Unlimited Vector Extension with data streaming support

João Mário Ribeiro Domingos

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.

1. INTRODUCTION

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 multi-core 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), thus allowing for larger processing throughput. 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 image processing and audio processing are ideal for being accelerated with Single Instruction Multiple Data (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.

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 [3, 4, 2].

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 Scalable Vector Extension (SVE) that limits the vector length to 2048-bits, but does not require the full length to be implemented in any processor [5]. Consequently, the same software is compatible with multiple implemented lengths. 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 [6, 7, 8]. 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 [9, 10, 11]. 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 [12, 11, 13, 14].

In this work, 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 [15] 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

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.

streaming poses as a new path forward in modern computer architectures.

1.1 Objectives

Considering the future computational needs of HPC applications, scalable vector 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 vector 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 vector 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.

2. ANALYSIS OF SIMD ISA EXTENSIONS

Currently, the three major Instruction Set Architectures, namely x86 [1], Arm [2] and RISC-V, are all equiped with instruction set extensions targeting data-level parallelism. Particullarly, the widelly available state-of-the-art SIMD extensions have a static VL (such as Arm NEON and x86 AVX), which forces all implementations to the Instruction Set Architecture (ISA) defined VL.

2.1 Static Vector-Length SIMD Extensions

While NEON defines the VL to 128-bits, AVX uses 256-bits, or 512-bits for the AVX-512 version. Clearly, there is no industry standard or golden vector-length, as it is highly dependant on the target applications. Actually, since SIMD instruction extensions made their way to computer architectures the vector-length as been adapting, through continuous increase, Figure 1 shows the trend in vector-length for the x86 family of SIMD extensions. The constant change in vector size, creates

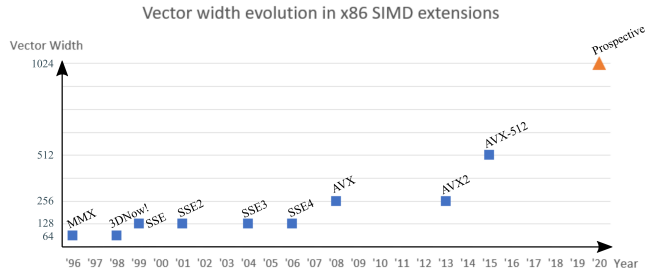

Figure 1: Evolution of vector width in x86 SIMD extensions. The rightmost orange mark is a prospective that represents the trend in the usage of wider vector registers [3, 4, 16].

a enormous difficulty in developing for these instruction sets. Particullarly, the use of these extensions ultimately leads to incompatible code in the future, as well as forcing the newer implementations to support pratically obsolete extensions to allow for application compatibility. All of these create an additional, and unnecessary, effort for hardware and software designers, integrators and their clients. In fact, it also cripples the implementations that would benefit from narrower (energy-efficient, low power processors) or wider vectors (HPC, datacenters) [17]. In sum, static vector-length is definitly not the way forward for SIMD extensions.

2.2 Scalable Vector-Length SIMD Extensions

Urging to overcome the difficulties imposed by using a static vector-length, both ARM and RISC-V created two new SIMD extensions. These extensions' focus is bringing flexibility to the implementations while retaining a programming model that is not excessively complex and that succeeds in hiding the underlying complexity and inherent problems of scalability. In particular, for a ISA extension to not define a particular vector-length, the instructions must not be dependant on positional references (e.g. element with index 3), or in a reference maximum number of elements. To overcome these difficulties the extension programming model must be completely vector-length agnostic. However, due to the unknown vector-length, this programing model means that the compiler does not have sufficient information to calculate the number of interations a given loop will take. Consequently, it is necessary to use an iteration method that makes use of run-time variables, the implementation vector-length, instead of compile-time heuristics.

Arm SVE is created with a vector length agnostic programming style coupled with an architecture that supports registers ranging from 128-bits up until 2048-bits in width, in increments of 128-bits [5]. The vector-length agnostic iteration method introcuded by SVE is centered on two sets of instructions named "while" and "inc". The "while" set is based on the well known while loop, where a given condition is tested in the beggining of each loop iteration. As an explanatory example, a "whilelt" instruction uses the less than comparison to condition the execution. Particullarly, the main idea is to request a size through the instruction operands and compare it with the vector length. However, this is implemented by giving a start value (X) in one operand and a comparison value (Y) in the other. The internal process is: X will be internally incremented with i and after each increment it is compared with Y, while $X+i$ is less than Y, the position (i) of a predicate register is set to 1; the iteration ends when i is equal to the number of elements of a given data width that fit the vector-length. These instructions have the following format:

$$whilelt \quad Predicate, Start, Comparison \tag{1}$$

In sum, this instruction creates a predicate register based on the comparison between the requested elements that can be processed and the elements that fit into the vector-length, the resulting predicate will constraint the following loop iteration instructions and control. In detail, the "while" instructions update the AArch64 NZCV condition code flags, this is essencial for branching, as these flags are tested by the branch instructions. In order to keep the start value updated from iteration to iteration, the "inc" set of instructions increments the destination operand with the number of elements of a given data width that fit the vector. Hence, it complements the "while" instructions, as they only write to the predicate register.

The RISC-V "V" Vector Extension (RVV) extension vector-length agnostic programming style is coupled with an highly scalable vector architecture. In detail, RVV does not impose any vector-length to be used, instead, the minimum length ($ELEN$) is determined by the maximum base ISA data width (e.g. byte, half, word, double), while the total length ($VLEN$) must be multiple of $ELEN$ [18, 15].

The agnostic vector-length programming of RVV is based on the "vsetvli" instruction, a vector configuration instruction.

[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.

In similarity with SVE, the idea behind the instruction is to request a size and compare it with the vector-length, however, "vsetvli" uses a quite different implementation. Specifically, the first source operand in "vsetvli" contains the requested size in elements, with the width of each element encoded in the instruction. The total size (elements times element width) is compared with the implemented vector-length, and the smaller is returned in the destination operand (configured size). This instruction has side-effects in the vector registers, as all vectors active length is constrained to the new configured size. The "vsetvli" instruction has the following format:

$$vsetvli\ \ Final\ Size,\ Size,\ \#Width,\ [\#Grouping] \qquad (2)$$

Additionally, through the "Grouping" paramenter, "vsetvli" is capable of configuring the vector registers in groups. Grouping registers results in a virtual increase of the register vector-length at the cost of reducing the available vector registers, hence, the active vector-length can be higher than $VLEN$, and up to $32 \times VLEN$. To clarify, with a grouping factor of 4 the vector registers 0 to 3 would be merged into a logical register 0 with $4 \times VLEN$, analogously the register 4 through 7 would form the logical register 4, and so on. By grouping registers the total number of loop iterations is reduced by the grouping factor, however, each instruction that operates on a group must be divided into multiple micro operations at execution time, a clear CISC behaviour that does not suit well the RISC-V architecture principles.

In the modern ISA SIMD extensions we have static and scalable vector-length extensions. Which the latter improve on the former by allowing the code to be reused in different implementations. And further, it allows for implementations to be tuned accordingly to the application specific needs. However, this work aims to add on top of that, and so, we will compare both the scalable extensions and from there propose a possible idealistic improvement. Figure 2 depicts both SVE and RISC-V approaches to the vectorization of a MEMCPY[3] kernel.

By directly comparing the RVV (Figure 2a) and the SVE (Figure 2b) assemblies, it is noticeable the difference in code structure due to the constrasting scalability technique approach ("vsetvli" versus "whilelt"). In detail, on the RVV side, a register (t0) is used to keep the size of the executing register and is used to calculate both memory addresses and iteration counters. This results in a reconfiguration of the vector registers (setting the vector-length) each iteration of the loop.

In comparison, the SVE assembly approach is to use the "whilelt" instruction to manage the predicate register (p0), who controls the active and valid elements of the vectors, and sets the branch condition flags. Also, it is necessary to use an additional instruction ("incb") that increments the loop iterator according to the data width (bite), and implemented vector-length.

In both RVV and SVE cases it is necessary to reconfigure the execution state (register width/elements). However, there is not much difference in terms of address calculation or counter management instructions. In particular, SVE has better memory access instructions that allow for address calculation with multiple registers in one instruction, however, this advantage may not transpose well to the microarchitecture, as it can be implemented as two micro operations.

The loop iteration and address calculation instructions do not directly contribute to the data processing, hence, they are overhead. Additionally, the configuration instructions that inside the processing loop, are also contributing to overhead. Consequently, in an ideal scenariou these would not exist, as we can remove them without affecting the core part of the work, assuming that the correct data is loaded and stored to the necessary registers.

Therefore, a set of opportunities are presented to us:

1. Remove unnecessary address calculation logic;
2. Remove unnecessary counter logic;
3. Remove looped configuration instructions.

Taking these into account, there is clearly space for some improvements in the state-of-the-art SIMD extensions. To demonstrate, an idealistic solution is conducted in Figure 2c. Starting from the loop routine, we can observe that the only operation that is executed is a vector move, hence we fulfilled our initial objectives of removing all the overhead instructions. In fact, the branch instruction can also be seen as overhead, however, removing it would seriously impact the architecture and thus is an opportunity left for the future.

3. DATA STREAMS FUNDAMENTALS

To achieve the objectives proposed in section 2, the data must be fed to the processing core and saved to the memory without additional instructions. Hence, a mechanism must be used to manage the data transactions to and from the processing unit. The core concept on which this relies is data streaming, where a transaction is configured and the data is "automatically" moved around. However, both the configuration and the consumption (data processing) of a stream, need a specific instruction set. This section paves the way into understanding how a stream is defined, and how a memory access can be configured to match a specific pattern.

In the context of this work, streams are defined as a set of data that is transported between the processor and the memory. Furthermore, streams are defined based on a memory access pattern description. In specific, this description defines the organization of the streameable data in memory. Finally, in order for a stream to be defined, we need to start by defining a convenient and simple way to specify the memory access patterns.

3.1 Related works

To allow for a memory access to be described with a set of simple representations, Nuno Neves et al. [19, 20] proposed that the majority of complex but still regular memory accesses can be described by linking together multiple 3-D descriptors. In detail, each descriptor contains the necessary information to fully represent a simple 3-D memory access: offset, hsize, stride, vsize, span, dsize. Where, the offset is the initial address, the stride and span are the spacement between elements for dimension 1 and 2, respectivelly. Additionally, hsize, vsize and dsize represent the size for each dimension, from the first to the third. Moreover, Neves et al. instroduced the usage of modification descriptors that will dynamically change the parameters of the 3-D descriptors. By combining everything, they show that it is possible to represent multiple complex patterns, some of which are depicted in Figure 3.

3.2 Simple Pattern Descriptors - Dimensions

Figure 4 contains multiple memory patterns and respective application code and stream representation, this examples will be analyzed throught this section.

To try and describe patterns, we will start by looking at a simple linear memory access pattern, like the one present in Figure 4a. This example specifies a memory access starting at memory position A and ending in memory position $A+N$ (size N), with an element spacing (stride) of 1. For simplicity, we will, for now, dismiss the presence of the data width offset.

[3]MEMCPY - Memory copy. Procedure that copies a source zone of memory to a destination one.

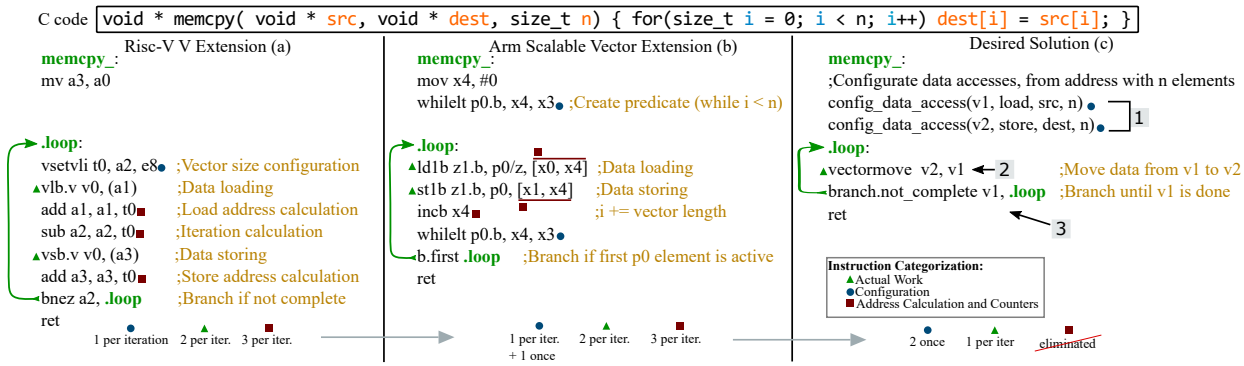

Figure 2: Side by side code complexity comparison. At the top, the C *memcpy* source code (linear memory copy), from whom the following are based. In the left, the RISC-V V extension assembly. In the center, the ARM SVE extension assembly. And in the right, a possible extension loosely based on RISC-V. Finally, at the bottom, a 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 [15]. ARM SVE example is based on the DAXPY code in [5].

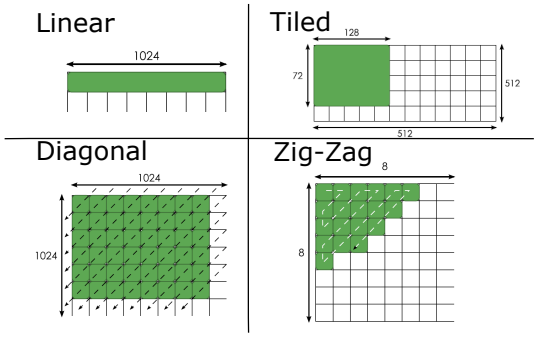

Figure 3: Set of describable patterns based on Neves et al. work [20].

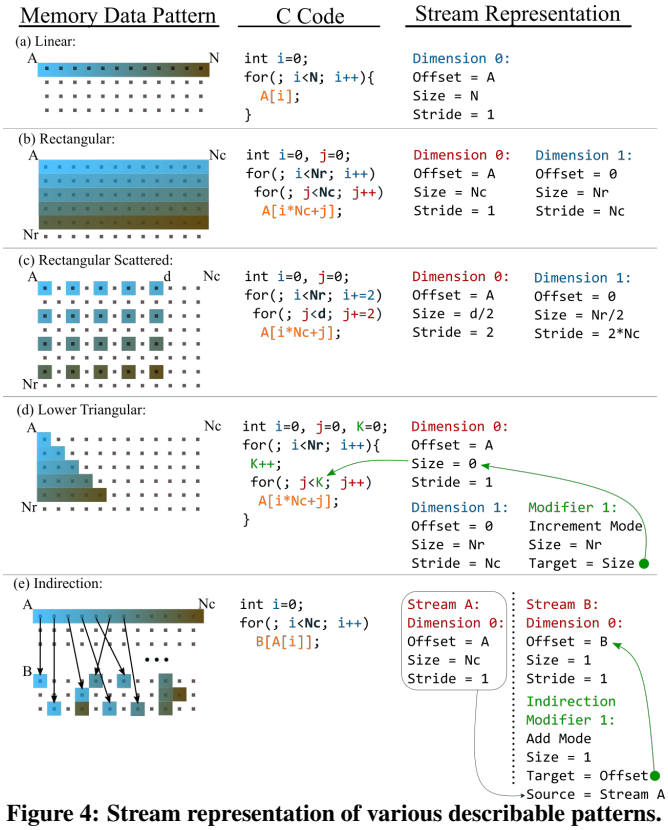

Figure 4: Stream representation of various describable patterns.

As a result, this memory access can be simply described with only three parameters, namelly: **Offset**, **Size** and **Stride**. A description of one memory access with these paramenters is named a descriptor. Also, any unidimensional descriptor can be represented by the following affine function:

$$i \in \{0 \dots E_i\}; \quad Y(i) = O_i + (S_i \times i); \tag{3}$$

Where, E_i, O_i and S_i represent, respectivelly, size, offset and stride for the dimension i.

By joining multiple descriptors it is possible to describe a multi-dimensional patterns, an example of a bidimensional access is given at Figure 4b. Particullarly it could be interesting to represent higher levels of dimensionality in an example, even so, in these examples only up to two dimensions are shown as it simplifies visualization.

In order to create that 2-D stream description, we start by defining the inner dimension from A to $A + Nc$ with stride 1, thus, creating the first descriptor. Additionally, the outer dimension will iterate Nr rows, with the spacing (stride) between the first elements of each row being the row size (Nc). Therefore, a new descriptor is created with size Nr and stride Nc. Also, as the memory access starts in A, the second dimension offset is 0. In sum, any regular 2-D memory pattern can be represented by two cascaded 1-D descriptors, as in the following function:

$$\begin{aligned} &i \in \{0 \dots E_i\}, j \in \{0 \dots E_j\}; \\ &Y(i,j) = O_j + (O_i \times S_i) + (S_j \times j) + (S_i \times i); \end{aligned} \tag{4}$$

Where, E_x, O_x and S_x represent, respectivelly, size, offset and stride for a given dimension x. In detail, the inner dimension offset O_j is generally the memory access base address. However, the base address of a stream description is obtained from all the dimensions' offsets and strides ($BA = O_j + (S_i \times O_i)$).

For the sake of completeness, Figure 4c depicts a 2-D memory access pattern that makes use of the stride parameter in order to skip memory elements.

3.3 Complex Pattern Descriptors - Modifiers

It is quite common in HPC workloads to have for loops that are not describable with only the introduced dimensions. Particullarly, it is quite common to have the conditions of an inner for loop being controlled by the iteration of an outer loop, as depicted in Figure 4d. The lower triangular example contains two for loops, therefore, two dimensions should suffice for the pattern representation. However, the outer loop (i) increments the parameter size of the inner loop (j), for each i iteration. To represent this behaviour in a pattern descriptor, the static modifiers need to be introduced with the parameters that fit the presented behaviour. In particular:

- 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.

In the example, the "displacement" parameter is not represented as it is unitary.

The modifier will modify the value of a lower-level dimension. Also, from what happens with the for loop, the modification is triggered when the outer loop is iterated. Based on these characteristics, it is clear that a modifier should be coupled with a dimension and their modifications should target the dimension immediately below. Hence, there is never a modifier coupled with the inner-most dimension.

Taking all of these into account, it is simple to represent the example's lower triangular pattern. To demonstrate such usage, the parameters are shown in Figure 4d.

Additionally, 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 N dimensions and N-1 modifiers, Figure 5 depicts a complete list of descriptors for $N = 4$. To achieve

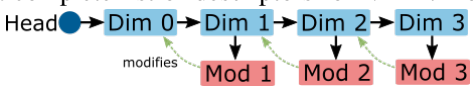

Figure 5: Example of a complete list of descriptors with 4 dimension descriptors and 3 modifier descriptors.

more intricate memory patterns, it is possible to cascade up to 8 descriptors.

The capabilities of the descriptions can be even further extended. In fact, it is also possible to use the contents of one stream to modify the values of another stream. Doing so, it is possible to create descriptions of indirection patterns and indexed scatter-gathers. Such descriptors are defined as dynamic or indirection modifiers, and extended the previous static modifiers by receiving the displacement value from another stream's data. Consequently, the displacement parameter is now a reference to the data stream. Also, more behaviours were changed to allow for the full potential of the dynamic modifiers:

- 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**.

By putting it all together into a pattern description, the result for a given pattern $B[A[i]]$ is depicted in Figure 4e.

4. PROPOSED EXTENSION

By understanding the novel capabilities that the streaming paradigm introduces, it is now required that the instruction set can make complete use of them. As it is expected, the standard instruction sets, such as x86, Arm and RISC-V do not include support for a streaming architecture. In fact, amongst the vast abundance of available ISA extensions, there isn't any vector extension that supports streaming. In fact, this work could be built on top of Arm SVE or RISC-V RVV, however, as seen in section 2, both these extensions were not developed with streaming in mind. For this reason, a new extension was developed from scratch.

As a means of defining such instruction set, the following requirements were established:

- 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.

Apart from this, the base instruction set chosen was RISC-V. In particular, this choice was taken based on three aspects that valued RISC-V in relation to Arm and x86. First, it is an open source instruction set, which means full access to documentation and source code, as well as no royalties. Second, it is simpler and cleaner in terms of instructions. Finally, it was made with academics in mind.

4.1 Vector Extension Design

4.1.1 Architectural State

Since the extension is vectorial, it is necessary to define a set of vector registers. Moreover, some instructions will certainly require scalar registers (e.g. Loads and Stores). The scalar registers are the same of the base RISC-V ISA [18]. In addition, to provide lane execution control, a set of predicate registers is present.

Vector Registers

Based on the state-of-the-art SIMD and vector architectures (section 2), the same ammount of 32 vector registers is present in both SVE and RVV extensions. Considering 32 architectural registers, an operand enconding uses 5 bits, taking almost half of the encoding space for a 3 operand instruction in a 32-bits encoding space. While it is possible to use 64 architectural registers (6 bits encoding), there is no research proof that the encoding limitations would be worthwhile for any possible performance increase. On the other hand, having only 16 registers could significantly limit the instruction set architecture capabilities. Considering that the RISC-V ISA reserves two opcodes in the 32-bit encoding space for custom intruction set extensions, this extension will make use of those spaces. As such, it is optimal to use 32 vector registers. While a 64-bits encoding space could be used, that would require all implementations to support 64-bits instructions. The 32 vector registers are named from "u0" to "u31".

The vector-length is not limited to any maximum size. However, a minimum value is defined, and corresponds to the maximum width of the elements supported by the architecture (i.e., byte, short, double, etc). On the other hand, the maximum vector length must be a multiple of the minimum length.

Each vector register is partitioned in multiple vector elements. The available element widths are byte (8-bits), halfword (16-bits), word (32-bits), double-word (64-bits). Hence, the minimum vector length is 64-bits. Also, the element width is configured independently for each vector register. In a case where a vector is not completelly filled with data, the vector register valid (active) bytes must be tracked, 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, the compute instructions specify the data-type [5, 15].

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 vector registers.

Predicate Registers

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 [5, 21]. 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.

4.1.2 Instruction Set

To guide through the developments of this section, the Memory copy. Procedure that copies a source zone of memory to a destination one. (MEMCPY) example will be used. In order to create support for the code proposed in Figure 2c (MEMCPY), it is necessary to divide the code in the following distinct portions:

1. Data access configuration;
2. Processing;
3. Execution control.

The first part (1), contains the configuration instructions for a stream. To start, the MEMCPY uses a linear uni-dimensional memory access, as shown in Figure 4a. Hence, MEMCPY can be described by a size of *n*, an offset of *src*, and an unitary stride. Additionally, we need the data width of the access, which is byte. Furthermore, the load and store data movements are selectable. With all this information the stream configuration is fully detailed, and therefore, all these details can be merged in the following instruction:

```
ss.[ld/st].[width]    Vector, Size, Offset, Stride
```

This instruction uses the opcode "streamsimple" (ss), which defines only one dimension pattern (decriptor). The parameter "ld/st" defines the direction of the memory access, while the width configures the accessed data width. The operand specified by "Vector" is the architectural register that will be coupled with the stream.

The materialization of the stream configuration in the MEMCPY example is depicted in Figure 7 (point 1).

After stream configuration instructions, the streams are fully configured and, from the ISA point of view it is to be assumed that the data is immediately ready for any consumption instructions. The stream configuration instructions are not limited to the "ss" instruction. As expected, there is a set of instructions that allow multiple dimensions to be defined, as well as static and dynamic modifiers. In detail, configuring multiple descriptors into one stream is obtained by executing multiple stream configuration instructions to the same vector. For this, a stream configuration begins with a "so.start" instruction and ends with a "so.end" instruction, any additional descripter is added through a "so.app" instruction.

To consume data from the streamed registers (code portion 2), a complete set of data processing and register manipulation instructions is needed. The complete set of instructions is not detailed in this paper, as it is extensive, however, a summary of the instructions is given in Table 1. The proposed 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).

All the data consumption instructions (e.g. arithmetic, logic, vector) do not need special modifications due to the streaming paradigm. The instruction that implements the data movement between vector registers is depicted in Figure 7 (point 2).

As it is inferable, there is nothing on this instruction that shows the existance of streaming. The "u2" and "u1" registers are vector registers, independently of the coupled stream, and "p0" is the always active predicate, that will allow all valid lanes to execute.

After the processing instruction executes, the affected streams must be iterated before the following instruction executes. In this extension, each consumption or writing of and to a vector register causes the streaming process to advance. To better illustrate, this process is depicted in Figure 6. To put it to words, each instruction that executes on a streamed register will iterate the stream automatically, there is no need for specific instructions that ultimately would be considered overhead. This is a destructive iteration process, and the associated tradeofs are discussed later.

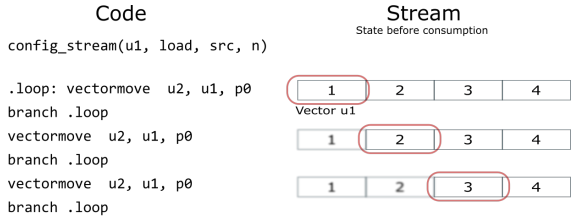

Figure 6: Stream consumption example for a vector move instruction.

Lastly, to address the execution control (3) it is necessary to define a type of conditional branch that is able to verify the stream state. As depicted in Figure 6, the stream will continuously feed data to the processor, but upon the stream termination there will be no more data. Hence, the branch instruction must be able to evaluate the end of the stream. In detail, a branch that would jump until a stream ends would be perfect for the runing example, this instruction is depicted in Figure 7 (point 3). The "nc" parameter specifies the condition "not complete", and it tests if the stream "u1" is complete, jumping to ".loop" while "u1" is not complete. The complementary test (complete) is also available. To give support for less generic cases, it is also possible to verify if any of the stream dimensions has or hasn't terminated.

The complete summary of the code changes, between the desired solution in Figure 2c and the proposed extension code, are denoted in Figure 7.

In overall, this ISA extension introduces 32 vector registers and 16 predicate registers. In detail, the 32 vector registers can be used to define a stream. Additionally, from the 16 available predicate registers, only the lower 8 are usable in execution control. Even so, the higher 8 predicate registers can be used to save other context predicates, allowing for easy

Table 1: Instruction set instructions overview, 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 Vector, Stream Start, Stream App, Stream End, Simple Stream

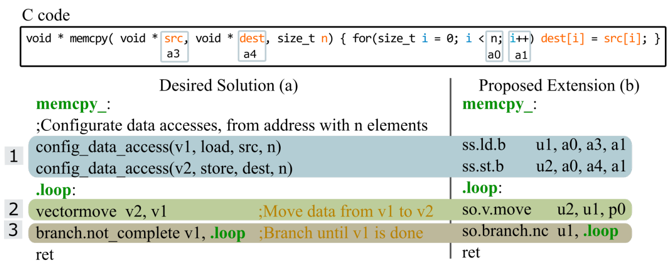

Figure 7: Relation between initial desired solution (in the left) and proposed extension assembly (in the right).

context swapping.

4.2 Streaming Support

In the following paragraphs, there is a collection of important topics for the extension. These discuss some problems and characteristics of the proposed extension.

Stream control: To support stream execution control (i.e. suspend, resume, stop, etc.), a set of instructions are included. The major reason for them, is to allow vector register momentary freeing and restoring, and therefore simple context swapping. This set of instructions can be used to allow the execution of parallel processes without interfering with the streams configuration.

Scalability influence: The vector extension is naturally scalable, even without streaming. This is accomplished with the scalable registers and dedicated instructions tha support the variable sized registers. In particular, this extension features vector load/store instructions that are also tailored for scalability, this is achieved by automatically updating the memory access address based on the active vector length. Thus, allows for automatic progression throughout the iterations. Even so, by mainly relying on streaming, the extension poses a more natural and elegant scalable solution, one that removes the necessity of knowing the vector size at any point in the code. In fact, the extension was designed with the principle that the vector length can be one element or infinite elements. All the instructions were based with this in mind.

Destructive behavior: As seen in Figure 6, the consumption of a register automatically iterates over the stream. Hence, the program must save the data for further use, as the already streamed data is not restorable. The advantage of the destructive behavior is that it removes the necessity of an additional step instruction in each loop, meaning that there is no need for explicit execution control.

Complexity limits: It is noticeable the plethora of memory access patterns that can be described with the actual capabilities of this extension. However, reasonable limits have to be applied in order to not compromise the implementations of supporting microarchitectures. Also, as most of the patterns have a dimensionality up to 4 dimensions, a possible limit could be 4 dimensions per stream with each dimension allowing one modifier. However, allowing 8 dimensions and 7 modifiers should not significantly constraint any possible implementations and, will easily allow for the vast majority of implementations.

Speculative execution and invalid memory access: As a stream is configured, the effective memory addresses are not validated. As a consequence, it is guaranteed that some streams will fetch data from invalid memory addresses. In that case, and only if the consuming instruction is committed, an invalid memory access exception must be raised. The loaded data is invalid in this case, but the fault is only effectivelly raised if an effective use of the data happens (upon commit). In the case where a stream is configured while in speculative execution, it will be iterated immediately. In particular, waiting for the configuration instruction to commit would create a severe negative impact in latency.

5. MICROARCHITECTURE SUPPORT

In order to support the vector streaming instruction set, a traditional processor needs to be extended with streaming specialized structures. In this section, a proposed suporting microarchitecture is detailed, alongside with an in-depth exploration of particularly important structures. In Figure 8, the modifications to a traditional pipeline are depicted. The streaming

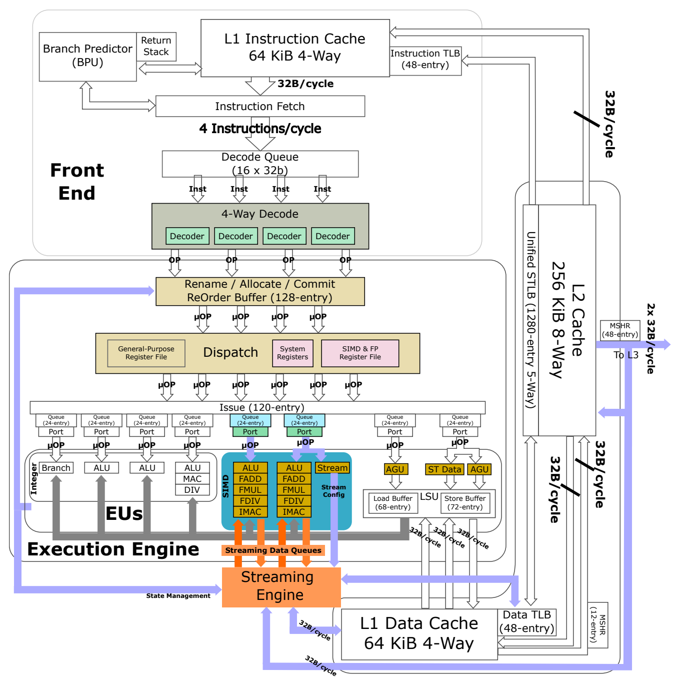

Figure 8: Microarchitecture diagram, with the introduced modifications highlighted.

engine is the core of the streaming process in this proposed microarchitecture. Hence, the microarchitecture modifications are centered on it. In detail, the streaming engine handles all the configuration and data transactions between the core and memory.

The modified structures in the reference architecture are detailed as follows:

1 - Core modifications: The adaptations at the processor's core level comprehend the register file extension and the stream

renamer structure. First, the register file must support the vector and predicate registers. In fact, the scoreboard also needs to be extended to support the new registers. Admittedly, it was possible to create an independent vector register file, however, that is not necessary, as the register file did not suffer extensive logic modifications. Secondly, to allow the hardware to take advantage of the out-of-order paradigm, the streams also have a specialized renamer who physically separates the architectural streams from the physical ones, improving consumption latency and not restricting the maximum speculation depth.

2 - Datapath compatibility: The following interfaces implement the communication between the pipeline and the streaming engine:

(a) Stream Configuration: As every stream is configured by the core (as an instruction), the execution engine is directly connected to the streaming engine. This communication channel allows for an unidirectional transaction, where each configuration instruction will sent the respective configuration data. Additionally, as the pipeline's execution core is executing out-of-order, the streaming engine will enforce order in the configurations.

(b) Streaming Data Queues: In order to buffer the non-synchronous data availability and data processing rates, two dedicated load and store first-in-first-out queues are used. Each queue module is composed of smaller queues, one for each physically available stream. As the processor intents to consume from a stream, the queue is notified of the request, then, when the data is available, it is streamed to the respective vector register inside the register file. On the store activity, upon instruction writeback, the data is writen to the store queue. Only after, when the respective instruction is committed, will the queue forward the store request to the streaming engine, who will then handle the memory transaction.

(c) State management: To give support for speculation, there must be state synchronization between the pipeline, streaming engine and queues. In detail, it is necessary to notify all structures when an instruction is issued, commited, and mispeculated. To guarantee a coherent state in the new structures and in the pipeline, two important phases of the streaming process are needed: **First, configuration:** when the configuration of a stream occurs, the streaming engine always starts processing if there are no validity errors (e.g. end configuration before a start configuration). However, if there is a mispeculation signal from the pipeline, the configuration and the already progressed processes must be reverted. **Secondly, consumption:** if there is a mispeculation targeting an instruction that already consumed data, the queues must be reverted. Generally, it would be necessary to reload the consumed data from memory or get the data from the register file. To avoid such time costly processes, the load queue will only discard the data after a commit. Obviously, this has impact on the effective size of the queues, as they will be full more often, and so leading to more stalls. In sum, it is a clear tradeoff between size and performance. Similarly, the store queue only effectively stores data upon instruction commit. The process of handling speculation in the load queue is depicted in Figure 9. **In addition,** the stream renamer and

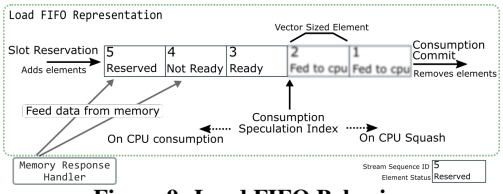

Figure 9: Load FIFO Behavior

the vector register file also support speculation. In particular, the speculation support is similar to the analogous structures, already present in a traditional processor.

3- Streaming Engine: An high-level representation of the streaming engine is shown in Figure 10. As the streaming engine is a complex module, it is not possible to represent the complete details in this document. The streaming engine is composed mainly of three internal processes: configuration, processing and memory access. All of the above are depicted in the figure.

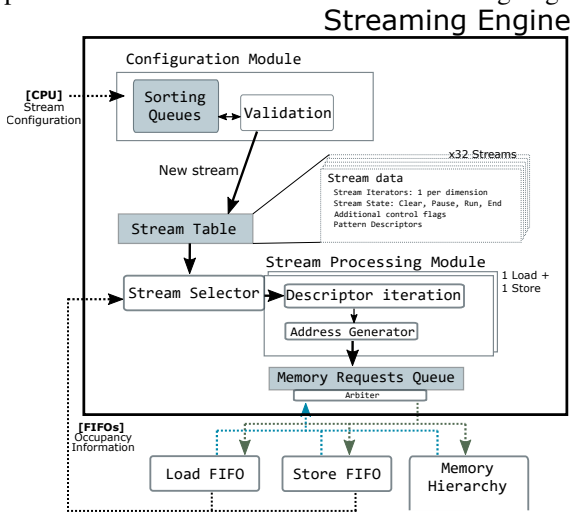

Figure 10: Streaming Engine Logical Block Diagram

Configuration: The configuration process is actuated each time a new signal from the processor execution engine arrives. In detail, the configuration signals can be one of: new stream configuration, pause or resume stream or other signal related with state control. Due to out-of-order execution, these aforementioned signals (commands) can arrive unordered. Hence, the configuration process must sort the commands, this is executed in the sorting queues module. To simplify the process of sorting, the table is first indexed when an instruction is renamed and the data arrives only during execution.After sorted, any complete chain of commands is validated and saved to the *Stream Table*. Each stream is composed of decriptor iterators, who are responsible to keep the current state of iteration. In addition to this, the configuration parameters (pattern descriptors) are also saved, these parameters are the ones already defined in section 3.

Processing: The processing is the core module of the streaming engine, where the memory addresses are generated based on the pattern decriptors. This processing module is divided into two phases: stream selection and iterator processing plus address generation.

(a) Stream selection: To keep the hardware complexity low, during each executing cycle, a stream is selected for processing. This reduces the processing part to a simple and easily replicable component, and thus, allows for other implementations to scale directly from this one. Subsequently, by processing only one stream per cycle, the hardware that is idle is minimized. Hence, for less instensive applications this solution is more power and area efficient. Alternatively, the implementations can choose to use more processing modules, with the known associated cost. Apart from this, the selection of streams takes into account the state of the stream and the associated (load/store) queue occupancy. As a result, the processing will always target streams that are running and that the respective queue will have enough free space.

(b) Iterator processing and address generation: The process of iteration is somewhat complex and the aims of this paper is to explore the overall solution. As a consequence, it's representation in more detail is not included here. In short, a set of iterators (one for each descriptor) are incremented each time a stream is processed, the iteration process starts with the outer

L1I Cache	64KB, 4-way set-associative, 64B line
L1D 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)

Table 2: CPU model configuration parameters, based on [5].

descriptor and consecutivelly arrives at the innermost descriptor. When the innermost descriptor is fully iterated, the descriptor in the above level are iterated once, after which the lower level descriptor starts iterating again. This process is repeated until all dimensions are iterated. However, to allow for multiplexed processing between streams, the above process is suspend and resumed multiple times. Each time it is suspended the processed iterations are used to generate memory addresses. The memory address generation is based on all the stream descriptors and the respective iteration states.

Memory access: The memory access is done through the memory hierarchy port, which directly connects to all levels of caches and memory. In addition, an arbiter schedules the transactions according to priority and availabily. In stream load scenarios, a request is made to the cache. When the data is ready, it is forwarded to the load queue. This process is handled internally by the streaming engine, in the memory requests queue. In fact, this removes the need to significantly modify the cache access mechanisms. Moreover, in store scenarios, the data from the store queue is merged with the address information in the memory requests queue, and then forwarded to the memory.

6. EXPERIMENTAL METHODOLOGY

With the objective to model, test and evaluate the instruction set architecture and the supporting microarchitecture, an ISA simulator and a microarchitecture simulator was required. With this in mind, the Gem5 simulator, which encompasses system-level architecture and processor microarchitecture simulation was used. By being open-source and modular, all the code is widely available and ready to be modified and extended. In addition to this, it holds support for the RISC-V instruction set architecture. Finally, the simulation of the out-of-order cpu pipeline alongside with an extensive memory architecture simulation allows the proposed microarchitecture to be thoroughly tested and evaluated. [22, 23]

In addition to the simulator, a realistic processor model is also needed to obtain results that are relevant. Not only this, but the inevitable comparison with Arm SVE, lead to the use of a processor model that is as close as possible as the one int the SVE paper [5]. In Table 2, a representation of the model main parameters is given.

Finally, to support the porting of benchmarks to the Unlimited Vector Extension (UVE) instruction set, the gnu compiler toolchain was extended to support every UVE instruction and registers [24, 25].

7. RESULTS

The evaluation of this work was divided into an instruction set evaluation and into a microarchitecture evaluation. Evaluating the instruction set independently removes the influency of the implementation parameters from the instruction set comparisons. In particular, the analysis of the instruction set allows us to do a first estimate of the achieveable performance. This estimate is only valid considering that the cpu model is the same, thus the difference in executed instructions of a given type (e.g. SIMD) is equiparable to the difference in performance.

The application code used throughout this document (MEMCPY) was also benchmarked. Figure 11 shows the differences in number of instructions for a 1 kilobite memory copy. UVE uses 10 times less address calculation and loop indexation (ACLI) instructions than SVE. In addition, the memory access (load/store) instructions are reduced in the same proportion. However, UVE needs to use a specific move instruction to active the streaming mechanisms iteration, a total of 64 moves were used. In sum, UVE uses a total of 0.17 instructions per transacted byte, while SVE uses 0.42 instructions per byte. UVE uses 2.49 times less instructions than SVE. In addition, Arm and RISC-V use more instructions, however these are executing scalar code.

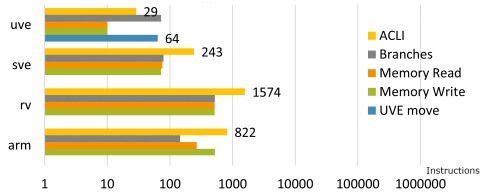

Figure 11: MEMCPY instruction set evaluation

Moreover, the SAXPY[4] benchmarked was also used to evaluate the instruction set performance. Figure 12 depicts the percentage of each instruction type (Integer, Memory, etc) for the evaluated instruction set architectures, and the relation between the total number of instructions between UVE, Arm, RISC-V and SVE. Based on the results, for the SAXPY benchmark, UVE completely removes the instructions categorized as overhead (non essential to processing instructions, e.g. comparisons). Additionally, UVE also removed all integer (address calculation and loop indexation) and all memory access instructions. UVE requires 4 times less instructions than SVE to execute SAXPY.

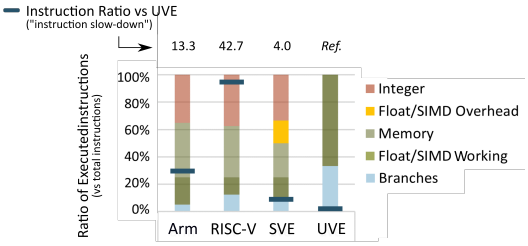

Figure 12: SAXPY instruction set evaluation

Based on both results, there is opportunity for an supporting microarchitecture implementation to have improvements by using UVE. Figure 13 shows a compilation of results for 4 different benchmarks. The IRSmk[5] benchmark is a memory-bound application, while HACCmk[6] is compute-bound. For the MEMCPY benchmark, UVE presents a speedup of 1.44 in relation to SVE. The previously detailed instruction set results suggested an improvement of 2.49 times less instructions, it clear however that the memory performance influences the microarchitecture results. With Single-precision computation of the product of A with each element of matrix X added to the respective element of the matrix Y. - simple 1-D memory access, low in compute intensity. (SAXPY), a speedup of around 3.7 times was obtained, and the instruction set evaluation showed

[4]SAXPY - Single-precision computation of the product of A with each element of matrix X added to the respective element of the matrix Y. - simple 1-D memory access, low in compute intensity.
[5]IRSmk - Scientific microkernel, used in radiation simulations - very intensive 3-D memory accesses, with low computational effort.
[6]HACCmk - Scientific microkernel, used in universe simulation - simple memory accesses with intensive computation.

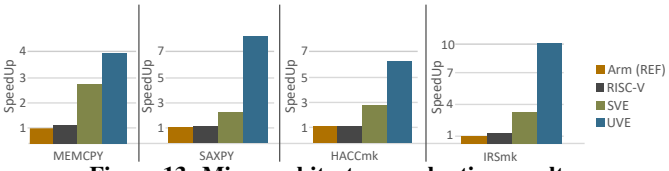

Figure 13: Microarchitecture evaluation results.

a decrement of 4 times in total number of instructions, comparing UVE with SVE. In less memory-bound scenarious, the microarchitecture results are much closer to the instruction set results, this is a consequence of less dependency on the memory. Furthermore, UVE show performance improvements of 2.1 times in HACCmk and 3.2 in IRSmk.

8. CONCLUSION

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, and new ISA extensions are emerging to improve their performance and scalability. In particular, with the emerging presence of scalable vector SIMD extensions, each processor implementation can be tuned to achieve any desired SIMD performance level. On the memory side, recent works show the concept of data streaming applied to a processor memory structure, while achieving improved 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 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 variants. 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, MEMCPY). 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 with 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).

9. REFERENCES

[1] ARM, "Introducing NEON™ Development Article," 2009.

[2] C. Lomont, "Introduction to Intel Advanced Vector Extensions," *Intel White Paper*, p. 21, 2011.

[3] 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*, pp. 302–310, 2000.

[4] 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.

[5] 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, pp. 26–39, 3 2017.

[6] 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, pp. 250–263, 2 2011.

[7] 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), pp. 495–496, ACM Press, 2009.

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

[9] 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, pp. 54–62, 8 2003.

[10] 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.

[11] A. López-Lagunas and S. M. Chai, "Memory Bandwidth Optimization through Stream Descriptors," 2006.

[12] 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.

[13] 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.

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

[15] A. Waterman and K. Asanovic, "RISC-V "V" Vector Extension," tech. rep., 2019.

[16] A. Sodani, "Knights Landing (KNL): 2nd Generation Intel® Xeon Phi™ Processor," tech. rep.

[17] 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*, pp. 355–365, 2012.

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

[19] 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.

[20] N. Neves, *Energy-Efficient Computing: Adaptive Structures and Data Management*. PhD thesis, Instituto Superior Técnico, Universidade de Lisboa, 1 2019.

[21] 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, (New York, New York, USA), pp. 697–710, ACM Press, 2016.

[22] 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, p. 1, 8 2011.

[23] 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.

[24] A. Griffith, *GCC: the complete reference*. McGraw-Hill, Inc., 2002.

[25] R. Stallman, *Using and porting GNU CC*, vol. 67. Free Software Foundation Cambridge, Massachusetts, 1992.