initial element width of 32-bits (Word) and the final element width of 16-bits (Half). The contents of the destination register are shifted before the conversion. The final block of data is discarded after the shift. The newly converted data may be automatically stored into the associated stream. The example assumes that the vector register length is 128-bits.

4.4 Stream Description Interface

This section explores the subset of stream configuration instructions. Having the memory pattern descriptors defined (chapter 3), it now is necessary to define instructions that can represent those patterns in a simple but efficient way. This section refers to the already defined pattern descriptors to define the stream configuration instructions. In addition to these, more instructions allow to dinamically configure the state of a stream (e.g. resume, suspend, etc..).

4.4.1 Basic Stream Definition

To start, let us pick a tri-dimensional description (Figure 4.14) and from there arrive to a complete stream configuration set of instructions. The example in Figure 4.14 is made of 3 dimensions. From which, the first dimension (dimension 0) can be used to describe a linear pattern that starts in "A" and as a size of 5, with an elemement stride of 1.

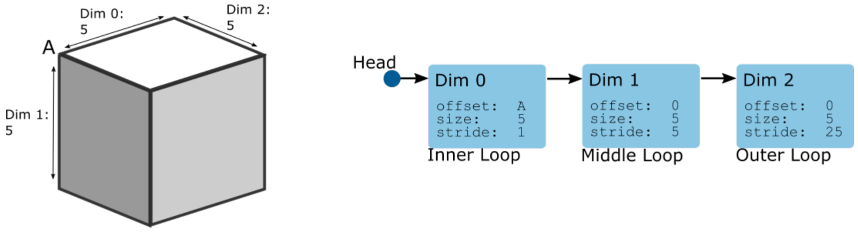


Figure 4.14: Example with a tri-dimensional pattern. In the left, a visual representation of the tri-dimensional pattern. In the right, the associated pattern descriptors.

Single-dimension streams To describe only the first dimension, a configuration instruction must provide the offset, size, and stride parameters. Additionally, a instruction must also set the data width ("Width") and the transaction direction ("Dir"). Consequently, it is possible to create a instruction with the following format:

```
ss.sta.<Dir>.<Width>   Vd,  Rs1 (Size),  Rs2 (Offset),  Rs3 (Stride)
```

Assuming that the destination vector register is u1, and that we are loading data composed of word-sized elements, the instruction becomes (a1-3 contain the dimension configurations):

```
ss.sta.ld.w   u1,  a1,  a2,  a3
```

By executing this instruction, a stream description is started (hence the instruction opcode ss.sta). However, it is necessary to end the description in order for it to be processed. In the case where only one dimension completely describes the stream, the use of the "ss" opcode without the ".sta" or other modifications is recommended. When using only the "ss" opcode, the stream is configured immediately and does not wait for further information. To clarify, the instruction would result in:

```
ss.ld.w   u1,  a1,  a2,  a3
```

Multi-dimension streams Even so, one dimension is not enough to describe the tri-dimensional access. Therefore, additional instructions are needed. Considering the list organization proposed in section 3.4, to add a higher level configuration, it is necessary to decribe the remaining configurations. Notice that, as described in chapter 3, a dimension that is defined after another, is implicitly an outer dimension. With this in mind, to define the dimension 1 in Figure 4.14, we only need to provide another instruction that targets the same stream/vectorial register. Additionally, considering that the complete description is not supposed to terminate (dimension 2 is not yet defined), the instruction will only append a configuration instead of terminating the configuration. When appending configurations to the stream description, the instructions use the opcode "ss.app" (append). With this, the following format concretizes the stream append instruction:

```
ss.app   u1,  a4,  a5,  a6
```

Note that, when appending a configuration the direction of the transaction and the element width are not repeated, this information is only necessary for the first configuration. The registers a4-6 contain the configuration of the dimension 1.

To define the last dimension and terminate the stream description, the instruction with the opcode "ss.end" is added. This behaves similarly to the append instruction; However, this sets the current dimension to be the last in the configuration. Any additional append or end configuration instruction to the same stream will be treated as an error, and an exception is be raised. In the case where an instruction starts defining a new configuration, the previous configuration is discarded and the new one begins. To cement the description of the terminating dimension, the following format corresponds to the end instruction:

```
ss.end   u1, a7, a8, a9
```

The registers a7-9 contain the configuration data of the dimension 3.

To summarize, the example given in Figure 4.14 can be defined by executing the instructions in Listing 4.1.

Listing 4.1: Configuration code example for a tri-dimensional access. The configuration parameters are detailed in Figure 4.14. This example assumes a word-sized elements loading scenario.

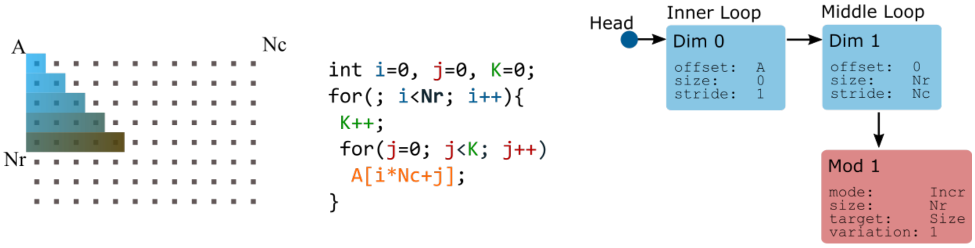
```
; Dim 0: a1 = A, a2 = 5, a3 = 1
ss.sta.ld.w     u1,  a1,   a2,  a3
; Dim 1: a2 = 5
ss.app          u1,  zero, a2,  a2
; Dim 2: a4 = 25
ss.end          u1,  zero, a2,  a4
```

4.4.2 Streams with Modifiers

In addition to the linear dimension-based pattern descriptions, in chapter 3 two types of modifiers were defined, which encoding is described next.

A) Static Modifiers

The most simple type of modifiers were named static due to the non-linear but constant behaviour. To specify how the static modifiers can be configured with instructions, let us remember the lower triangular example sumarized in Figure 4.15. In this example, the two dimensions are defined by the instructions already specified. The first dimension must be defined with the instruction stream start and the second dimension with the instruction stream append. However, to constrain the stream to only fetch the lower triangle a static modifier must be introduced before ending the stream configuration.



```
int i=0, j=0, K=0;
for(; i<Nr; i++){
 K++;
 for(j=0; j<K; j++)
  A[i*Nc+j];
}
```

Figure 4.15: Example with a lower diagonal pattern. In the left, a visual representation of the pattern. In the center, the C for loop representation of the pattern. In the right, the associated pattern descriptors.

To start the definition of a modifier instruction, let us identify the necessary information that the instruction must provide. First, this instruction must be distinguished from the previous append or end dimensions. And second, the mode, size, target, and displacement must be provided. With all this, the format for the modifier instruction is:

```
ss.<type>.mod.<target>.<mode>   Vd,  Rs1 (Size),  Rs2 (Displacement)
```

The type is one of "app" (append) or "end", and defines if this modifier terminates the stream description. The target, mode and displacement, who were detailed in chapter 3, define the impact of the modifier in the targeted dimension.

B) Dynamic Modifiers

The dynamic modifiers are similar to static modifiers. With the exceptions that: an already configured stream gives the displacement value, and slightly different modification modes are available. The corresponding instruction format is:

```
ss.<type>.ind.<target>.<mode>   Vd,  Rs1 (Size),  Vs2 (Source Stream)
```

In this format, the ".ind" parameter is set, which represents indirection (the displacement value is obtained from an indirect source). The target and mode paramenters were already detailed in chapter 3.

4.4.3 Stream State Manipulation

When executing, the processor may need to swap context (e.g. exception handling). In those cases the utilization of the streams and vector registers may be limited by the previous context. In order to momentarly disable the configured streams, a set of control specialized instructions are provided.

These instructions allow the state configuration of each stream at any point of the code. Namely, the available instruction allow for suspend, resume, and terminate a stream. Particularly, the suspend instruction will momentarly disable the automatic iteration of a stream until the resume instruction is executed. During the suspended period, the streaming register will be disabled (no consumption or production), all will work as if no stream was configured.

In addition, a load and store instructions allow for moving data to and from a suspended stream. This instructions will force the stream to load one vector of data into the coupled register (or store), this only works for a suspended stream.

With a suspended stream, all the instructions that would consume or produce data from/to the stream will use the data from the standard vector registers, while the stream is suspended. When a stream is suspended and a new is to be configured, the suspended stream needs to be terminated beforehand. Otherwise, an exception is raised.

4.4.4 Stream Cache Control

Considering that the streams can target large memory spaces, the streaming mechanism can have a negative impact on the cache, namely by polluting the cache. To prevent this, the streaming mechanism could be configured to get the data from a specific cache or directly from memory. To allow the cache level configuration, a specific instruction was defined. This instruction specifies the target cache level for an individual stream.

4.4.5 Stream-dependent Execution Flow Control

The last set of instructions that target the configuration and management of streams are related with the conditional repetition of loops with account for the stream state. These last instructions provide to the programmer a way to conditionally jump when a stream is at a given state. In general, SIMD/vectorial extensions are used to accelerate loops. Knowing that a stream is terminated/complete is sufficient for the great majority of the looping situations. With this into account, two branch instructions are proposed. One instruction branches when the stream has completed (s.c) and the oposite one banches when the stream is not completed (s.nc). The format of both intructions is the following:

```
b.<condition>   Vd,  <.gotolabel>
```

4.4.6 Flow Control with Inter-loop Constraints

It is common, for a code to process data in an inner for loop and only write to memory at the end of that loop, as Listing 4.2. In this example, UVE is limited to accelarating the inner loop, even if it is possible to describe both X and Y accesses in two streams, it is not possible to distinguish the terminations of the loop "j". In fact, it is possible to consecutivelly load and store the Y array, however, that would lead to a negative impact in performance due to the unnecessary memory accesses.

Listing 4.2: Code example of inner loop processing with outer loop memory access.

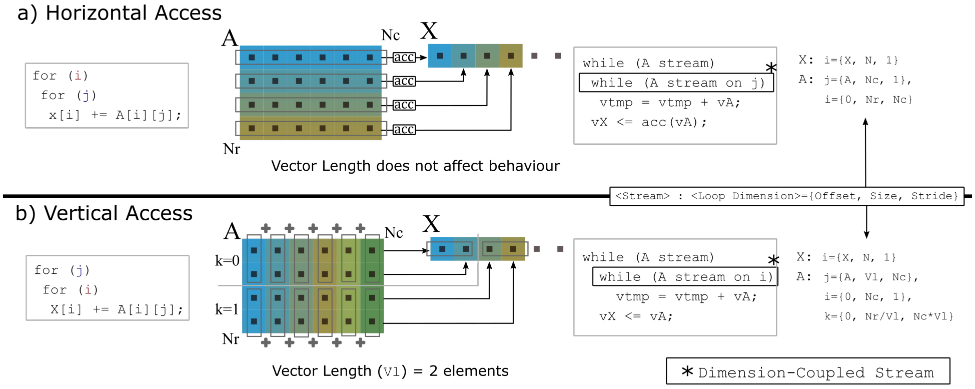
```c
register float aux = 0;
for(int i=0; i<N; i++){
    for(int j=0; j<M; j++){
        aux += X[i][j];
    }
    Y[i] = aux;
    aux = 0;
}
```

Vector Length Control To control the exact number of instructions the inner loop needs to execute. However, due to the scalability nature of UVE, this is not achieveable. Even so, at executing time the vector length is known and can be read from the processor state registers. While knowing the vector length is a clear advantage, it is also possible to improve the control over the execution by configuring the vector length at runtime, as such, there is a clear need of vector length control instruction. This is done with the "setvl" instruction, which accepts a requested vector length and returns the available vector length to the destination operand. The available vector length is the minimum-value between the requested and the implemented vector lengths. This is not valid when the requested vector length is zero, in this case the available vector length is the implemented vector length. This removes the need for a "getvl" instruction.

The configuration of a stream can be prepended with the "setvl" instruction that will set the vector length in all transactions of that stream, where the vector length is set to the mininum of the implemented and requested vector lengths. In short, the "setvl" instruction presents the following format:

```
so.c.setvl   Rd(Final VL),  Vs1(Stream),  Rs2(Requested VL)
```

Vector/Dimension-Coupled Streaming Having full control over the vector length, while an improvement, lacks simplicity and would lead to intricate and unnatural codes. Hence, a complementary solution is proposed, as motivated in the two examples in Figure 4.16. Figure 4.16a) represents a code where the knowledge of the innermost loop boundaries is used to synchronize between the accumulation and store, i.e. all the data from the innermost loop is loaded and accumulated first, then stored to X. It requires for X to be configured as a single element store stream, where the accumulation instruction output would write to. In the pseudo code portraing the vectorial execution, the innermost while condition is dependant on knowing if the stream A is processing the dimension j (innermost dimension), this indicates that the stream passes the processing status to the processing core. Additonally, for this example to work as expected, the vector length must be a multiple of the innermost loop access size (Nc), otherwise, data from iteration "i+1" would be loaded and processed in iteration "i", leading to incorrect behaviour and data loss. To circumvent this, we propose a vector-coupled behaviour that can be enabled for any dimension of any stream. The vector-coupled behaviour, depicted in Figure 4.17, allows for a multi-dimensional stream to be logicaly divided, from the data organization view, onto multiple streams with lower dimensionality, e.g. it automatically turns a bi-dimensional stream (with dimensions


a) Horizontal Access
A
Nc
X
acc
acc
acc
acc
Nr
Vector Length does not affect behaviour
<Stream> : <Loop Dimension>={Offset, Size, Stride}
b) Vertical Access
A
Nc
X
k=0
k=1
Nr
Vector Length (Vl) = 2 elements
* Dimension-Coupled Stream

```
for (i)
 for (j)
  x[i] += A[i][j];
```

```
while (A stream)            *
 while (A stream on j)
  vtmp = vtmp + vA;
 vX <= acc(vA);
```

```
X: i={X, N, 1}
A: j={A, Nc, 1},
   i={0, Nr, Nc}
```

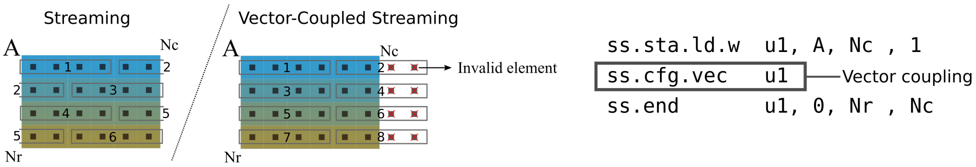
```
for (j)
 for (i)
  X[i] += A[i][j];
```

```
while (A stream)            *
  while (A stream on i)
    vtmp = vtmp + vA;
  vX <= vA;
```

```
X: i={X, N, 1}
A: j={A, Vl, Nc},
   i={0, Nc, 1},
   k={0, Nr/Vl, Nc*Vl}
```

Figure 4.16: Examples of dimension-coupled streaming. In part a (top), the code transverses the memory region horizontally, the data that corresponds each iteration of the innermost loop is accumulated and stored after each outer loop iteration. In part b (bottom), the memory is accessed vertically and all of the innermost accesses are now executed before storing.

"i" and "j") into multiple "i" linear dimensions "j", while maintaining the stream processing unchanged. To activate vector-coupling, the instruction "ss.cfg.vec" is put after the target dimension configuration.



```
ss.sta.ld.w   u1, A, Nc , 1
ss.cfg.vec    u1
ss.end        u1, 0, Nr , Nc
```

Figure 4.17: Distinction between vector-coupled and non vector-coupled streaming. The leftmost figure represents the organization of a 2-D memory access in the resulting vectors, here all the elements are placed consecutively inside the vectors. In contrast, in the vector-coupled stream figure the vectors are aligned with the inner dimension of the stream. Vector-coupling gives the streaming process knowledge of the data destination format.

The fundamental side-effect of the vector coupling instruction is that the last vector of the vector coupled dimension is flaged as that dimension termination, this flag can be captured by a new branch instruction. Both "b.de.#" (branch on dimension # end) and "b.nde.#" (don't branch on dimension # end) instructions can capture the dimension termination flag for dimension # and branch accordingly. For comparison, the branch on stream complete, introduced in subsection 4.4.5, corresponds to the case where the outermost dimension flag is compared.

Figure 4.16b) depicts the same processing as before with a vertical (trasposed) memory access. In this example, the processing is executed in bands, this showcases that another processing pattern can be handled by resorting to dimensional coupling. In fact, this example makes use of the execution time vector length knowledge as well as dimension coupling in dimension "i". In sum, the tools/instructions introduced in this chapter are absolutelly necessary to tackle a large but not specialized set of benchmarks.

4.5 Encoding Space Management

As in all instruction sets, each instruction is encoded with encoding space limitations. On the one side, more operands and parameters make more powerful instructions possible. On the other side, the processor needs to decode all the instructions; therefore, there is a limitation of complexity. Additionally, simpler and smaller instructions generally execute faster.

In this thesis, it was chosen for this extension to target the 32-bit encoding space of RISC-V. It occupies the two major opcodes reserved for custom extensions (see also Figure 4.18, which illustrates the RISC-V opcode map), namely custom-0 and custom-1. These were chosen to avoid interfering with the standard RISC-V extensions. The stream configuration instructions were given the full custom-0

inst[4:2]	000	001	010	011	100	101	110	111
inst[6:5]								($> 32b$)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	$48b$
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	$64b$
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	$48b$
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	$\geq 80b$

Figure 4.18: RISC-V base opcode map, inst[1:0]=11 [27].

opcode. This is due to the stream configuration instructions taking up the majority of the enconding by using up to 4 operands (20-bits of encoding). The group of streaming configuration instruction in custom-0 opcode is named StreamSet.

Consequently, the custom-1 opcode contains all the other instructions, such as arithmetic, logic, and predicate. This group instruction is named StreamOps. Table 4.4 summarizes the encoding map for the proposed extension.

Table 4.4: Encoding map of the StreamOps (custom-0) and StreamSet (custom-1) instructions.

StreamOps		inst[30:29] 00	01	10	11
inst[31]	0	Arithmetic		Misc	Empty
	1	Predicate	Vector/Stream Config	Logical	Branch

StreamSet		inst[30:29] 00	01	10	11
inst[31]	0	Stream configuration (ss, ss.sta, ss.app, ss.end)			
	1				

The encoding was limited to 32-bits by following an initial consideration that this extension would be able to target embedded and HPC processors. Consequently, the usage of 32-bits was essential to support embedded platforms. However, after carefull considering, including the requirement to add additional instructions, a 64-bits encoding would be more appropriate for HPC applications.

4.6 Summary

This chapter summarizes the proposed ISA, which is directed towards minimizing the number of loop instructions, as well as for providing the means to explore the stream paradigm and to support unlimited vector sizes. In Figure 2.6, the desired ISA removed the overhead instructions to the absolute minimum. To turn the conceptual pseudo-code to the a real implementation, Figure 4.19 presents both the pseudo-code and a possible implementation of the UVE extension. As displayed, there is a direct correspondence between the pseudo-code instructions and the UVE instructions. To recall the differences between the Arm SVE ISA and the UVE ISA, Figure 4.20 shows the comparison between both and focus on the improvements achieved by the Unlimited Vector Extension. The UVE ISA reduces the number of instructions in the part of the execution where the great majority of the time is spent (inside the loop). This optimization is especially relevant for long and time consuming loops, representing a large percentage of the application execution time, as well as in cases as the indexing and loop control operations otherwise require a relative high number of instructions. The downside is that, while the UVE stream configuration (loop preamble) may take more execution time than in SVE code, this configuration time may be negligeble in large loops. In summary, the Unlimited Vector Extension uses less instructions than SVE, and therefore can be considered a more efficient vector instruction set extension. The UVE instructions used in this chapter and the remaining available UVE instructions are summarized in

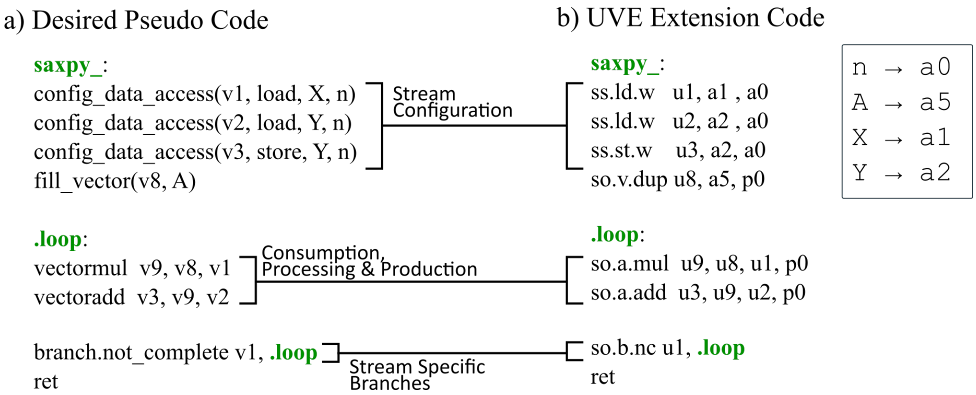


Figure 4.19: Concretization of the SAXPY example with UVE. On the left, the pseudo code defined in Figure 2.6. In the right side, the concretization of the aforementioned pseudo-code with the UVE ISA. The equivalence between the registers and the C variables counterparts are shown in the box placed to the right.

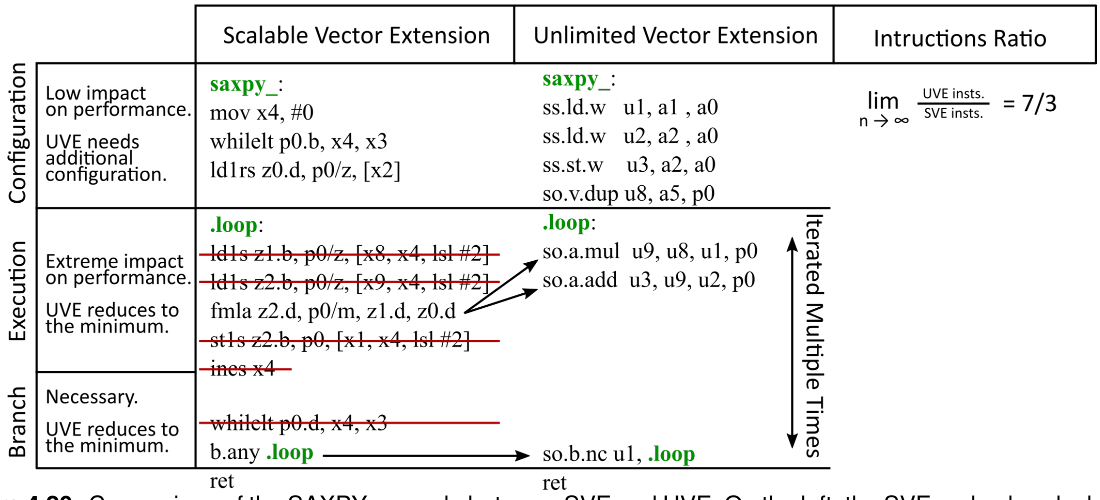

Scalable Vector Extension
Unlimited Vector Extension
Intructions Ratio
Configuration
Low impact on performance.
UVE needs additional configuration.
saxpy_:
mov x4, #0
whilelt p0.b, x4, x3
ld1rs z0.d, p0/z, [x2]
saxpy_:
ss.ld.w u1, a1 , a0
ss.ld.w u2, a2 , a0
ss.st.w u3, a2, a0
so.v.dup u8, a5, p0
$\lim_{n \to \infty} \frac{\text{UVE insts.}}{\text{SVE insts.}} = 7/3$
Execution
Extreme impact on performance.
UVE reduces to the minimum.
.loop:
~~ld1s z1.b, p0/z, [x8, x4, lsl #2]~~
~~ld1s z2.b, p0/z, [x9, x4, lsl #2]~~
fmla z2.d, p0/m, z1.d, z0.d
~~st1s z2.b, p0, [x1, x4, lsl #2]~~
~~incs x4~~
.loop:
so.a.mul u9, u8, u1, p0
so.a.add u3, u9, u2, p0
Iterated Multiple Times
Branch
Necessary.
UVE reduces to the minimum.
~~whilelt p0.d, x4, x3~~
b.any .loop
ret
so.b.nc u1, .loop
ret

Figure 4.20: Comparison of the SAXPY example between SVE and UVE. On the left, the SVE code already defined in Figure 2.6. In the middle, the respective UVE code. In the right, the limit of the instructions ratio when the number of processing elements (n) tends to infinite.

Table 4.5.

Table 4.5: Instruction set instructions, organized by type.

Instructions Group	Instructions
Arithmetic	Add, Subtract, Multiply, Divide, AddElements, Multiply and Accumulate, Absolute, Increment, Decrement, Negative
Logic	AND, NAND, OR, NOR, XOR, NOT
Shift	Shift Left Logical, SLL scalar, Shift Right Logical, SRL scalar, Shift Right Arithmetic, SRA scalar
Misc	Minimum, Minimum Element, Maximum, Maximum Element
Predicate (Manipulation)	Zero, One, Vector, Move, Move and Transpose, Exchange, Convert
Predicate (Logic and Comparison)	NOT, AND, OR, Equal or Less Than, Equal or Less Than Scalar, Equal, Equal To Scalar, Less Than, Less Than Scalar
Branch	Dimension [Not]Complete, Stream [Not]Complete, Predicate-Based(AND, OR)
Vector Manipulation	Load, Store, Duplicate, Move, Move and Transpose, Move Scalar To Vector, Move Vector To Scalar, Convert
Stream State	Suspend, Resume, Break, Manual Load, Manual Store
Stream Configuration	Set Vector-Length, Get Vector-Length, Config Vectorial, Stream Start, Stream App, Stream End, Simple Stream

5 ISA Evaluation and Discussion

Accessing the performance of the instruction set through simulation brings a more realistic comparison than a side-by-side assembly comparison. To access the performance of the instruction set, an ISA simulator is used. The instruction set simulators are an integral part of a today's processor and software design process; they have an essential and undisputed role within the architecture exploration and early system verification [49, 50]. Admittedly, it is also possible to go further and model the entire system; however, doing this would require us to model the streaming supporting mechanisms. In this chapter, the required tools to simulate the extension are presented, alongside an overview of the modifications applied to these tools.

This chapter is divided into three sections. The first section describes the compilation tools that allows the code to be compiled with this extension semantics. The second section presents a cycle-based simulator with the capability to simulate the instruction set alongside a supporting micro-architecture model. The final section depicts benchmark results and performance evaluations of the UVE vector extension from the instruction set point-of-view.

5.1 Compilation Tools

In the space of C/C++ compilers, only two are worth considering for programming support to this extension. On the one hand, the GNU Compiler Collection (GCC), which is an official compiler for the GNU and Linux systems [51]. On the other hand, the Clang/Low Level Virtual Machine (LLVM) which is a newer and more modularized compiler [52]. Both these C/C++ compilers now support the RISC-V ISA; however, that was not the case at the initial development phases of this work. Consequently, the choice was to extend on top of GCC, who supported the RISC-V ISA earlier.

To support a new extension, the new architectural state and the set of new instructions must be added to the compilation toolchain. After this, it is possible to create C/C++ intrinsic functions that allow the compiler to generate the extension instructions based on the provided C/C++ intrinsics. Taking it further, it is even possible to extend the compiler into vectorizing the code and produce instructions accordingly. Due to the GCC source code complexity and to time constraints, the later was left for future work. Hence, at this stage only assembly support is provided.

To support the new instructions, the assembler tool was extended. To aid in the process of creating new instructions, a fully automated system was created. This system is given a set of files that describe the new instructions, and the available registers, from which the appropriate toolchain supporting files are generated, and the cross-compilation tools are then built. Figure 5.1 contains a flowchart that details the work-flow of the automated process. To define instructions in the toolchain is not a simple process. This

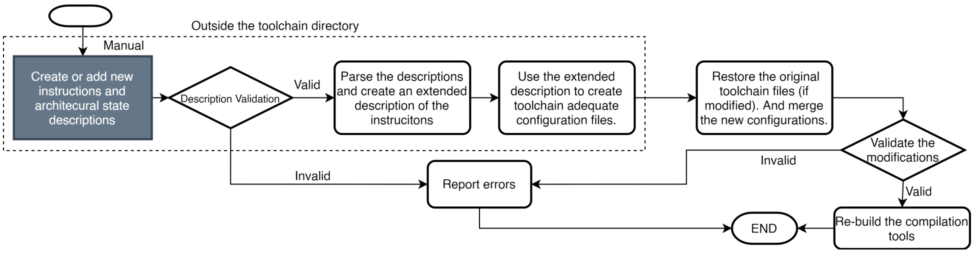

Outside the toolchain directory
Manual
Create or add new instructions and architecural state descriptions
Description Validation
Valid
Parse the descriptions and create an extended description of the instruciton
Use the extended description to create toolchain adequate configuration files.
Restore the original toolchain files (if modified). And merge the new configurations.
Validate the modifications
Invalid
Invalid
Report errors
Valid
Re-build the compilation tools
END

Figure 5.1: Flowchart detailing the steps in the automated process of extending the compilation tools with the new instructions. The first process involves the creation of description files that specify the instructions and auxiliary files that specify available registers and specific expansion rules. From the description, the new instructions and registers are converted to appropriate formats and used in the toolchain compilation.

involves creating appropriate representations of instructions for the toolchain code and understanding the underlying mechanisms of the assembly process. In detail, when handling branch/jump instructions, there are special considerations that must be provided to the address relocation process where the jump labels are matched with the final instruction addresses.

The implementation of instructions at the assembly level allows for the usage of inline assembly and extended assembly directives. These directives allow the programmer to embed assembly instructions inside the C/C++ code. In fact, with extended assembly directives, the C/C++ code is able to interact directly with the assembly code. The compiler also automatically prepares the data and registers to be used in assembly. Figure 5.2 gives an example where the malloc function is vectorized using extended assembly directives. The extended assembly syntax is extremelly useful in the integration of assembly

Original C code

```
void * memcpy( void * src, void * dest, size_t n) {
  for(size_t i = 0; i < n; i++)
    dest[i] = src[i];
}
```

Vectorized C code

```
void * memcpy( void * src, void * dest, size_t n) {
  asm ("ss.ld.b u1, %[src], %[size], %[stride]\n\t"
       "ss.st.b u2, %[dest], %[size], %[stride]\n\t"
       ".loop:\n\t"
       "so.v.mv u2, u1, p0\n\t"
       "so.b.nc u1, .loop\n\t"
       :
       : [dest] "r" (dest), [src] "r" (src),
         [size] "r" (n), [stride] "r" (1)
       : memory);
}
```

Figure 5.2: Extended assembly directives support, exemplified with the memcpy code. The compilation process will match the C variables with the assembly extended operands.

code alongside C code. In detail, the C code variables are available for usage in the assembly, with all the conversion details being left to the compiler. The ending portion of the extended assembly function allow the programmer to specify wheter a C variable will be used as either destination or source operand. In detail, the code in front of the first colon (":") represents registers that will be destination operands.

Similarly, after the second colon (":") the source operands are defined. Each operand is defined as $[ref]\ ”mode”\ [cvar]$ where the "ref" is the identifier to be used in the assembly code, the "cvar" is the C variable name and the "mode" specifies how that variable should be interpreted. Particularly, the "r" mode specifies that the variable should be put in a scalar register. Finally, the last colon (":") is for definition of the registers (or memory) that were modified inside the assembly code, in this example the memory was modified. The extended assembly is extensively documented in [53].

5.2 Simulation Environment

The simulator used through this work is gem5 [54]. In the space of computer architecture simulators, there are many levels of simulation accuracy that make a tradeoff with simulation execution time. Figure 5.3 synthesises the positioning of various simulators in the speed/accuracy tradeoff.

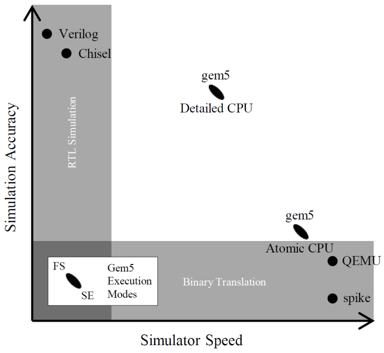


Figure 5.3: Accuracy and speed positioning of computer architecture simulators [5].

Whereas the RTL simulation is the most accurate level of simulation, it requires that the system is completely implemented. This is not suitable for a work that needs to modify the instruction set, the CPU architecture, and the memory system, particularly considering that the objective of this thesis is to propose and make a first evaluation of the UVE instruction set extension. On the other side, the binary translation level is only accurate at the instruction simulation level and does not contain any performance assessment of an implementing system. This would be useful for this part of the work; however, the complete work also simulates an implementing microarchitecture. Considering all this, the gem5 simulator sits right between the RTL accuracy and the binary translation speed. The gem5 simulator is a computer systems simulator that simulates system architectures and supports microarchitectures with cycle-by-cycle precision. To support ISA only simulation, a simplistic CPU model (atomic) is available. This model does not include any microarchitecture modelling. In fact, the model only executes instructions sequentially and maintains an architectural only register file. There are no considerations into memory access or instructions latency and throughput, as there is no underlying realistic CPU model. It can be further extended to support custom instruction set architectures and microarchitecture models, being the perfect candidate for this work [5, 54, 55].

The gem5 simulator is an open-source project that is developed by a large number of developers. Unfortunately, the documentation is not extensive enough, and there is a lack of code commenting. This resulted in a prolonged and severe process to understand the majority of the code. To implement UVE, the simulator instruction set architecture framework was extensively modified. The gem5 simulator and the carried modifications are written in C++ and Python programming languages. Figure 5.4 contains a simplified overview of the modifications to the gem5 simulator. The figure represents the gem5 modifications through a view of a simplistic pipelined processor organisation. The choice of representation is due to the close relationship between the gem5 internal organisation, at the instruction set level, and the represented pipelined. The modifications are explained in the following paragrahps. To support the modified architectural state, the vectorial and predicative registers were specified in gem5. Also, the added configuration registers were added. Finally, supporting the vector register scalability involves the specification of the vector register length, this involves allowing the configuration to be changed for each simulator execution and set the configuration registers accordingly.

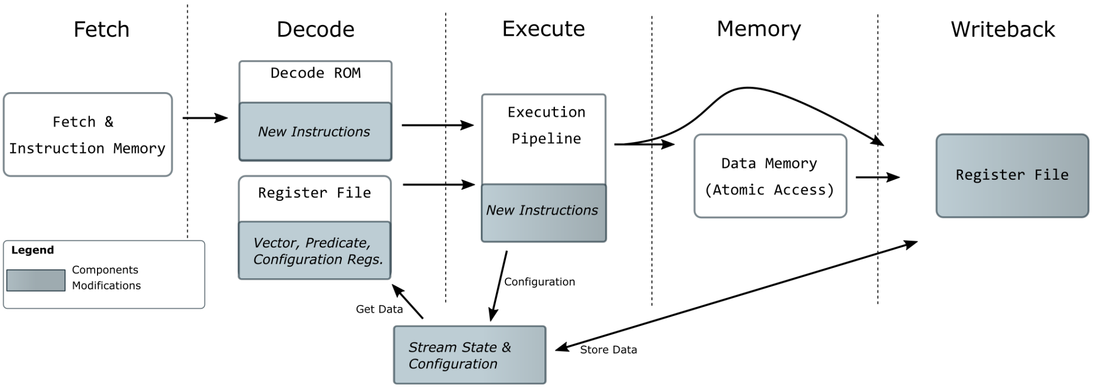


Figure 5.4: Overview of the modifications to the gem5 simulator. The choice of representation is a simplistic pipelined processor structure, due to the close relationship between the gem5 instruction set organization and the aforementioned processor structure. The modifications/additions are shown in grey background. The stream state and configuration block guarantees is not part of the instruction set modifications. It is represented to indicate the connections between the standard pipeline blocks and the streaming mechanisms.

The instruction set was added to the gem5 simulator. The gem5 simulator contains a decoding mechanism that interprets the bits of the machine instructions and selects the appropriate simulation code. Each instruction is written in a domain-specific language (DSL), in which the syntax is a mix between the Python and C++ languages. This DSL allows templating of the instructions in order to reuse code. However, the gem5 simulator was not prepared for scalable registers with the element width as a part of the registers. This resulted in extensive modifications to the code source. In detail, the DSL interpreting mechanism was modified, which involved the usage of regular expressions [56] when modifying the instruction set parser. The instruction set parser is based on Lex & Yacc [57], and the modifications required the basic understanding of how the parser works, and mainly how the interpretation code is structured. The mechanism of parsing the registers was modified to support the new vector registers, alongside with the width and valid index register information. To provide the registers with the width and register information, the vector and predicate register classes were modified with new specific methods and properties.

The process of defining new instructions is not a simple one. Each templated instruction is expanded by the parser to the decode, header, and exec sections. First, the decode section contains the code to decode a machine instruction into a C++ function. Second, the header definitions of the C++ functions that simulate the instruction behaviour are contained in the header section. Finally, the exec section contains the implementation of the functions that simulate the behaviour of the instruction. These sections are output to a set of C++ files and then compiled. To produce the output for all these sections, a system of format, template and programmatic (insts) blocks are in place. These blocks are then filled by hand with code that will be expanded to the respective sections. Each instruction implemented extends on a base C++ class. In order to inherit the methods of the parent class. To better clarify the complexity of this process, Figure 5.5 depicts the process and components of creating one instruction. Although the usage of the ISA definition DSI presents an excellent opportunity to reuse code, not all instructions can share the same template. Thus, the process of defining new templates, formats, insts and base classes must be repeated multiple times, such as for every instruction that uses different operands. Additionally, the presented organization is a complex one and therefore, not easy to understand without proper documentation, which unfortunately is lacking in gem5.

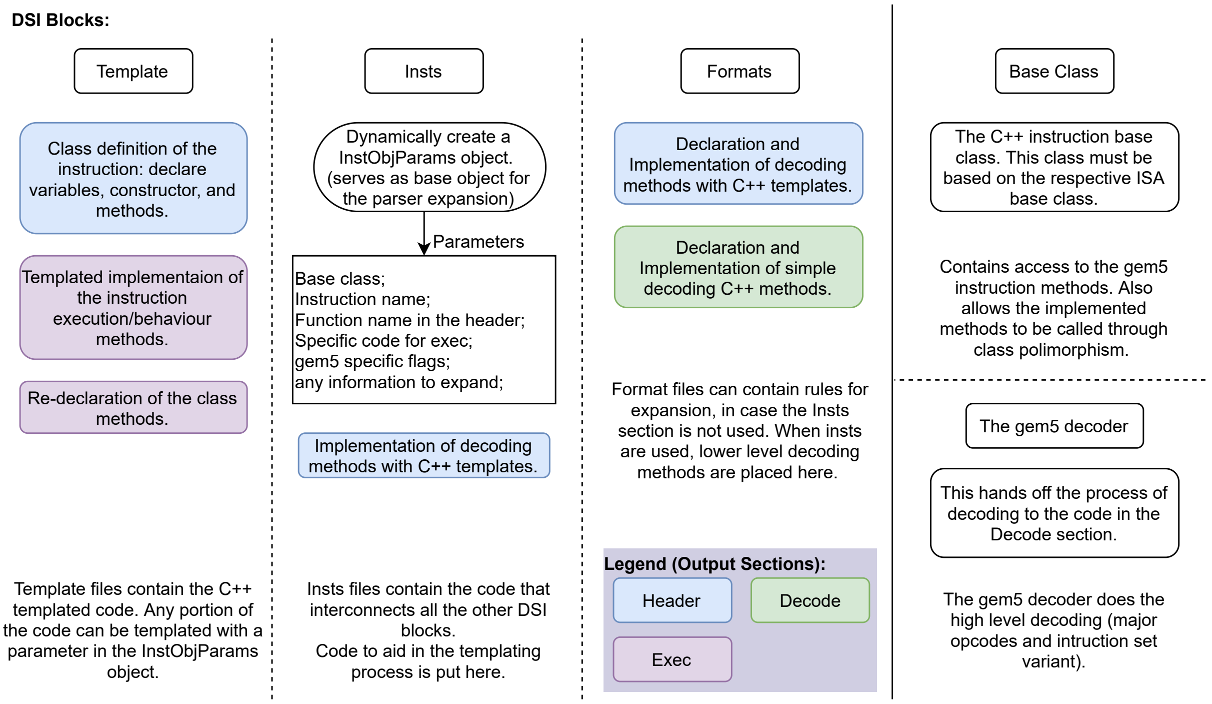

DSI Blocks:
Template
Class definition of the instruction: declare variables, constructor, and methods.
Templated implementaion of the instruction execution/behaviour methods.
Re-declaration of the class methods.
Template files contain the C++ templated code. Any portion of the code can be templated with a parameter in the InstObjParams object.
Insts
Dynamically create a InstObjParams object. (serves as base object for the parser expansion)
Parameters
Base class;
Instruction name;
Function name in the header;
Specific code for exec;
gem5 specific flags;
any information to expand;
Implementation of decoding methods with C++ templates.
Insts files contain the code that interconnects all the other DSI blocks.
Code to aid in the templating process is put here.
Formats
Declaration and Implementation of decoding methods with C++ templates.
Declaration and Implementation of simple decoding C++ methods.
Format files can contain rules for expansion, in case the Insts section is not used. When insts are used, lower level decoding methods are placed here.
Legend (Output Sections):
Header
Decode
Exec
Base Class
The C++ instruction base class. This class must be based on the respective ISA base class.
Contains access to the gem5 instruction methods. Also allows the implemented methods to be called through class polimorphism.
The gem5 decoder
This hands off the process of decoding to the code in the Decode section.
The gem5 decoder does the high level decoding (major opcodes and intruction set variant).

Figure 5.5: The process of defining and expanding one instruction with the gem5 instruction set parser. Each output section is shown in color and positioned in the block that contains the templating code.

UVE relies on a streaming engine, closely related to a prefetcher. However, for instruction set level simulation, it is not possible to define a microarchitecture component. Hence, to obtain results and validate the instruction set part, the supporting microarchitecture was implemented (decribed in Part II), and the results obtained. Even though the ISA has not been simulated independently, this chapter uses results that are exclusively related to the instruction set, and are not affected by the underlying microarchitecture or streaming engine configuration. Naturally, the vector length parameter is not specified by the instruction set, but affect the instruction set results, such as the number of issued instructions. Hence, to assure the validity of the results, all the comparisons between vectorial extension use exactly the same vector length, which is herein set to 512 bits.

5.3 Performance Evaluation

UVE is a vectorial extension with streaming that reduces overhead instructions inside the accelerated loop bodies. To show the impact of UVE, SVE is used as a comparison. The SVE extension is the most developed state-of-the-art technology in vectorial extensions. Additionally, the gem5 simulator and the compilation tools were available and give the needed support for the SVE extension. In contrast there is still no implementation of the RISC-V V extension in gem5, which compromises the comparison of results. Regarding Intel AVX-512, it is based on a completely different paradigm (CISC), which makes the comparison invalid. Hence, such results were excluded from this report.

The comparison between the SVE and UVE extensions is mainly affected by the differences in the number of unnecessary (overhead) instructions, as already seen in Figure 4.20. Hence, with a reduced number of address indexation and memory access intructions it is expected that UVE has a significantly lower number of instructions. In fact, UVE also requires that stream configuration instructions are added, however, these are not looped through, hence do not affect significantly the results.

5.3.1 Memcpy performance evaluation

The memcpy example (depicted in Figures 5.2 and 2.7) can be considered the most simplistic usage of an extension that interacts with memory. I.e. memcpy is the less demanding example in terms of instruction set implementation, as it only loads and stores words from memory in a very simple pattern, and making no computations. Hence, as it is the less demanding case of memory addressing complexity it should represent one of the worst cases (in performance differences) regarding other vectorial extensions that do not employ streaming.

The obtained results are shown in Figure 5.6. The memcpy kernel can be executed with any arbitrary size, i.e. the number of elements to copy can be chosen arbitrarily. The results shown in Figure 5.6 were obtained with size 512 B. This is a small size, considering that it is only eight times higher than the vector registers length, resulting in 8 loop iterations. Even so, small bursts are where the vectorised code would give fewer benefits.

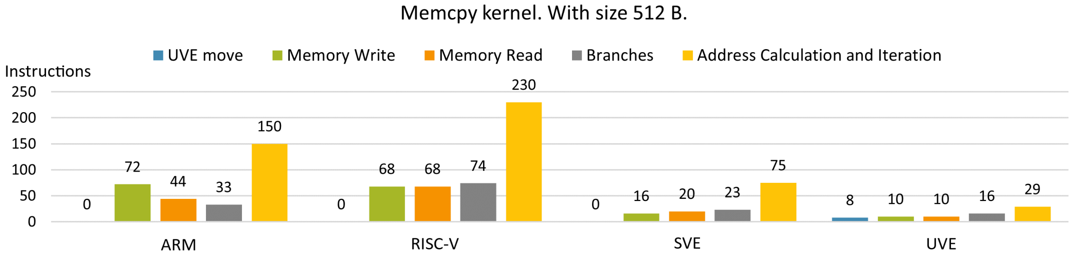
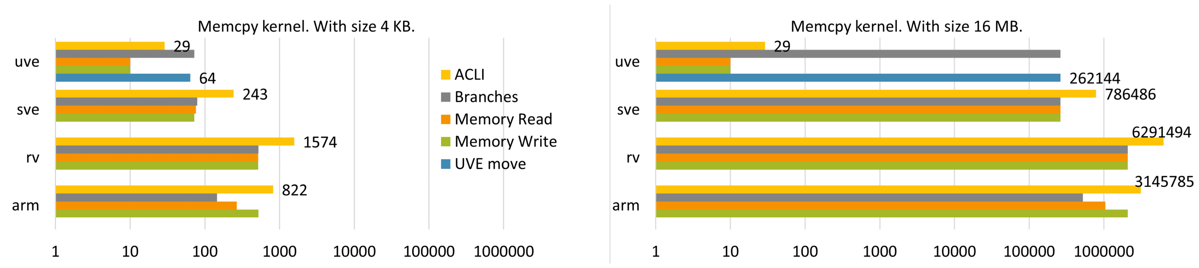


Figure 5.6: Comparison of the number of instructions executed with Memory copy. Procedure that copies a source zone of memory to a destination one. (MEMCPY) kernel. The copied memory is of size 512 B. The UVE move series represents the number of move instructions executed by UVE. The address calculation and iteration series contains all the arithmetic instructions that calculate addresses or control iteration counters. The RISC-V, Arm, and SVE kernels are compiled with full optimization (O3). The UVE kernel was hand optimized with assembly. Both UVE and SVE are configured with 512-bits of vector length.

Analysing the results between the RISC-V and Arm implementations, the latter uses less instructions. This is due to the difference in the optimization methods. In the Arm side, the software made use of the quadword (128-bit) registers, present in the aarch64 ISA, which led to a lower number of reads. However, the store instruction is segmented in two 64-bit stores. In the RISC-V side, the store and load instructions are even. The number of branches is much higher in the RISC-V side due to the compiler handling of the possible edge cases. The Arm pre-fix memory instructions leads to a lower number of address calculation instructions. These instructions calculate the memory address and execute the access in a single instruction whereas RISC-V does each operation in a separate instruction. In sum, RISC-V and Arm are very different approaches in terms of the ISA, which does not directly imply that one performs better than the other in a real implementation.

However, the real comparison for the context of this thesis regards the difference between SVE and UVE. As a note, the UVE move instruction is necessary as the used UVE code uses streaming and not standard load/stores instructions. In terms of memory transactions, the values for UVE are only relative to the configuration instructions, as the streaming mechanism executes the memory transactions. For the number of branches, the minimum theoretical value is given by the size of the data divided by the vector length, which is 8. However, there are configuration processes involved, which require UVE to issue 16 branch instructions. On the SVE side, the number of branches increases due to the compiler handling of the code. Finally, the differentiating result is shown in the address calculation and iteration series. The reduction in instructions is more than 2 times, between the SVE and the UVE results. It is necessary to remember that this case is only at 512 B of data size. To showcase the potential, Figures 5.7a and 5.7b show the results for the sizes 4 kB and 16 MB. The results show that the reduction in address calculation and iteration control is high and increases with higher data set sizes. In addition, the memory load and store instructions are constant and low for the UVE extension. However, the UVE move instruction is now needed to control the data flow. This results in a two to one ratio between the memory instructions in SVE and the UVE move. When observing only the address calculation instructions, the ratio of instructions ranges from 258% (512 B, Figure 5.6) to 27120% (16 MB). This improvements are



(a) Size 4 KB. Equals to 64 vector registers in length.

(b) Size 16 MB. Equals to 262144 vector registers in length.

Figure 5.7: Number of instructions comparison in the execution of the memcpy kernel. The instructions are categorized in 5 series. The UVE move series represents the number of move instructions executed by UVE. The address calculation and iteration series (ACLI) contains all the arithmetic instructions that calculate addresses or control iteration counters. The SVE kernel is compiled with full optimization (O3). The UVE kernel was hand optimized with assembly. Both UVE and SVE are configured with 512-bits of vector length.

washed out by the remaining instructions. Effectivelly, by comparing the total number of instructions between UVE and SVE, a reduction of 300% is observed.

5.3.2 Strided memory transactions performance evaluation

To further explore the potential of UVE, the memcpy example was adapted to use a specific stride. I.e. the elements to transfer are spaced with a given value. In addition to this, the data is also processed and reduced, i.e. the data output is accumulated to a register instead of stored in memory. Figure 5.8 shows the generic form of the example kernel, named paimpac (product-accumulation impact).

```
float paimpac_1( float * src, size_t n, size_t stride) {
  float dest = 0.0;
  for(size_t i = 0; i < n; i += stride)
    dest += src[i]; }
```

```
float paimpac_4( float * src, size_t n, size_t stride) {
  float dest = 0.0;
  for(size_t i = 0; i < n; i += stride)
    dest += src[i]*src[i]*src[i]*src[i]; }
```

Figure 5.8: Paimpac code.Left: variant with kernel complexity one /(accumulation). Right: variant kernel with complexity 4 (accumulation of a power of 4, requiring 3 multiplications). The dest variable is the kernel output.

The kernel was first benchmarked with unit stride and the lowest complexity (accumulation only). The results are depicted in Figure 5.9. For larger input sizes, the instructions ratios remain stable and seem similar to the largest case in Figure 5.9. These results show that the proposed UVE is able to reduce the number of instructions by up to 28x, 24x and 2.5x when comparing with the Armv8, RISC-V and SVE instruction sets. These results also show that the RISC-V ISA is more instruction efficient than the Arm ISA, due to a lower number of integer processing instructions. In the vector extension comparison, the SVE ISA is less efficient than the UVE ISA, as seen in the memcpy example. In fact, although UVE uses more data processing instructions due to the simpler set of instructions, the reduction in the number of memory access and data indexing instructions allows to attain a more efficient kernel encoding. In particular, even for the smallest input size (2^7 elements) it is significantly more efficient than the SVE counterpart.

The kernel was also benchmarked while tuning the kernel complexity and the element stride. Adjusting the stride will test the capabilities of the ISA in dealing with non-contiguous memory accesses. This test will have more impact in non-scalar implementations, due to the need to merge the data into the

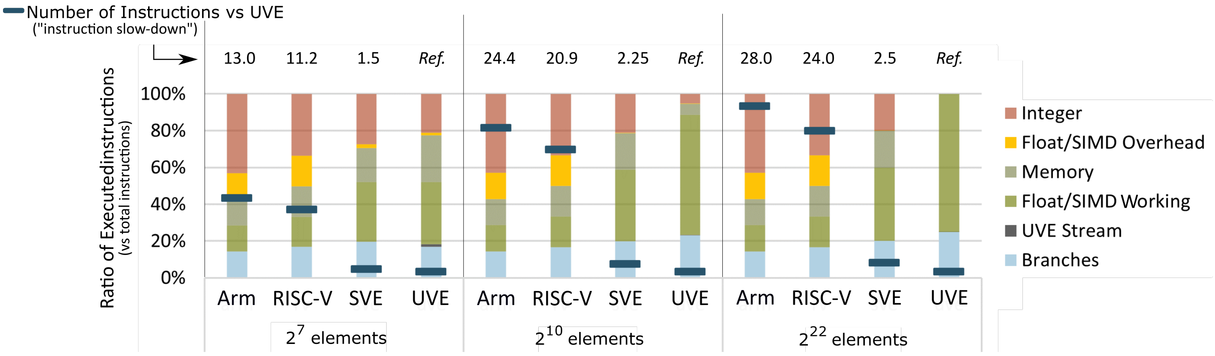


Figure 5.9: Instructions ratio comparison for the paimpac kernel. Each sub-graph depicts the ratio of instructions at a different number of single-precision floating-point elements. The instructions are categorized in 6 series. The integer instructions are directly related with memory address calculation and loop control. The Float/SIMD overhead instructions are data preparation instructions. In oposition, the Float/SIMD instructions are data processing instructions. The UVE stream instructions are the stream configuration instructions in the UVE code. The Arm, SVE and RISC-V kernels are compiled with full optimization (O3). The UVE kernel was hand optimized in assembly. Vector length is set to 512-bits.

vector registers. Additionally, rising the kernel complexity will increase the number of working instructions. Hence, this test is interesting to evaluate the complexity of the available instructions. The results of the explored dimensions are presented in Table 5.1. As expected, by tuning the elemente stride, the scalar ISAs do not suffer any changes in performance. In oposition, the SVE results show a decrease in performance between 2.6 and 3 times, at complexity 1. UVE is not affected by the change in stride, as the streaming mechanism will handle the different strides withouth the affecting the execution of the code. On the other hand, by executing the kernel with complexity 4, the instructions ratios decrease. This is due to the lower complexity of instructions in the UVE extension.

Table 5.1: Paimpac kernel benchmarking with varying kernel complexities and element strides. The Arm, SVE and RISC-V kernels are compiled with full optimization (O3). The UVE kernel was hand optimized in assembly. UVE and SVE at vector length 512-bits.

		Stride at Complexity 1			Stride at Complexity 4		
Size	ISA	1	2	4	1	2	4
2^7	Arm	13.0	13.0	13.0	12.4	12.4	12.4
	RISC-V	11.2	11.2	11.2	11.0	11.1	11.1
	SVE	1.5	3.7	3.7	1.4	3.1	3.1
2^22	Arm	28.0	28.0	28.0	20.6	20.6	20.6
	RISC-V	24.0	24.0	24.0	18.3	18.3	18.3
	SVE	2.5	7.5	7.5	2.0	4.9	4.9

5.3.3 SAXPY benchmark

The SAXPY benchmark was also evaluated for the sake of completeness, and for comparison with the theoretical result given in Figure 4.20. The obtained results are summarized in Figure 5.10. The instructions ratio, at the top of the chart, shows that UVE is effectivelly more instruction efficient than the SVE ISA. However, the compiler generated assembly includes more Arm instructions than the ones used in the comparison (Figure 4.20). Consequently, due to the compilater ineficiencies, the simulation ratio of instructions is higher than the theorectical one. Another comparison to note is the highly efficient

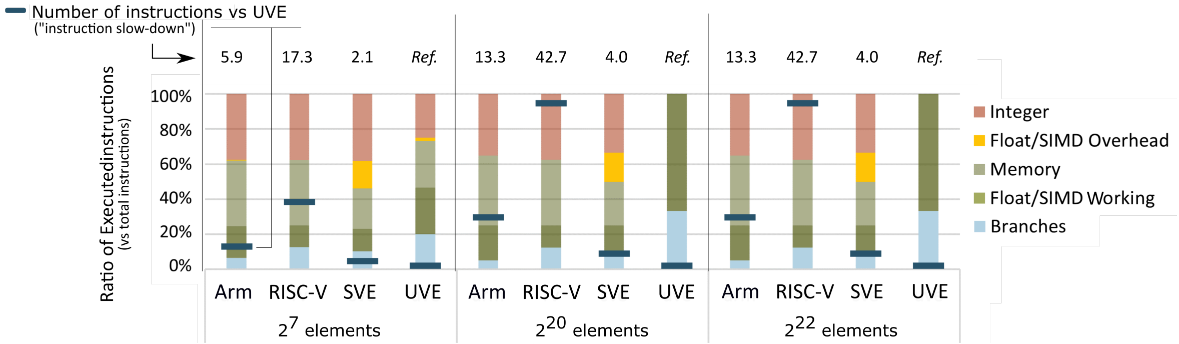
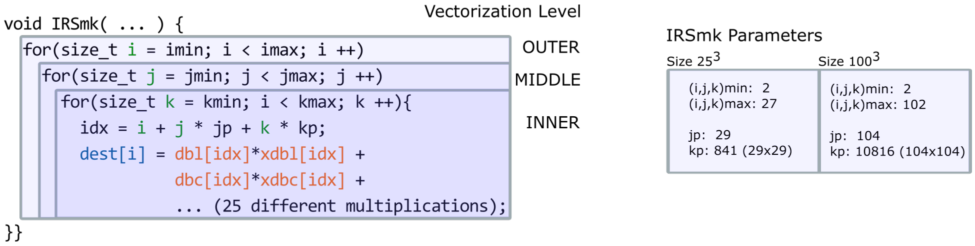


Figure 5.10: Instructions ratio comparison for the SAXPY kernel. Each sub-graph depicts the ratio of instructions at a different number of single-precision floating-point elements. The instructions are categorized in 5 series. The integer instructions are directly related with memory address calculation and loop control. The Float/SIMD overhead instructions are data preparation instructions. In oposition, the Float/SIMD instructions are data processing instructions. Arm, SVE and RISC-V kernels are compiled with full optimization (O3). The UVE kernel was hand optimized in assembly. UVE and SVE at vector length 512-bits.

use of instructions in UVE, where two thirds of the instructions are processing data. Finally, in spite of the higher ratio of instructions in the RISC-V results, the UVE extension is more instruction efficient due to the fact that it is a vectorial extension. In addition to this, the UVE vector extension code susbstitutes directly the less instruction efficient portion of the RISC-V code, which is the loop.

5.3.4 IRSmk benchmark

The IRSmk is a microkernel of the Implicit Radiation Solver (IRS) benchamark [58]. The IRSmk kernel code is depicted in Figure 5.11. This kernel exploits the limitations on the UVE ISA in terms of the number of streams. Due to having 54 vectors that are multiplied and accumulated, it would be impossible to vectorize due to lack of available streams. Even so, it is possible to divide the accumulation into two parts and compute each one separately. In the scalar and SVE ISAs, there is no such problem, as the vector registers can be reused after the results are accumulated. The results of this benchmark are shown in Table 5.2 for a cube size of 25, and in Table 5.3 for a cube size of 100. The results again show that UVE provides a considerable reduction in instructions when compared to the comparing setups. The Arm ISA uses fewer instructions than the RISC-V ISA due to the usage of the Arm quadword registers, whereas RISC-V is limited to 64-bit registers. On the other hand, when comparing the different vectorization levels in the UVE results, it is observed that the ratio between the MIDDLE and OUTER



```
void IRSmk( ... ) {
for(size_t i = imin; i < imax; i ++)
 for(size_t j = jmin; j < jmax; j ++)
  for(size_t k = kmin; i < kmax; k ++){
    idx = i + j * jp + k * kp;
    dest[i] = dbl[idx]*xdbl[idx] +
              dbc[idx]*xdbc[idx] +
              ... (25 different multiplications);
}}
```

IRSmk Parameters

Size 25^3	Size 100^3
(i,j,k)min: 2 (i,j,k)max: 27 jp: 29 kp: 841 (29x29)	(i,j,k)min: 2 (i,j,k)max: 102 jp: 104 kp: 10816 (104x104)

Figure 5.11: IRSmk code and parameters. In the left, the IRSmk kernel with identification of the levels of the possible vectorizations. In the rigth, the reference configuration parameters for cubes of size 25 and 100.. The IRSmk kernel computates the accumulation of 27 multiplications of 54 different vectors.

levels is almost unitary. This is due to the number of constant configuration instructions surpassing the number of data processing instructions, i.e. the advantage of vectorizing more loops is overcome by the increasing complexity of the configuration instructions. Hence, there is no advantage in allowing for arbitrarily extensive levels of pattern description.

Table 5.2: IRSmk results at size 25^3. The overhead ratio relates to the number of overhead (nom-processing) instructions. The instructions ratio relates to the total number of instructions. Both ratios use as reference the UVE architecture with OUTER vectorization level.

Architecture	Vectorization Level	Overhead Ratio	Instructions Ratio
RISC-V	None	1885.86	37.34
Arm	None	445.07	18.59
SVE	INNER	313.79	7.14
UVE	INNER	169.99	2.95
UVE	MIDDLE	10.56	1.06
UVE	OUTER	1.00	1.00

Table 5.3: IRSmk results at size 100^3. The overhead ratio relates to the number of overhead (non-processing) instructions. The instructions ratio relates to the total number of instructions. Both ratios use as reference the UVE architecture with OUTER vectorization level.

Architecture	Vectorization Level	Overhead Ratio	Instructions Ratio
RISC-V	None	119727.93	38.49
Arm	None	19571.73	16.99
SVE	INNER	14545.31	5.59
UVE	INNER	2716.09	1.47
UVE	MIDDLE	40.94	1.00
UVE	OUTER	1.00	1.00

5.4 Summary

The evaluation of the UVE instruction set, shows that UVE uses a significantly lower number of instructions than SVE. In particular, a reduction in instructions of 400% was seen in the Computation of the product of the Real Value A with each element of the Real matrix X added to the respective element of the Real matrix Y (SAXPY) benchmark, while achieving 559% in the IRSmk benchmark.

Taking into account the presented results through this chapter, it can be concluded that the proposed UVE extension is more efficient, however, the comparison could be more fair. Particularly, while the UVE code was hand-optimized through extended assembly, the other codes (RISC-V, Arm, SVE) were not. Even though, all the codes of RISC-V, Arm and SVE were compiled with optimization level 3, which allows the compiler to do instruction scheduling, loop unroling and loop vectorization.

Despite these encouraging results, a CPU is highly dependent on a the actual implementation of the instructions, much like a reduced CISC code (vs RISC) does not necessarily translates to a higher performance. In the following chapters a possible supporting microarchitecture design and implementation are discussed.

Part II

Microarchitecture

6 Microarchitecture support for data streaming

The results of simulating the instruction set already show the potential for the proposed UVE extension. However, a realistic assessment can only be made by also considering the architecture details. Therefore, to better evaluate the proposed solution, it was implemented in gem5 by considering realistic processor models. Although both in-order and out-of-order processor models could be used for this, to assure a valid assessment when targeting HPC applications, it was chosen to base this work on a more complex (but also more relevant) out-of-order model.

The implementation involved multiple modifications to a standard processor (see also the functional diagram presented in Figure 6.1), including: on the register file, on the decode, renaming, execute, write-back and commit stages, as well as on the memory interface to provide support for the proposed streams. Most of these modifications are due to the requirements to support the proposed stream management solutions, as most of the remaining instructions are usually already available in processors that feature SIMD/vector extensions. The following sections detail the architectural changes to the processor model.

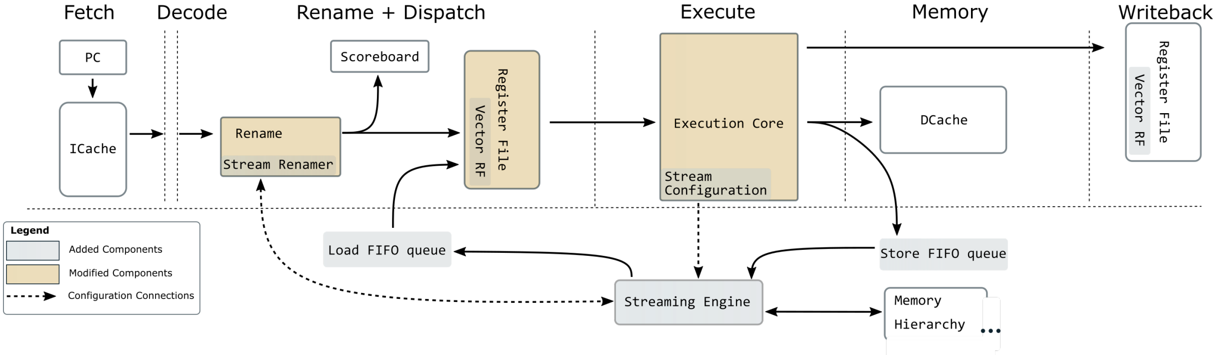
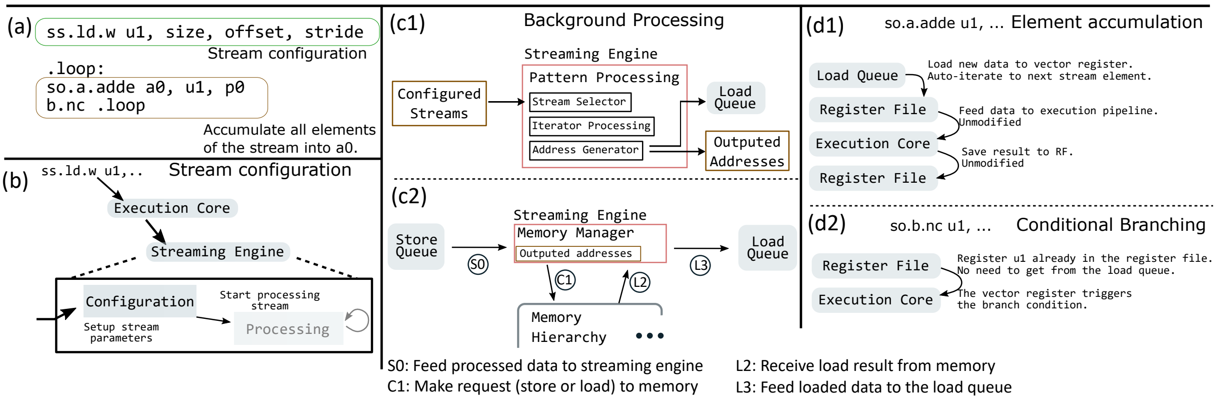


Figure 6.1: Microarchitecture overview block diagram. The added and modified components are highlighted.

6.1 Stream Life-Cycle

The streaming engine manages the streams and the data transactions between the memory and the processor pipeline. Despite being the central part of the streaming system, the streaming process is distributed along the processor. In detail, the life-cycle of a stream spans across all the highlighted components of Figure 6.1, such as: rename, register file, execution, streaming engine and queues. In addition to these, the commit and squashing mechanisms (required to handle miss-speculation issues) are also part of the stream life-span. The commit and squashing mechanisms are distributed throughout the processor and therefore not represented with a block in the diagram. Figure 6.2 succinctly describes the progress of a stream inside the system. The chosen example executes an accumulation loop through all elements of a memory array. When the stream configuration instruction is executed, the execution core will start a new stream in the streaming engine. After the configuration phase, the streaming engine will start processing the configurations and creating the memory access addresses. The generated addresses will then be used to fetch/store the data to/from memory. In the case where a load stream is being used, the data is then fed to the register file and the instructions are allowed to execute.



```
ss.ld.w u1, size, offset, stride
.loop:
so.a.adde a0, u1, p0
b.nc .loop
```

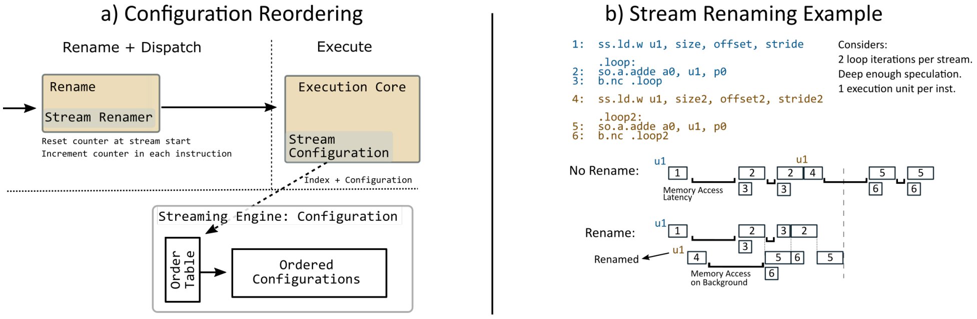
S0: Feed processed data to streaming engine
C1: Make request (store or load) to memory
L2: Receive load result from memory
L3: Feed loaded data to the load queue

Figure 6.2: Load stream life-cycle inside the system. In (a), the example code used to describe the multiple phases of the load stream. The code executes an accumulation through the elements of an array in memory. In (b), the stream configuration phase, which starts the streaming process. In (c1) and (c2), the processes executed in the background are represented. In (d1) and (d2), the processes of vector register consumption are depicted.

6.1.1 Out-of-order stream configuration

The configuration process reveals the first problem regarding the implementation of the proposed UVE on out-of-order processor architectures. If the stream configuration process is allowed to work speculatively, there is no guarantee that the order of stream configuration is maintained. In particular, consider the case where multiple instructions are required to fully describe one stream. Since different instructions may have different dependencies that are satisfied out of order, the configuration order may be changed during execution.

Hence, a solution must be devised that: 1) ensures that the configuration is performed in-order; and 2) does not impose any large penalty while speculatively executing code out of order. To account for this, the proposed solution is to make use of rename stage (which is performed in order) to mark each stream configuration instruction with its order. Figure 6.3a exemplifies the process of reordering the stream configuration instructions. In the rename stage, a set of counters is used to enumerate the configuration instructions, while still in-order. Each counter is associated with a stream, which simplifies the reordering logic. The execution core will send the instruction index alongside the configurations to the streaming engine. The reordering of the instructions is carried in an ordered table which indexes match the configuration instructions indexes.


a) Configuration Reordering
Rename + Dispatch
Execute
Rename
Stream Renamer
Reset counter at stream start
Increment counter in each instruction
Execution Core
Stream Configuration
Index + Configuration
Streaming Engine: Configuration
Order Table
Ordered Configurations
b) Stream Renaming Example
Considers:
2 loop iterations per stream.
Deep enough speculation.
1 execution unit per inst.
No Rename:
u1
1
2
3
2
4
3
5
6
5
6
Memory Access Latency
Rename:
Renamed
Memory Access on Background

```
1:  ss.ld.w u1, size, offset, stride
    .loop:
2:  so.a.adde a0, u1, p0
3:  b.nc .loop
4:  ss.ld.w u1, size2, offset2, stride2
    .loop2:
5:  so.a.adde a0, u1, p0
6:  b.nc .loop2
```

Figure 6.3: Out-of-order configuration reordering and stream renaming. In the left, the out-of-order configuration reordering processing. In the right, a motivational example for the usage of renaming in the streaming registers.

6.1.2 Stream renaming

Each architectural stream can be renamed to a physical stream. To translate the architectural streams to physical streams, the stream rename block is was created. By allowing stream renaming, the processor can configure an additional stream before the previous one has finished. This allows the processor to start prefetching the new stream, before the last one has completed execution (see Figure 6.3b). The principle is the same as in the standard register renaming. In the standard register renaming, some data dependencies are eliminated through the separation of architectural and physical registers [59]. In the stream renaming, the dependencies are eliminated between subsequent stream configurations.

It should be noticed that, dependencies within the same stream (related with data consumption) may also occur. However this is solved with vector register renaming and will be discussed later.

6.1.3 Background stream processing

After the stream is configured (Figure 6.2b), the stream engine is ready to start processing it. The background stream processing is depicted in Figures in 6.2 c1 and c2. This processing phase is clearly distinguished in two portions. First, the generation of the memory addresses uses the stream state and configuration to calculate the memory addresses (Figure 6.2c2). Second, the memory access manager (Figure 6.2c2) uses the addresses generated in the first portion to address the memory hierarchy.

The address generation process is herein named pattern processing.

A) Pattern Processing

The pattern processing process starts by choosing one stream to process. The configured streams are kept in the streaming engine memory. As a design decision for this particular microarchitecture, only two streams (one load and one store) can be processed each cycle. There are multiple techniques available to select the streams to be processed, e.g. random, round-robin, throttled. The random selection strategy is not optimal, as the same stream could be iterated consecutively. Moreover, using a real random seed could prove difficult. With the round-robin strategy, each stream receives the same amount of weight, as this mechanism will circulate all the streams in order, also, it is a simple mechanism to implement. However, giving the same amount of importance to all streams is not optimal. Consider the case where a stream is configured much before the consumption of the corresponding data. When compared to a recently configured stream that requires new data as fast as possible, the first stream processing can be delayed. An ideal mechanism would throttle down the streams with lower throughput. For the sake of implementation time, a fully-fledged throttling mechanism was not implemented. Instead, the round-robin strategy was chosen, but by taking into account the state of the stream queues, hence skiping the streams that do not have space in the queues. Figure 6.4a summarises the stream selection process.

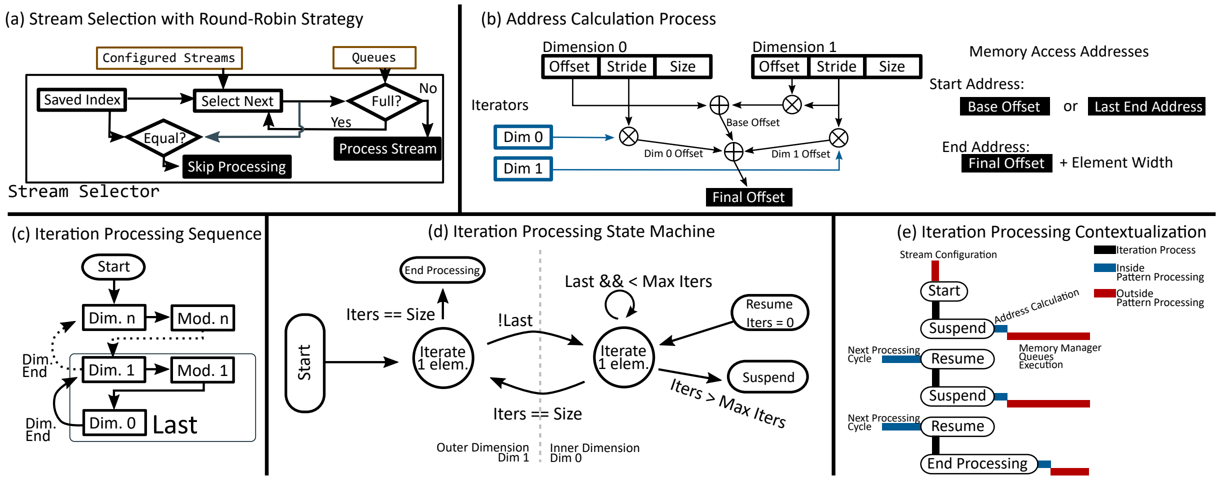


Figure 6.4: Stream pattern processing diagrams and state machines. In the top left (a), the stream selection round-robin based diagram. In the top right (b) the address calculation diagram that is based on the pattern configurations and the iterative state. In the bottom left (c), a overview of the dimensions and modifiers processing sequence. In the bottom centre (d), the state machine relative to the configured patterns iteration. In the bottom right (e), a contextualisation of the state machine (d) in the overall stream life-cycle.

Before going into details on the pattern processing itself, it may be useful to define the necessary fields that represent a stream configuration and the corresponding state. Each stream is composed of a memory access pattern. Each pattern is currently limited to support up to five dimensions and four modifiers. In addition, the state of each stream must state if the stream is not configured, executing or finished. Moreover, each stream state contains the current iterators of the modifiers and dimensions.

The address generation is carried by the state machine depicted in Figure 6.4b. Each memory access can be simply represented by an initial address and a length (in bytes). The diagram in Figure 6.4b depicts the calculation of a final offset based on the configured dimensions and the respective iterative states. Even so, the initial offset must also be calculated. There are three situations where the initial offset must be calculated with different methods. First, in the initial processing of a stream, the initial offset must be calculated for the iterators set to zero (base offset). A second situation is encountered when the memory addresses between address calculation are contiguous, i.e. the second iteration has an initial address that points to one element ahead of the first iteration final address. In short, the last final address can be used to calculate the new initial address. Finally, when the memory addresses are not contiguous, the initial address must be re-calculated from the dimensions and the respective iterators.

Each new pattern processing starts by iterating from the outer dimension to the inner dimension, similarly to a for loop. Each time a lower dimension ends, the iteration process will return to the above dimension, maintaining coherence with a for loop. To speed up the processing of the pattern iterations, multiple iterations are processed sequentially. However, not all addresses will be generated immediately, as this could lead to significant wait times for concurrent streams. Consequently, the processing mechanism will only produce a limited number of iterations per execution. Figure 6.4d depicts an example where two dimensions are processed. The pattern processing method has the following four control steps: Start, Suspend, Resume and End Processing. These control steps allow the processing to be split across multiple instances of the same. Figure 6.4e depicts the control steps inside the more general stream life-cycle. By splitting the same stream pattern processing into multiple phases, multiple streams can be processed in an interleaved style, which reduces the maximum latency between concurrent streams. Figure 6.5 represents the processing of two streams with split and non-split configurations.

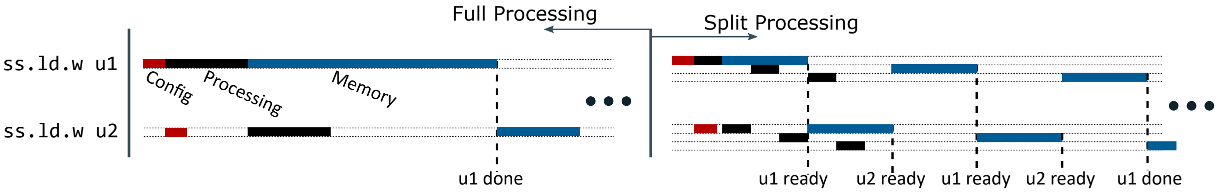


Figure 6.5: Stream processing methods comparison. Two streams are configured sequentially. With the full processing, the stream u1 is completely processed before u2 starts. In contrast, split processing will divide each stream in multiple processing steps, which allows interleaving between streams.

B) Special cases

Multiple distinct configurations can lead to memory access patterns with non-contiguous memory addresses. The simplest example of this is a strided memory pattern. In this case, the distance between elements is more than one element. Therefore, by using the proposed processing mechanism would lead to incorrect memory accesses. In detail, in a case where the stride is 2, the memory addresses would be correctly produced, but only half of the addressed data is needed.

One of the possible solutions to deal with non-contiguous patterns would be to would be to fetch all the data and discarding the unnecessary ones. However, such a solution leads to a waste in memory bandwidth. Another way is to process all the addresses and fetch the data element-by-element. In this case, the processing cycles would be lower, but the bursting mechanisms would not be used and more transactions would lead to poor performance. Still, none of these considerations took into account the caching mechanisms of the memory hierarchy or the possible implementation of a caching mechanism inside the streaming engine. Despite the stride or the efforts to use memory bandwidth more efficiently, when an address is requested to memory, a line sized data-block is loaded into the caches. Consequently, there is no advantage in requesting one element at a time. Even so, it is possible to request data directly to the memory (avoiding the caches); in this case, the streaming engine could benefit from implementing a small caching mechanism, to improve performance in cases where the data is contained to one location but the access order is complex. As a reminder, these were not implemented.

A second example where the memory accesses are not contiguous is related to multi-dimensional patterns. In this case, when an outer dimension is iterated, the produced addresses may not be contiguous. This case presents an excellent opportunity to suspend the processing and produce the memory addresses. When a stream is multi-dimensional, the processing is splitted every time the innermost dimension ends, being resumed when the stream resumes processing, this happens when the stream selector chooses the stream for processing.

6.1.4 Memory access management

The memory access management controls all the streaming data transactions from and to the processor. The memory management is composed by two queues (between the CPU and the streaming engine), and the streaming engine memory controller. Figure 6.6a depicts the organisation of these elements. When streaming data in the context of a load stream (memory to CPU), the first step (1) is to use the addresses produced by the pattern processing to access and address the memory subsystem. However, this process is rather complex due to three major problems:

- Virtual addresses translation;
- Out-of-order memory subsystem;
- Memory level selection.

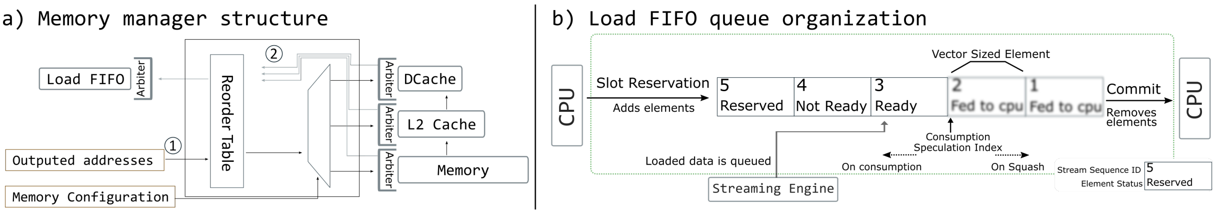


Figure 6.6: Memory manager and queue structures when streaming load data. In the left (a), the memory controller inside the streaming engine and the connections to the surrounding components. In the right (b), the organisation of of the load queue, depicting the connections to the CPU and the streaming engine.

First, the addresses produced by the pattern processing are virtual, and thus need be translated to physical memory addresses. However, this leads to a problem where contiguous virtual addresses may not be contiguous in case of page crossing. Consequently, the memory controller needs to do the translation after the virtual addresses are ready. To translate the addresses, the streaming engine connects directly to the CPU translation lookaside buffer (TLB). However, a further optimization is implemented.

By noting that the n-th least significant bits of the address correspond to offsets within a virtual page (e.g., n=12 for a 4KB page), it is possible to reduce the number of accesses to the TLB. Hence, the memory controller keeps track of the last virtual address translation per stream and, if the next address sits on the same virtual page, the previous translation is used.

The second problem takes into account the out-of-order nature of the memory subsystem, where a requested portion of data may arrive before or after any other request. With this in consideration, when a request is made it is possible to wait for it to be ready and only then issue the following request. However, this would lead to a major inefficiency in getting data from the memory. Consequently, the best approach is to execute multiple requests sequentially and then reorder the data upon arrival, as depicted in Figure 6.7. To be able to reorder data, any memory request is identified and its respective identifier is always attached to the same memory request, from start to finish. This identifier is also saved in the memory controller and used to insert the data in the load queue. Finally, there are multiple levels in the memory hierarchy, and so it may be beneficial to select the memory level for each stream. As an example, the smaller caches could be polluted when streaming a larger stream, which would hurt the CPU processing performance, especially in multi-threaded environments. To select the memory hierarchy, each sream contains information on the cache level from which to get or store the data. This level can be configured via the configuration instructions. One additional consideration to note is that when addressing a cache, it is useful to take into consideration the size of the cache line and to align the requests to that size. The memory controller will separate the initial memory request into multiple requests that are aligned with the cache lines, thus improving the memory access performance.

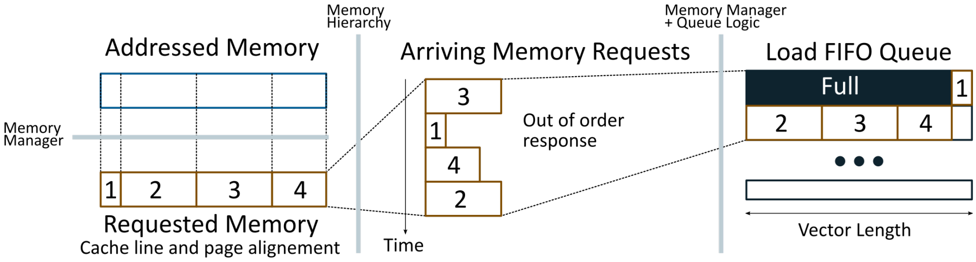


Figure 6.7: Data reordering when streaming load data. Depiction of the memory requesting, reordering and queue insertion process. In the left, the addressed memory (already translated) is split into cache line aligned requests. In the centre, the requests which arrive to the memory controller out-of-order. In the right, the elements are reordered in the load queue. Each state transition is carried by the components identified by the vertical grey lines.

After the data as arrived to the memory controller (following the memory request), the data is inserted into the load queue. The queue organisation is represented in Figure 6.6b. Each data element that arrives to the queue is inserted according to the identifier (stream sequence id), i.e. the data is always reordered upon insertion on the queue. The process of inserting a data element into the queue starts before the data is available. In fact, once the memory request is made, the CPU requests data from the queue which marks a queue element as reserved, meaning that the respective element will receive data from the streaming engine. However, the reverse can also happen if the data arrives before the CPU request. In this case the slot is marked as having data, and will be reserved and ready when the CPU request arrives. The data can easily arrive before the CPU request when a stream is configured much before the CPU starts to rename the consumption instructions.

After the slot is ready (reserved and full), it is time for the CPU to get the data. Due to the speculative execution inside the CPU pipeline, it is necessary to provide speculative execution mechanisms inside the queues, otherwise it would harm the CPU performance by forcing the CPU to be in non-speculative execution. To allow for speculative execution in the load queue, a speculative iterator was created. This iterator indicates which queue element is the next to be fed to the CPU. In addition to this, the same iterator will advance when a consumption of data occurs and goes back on the event of mispeculation (achieved by saving a non-speculative iterator, which advances only upon commit of the consuming instructions). Taking into account that the CPU makes the queue slot reservation upon the renaming stage, and that the data is put into the vector registers (inside the register file), in spite the data being fed orderly to the registers, the instructions can still execute out-of-order. In other words, with the slot reservation being an in-order operation it is expected that the consumption being also orderly does not injure the overall performance. In fact, the additional latency of the in-order consumption can be hidden by filling the queue in the background, i.e. while the CPU waits for the next element to be ready to consume, other elements are becoming ready at the same time, thus removing the wait in further consumptions. Admittedly, this is a point to improve in the queue architecture. However, by allowing orderly consumption we can significantly simplify the speculation mechanism. To end, after a commit of a consuming instruction happens in the CPU, the queue is notified and the element is removed from the queue as it is no longer necessary. In fact, when the last element is committed, the stream is considered as terminated.

6.1.5 Data buffering and consumption

The presented load queue organisation implies that each stream has it own queue. In fact, that organisation was the one used in this work. However, it is recognised that it is not the most performing in space utilisation. In fact, it may also not be the one that gives best processing performance. In spite of this work using a simple implementation of a queue, it is useful to discuss other alternatives that could improve the overall performance of the queues. Figure 6.8 depicts three different queue configurations that we intend to discuss. The first configuration (Figure 6.8a) is the aforementioned simple configuration where each stream has one independent queue. Considering that each vector register can only be coupled with one stream, and consequently that stream is either not configured, a load stream, or a store stream, it is simple to arrive at a unified version of the queue (Figure 6.8b), where the same queue is shared by the load and store stream. This effectively reduces the queue size in half, while not affecting the performance. However, it is still possible improve the performance figures.

Taking into account that not all codes will saturate all the streams, there may be gains in prioritising

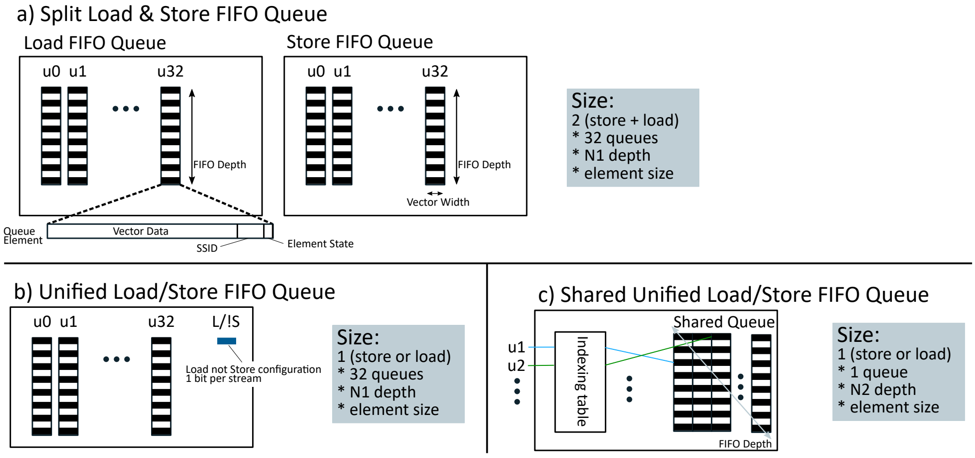
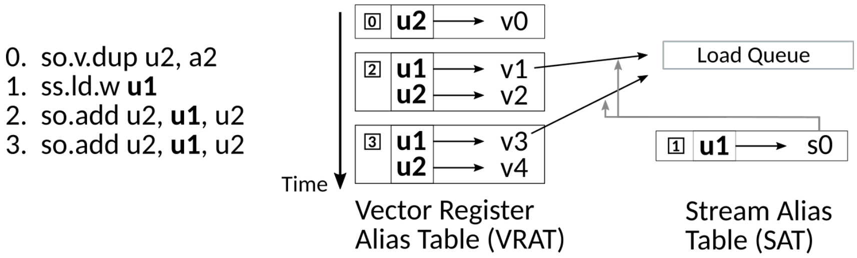


Figure 6.8: FIFO queues configurations comparison. In the top, the most basic configuration where each stream has it own queue. In the bottom left, a unified queue where each load and store stream share the same respective queue. In the bottom right, a more complex shared queue, where the same queue is share among all the streams.

the one stream over another. In the case of the queues, a higher priority stream would have more stream slots available than the lower throughput streams. To allow for dynamic sizing of the queues, an unified and shared queue is proposed, based on the design proposed by Crago et al. in the Outrider architecture [19]. By using an indexing table it is possible to virtualise a wide buffer into multiple queues. This design has a clear impact on the complexity of the queue logic. However it allows for more flexibility on the total queue size and brings dynamic sizing to the queues, which in place gives the ability to do dynamic throttling of the streams.

The consumption process happens when a new data processing instruction that reads from the stream register passes through the renaming stage. When this happens, the load queue will be advised of a new consumer. Before the execution of the named instruction, the load queue will have the data in the vector registers which will be read by the instruction. However, when two subsequent stream consumption instructions arrive at the rename, how does the load queue distinguish between the vector registers of the two instructions. This problem was ignored until now, which simplified the overview of the consumption process. The solution to this problem is vector register renaming. Here, the proposal is to extend the standard register renaming methods and allow for source registers renaming. In common architectures, only the destination registers are renamed, while the source registers are pointed to the previously renamed destination registers. Figure 6.9 represents a typical renaming situation in a streaming situation. Notice that the register u1, which is used as source operand twice, is always renamed. The physical register (renamed) is then passed to the load queue, which will couple the vector register data with the vector register identifier. This guarantees that each instruction receives the correct streaming data.



```
0. so.v.dup u2, a2
1. ss.ld.w u1
2. so.add u2, u1, u2
3. so.add u2, u1, u2
```

Figure 6.9: Vector register renaming upon consecutive data consumptions. The changes in the vector register alias table and stream alias table are depicted in relation to the presented code. The load queue is updated with the renamed vector registers that are tied to a stream.

6.1.6 Store streams

Although the store streams have been previously mentioned, there is no detail on how they work and differ from the load streams. This subsection explores how the store streams work, specifically in the queue, memory management and CPU production processes. The pattern processing component bares no differences between the load and store processes.

When processing store streams two processes happen in parallel. On the CPU side, instructions write data to the store queue (writeback stage). On the streaming engine side, the memory addresses are generated. Both the data and the addresses are then needed by the memory controller to issue the store requests to the memory hierarchy. These steps are represented in Figure 6.10a, where the addresses are saved in the address queue while waiting for the data in queue to be ready for store. However, the opposite can happen, where data is ready before the addresses, in this case the store queue will keep the data until there are addresses. Similarly to the load memory requests, the store requests are also aligned with the cache line and the addresses translated in the TLB.

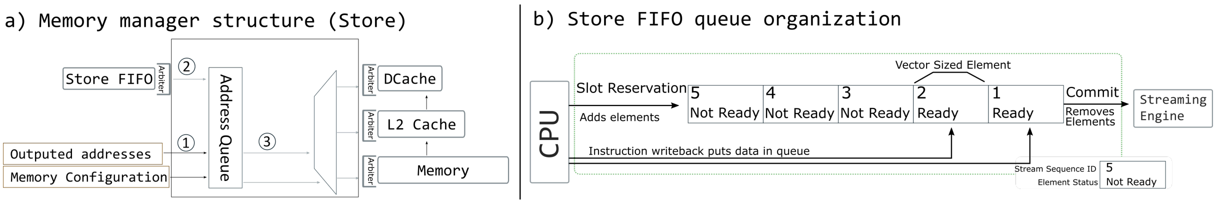
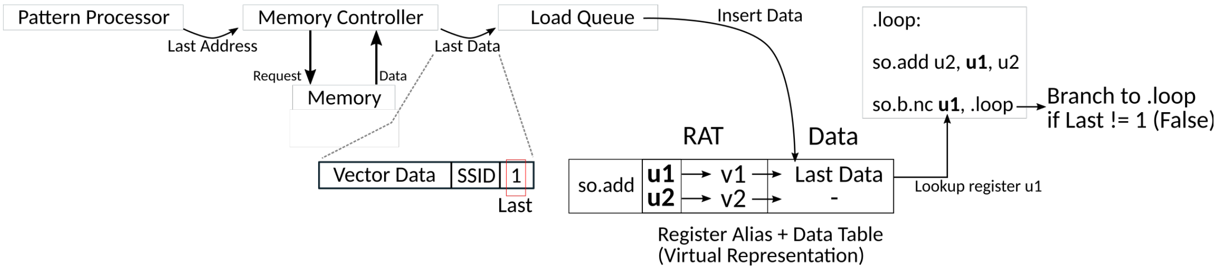


Figure 6.10: Memory manager and queue structures when streaming store data. In the left (a), the memory controller inside the streaming engine and the connections to the surrounding components. In the right (b), the organisation of of the load queue, with depiction of the connections to the CPU and the streaming engine.

On the store queue side (Figure 6.10b), the logic is quite different from the load queue. In detail, upon CPU writeback, the data is inserted into the respective queue slot. The reservation of the slot happens in the rename stage, to guarantee that the queue has enough space before execution, and stalling the pipeline if it does not. After the commit signal arrives from the CPU, the queue slot is marked as committed and the data will be transferred to the streaming engine (if there are available addresses). In contrast, when a squash signal is given, the queue must discard the squashed slot, and so the most recent elements are discarded.

6.1.7 Detection of stream termination

The streaming process ends when all the data as been consumed by the CPU (load streams) or stored to memory (store streams). When loading data, the iterations will be controlled by the branch instructions, which depend on the actual stream status (terminated or not). To give the stream status information to the branch instruction, an additional flag is encoded to the vector register data. This flag is set by the streaming engine when the stream has terminated. The stream branch instruction will reuse the last register used by a stream consumption instruction. The stream branch will only consume a new register if the last instruction that used the same architectural register is also a stream branch. The stream ending detection on loading scenarios is depicted in Figure 6.11.



```
.loop:
so.add u2, u1, u2
so.b.nc u1, .loop
```

Figure 6.11: Stream life-cycle on the last stream element. The major components of the streaming process, when consuming, are depicted alongside a possible life-cycle for the last element of data of a stream. The last element carries the last flag which is enabled in the pattern processing block and carried to the stream branch execution, where it is used.

When storing data, this same process is not replicable, as the production instructions will only communicate with the streaming engine after the commit. Here, the stream branch will communicate with the store queue and access if the corresponding slot was marked as being the stream end. This marking is handled by the streaming engine when the last address has been produced.

6.2 Speculative execution

This section aims to provide a more detailed view of the speculative execution inside the streaming components. There are two main processes when streaming data, namely: the stream configuration where the stream is allocated and configured with the memory access pattern; and the data production (store stream) or consumption (load stream) where the actual data transaction occur. Figure 6.12 breaks these processes into the respective pipeline stages. Notice that the commit and squash stages are represented, however these are not actual stages and are distributed throughout the processor. In fact, the squash event can occur before a instruction is in either of the pipeline stages (e.g. rename, issue, execute, writeback).

6.2.1 Configuration

The stream configuration process starts in the rename stage where a reorder identifier is produced and used to reorder the instruction in the execution stage, thus allowing for out-of-order execution while

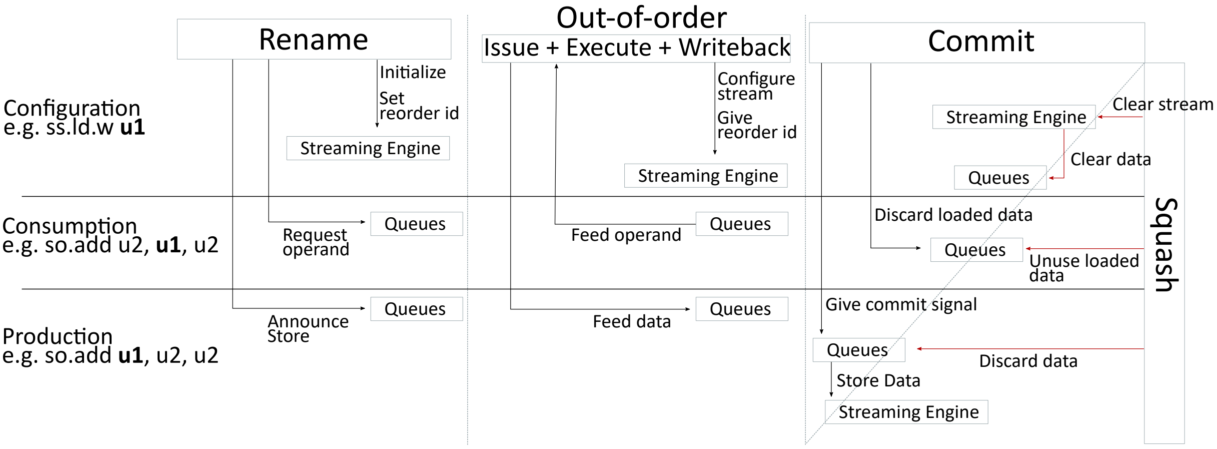


Figure 6.12: Speculation handling in multiple streaming phases. The streaming process phases (configuration, consumption, production) are related with the various pipeline stages and pseudo-stages (commit and squash). The depicted elements relate the speculation mechanisms inside the pipeline and the respective events in the streaming components.

maintaining configuration coherency. When the instruction executes it is seen by the streaming engine as valid, until a squash is triggered. In case a squash occurs, the streaming engine will clear the configuration data given by the instruction and send a clear signal to the queues. This results in all the in-flight memory transactions for that stream to be invalidated on arrival. Additonally, the stream is marked as not configured in the streaming engine, and the respective elements in the queues are emptied. When a commit occurs, there is no change to the streaming engine, as it previously assumed a valid configuration.

6.2.2 Consumption

With consumption instructions (e.g. adds and subtracts), the rename stage will communicate a load intent to the load queue. The rename stage is in-order, and consequently the load intent will also be in-order. This way the data is guaranteed to be processed orderly. When the load queue is ready to feed the requested data, it will communicate to the pipeline that the data is ready. This step involves that the data is loaded to the respective vector register in the register file. The connection between the data request and data feed is made through a identifier of the target vector register. This register is allocated by the rename stage and filled by the streaming engine. When the filling process ends, the scoreboard is updated, allowing the instruction issue. When an instruction is squashed, the data in the queue is marked as unused (the speculation pointer is reversed). If the instruction is valid, the data will be discarded from the queue, as it is not needed anymore.

The load queue speculation mechanisms are shown in Figure 6.13b. The load stream is considered to have 5 slots of depth in this example, the initial state has 2 slots available, 2 reserved but without data (RDIn) and one reserved and filled with data (RDCo). The position of the speculation pointer indicates that the ready slot was already consumed by the processor. On the event of new data from the streaming engine, the respective element is marked as filled, in this example an already reserved slot became

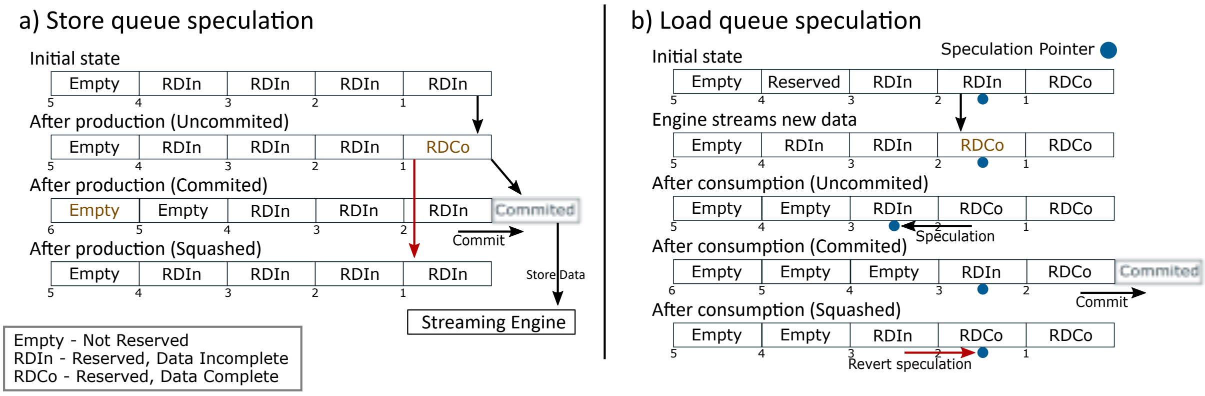

a) Store queue speculation
Initial state
Empty
RDIn
RDIn
RDIn
RDIn
After production (Uncommited)
RDCo
After production (Commited)
Commited
Commit
After production (Squashed)
Store Data
Streaming Engine
Empty - Not Reserved
RDIn - Reserved, Data Incomplete
RDCo - Reserved, Data Complete
b) Load queue speculation
Speculation Pointer
Initial state
Reserved
Engine streams new data
After consumption (Uncommited)
Speculation
After consumption (Commited)
Commit
After consumption (Squashed)
Revert speculation

Figure 6.13: Speculation mechanisms in the queues. In the left, the load queue speculation mechanism. In the right, the store queue speculation mechanisms.

ready. This slot can now be fed to the CPU. When consumed, the speculation pointer advances. After this, in case the speculation is valid, the already consumed slot is committed and therefore discarded. This advances all the elements on the queue. Notice that the speculation pointer did not move. In case of a squash, only the speculation pointer is reverted, and no queue elements are moved.

6.2.3 Production

For instructions producing data to the output streams, the speculation mechanisms are rather simplified. Here, the CPU has complete control of what executes speculatively. In fact, all the output addresses can be executed speculatively, with no influence on speculation or security. In contrast for output streams the only source of hazards is originates on the writing of output data, which must be delayed until the instruction is committed. To cope with this, the streaming system was designed following the CPU storing mechanisms speculation structure.

The production instructions behave as depicted in Figure 6.12. In the rename phase, the CPU announces a store (production) event to happen. This event will allocate a slot in the store queue, that will be later used to actually store data. The slot reservation provides two essential capabilities. First, in the case where the queue is full, the rename will be forced to wait for resources. This allows for maintaining the data order in the out-of-order execution paradigm. It is also worth mentioning that the streaming process will retain the data inside the queue until a commit or squash signal arrives. In the event of a commit (Figure 6.13a:Committed) , the queue will announce to the streaming engine that the data is ready to be stored. In the case of a squash (Figure 6.13a:Squashed), the queue slot will be discarded, as the produced data is not valid.

6.2.4 Faulting memory accesses

Executing speculatively will use more data than the one loaded by the streaming engine, or try to store more data than the produced addresses. This would ultimately result in invalid data loading and the store queue becoming full. When loading data, if no more data is available we must allow the CPU to continue the execution as if it was valid. In this case, the queue will not give additional data to the CPU, instead the queue will update the scoreboard as if the data in the register was valid, and the execution proceeds normally while invalid. To keep control over the speculation, the streaming engine and queues will only trigger an exception if a commit is made on invalid data. The real problem here is how to distinguish if the invalid execution is created by the program or by speculative execution. When in speculative execution, the commit will not arrive, and instead a squash is triggered, not causing an exception. If the error is caused by the program, then the commit will trigger the exception, and the program will be notified of the error. By acting only on the commit, we guarantee that the invalid execution is correctly handled.

When storing, there are no concerns about the speculative execution. Therefore, if a commit is made and there are no addresses in the queue, the streaming engine is checked to confirm if the stream is executing or not. If it is not executing, then a program error was detected and the memory access is invalid, causing an exception.

6.3 Summary

The presented microarchitecture design was implemented on top of the gem5 simulator. To support the modifications, the file and code structure of the gem5 simulator was studied in depth. In detail, the simulator had a complete out-of-order CPU model implemented which was modified and extended to support the stream supporting microarchitecture. The majority of the out-of-order CPU blocks were modified in some extent. In addition, new simulation objects were created to support the streaming engine and related blocks. Figure 6.1 presents a simple overview of the modified code blocks of the CPU model, and the created streaming engine blocks. To complement it, Figure 6.14 depicts a microarchitecure overview that is closer to a real CPU organization, this microarchitecture is based on an ARM Cortex-A76 model with modified parameters [60]. The modifications are further detailed as follows:

- **Rename**: Added a stream rename unit; enabled vector register renaming to rename source operands when streaming; scoreboard updated; connections to the queues and streaming engine created.
- **Register File**: Added additional flags and control fields to the vector registers; connections to the queues added.
- **Issue, Execute, Writeback (IEW)**: Added connections to the streaming engine for stream configuration operations.
- **Reorder Buffer**: Created a connection to the streaming engine and queues to support the commit signal; created support for the squashing signal to be sent to the queues and streaming engine.

In addition to these, the streaming engine and queues were also created from scratch. In short, the implemented functions consinst in:

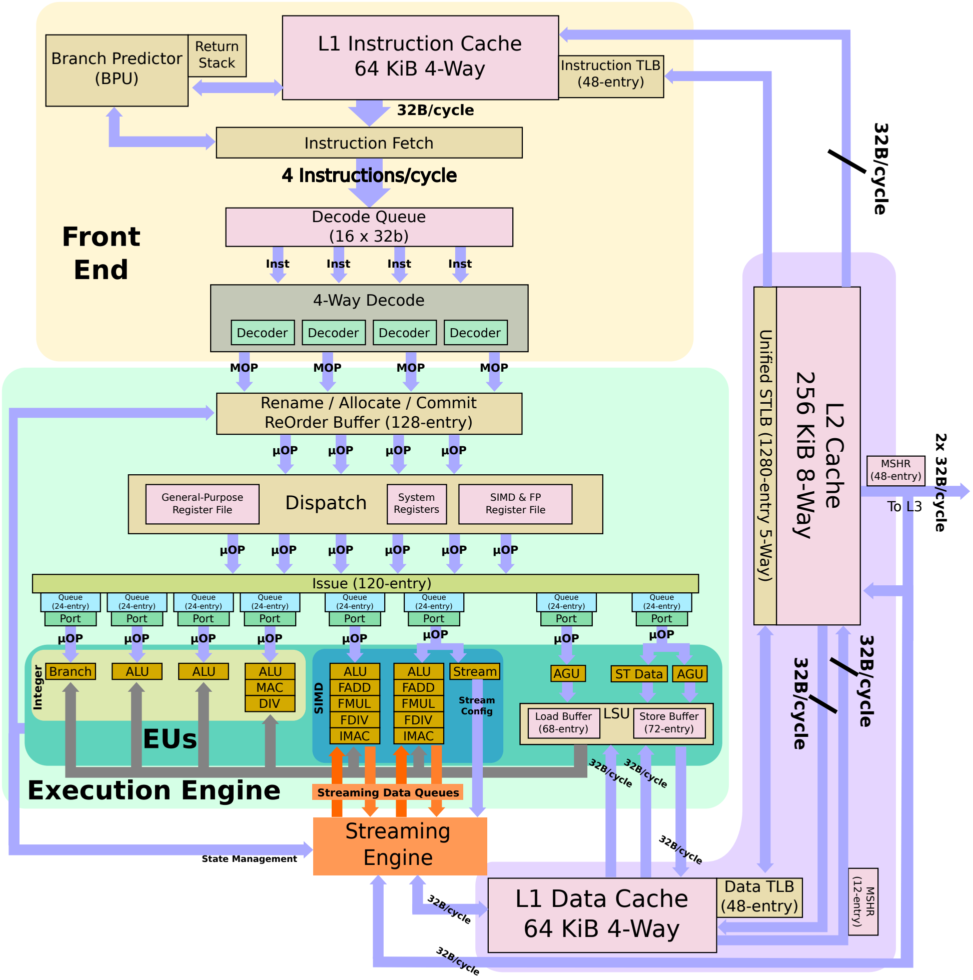


Figure 6.14: Microarchitecture overview of the processor model. The various processor units and in-between connections are depicted. Figure adapted and extended from [60].

- **Streaming Engine**:
  - **Configuration Module**: Configuration reordering and validation; conversion of ISA level pattern representation to streaming engine pattern representation.
  - **Pattern Processing**: Address generation from stream (pattern) representation; control of stream processing flow.
  - **Memory Controller**: Management of address translation and memory requests; management and forwarding of data to queues; reordering of memory requests and data.
- **Buffering Queues**: Enforce coherency between data requested by instructions and the streamed data; decoupling between data request and access of the core pipeline; handling of speculation on the streaming process;

In total, the modifications and additions to the gem5 simulator source code make up a total of twenty-five thousands lines of source code. In addition to these, thirty thousand lines of code were also written to create supporting scripts, tests, tools, and utilities.

Parameter Tuning When creating a streaming engine and any support blocks some parameters were decided based on simple tuning or by reasonable guesses. As an example, parameters such the latency of calculating a memory address or the queue depth were coarsely tuned based on the latency of other similar architectural components. By being on a high-level simulation, the lower level constraints (timing, area and energy) do not impact the feasibility of the higher level simulation. However, during this work an effort was made to choose reasonable parameters, optimize the data structures, and foresee eventual hardware constraints. One example of this effort is the queue depth, were it could be set to any value to increase performance. Even so, this work uses a size of 8 vector elements of queue depth, which value was validated through a set of tests to present a good balance between hardware resources and system performance. The tuning of this parameter could be improved by using queueing theory [24, Appendix D-24], where the implementation could be tuned for a specific set of benchmarks or applications. Another example is the number of addresses generated per cycle, in this case a maximum of one vector element (64 addresses) per cycle was defined. In fact, this value can only be achieved in one dimension linear and consecutive memory accesses, where the address calculation is straightforward. In more complex cases the address throughput can be limited to one address per cycle.

7 Results Overview and Discussion

To approach a realistic simulation, not only the simulator needs to be accurate but also the respective configuration parameters. To ensure the obtain results are meaningful, CPU model was heavily based on the one used by ARM in the SVE introductory paper [7]. We believe that by comparing this implementation with SVE while using the aforementioned model, the results achieve a scrupulous level of comparison and validation. To further guarantee a valid model, a prefetcher was used in the L2 cache. Particularly, the Access Map Pattern Matching (AMPM) [12] and Best-Offset Prefetcher (BOP) [13] prefetchers were used as they were the most recent prefetchers with gem5 implementations. In particular, there are newer and better prefetchers such as the Bingo spatial data prefetcher [61], however, the performance improvements over the AMPM and BOP prefetchers are negligible (around 10% better). Specifically, only AMPM for the evaluations was used, as the BOP implementation did not worked correctly in the chosen gem5 version, the poor BOP results are depicted in Figure 7.2.

The used CPU model is displayed in Table 7.1. The model was used through all the following benchmarks, except in the cases where the experiment aims to assess the sensitivity to parameter variation.

Table 7.1: CPU model parameters. The base parameters of this model were collected from the Arm SVE paper [7]. The vector size was set to 512-bits, which was the best performing vector size in the SVE paper.

Common (UVE and SVE)	L1 instruction cache	64KB, 4-way set-associative, 64B line
	L1 data cache	64KB, 4-way set-associative, 64B line, 12 entry MSHR
	L2 cache	256KB, 8-way set-associative, 64B line
	Decode width	4 instructions/cycle
	Retire width	4 instructions/cycle
	Reorder buffer	128 entries
	Integer execution	2 x 24 entries scheduler (symmetric ALUs)
	Vector/FP execution	2 x 24 entries scheduler (symmetric FUs)
	Load/Store execution	2 x 24 entries scheduler (2 loads / 1 store)
	Vector Size	512-bits
UVE	Queue Depth	8 x Vector Size
	Total Queue Size	64 x Queue Depth (32 load, 32 store)
	Stream processing units	2 (1 load, 1 store)
	Address generation throughput	max. 64 addresses/cycle/unit
	Stream configuration latency	2 cycles
	Address to request latency	min. 2 cycles (4 cycles when translating)
	Stream engine to register latency	min. 3 cycles

All the simulations were executed in system call emulation mode. The gem5 simulator supports full system simulation where the entire system and peripherals are simulated; however, at the time, no support for full system simulation of the RISC-V instruction set existed. The system call emulation mode emulates all the system calls made by the executed code and removes the need for an underlying kernel. Taking into account that this work aims to create and evaluate a scalable SIMD vector extension, the detail of the system call emulation simulation is sufficient. To justify, the full system simulation allows for a complete operating system to be run with the simulated processor. While, in system call emulation that is not an option and any necessary calls to an operating system are forwarded to the host operating system and thus not simulated. For this work, which defines a first iteration of the UVE extension, a complete system simulation is considered to be out of the scope, and therefore there were no intents in allowing for a full system simulation.

The following sections contain results for the executed benchmarks and the respective analysis.

7.1 MEMCPY performance evaluation

The memcpy (Memory Copy) benchmark executes a simple, uni-dimensional memory transaction from a point in memory to another one. As memcpy is a memory-bound benchmark, it allows for the comparison between the memory access performance of the different ISAs. This benchmark starts by filing the source portion of the memory with random data, and then cleaning the caches by overwriting. Then, the time is measured between start to finish of the transaction. Figure 7.1 shows the relative execution times (speedup) versus the base ARM processor using NEON, for a given transaction size with an AMPM L2 prefetcher or without any prefetcher. The UVE results are presented with 3 memory connection configurations, each one representing the memory level from where the data is requested in loads (bypassing the upper levels).

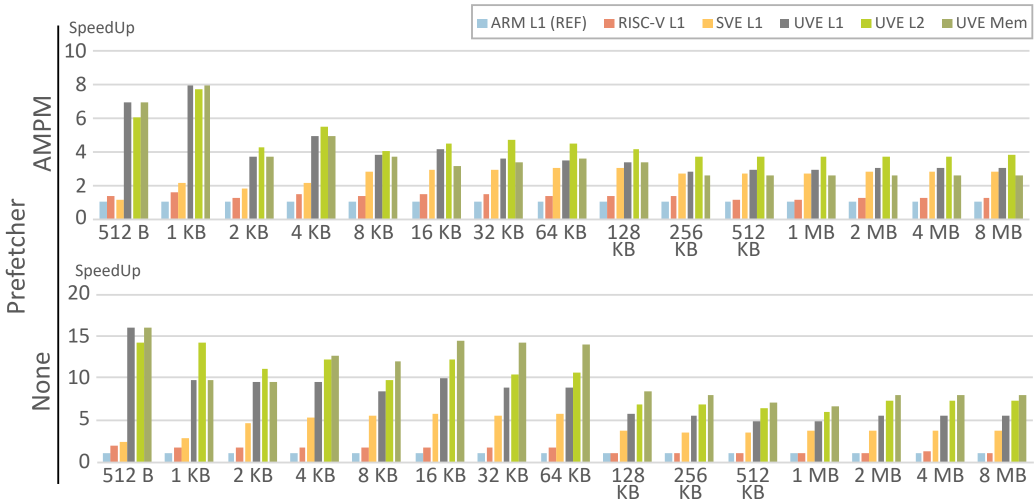


Figure 7.1: Memcpy performance. Comparison between ISAs and memory connection with (AMPM, at the top) and without (none, at the bottom) prefetcher. In each group of columns, from left to right: Arm result (as the reference time); RISC-V result; SVE result; UVE results with different streaming memory connections.

In general, for the AMPM results, the UVE connected to the L2 cache has the higher speedup. Here, the L1 and L2 caches make use of the AMPM prefetcher installed in the L2 cache. While connected to memory there is no prefetcher. The direct memory access could be improved by increasing the data queue size, and consequently allowing for a deeper prefetching. In similarity, if the streaming engine fetches data from L1 or L2, the L2 prefetcher will virtually increase the total prefetching depth, resulting in better performance. As a concluding note on UVE, the connection to the L1 suffers from the additional cycles of the L1 to L2 accesses.

For the SVE performance, it is close to the UVE speedups. As memcpy is a memory-bound benchmark, the SVE results are expected to be close to the maximum achievable bandwidth for the implementation, and consequently there is not much performance room for the UVE streaming prefetching mechanisms.

When using no prefetcher, the differences between the UVE and SVE results are augmented. By having no L2 prefetcher, all the implementations are slowed down with the slower memory access. In this case, the UVE results are much better with the streaming engine connected directly to memory, as this represent the lowest latency when loading the data.

To better compare the effects of using a prefetcher with the UVE streaming engine, Figure 7.2 represents the relative times for a set of prefetcher configurations. The prefetcher configurations are none, stride, best-offset (BOP) and AMPM. The Arm and RISC-V results are hugely improved (2.6 times) by using the AMPM prefetcher. Accordingly, the SVE (1.9 times) and UVE L1 (1.5 times) and L2 (1.5 times) also improve with the AMPM prefetcher. Notice that UVE is the one that improves less with the addition of the prefetcher, as the streaming engine is already prefetching. For UVE connected to the memory, it shows that the prefetcher does not significantly affect performance, as it is not connected to the to the L2 cache where the prefetcher sits. In detail, there is some effect on the performance, as the scalar and floating-point loads are still using the prefetchers. However, this effect is meaningless for the longer transactions.

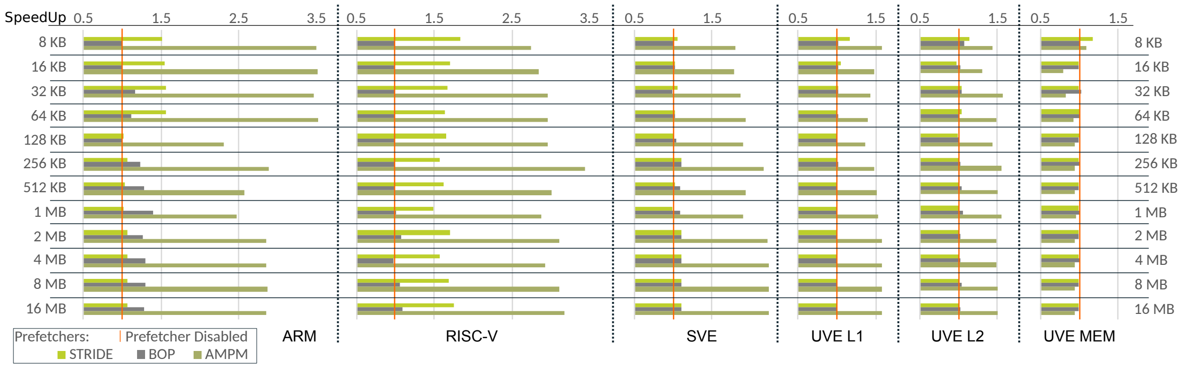


Figure 7.2: Memcpy benchmark, comparison between prefetchers and their effect on the ISAs. The speedup reference value is given by the prefetcher disabled series, represented by the vertical line.

To complete the memcpy benchmark analysis, Figure 7.3 shows the memory utilization of the memory access bus, with AMPM prefetcher enabled. In the figure, the darker areas represent a higher bus utilization. Notice that, since the amount of transaction data remains approximately the same, for the same problem size, the lower the execution time, the higher the bus utilization. On the other hand, the lower average utilization between UVE L2 and the other UVE connections represents a higher efficiency on the utilization of the bus. In general, for the same transaction size, UVE is more efficient and attains a higher memory access performance than the counterparts.

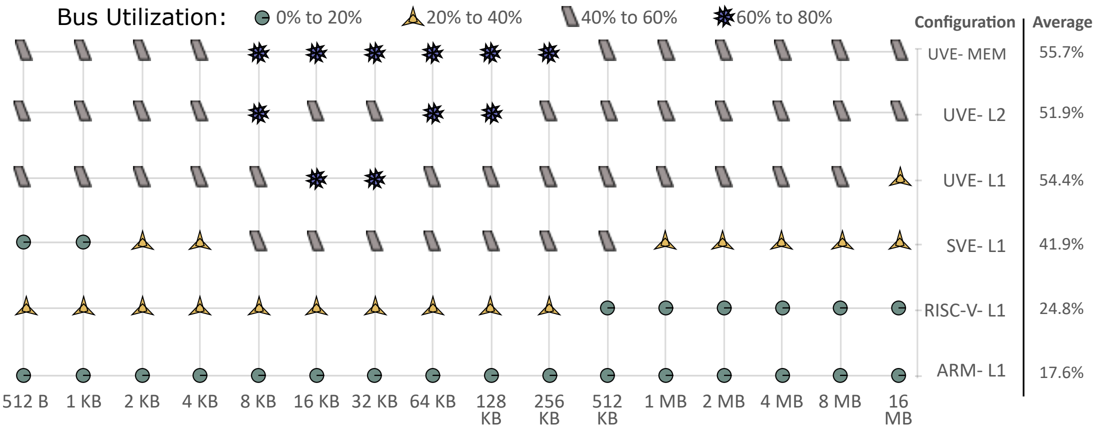


Figure 7.3: Memcpy benchmark, bus utilization with AMPM prefetcher enabled.

7.2 Non-unitary stride in memory transactions

As a consequence of such a data organization, the caching mechanism efficiency will have a reduced efficiency. Hence, while on unit stride all elements within a cache line are used, with non-unit strides only a few number of elements are used. To compensate the lower reusability of the cache line, the prefetcher will preload data to the cache lanes, improving the hit ratio on the caches. Figure 7.4 depicts the differences in memory access performance between the studied ISAs. In general, a stride increase will give worst performance, which is clearly depicted in Arm, RISC-V and SVE results, and coroborated by the fact that an higher stride will lead to a higher number of loaded lines per data element. Even in the presence of a prefetcher, the prefetching depth is limited and will saturate more quickly as more cache lines are needed.

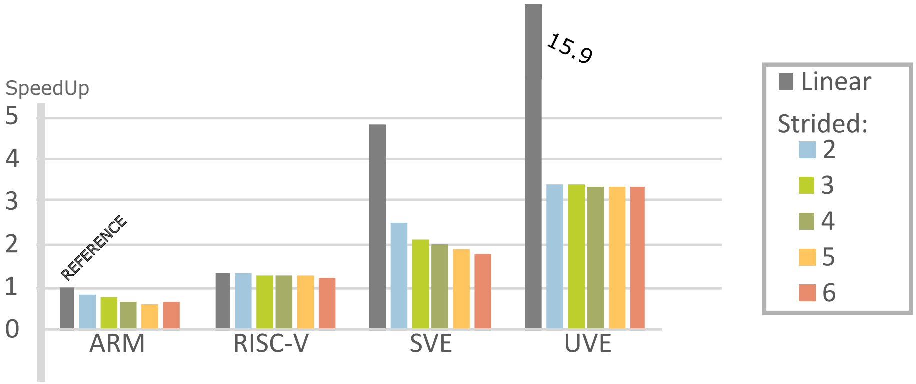


Figure 7.4: Memory transaction impact of non-unitary stride.

With UVE as the L1 cache is being bypassed, the latency between L1 and L2 cache missed accesses is completelly removed which leads to overall faster accesses that are less affected by an higher stride.

In the cases where the vectorial extensions process with a linear memory access, the processing performance is maximized as well as the memory access. In fact, the UVE memory access could be improved in smaller strides, as there should be a clear performance increase between stride 3 and stride 2. This limitation is imposed by the address generation/processing architecture, which processes only one element address per cycle when the stride between two elements is higher than one.

7.3 SAXPY benchmark

The SAXPY benchmark is more compute-bound than MEMCPY. Due to the vectorial nature of UVE it is expected that compute-bound applications show better performance in vectorial or SIMD architectures. In addition, the improved memory access mechanism in UVE will allow for a potential advantage in relation to SVE. Figure 7.5 depicts the results of SAXPY in terms of speedup to the Arm scalar ISA and total bus utilization in percentage. As it was expected the UVE performance is much higher than the counterparts. In particular, for the largest dataset UVE shows a performance increase of roughly 4 times to SVE and 8 times to Arm. In addition, UVE makes a better usage of the memory bus,and, by decreasing the number of loop instructions, is also able to attain a higher performance in a compute bound kernel. In facto, these results are highly correlated with those presented in subsection 5.3.3 and Figure 5.10, where UVE showed to reduce the number of executed instructions by a factor of 4.

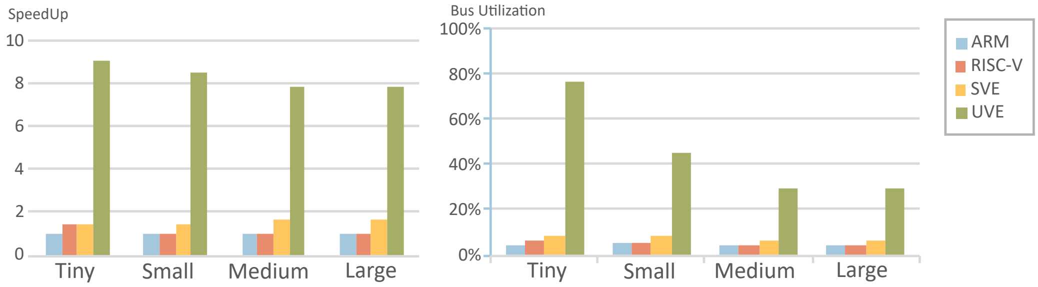


Figure 7.5: SAXPY benchmark results. Comparison of performance and bus utilization between ISAs.

For this particular benchmark it is also worth analyzing the gem5 statistics related with processor stage stalls. Figure 7.6 represents the blocked cycles during execution as well as their distribution. In Arm and SVE the majority of blocked cycles are in the decode and rename stages, showing that the performance is limited by a lack of available resources (registers, instruction queue entries) in the pipeline. UVE also presents the same blocked cycles distribution, however, the total ammount of blocked cycles is 7 times lower than SVE.

7.4 IRSmk benchmark

As introduced in subsection 5.3.4, the IRSmk microkernel makes high usage of the total number of streams. In detail, the microkernel execution corresponds to 54 arrays being loaded and multiplied in pairs, then accumulated into a final array. IRSmk is undoubtedly a memory-bound application.

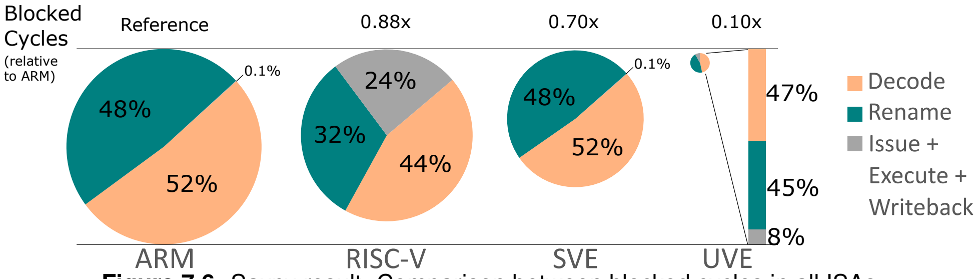


Figure 7.6: Saxpy result. Comparison between blocked cycles in all ISAs

Figure 7.7 shows the obtained performance for the reference architectures (Arm, RISC-V and SVE), and for three levels of vectorization with UVE (Inner, Middle, Outer), as detailed in subsection 5.3.4. The IRSmk kernel is composed of three cascaded loops (i,j, and k) with k being the innermost loop. The inner level only vectorizes k, while the middle and outer levels vectorize (k, j) and (k, j, i), respectively. Also, three sizes were executed (25, 50 and 100), where each size represents the number of elements in each dimension of the loop (i,j, and k).

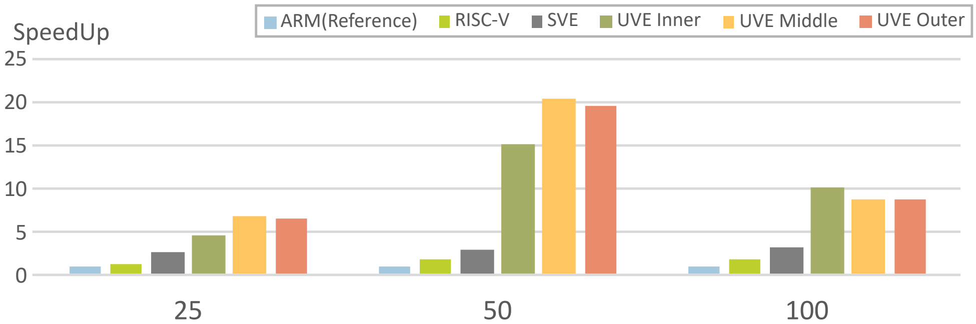


Figure 7.7: IRSmk performance results, comparison of UVE performance in relation to the reference architectures (Arm, RISC-V, SVE). Three execution sizes are shown 25, 50 and 100.

As expected, the performance with the UVE extension is significantly better than the reference architectures, mainly due to the application being memory-bound. In detail, at size 25 there is not much advantage in vectorizing (UVE 3 to 7 times faster than Arm), as there is not much data to fully explore the SIMD potential. On larger datasets (50 and 100), there is much room for vectorization, as shown in the results for the size 50 (15 to 20 times faster than Arm). On the other hand, the UVE relative performance at size 100 appears to be significantly lower. This is a consequence of the prefetching mechanisms attaining a better accuracy in larger datasets, thus improving the performance of the reference designs. Also, with a larger dataset more cycles are spent processing the same loop iteration, thus increasing the prefetcher predicability. In conclusion, UVE shows up to 10 times better performance than SVE for a size of 50, while showing up to 4 times better performance than SVE for a larger size of 100.

In contrast, as it would be expected, deeper vectorization (outer level) did not result in better performance. When processing large data sets with simple and linear multi-dimensional memory accesses, the outermost loops are iterated with a reduced frequency and hence do not significantly affect the performance. As an example, for a size 100 the outermost loop is iterated only after 10000 iterations of the innermost loop. Also, considering that an innermost loop iteration takes some time, due to memory accesses and processing, there is no perceivable performance advantage in vectorizing the outermost loop.

Figure 7.8 and Figure 7.9 show that UVE considerably improves memory access performance (around 7 times more bus utilization) and considerably reduces the ammount of cycles the pipeline was blocked. The reduction in total blocked cycles is a clear result of the increased memory bus utilization. The results in Figure 7.9 are only for the size 50, where the best performance results were obtained.

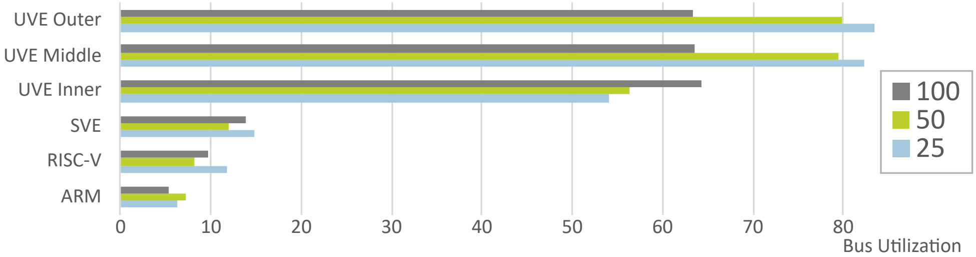


Figure 7.8: IRSmk bus utilization for UVE and the reference architectures (Arm, RISC-V, SVE). Three execution sizes are shown 25, 50 and 100.

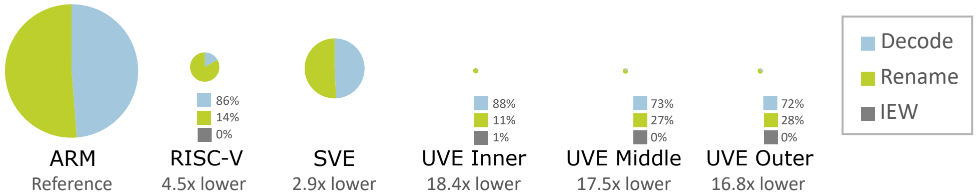


Figure 7.9: IRSmk blocked cycles for UVE, RISC-V, SVE and Arm, with the Arm architecutre as reference. On the execution size of 50 is depicted.

7.5 HACCmk benchmark

The HACCmk benchmark [62] is a microkernel based on the Hardware Accelerated Cosmology Code (HACC) framework. While the HACC framework mainly uses N-body computation to simulate the evolution of the Universe, the HACCmk is the kernel routine that calculates the short force evaluation based on an $\mathcal{O}(n^2)$ algorithm. This benchmark is part of the CORAL Benchmark Codes suite [63].

The short force evaluation kernel is characterized by an high compute intensity with unit stride memory accesses, generally resulting in a compute-bound application. Figure 7.10 depicts the results for the HACCmk microkernel. The execution parameters, in particular the dataset size used in the results were reduced in order to constraint the total simulation into the feasible range. Even so, it was validated that the selected parameter would not meaningfully affect the comparison results. The HACCmk results show a performance increase with both SVE and UVE over the Arm and RISC-V, which is supported by the higher compute intensity. Having wider vector registers resulted in improved performace. However, UVE shows a speedup of 2 times when compared to SVE, this is supported by the 2 times increase in bus utilization. The low bus utilizations and high ammounts of blocked cycles mirror the difficulty of

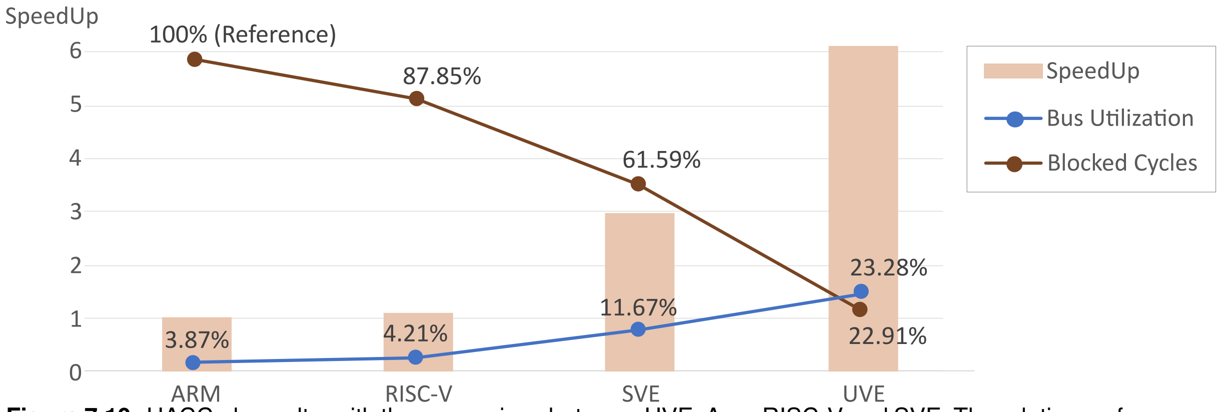


Figure 7.10: HACCmk results, with the comparison between UVE, Arm, RISC-V and SVE. The relative performance is shown alongside with the relative pipeline blocked cycles and the percentual memory bus utilization.

the pipeline in executing data processing instructions and, at the same time, handle memory address calculation, and memory transactions.

7.6 Summary

Throughout the performance analysis of the microarchitecture, it is clear that UVE improves the performance in both compute and memory intensive applications. Particularly, in memory intensive applications, such as IRSmk UVE shows a substantial speedup of at least 3 times in comparison with SVE. In compute intensive applications such as SAXPY a speedup of around 4 times between UVE and SVE was registed. The speedup between UVE and the scalar instruction set architecuters was considerably higher. Specifically, UVE has higher performance due to the reduction in executed instructions combined with improved memory access. Particularly, using streaming provides reduced latency and higher throughput.

8 Conclusions and Future Work

8.1 Conclusions

The Unlimited Vector Extension (UVE) explores two totally distinct and not yet combined state-of-the-art industry and scientific computer architecture areas. On the computational side, the SIMD extensions are becoming increasingly predominant in General Purpose Processors (GPPs), and new ISA extensions are emerging to improve the performance, scalability and the implementations flexibility. In particular, with the emerging presence of scalable vector SIMD extensions, each processor implementation can be tuned to achieve any desired SIMD performance level. Moreover, there is an increase of data analysis, machine learning and artificial intelegence applications leaning on SIMD extensions to significantly improve performance without resorting to external accelerators, which would imply additional costs. On the memory side, recent works show that data streaming applied to a processor memory structure improves memory access performance, both in latency and bandwidth. Moreover, streaming is an excelent opportunity to decouple the memory access procedures from the computational operations, moving the memory access logic to external co-processors.

UVE is a streaming scalable vector extension that merges both concepts into an instruction set architecture containing a comprehensive set of 41 vector compute instructions (integer and floating-point), execution control and vector manipulation instructions, and a set of specialized stream configuration and manipulation instructions - a total of 82 instructions, resulting in 450 available instructions, considering all variants. Moreover, a set of 32 vector/streaming registers are provided with support for individual vector lane execution control through a set of 16 predicate registers. Furthermore, a version of the GNU Compiler Collection (GCC) was extended to support UVE instructions.

By decoupling the memory access and loop iteration from the core pipeline, UVE can completely remove the control and memory indexation instructions from simpler codes (e.g. SAXPY), while significantly reducing the overhead instructions from more complex codes (e.g. IRSmk, HACCmk). UVE uses, in average, half of the instructions than SVE for the same application kernel.

To evaluate the impact of using UVE in a realistic superscalar out-of-order architecture, a reference CPU model based on ARM Cortex-A76 and implemented in a cycle-accurate architecture simulator (gem5) was modified and extended. The modifications comprehend the implementation of the UVE ISA, a stream processing and management module, and the integration of the stream module with the out-of-order pipeline.

The conducted evaluation of UVE and the supporting microarchitecture, indicates that UVE is 3 times faster than SVE in memory-bound kernels (IRSmk) and 2 times faster in compute-bound kernels (HACCmk).

8.2 Future Work

With this work being a proof-of-concept it is clear that some idealized objectives were defered due to the time constraints. The proposed future work is divided into two categories:

ISA: To make UVE a distributable and useful instruction set extension, it is necessary to improve the specification through application porting analysis and testing, particularly it is essential that characteristics such as the number of registers, the ammount of configurable descriptors and the available descritpors are further tuned. To allow for easier application porting, the compiler support must also be extended. In detail, intrinsics must be created to enable a higher-level programming style for UVE. Furthermore, the compiler should be extended to support auto-vectorization. This, would require the implementation of pattern analysis and extraction in the compiler front-end. Additionally, the resulting patterns must be converted to pattern representation that is compatible with UVE. Clearly, this work involves deep modifications to the compiler and may also result into changes to the extension. In addition to the compiler modifications, it is possible to ease the porting of applications to UVE by integrating the extension in already available SIMD libraries. Particularly, libraries such as libvolk [64], libsimdpp [65], yeppp! [66], E.V.E. [67], and the appropriatelly named Generic SIMD Library [68], would significantly improve UVE portability.

Microarchitecture: The microarchitecture implementation is fundamental in testing and evaluation of the UVE extension. In particular, there some improvements to the current microarchitecture that were left for future work. Namely, the processing module of the streaming engine has a simple but slow processing algorithm, improving this would lead to faster and more efficient address processing. Also, in the current implementation there is no internal reutilization of the memory requests, a possible approach to improve request utilization would be to create a structure similar to a miss-status holding register (MSHR) for the memory requests. Furthermore, there is currently no implemented coherence protocol for loading and storing to the same memory address, also, there is no coherence protocol for coincident loads and stores between the pipeline load-store unit and the streaming engine. Finally, the FIFO data queues structure must be optimized to support dynamic scaling of individual stream queues and allow for dynamic prioritization of streams. Considering all the proposed changes, the final microarchitecture could be used as a reference design for future implementations. Moreover, there is also space for exploring the performance benefits of UVE in lower performance and power architectures, such as in-order cores. To further validate UVE and the streaming mechanisms the reference microarchitecture or a relevant variant could make is way into RTL, further validating the performance, area and energy of this solution. Moreover, as a future research path, changes to the streaming engine architecture should be explored. Particularly, it is possible to create a distributed architecture where the streaming engine is scattered throughout the memory hierarchy, paving the way to achieving lower latency memory accesses. Finally, as it became clear this work only scratched the surface of employing streaming in vector architectures, particularly, the evaluation was only made in a single-core processor model, as implementing the streaming mechanisms in multi-core heterogeneous systems is another work for the future.

Bibliography

[1] ARM, “Introducing NEON™ Development Article,” 2009. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.dht0002a/DHT0002A_introducing_neon.pdf

[2] C. Lomont, “Introduction to Intel Advanced Vector Extensions,” *Intel White Paper*, p. 21, 2011. [Online]. Available: http://www.obpm.org/download/Intro_to_Intel_AVX.pdf

[3] N. Neves, “Energy-Efficient Computing: Adaptive Structures and Data Management,” Ph.D. dissertation, Instituto Superior Técnico, Universidade de Lisboa, 1 2019.

[4] Z. Wang and T. Nowatzki, “Stream-based memory access specialization for general purpose processors,” in *Proceedings - International Symposium on Computer Architecture*. Institute of Electrical and Electronics Engineers Inc., 6 2019, pp. 736–749.

[5] A. Roelke and M. R. Stan, “RISC5: Implementing the RISC-V ISA in gem5,” *First Workshop on Computer Architecture Research with RISC-V (CARRV)*, vol. 7, no. 17, 2017. [Online]. Available: https://carrv.github.io/2017/slides/roelke-risc5-carrv2017-slides.pdf

[6] A. Waterman and K. Asanovic, “RISC-V "V" Vector Extension,” Tech. Rep., 2019. [Online]. Available: https://riscv.github.io/documents/riscv-v-spec/riscv-v-spec.pdf

[7] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, and P. Walker, “The ARM Scalable Vector Extension,” *IEEE Micro*, vol. 37, no. 2, pp. 26–39, 3 2017. [Online]. Available: http://ieeexplore.ieee.org/document/7924233/

[8] G. Conte, S. Tommesani, and F. Zanichelli, “The long and winding road to high-performance image processing with MMX/SSE,” in *Proceedings Fifth IEEE International Workshop on Computer Architectures for Machine Perception*, 2000, pp. 302–310. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=875989

[9] N. Firasta, M. Buxton, P. Jinbo, K. Nasri, and S. Kuo, “Intel AVX: New Frontiers in Performance Improvements and Energy Efficiency,” *White Paper*, vol. 19, pp. 1–9, 2008. [Online]. Available: https://software.intel.com/sites/default/files/7b/f7/16820

[10] D. D. Habich, P. P. Damme, A. A. Ungethüm, and W. W. Lehner, “Make larger vector register sizes new challenges?: Lessons learned from the area of vectorized lightweight compression algorithms,” in *Proceedings of the Workshop on Testing Database Systems, DBTest 2018*. New York, New York, USA: Association for Computing Machinery, Inc, 6 2018, pp. 1–6. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3209950.3209957

[11] Y. Guo, P. Narayanan, M. A. Bennaser, S. Chheda, and C. A. Moritz, “Energy-efficient hardware data prefetching,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 2, pp. 250–263, 2 2011.

[12] Y. Ishii, M. Inaba, and K. Hiraki, “Access map pattern matching for data cache prefetch,” in *Proceedings of the International Conference on Supercomputing*. New York, New York, USA: ACM Press, 2009, pp. 495–496. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1542275.1542349

[13] P. Michaud, “Best-offset hardware prefetching,” in *Proceedings - International Symposium on High-Performance Computer Architecture*, vol. 2016-April. IEEE Computer Society, 4 2016, pp. 469–480.

[14] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens, “Programmable stream processors,” *Computer*, vol. 36, no. 8, pp. 54–62, 8 2003.

[15] B. K. Khailany, T. Williams, J. Lin, E. P. Long, M. Rygh, D. F. W. Tovey, and W. J. Dally, “A Programmable 512 GOPS Stream Processor for Signal, Image, and Video Processing,” *IEEE Journal of Solid-State Circuits*, vol. 43, no. 1, pp. 202–213, 2008.

[16] A. López-Lagunas and S. M. Chai, “Memory Bandwidth Optimization through Stream Descriptors,” 2006. [Online]. Available: https://www.researchgate.net/publication/220244701_Memory_bandwidth_optimization_through_stream_descriptors

[17] P. T. Hulina and L. D. Coraor, “Memory Latency Effects in Decoupled Architectures,” *IEEE Transactions on Computers*, vol. 43, no. 10, pp. 1129–1139, 1994.

[18] N. Neves, P. Tomás, and N. Roma, “Efficient Data-Stream Management for Shared-Memory Many-Core Systems,” in *International Conference on Field-programmable Logic and Applications (FPL 2015)*, 9 2015.

[19] N. C. Crago and S. J. Patel, “OUTRIDER: Efficient memory latency tolerance with decoupled strands,” in *Proceedings - International Symposium on Computer Architecture*, 2011, pp. 117–128.

[20] Arm, “Technologies | NEON – Arm Developer.” [Online]. Available: https://developer.arm.com/technologies/neon

[21] R. Ramanathan, R. Curry, S. Chennupaty, R. L. Cross, S. Kuo, and M. J. Buxton, “Extending the World’s Most Popular Processor Architecture,” *White Paper*, 2006. [Online]. Available: http://www.ele.uva.es/~jesman/BigSeti/ftp/Microprocesadores/Intel/new-instructions-paper.pdf

[22] A. Sodani, “Knights Landing (KNL): 2nd Generation Intel® Xeon Phi™ Processor,” Tech. Rep. [Online]. Available: http://www.Intel.com/performance

[23] I. Corporation, “Intel® Streaming SIMD Extensions Technology Defined,” 2016. [Online]. Available: https://www.intel.com/content/www/us/en/support/articles/000005779/processors.html

[24] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed. Morgan Kaufmann, 2017.

[25] Y. Dai, Q. Li, Q. Zhang, and C. C. Jay Kuo, “SIMD-efficient loop unrolling design for embedded multimedia applications,” in *2004 IEEE International Conference on Multimedia and Expo (ICME)*, vol. 3, 2004, pp. 1851–1854.

[26] R. F. Barrett, S. D. Hammond, C. T. Vaughan, D. W. Doerfler, M. A. Heroux, J. P. Luitjens, and D. Roweth, “Navigating an evolutionary fast path to exascale,” in *Proceedings - 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, SCC 2012*, 2012, pp. 355–365.

[27] A. Waterman and K. Asanovic, “The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA Document Version 20190608-Base-Ratified,” 2019.

[28] S. S. Baghsorkhi, N. Vasudevan, and Y. Wu, “FlexVec: auto-vectorization for irregular loops,” in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2016*, vol. 51, no. 6. New York, New York, USA: ACM Press, 2016, pp. 697–710. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2908080.2908111

[29] S. Flur, K. E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon, and P. Sewell, “Modelling the ARMv8 architecture, operationally: Concurrency and ISA,” *ACM SIGPLAN Notices*, vol. 51, no. 1, pp. 608–621, 4 2016. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2837614.2837615

[30] F. Petrogalli, “A sneak peek into SVE and VLA programming,” *White Paper*, 2018.

[31] T. J. Ham, J. L. Aragón, and M. Martonosi, “DeSC: Decoupled Supply-Compute Communication Management for Heterogeneous Architectures,” 2015. [Online]. Available: http://dx.doi.org/10.1145/2830772.2830800

[32] D. L. Kuck, *Structure of Computers and Computations*. USA: John Wiley & Sons, Inc., 1978.

[33] P. M. W. Knijnenburg and A. J. C. Bik, “On Reducing Overhead in Loops,” Tech. Rep., 2000. [Online]. Available: https://www.researchgate.net/publication/2597549_On_Reducing_Overhead_in_Loops

[34] B. Jang, D. Schaa, P. Mistry, and D. Kaeli, “Exploiting memory access patterns to improve memory performance in data-parallel architectures,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 1, pp. 105–118, 2011.

[35] A. López-Lagunas and S. M. Chai, “Compiler manipulation of stream descriptors for data access optimization,” in *Proceedings of the International Conference on Parallel Processing Workshops*, 2006, pp. 337–344.

[36] N. Neves, P. Tomás, and N. Roma, “Adaptive in-cache streaming for efficient data management,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 7, pp. 2130–2143, 2017.

[37] S. M. F. Paiágua, F. Pratas, P. Tomás, N. Roma, and R. Chaves, “HotStream: Efficient Data Streaming of Complex Patterns to Multiple Accelerating Kernels,” in *25th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD’2013)*, 10 2013, pp. 17–24.

[38] B. Khailany, W. J. Dally, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, A. Chang, and S. Rixner, “IMAGINE: MEDIA PROCESSING WITH STREAMS,” *IEEE Micro*, vol. 21, no. 2, pp. 35–46, 2001.

[39] S. Ciricescu, R. Essick, B. Lucas, P. May, K. Moat, J. Norris, M. Schuette, and A. Saidi, “The reconfigurable streaming vector processor (RSVP™),” in *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, vol. 2003-January. IEEE Computer Society, 2003, pp. 141–150.

[40] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross, “Q100: The Architecture and Design of a Database Processing Unit.” [Online]. Available: http://dx.doi.org/10.1145/2541940.2541961

[41] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, “Stream-dataflow acceleration,” in *Proceedings - International Symposium on Computer Architecture*, vol. Part F128643. Institute of Electrical and Electronics Engineers Inc., 6 2017, pp. 416–429.

[42] N. Clark, A. Hormati, and S. Mahlke, "VEAL: Virtualized Execution Accelerator for Loops," in *Proceedings - International Symposium on Computer Architecture*, 2008, pp. 389–400.

[43] G. Weisz and J. C. Hoe, "CoRAM++: Supporting data-structure-specific memory interfaces for FPGA computing," in *25th International Conference on Field Programmable Logic and Applications, FPL 2015*. Institute of Electrical and Electronics Engineers Inc., 10 2015.

[44] C. H. Ho, S. J. Kim, and K. Sankaralingam, "Accelerating the Accelerator Memory Interface with Access-Execute and Dataflow," *IEEE Micro*, vol. 36, no. 3, pp. 31–41, 5 2016.

[45] S. Thomas, C. Gohkale, E. Tanuwidjaja, T. Chong, D. Lau, S. Garcia, and M. Bedford Taylor, "Cortex Suite: A synthetic brain benchmark suite," in *IISWC 2014 - IEEE International Symposium on Workload Characterization*. Institute of Electrical and Electronics Engineers Inc., 12 2014, pp. 76–79.

[46] J. Bucek, K. D. Lange, and J. V. Kistowski, "SPEC CPU2017 – next-generation compute benchmark," in *ICPE 2018 - Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, vol. 2018-January. Association for Computing Machinery, Inc, 4 2018, pp. 41–42.

[47] J. E. Smith, "Decoupled access/execute computer architectures," in *Proceedings - International Symposium on Computer Architecture*, vol. 10, no. 3. Institute of Electrical and Electronics Engineers Inc., 4 1982, pp. 112–119. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1067649.801719

[48] Y. Lee, A. Waterman, R. Avizienis, H. Cook, C. Sun, V. Stojanovic, and K. Asanovic, "A 45nm 1.3GHz 16.7 double-precision GFLOPS/W RISC-V processor with vector accelerators," in *European Solid-State Circuits Conference*, 2014, pp. 199–202.

[49] A. Nohl, G. Braun, O. Schliebusch, R. Leupers, H. Meyr, and A. Hoffmann, "A universal technique for fast and flexible instruction-set architecture simulation," in *Proceedings of the 39th conference on Design automation - DAC '02*. New York, New York, USA: Association for Computing Machinery (ACM), 2002, p. 22. [Online]. Available: http://portal.acm.org/citation.cfm?doid=513918.513927

[50] M. Reshadi, P. Mishra, and N. Dutt, "Instruction set compiled simulation." Association for Computing Machinery (ACM), 2003, p. 758.

[51] R. Stallman, *Using and porting GNU CC*. Free Software Foundation Cambridge, Massachusetts, 1992, vol. 67, no. 5.

[52] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, CGO*, 2004, pp. 75–86.

[53] "Extended Asm (Using the GNU Compiler Collection (GCC))." [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html

[54] N. Binkert, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, D. A. Wood, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, and T. Krishna, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, p. 1, 8 2011. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2024716.2024718

[55] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, A. Saidi, and S. Reinhardt, "The M5 Simulator: Modeling Networked Systems," *IEEE Micro*, vol. 26, no. 4, pp. 52–60, 7 2006. [Online]. Available: http://ieeexplore.ieee.org/document/1677503/

[56] J. E. F. Friedl, *Mastering Regular Expressions*. O'Reilly Media, 2006. [Online]. Available: https://books.google.pt/books?hl=en&lr=&id=P5UXAwAAQBAJ&oi=fnd&pg=PR7&dq=regular+expressions&ots=HBnT84keSr&sig=U72sj3wvFsGwNbME0lLkTiaEddo&redir_esc=y#v=onepage&q=regular%20expressions&f=false

[57] J. R. Levine, J. Mason, and J. R. Levine, *Lex &amp; Yacc*, 1st ed., 1992. [Online]. Available: https://books.google.pt/books?hl=en&lr=&id=fMPxfWfe67EC&oi=fnd&pg=PP2&dq=lex-yacc&ots=RdRVjj4OBS&sig=m9fzB01RswCxkHwdv32LryYIlig&redir_esc=y#v=onepage&q=lex-yacc&f=false

[58] L. L. N. Laboratories, "ASC Sequoia Benchmark Codes," 2009.

[59] D. Sima, "Design space of register renaming techniques," *IEEE Micro*, vol. 20, no. 5, pp. 70–83, 2000.

[60] "Cortex-A76 - Microarchitectures - ARM - WikiChip." [Online]. Available: https://en.wikichip.org/wiki/arm_holdings/microarchitectures/cortex-a76

[61] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Bingo spatial data prefetcher," in *Proceedings - 25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019*. Institute of Electrical and Electronics Engineers Inc., 3 2019, pp. 399–411.

[62] Z. Jin, V. Morozov, and H. Finkel, "A Case Study on the HACCmk Routine in SYCL on Integrated Graphics." Institute of Electrical and Electronics Engineers (IEEE), 7 2020, pp. 368–374.

[63] "CORAL Benchmark Codes | Advanced Simulationand Computing." [Online]. Available: https://asc.llnl.gov/coral-benchmarks

[64] "Vector Optimized Library of Kernels." [Online]. Available: https://www.libvolk.org/

[65] “p12tic/libsimdpp: Portable header-only C++ low level SIMD library.” [Online]. Available: https://github.com/p12tic/libsimdpp

[66] “Yeppp! library .” [Online]. Available: https://sites.google.com/site/yeppplibrary/

[67] “E.V.E., An Object Oriented SIMD Library | Scalable Computing: Practice and Experience.” [Online]. Available: https://www.scpe.org/index.php/scpe/article/view/345

[68] H. Wang, P. Wu, I. G. Tanase, M. J. Serrano, and J. E. Moreira, “Simple, portable and fast SIMD intrinsic programming: Generic SIMD Library,” in *WPMVP 2014 - Proceedings of the 2014 ACM SIGPLAN Workshop on Programming Models for SIMD/Vector Processing, Co-located with PPoPP 2014*. New York, New York, USA: Association for Computing Machinery, 2014, pp. 9–16. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2568058.2568059