RISC-V Streaming Extension Support on the Spike Simulator

João Pedro Rosa Baptista
Instituto Superior Técnico
University of Lisbon
Lisbon, Portugal

***Abstract*—Vectorial and Singe-Instruction-Mutiple-Data (SIMD) instruction-set extensions have gained added attention in the last decade, as a result of an increased prevalence of computational demanding application domains, pushing the need to exploit as much data-level parallelism (DLP) as possible. The numerous Streaming SIMD Extensions (SSE) and Advanced Vector Extensions (AVX) extensions from Intel/AMD or the NEON and Scalable Vector Extension (SVE) extensions from ARM are some well-known examples of these Instruction Set Architecture (ISA) extensions. Following these same steps, the well-known RISC-V ISA has recently established a comparable vectorial extension, known as the RISC-V Vector (RVV). Other ISA extensions have also been developed to enhance the performance and power/energy efficiency of computing systems. The extremely successful Unlimited Vector Extension (UVE), developed at the INESC-ID HPCAS lab, is one of those extensions. Its prime objective is to provide consolidated support for data-stream processing, alleviating the Central Processing Unit (CPU) from the memory indexing/addressing tasks, while also simplifying loop control. Spike is recognized as the golden reference functional RISC-V ISA software simulator. Making justice to its title, Spike already supports a large collection of extensions including the pertinent RVV extension. However, it still lacks accompany the recent arise of stream-based ISA extensions. Having what was stated in mind, this work proposal aims to define a new RISC-V stream-based extension, integrate it with the rest of the RISC-V ISA and introduce support for the defined extension on the Spike functional simulator. As a result, users will be able to explore such extension on C/C++ applications, reaping all the benefits of this highly promising extension to RISC-V ISA.**

***Index Terms*—Vectorial and SIMD instruction-set extensions, Stream, RISC-V ISA, RVV, UVE, Spike**

I. Introduction

A. Motivation

Since the beginning of the computing area, there is a perpetual competition to maximise processor performance. There were times when the growth of the processing power would be predictable as the number of transistors increased, as stated in Moore's Law [1]. Helping Moore's Law was the Dennard Scaling [2] projection, which stated that as transistor density increased, power consumption per transistor would drop, making computers energy efficient. Although, this era has ended [3]. With the emergence of the *"Thermal Wall"* problem, the slow down of Dennard scaling and the end of Moore's Law, the procedure of shrinking transistors, ensuring even smaller circuits, is reaching its physical boundaries.

One consequence in the computing world was the stagnation of the single core frequency. In an effort to mitigate this issue, the main focus shifted from exploring Instruction-Level-Parallelism (ILP) to finding ways of combining instruction, task and Data-Level-Parallelism (DLP), which constitute the core of the architecture paradigm present in modern high-performance processors. Furthermore, the recent proliferation of certain application domains, such as deep learning, with increasing computational performance demands, has reinforced the need to further exploit DLP. In this scenario, the development of vectorial and Single-Instruction-Multipe-Data (SIMD) instruction-set extensions, which allow multiple data elements to be processed at the same time, has been gaining renewed attention.

Traditional concepts of these extensions, such as the numerous extensions from Intel [4], [5], [6] and the NEON [7] extension from ARM are based on apriori fixed-size registers, which despite fulfilling their purpose, pose questionable issues. First, having a fixed vector length leads to portability issues, since any modification of the length requires a new instruction set to be defined, and therefore new code needs to be written, compiled and deployed. Next, since the optimal vector length always depends on the workload, with a fixed vector length, it is often hard to choose the perfect vector length for a given application. This problem is addressed by Vector-Lrength Agnostic (VLA) SIMD extensions that do not rely on a fixed vector size. In [8], the recognized RISC-V organization denoted the development of a VLA SIMD extension named RISC-V Vector (RVV), which mainly adds the feature of tuning the vector length at runtime. In [9], ARM also denoted the development of a VLA SIMD extension named Scalable Vector Extension (SVE), which mainly relies on predication to drive vectorized loop control flow decisions. When compared with the typical SIMD extensions, the added overhead, resulting from the instructions needed to attain a variable vector length, can constrain the application throughput.

Despite parallelism being a solution in terms of performance, by itself, does not solve the challenge of energy-efficient computation that was exacerbated by the end of Dennard scaling. To tackle the lack of energy-efficient computation, Domain-Specific-Architectures (DSA) and Domain-Specific-languages (DSL) emerged, which rely on designing architectures tailored to a specific problem domain, offering significant performance and efficiency. Attached to the growth of DSA, some techniques, such as memory decoupling, started

to gain relevance [10], [11]. By decoupling the memory accesses from the computation, it is possible to achieve more performance and efficiency in both domains. This is mainly attained by allowing data acquisition to occur in parallel with data manipulation. In fact, DSA are in most cases related to the use of data-flow and stream-based techniques to improve memory transactions [12], [13], [14], [15].

Based on the growth of DSA, general-purpose processors adopted data streaming techniques to mainly combat the Von-Neumann architecture limitations [16], [17]. These limitations are clear by the fact that the performance of memory devices has not improved at the same rate as that of processors, leading to an increase in the gap between Central Processing Unit (CPU) and memory speeds.

Bonded with this adoption, several ISA extensions featuring streaming techniques were developed. One novel example is the Unlimited Vector Extension (UVE) extension [18], which by aggregating both VLA SIMD and data streaming approaches promises to reduce the CPU's workload associated with memory addressing/indexing in general-purpose RISC-V-based processors. Despite the considerable number of streaming extensions being developed, there is still not one associated with the official RISC-V extensions.

Despite the growth of stream-based approaches, there is still little hardware support for such approaches. By the beginning of this work, there was no hardware support for the stream-based extension that is going to be proposed. Therefore, the only valid solution to test the proposed extension is through a functional simulator. Connected with this whole paradigm, Spike [19] is a RISC-V ISA simulator, which implements a functional model of one or more RISC-V harts. Spike, more known as the golden reference RISC-V ISA simulator, supports all standard RISC-V instructions as well as a vast number of extensions, in which the previously mentioned RVV extension. However, despite its title, Spike still lacks in providing support for the newly raised stream-based approaches.

B. Objectives

In general-purpose processors, the performance is majorly limited by the time spent with memory addressing/indexing actions. To solve this issue, new streaming techniques have been adopted into general-purpose processors [16], [17], promising to reduce the existent gap between processing and memory access speed. Despite the considerable number of developed extensions based on streaming, the RISC-V paradigm still does not have an official stream-based extension. Based on what was stated, this represents an open chance to **propose a new RISC-V stream-based extension**.

Moreover, to exploit Data-Level-Parallelism (DLP) at maximum in general-purpose processors, the development of Single-Instruction-Mutipe-Data (SIMD) extensions reached its peak [4], [5], [6], [7]. Furthermore, by making use of a variable vector length, Vector-length-Agnostic SIMD extensions further enhance the performance and energy efficiency [8], [9]. This opens a chance to **combine the performance enhancement delivered by RISC-V Vector (RVV), with the addition of decreasing the memory access latency by using streaming approaches**.

Additionally, Spike, recognized as the golden reference Instruction Set Architecture (ISA) simulator by RISC-V, implements a functional model of one or more RISC-V harts. However, **Spike does not support any extension that denotes streaming techniques**.

Following the apriori statements, the main objective of this work is to introduce support for a new RISC-V stream-based extension, fully compatible with the RISC-V paradigm, despite being more focused in RVV, on the Spike functional simulator. Moreover, this goal can be divided into three major objectives:

- Assess the main requirements to introduce a new RISC-V stream-based extension.
- Investigate the possibility of interaction and interoperability between a stream-based extension and standard RISC-V extensions, more specifically the RVV extension.
- Validate the defined extension through ISA simulation.

C. Contributions

This work is grounded on previous works [16], [17], [18] that demonstrated that the use of streaming mechanisms is the go-to in order to improve the overall performance in the memory transactions present in High-Performance Computing applications. Accordingly, this work first specified the necessary requirements to define a new RISC-V stream-based extension. Furthermore, it discussed the possibility of integrating a full stream-based extension in the RISC-V paradigm. Finally, to give evident proof of the enhancement delivered by the stream-based extension, this work aimed to provide support for the defined extension on the Spike Simulator. Therefore, during the development of this work, the following contributions were achieved:

- Definition of a novel stream-based extension, whose main focus is to increase performance in memory access.
- Integration of the new stream-based extension with the subsets of the RISC-V Instruction Set Architecture (ISA), more specifically the RISC-V Vector (RVV) extension.
- Deployment of the proposed extension in the Spike Simulator and its functional validation and interoperability verification with other standard RISC-V extensions.

II. Background and State-of-the-Art

A. Vectorial and SIMD extensions

Single-Instructions-Multipe-Data (SIMD) systems [4], [5], [6], [7], [9], [8] enhance, as the name indicates, the execution of the same operation on multiple elements present in a dataset. This SIMD parallelism is commonly accomplished by resorting to vectorization, where successive instances of a scalar operation, which operates on a set of single operands, are transformed into a vector instruction that operates on multiple operands at once.

In the last two decades, an increasing number of processors that adopt SIMD operations have been developed. This adoption was majority handled by adding extensions to the existing Instruction Set Architecture (ISA). These extensions have been established as the essential go-to method to potentiate the exploitation of Data-Leve-parallelism (DLP) in High-Performance-Computing (HPC) workloads.

To represent this SIMD mechanism, Figure 1, denotes the difference between using Single-Instruction-Single-Data (SISD) or SIMD operations on performing a simple sum between two sets of four elements each.

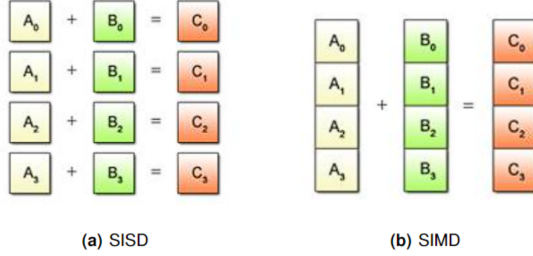

Fig. 1. SISD vs SIMD Operations.

The evolution of the development of SIMD extensions is emphasized in specific SIMD extensions that are worth mentioning.

Multi-Media Extensions (MMX): Intel's Multi-Media Extensions (MMX) [4] extensions to the x86 ISA defined four new data types. To support these new data types MMX provided eight 64-bit general-purpose registers, named MM0 through MM7. Despite the delivered enhancements, MMX still presents a major weakness. In addition to MMX providing only integer operations, the MMX registers are aliases for the existing floating-point registers. As a result, operations involving the floating-point stack might also affect the MMX registers and vice versa, making it almost impossible to work with floating-point and SIMD operations in the same program.

Streaming SIMD Extensions (SSE): Intel's Streaming SIMD Extensions (SSE) [5] expanded the x86 ISA, emphasizing floating-point instructions and introducing 128-bit registers, named XMM0 through XMM7. Subsequent SSE generations (SSE2, SSE3, SSE4) [20], [21], [22] extended support for multimedia and introduced instructions for cryptography, achieving speedups in diverse applications [23].

Advanced Vector Extensions (AVX): Intel's Advanced Vector Extensions (AVX) [6] main feature was the support for 256-bit vector operations. AVX defined sixteen 256-bit registers, named YMM0 through YMM15 and a 32-bit control/status register called MXCSR. AVX2 [24] expanded vector integer instructions and added support for various data types. AVX-512 [25] provided support for 512-bit vector operations, defining thirty-two 512-bit registers, named ZMM0 through ZMM31. These extensions delivered substantial speedup in applications like image processing and N-body simulations [26]

NEON Extension: ARM also introduced its own SIMD extension called NEON [7], which was designed to be fully compatible with the ARMv7 and ARMv8 architectures.

NEON instructions and floating-point instructions share different register files, which is a collection of registers that can be accessed as 32-bit, 64-bit or 128-bit. NEON registers can be represented as thirty-two 64-bit registers, named D0 through D31, or sixteen 128-bit registers, named Q0 through Q15. Summing up, while D registers can keep vectors containing either eight 8-bit, four 16-bit, two 32-bit or one 64-bit element/s, Q registers can keep vectors containing sixteen 8-bit, eight 16-bit, four 32-bit or two 64-bit elements.

Traditional SIMD Extensions Limitations: The numerous SIMD extensions ascribed so far achieved innumerable performance enhancements in multiple domains. However, all of them were developed to operate with fixed-size registers. Despite simplifying the implementation and limiting the hardware requirements, since the ideal vector length often depends on the task being attained, this approach could never reap all the possible performance. Additionally, any change to the register length normally entails the use of newer instruction-set extensions, leaving earlier implementations of code outdated. In [27], the authors denoted the referenced problems, exploring dynamically-sized vector operations on SIMD architectures and the potential for variable-length vector registers to overcome the apriori limitations.

B. Vector-Length Agnostic (VLA) SIMD extensions

To overcome the problems mentioned, a different approach of SIMD extensions denominated Vector-Length Agnostic (VLA) SIMD extensions started to appear. VLA SIMD extensions are a set of instructions and data types that allow a processor to perform SIMD operations on vectors of varying lengths. This way, these extensions allow a programmer to write code that can operate on vectors of any length, rather than being limited to a fixed set of vector lengths. This can be useful in situations where the length of the vector is not known at compile time, or when the vector length may vary based on the input data. Furthermore, these extensions grant more flexibility in terms of the architectures that can perform SIMD instructions. For example, HPC processors can make use of large vectors to attain high throughput, while low-power processors can adopt smaller vectors to fulfil power and resource constraints. Moreover, these extensions grant portability, as the same code can be re-utilized in processors with different vectorial architectures without needing to recompile it. Two dominant examples of these type of extensions are the Scalable Vector Extension (SVE) [9] and the RISC-V Vectors (RVV) [8].

Scalable Vector Extension (SVE): The SVE [9] is a VLA SIMD extension integrated into the ARMv8-A architecture. SVE introduces flexible vector processing capabilities by supporting vector lengths ranging from 128 bits to 2048 bits in 128-bit increments. This adaptability is a primary feature of SVE. SVE introduces a set of registers, including thirty-two vector registers (Z0-Z31) and sixteen predicate registers (P0-P15). The size of the Z registers is implementation-dependent within the specified range, allowing for the accommodation of 64-bit, 32-bit, 16-bit, and 8-bit data elements. The low 128 bits of these registers overlap with ARM NEON registers. SVE heavily relies on predicated instructions to determine which vector elements to process. Predicate registers (P0-P15) hold one bit for each 8-bit data element in the Z registers, indicating whether the element is active or not. SVE's design leverages predication to drive vectorized loop control flow decisions. It employs a family of "while" instructions in combination

with scalar count and limit registers to generate predicates that control loop iterations. In performance comparisons, SVE outperformed ARM NEON in vector utilization and achieved speedups of up to three times in certain applications [28].

In summary, SVE introduces a versatile SIMD extension with variable vector lengths and advanced predicated processing capabilities, making it a valuable addition to ARM's architecture for vector processing tasks.

RISC-V Vector (RVV): The RISC-V Vector (RVV) extension [8] is a recent VLA SIMD extension designed for the RISC-V ISA. Initially proposed in 2015 and refined in 2021 as part of the RISC-V Vector 1.0 specification, RVV has gained popularity due to RISC-V's open-source nature and its adaptability to various applications. RVV introduces thirty-two vector registers (v0-v31) and seven unprivileged Control Status Registers (CSRs) (vstart, vxsat, vxrm, vcsr, vtype, vl, vlenb) to the base scalar RISC-V ISA. Unlike SVE's predication approach, RVV achieves Variable-Length Architecture through the dynamic configuration of a specific vector length (VLEN). VLEN is constrained by the constant ELEN, which represents the maximum size in bits of a vector element produced or consumed. RVV instructions are monomorphic, meaning there are specific instructions for each data type. RVV uses mask bits to support instruction predication, with one mask bit per element in a vector. These bits determine whether an instruction is performed for the corresponding element. Unlike SVE's dedicated predication registers (P registers), each of RVV's thirty-two registers can serve as mask registers, where mask bits are packed sequentially.

The dynamic configuration of vector size and element size is executed through the "vsetvli" instruction, allowing for adaptive vector processing. The "vsetvli" instruction starts with a request for vector size (in elements) and element size, followed by a comparison to the implemented vector length. The minimum of the two values is used to configure all vectors and is written to the destination operand. The "grouping factor" is used to group consecutive registers, particularly useful when the requested size exceeds the implemented vector size.

In summary, RVV is an adaptable VLA SIMD extension designed for RISC-V ISA, providing dynamic vector length configuration, mask bits for predication, and flexibility in vector register usage. It has gained attention for its open-source nature and application versatility.

SVE and RVV Limitations: Despite the crucial improvements delivered by the referenced VLA SIMD extensions, these extensions also come with issues that limit their performance and range of uses. in [18], the authors were able to identify these extensions' bottlenecks. While in SVE, its large instruction overhead comes from the dependence on predication, in RVV this overhead mainly falls on the vector control instructions needed. Figure 2 illustrates the referenced overhead instructions on the saxpy kernel implementations on SVE and RVV.

Overall, both SVE and RVV can impose a significant amount of instruction overhead (shaded instructions in both figures), which is caused by memory indexing, loop control, and even memory access, none of which directly increase the throughput of data processing. The majority of the loop code frequently consists of these overhead instructions, wasting CPU resources and adversely affecting performance.

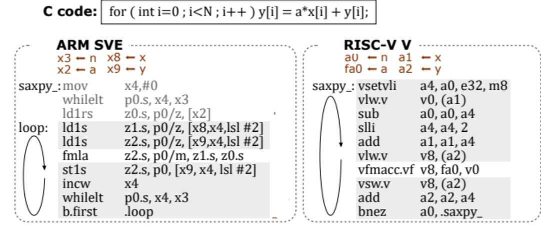

Fig. 2. Saxpy kernel implementation on ARM SVE and RISC-V V. Image from [18].

C. Data-flow and Stream-based approaches

From a different point of view, more recently, in general-purpose processors, the speed at which a processor can process data is outscaling the speed at which data can be accessed from memory. The existent gap, known as the "memory wall" [29], is a major bottleneck in the performance of multiple systems. For example, the performance of data-parallel applications is no exception to the rule and is often constrained by this memory hierarchy problem. Coupled with the emergence of domain-specific architectures, data-flow and stream-based approaches have been gaining ground [12], [13].

Overall, these new approaches lead the way for programmers to investigate various complementary features to boost throughput, such as memory access decoupling and specialization, data prefetching, and efficient parallel computation. Owing to these features, data streaming has lately been adopted beyond domain-specific computing into general-purpose processors to overcome the Von Neumann architecture's limitations. In [16], the authors realized a vast and relevant study to find and tune the best configuration to specialize memory primitives. The first conclusion was that the best way of exposing rich semantic information about memory operations at fine grain at the ISA level was by using streams as the structure for memory accesses. Streams denote repeated patterns of memory access, occurring due to loops and nested loops.

Unlimited Vector Extension (UVE): The Unlimited Vector Extension (UVE) [18] is a modern SIMD extension designed for RISC-V-based general-purpose processors. Unlike previous extensions, UVE combines VLA SIMD processing with data streaming to enhance performance. UVE includes a streaming interface that allows effective prefetching of data and linearizing non-coalesced memory accesses. Each data stream is associated with a vector register, enabling instructions to directly use the corresponding stream. This ensures the progression of streams after each interaction with the vector, simplifying loop control with a basic set of stream-conditional branches. UVE employs a hierarchical descriptor-based representation to describe its stream model. UVE provides 32 vector registers (u0-u31) with lengths defined according to the supported data types (byte, half-word, word, double-word). It also includes

32 predicate registers (p0-p31) similar to those in SVE, allowing per-lane execution control and automatically handling boundary conditions.

In terms of performance, UVE significantly reduces the instruction overhead compared to SVE and RVV extensions. In benchmark comparisons, UVE demonstrated an average performance advantage of 2.4× over SVE for vectorized workloads [18].

D. Spike Simulator

Spike [19] is an open-source simulator designed to implement a functional model of one or more RISC-V processor cores. It uses a combination of C++ language features, design patterns, and object-oriented programming techniques to represent the components of a simulated processor, including registers, memory, and pipeline. Spike supports all standard base integer and floating-point RISC-V instructions and a wide range of extensions. It has a modular design that allows for easy modification, extension, and ISA testing. Figure 3 portrays a straightforward view of the Spike Simulator, depicting its most relevant simulated components.

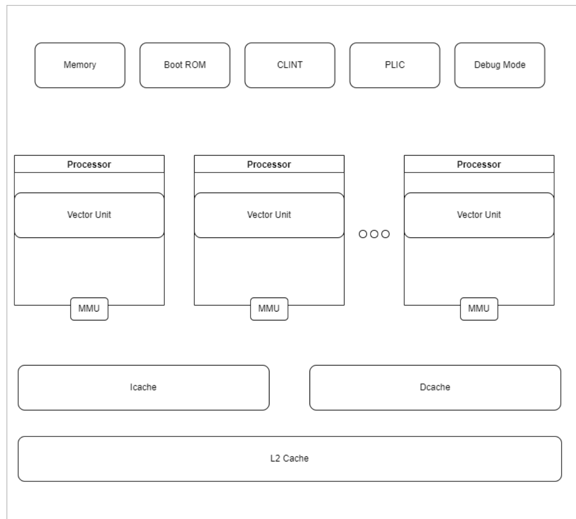

Fig. 3. Spike main modules Overview.

Spike's main modules are composed of the Processor-Dependent Code, the Memory Management, other components such as the Data Bus and the Read-Only Memory and the Vector Unit (to support the RVV extension). In summary, Spike is a versatile RISC-V simulator with a modular design that facilitates processor development and testing. It encompasses various functional modules to simulate RISC-V processor cores and their associated components, making it an invaluable tool for RISC-V developers and researchers.

III. RISC-V STREAM-BASED EXTENSION

In this Section it is presented the proposed RISC-V stream-based extension. To do so, first, it is described the underlying memory access descriptor representation. Following, it is discussed the architectural state by defining the necessary architecture features and mechanisms to support the proposed extension. Finally, it is presented the set of proposed instructions for stream configuration, manipulation and stream-based control flow.

A. Memory Access Pattern Description

In the context of this work, a **stream** is defined as a continuous flow of data that often exhibits a predictable or constant pattern as soon as it starts executing. Because of this, even though the pattern, length, or data type may not be known at compile time, they must be defined through a collection of variables whose values may be determined when the stream is generated. Furthermore, sequential and orderly processing (loading and storing) of the streams must be guaranteed.

To represent a stream of data, the concept of a descriptor is herein used. An access pattern to memory must be described by a descriptor, which in most cases specifies a starting address, word length (in bits), and size. To express more complex patterns, such as multidimensional strides, extra fields might be added. Descriptors may also be made dependent or hierarchical to reflect advanced memory access patterns.

1) Multidimensional Access Encoding: In the extension being defined, a similar solution to the one successfully demonstrated at the UVE extension [18] is proposed. The authors stated that by using a descriptor composed of a three-element tuple {offset, size, stride}, all the straightforward single-dimension patterns can be described. Furthermore, multidimensional access patterns can be described by chaining these descriptors. The formal function for calculating each memory address is defined as seen in the following equation:

$$y(X) = \text{offset} + x_0 + \sum_{k=n}^{n} x_k \times \text{stride}_k, \quad x_k \in \{0, \ldots, \text{size}_k\} \tag{1}$$

To describe straightforward patterns, a single three-element tuple is needed, constituted namely by an offset, a size and a stride. In a single dimension, the offset represents the starting memory location of the pattern that is being described. However, for multidimensional descriptions, the starting memory location is dependent on the offsets and strides of every dimension. The size indicates how many components to produce in a particular dimension. The stride denotes the location of the following element inside the same dimension. These parameters are what makes a descriptor, the minimal representation for a stream. Based on that, to represent a simple stream, a descriptor constituted with an **offset**, a **size** and a **stride** is defined. This introduced descriptor is called **dimension**.

2) Imperfect Loop Access Encoding: Although it is possible to describe a decent amount of memory access patterns relying only on the dimension descriptor, a vast amount of examples are still not possible to be represented. Certain examples like the Lower Triangular Memory Access Pattern where for each outer loop iteration, the inner loop parameter is modified can not be described. To provide the described functionality, a new descriptor type called **modifier** is introduced. This new descriptor has the ability to change a dimension descriptor's parameter. This modifier is represented by a tuple comprised

of four parameters. The first parameter is the **target**, which serves as an identifier of the descriptor parameter to edit (offset, size or stride). The second parameter is the **behaviour**, which specifies the operation to be carried out (increment or decrement). The third is the **displacement**, which is the constant value utilised by the operation to update the parameter value. The final value of the tuple is the **size** of the modifier and it specifies the total number of iterations for which the modification should occur.

3) Indirect Memory Access Encoding: In some applications, it is common to find the use of indirect memory accesses. Listing 1 denotes the C code of one example of an indirect memory access pattern. The denoted memory access pattern cannot be represented by the use of the previously introduced descriptors.

Listing 1. Indirect Memory Acces Pattern C code
```
int i = 0;

for (; i < Nc; i++) {
    B[A[i]];
}
```

The description needs to use the values of $A[i]$ to modify the parameter offset of the pattern B dynamically. To be able to represent such a pattern, it can be re-utilised the idea of the already explained modifier with the slight chance that the source value (displacement) needs now to be dynamic. In addition, the incrementation and decrementation behaviours would restrict the description's potential given that the displacement value is now taken via a stream. Therefore, a modified descriptor called **dynamic modifier** is introduced. This modifier is still represented by a tuple of four parameters. Two of them, the **target** and the **size**, still hold the same behaviour as in the already explained modifier descriptor. The last two parameters are the **behaviour** and the **source**. The behaviour still has the same functionality with the addition that it has a total of five possible etiquettes: Add, Sub, Inc, Dec and Set. The source acts like the displacement but in this case, is dynamic.

B. Memory Access Pattern Descriptors Summary

This section covered the core ideas of pattern description, which serve as the base for the instruction set design. Summing up, the descriptor is the fundamental building block for a stream description; many descriptors can be linked together to provide a demanding description for streams with a high degree of memory access complexity. In our solution, we defined three different forms of a descriptor: the simple three-element tuple descriptor that represents a **dimension**, the four-element tuple that represents a **modifier** and the other four-element tuple that represents a **dynamic modifier**. Figure 4 denotes an overview of the available descriptors.

C. Architectural State

The process of designing the Instruction Set Architecture (ISA) extension is the most crucial part of the work, as the supporting microarchitectures will be affected by any mistake in the ISA definition. Therefore, before going to the actual definition of the proposed extension, it is important to understand a brief overview of the architectures that will support this extension. As so, this Section tackles the architectural components that will support the behaviour specified by the stream-based extension.

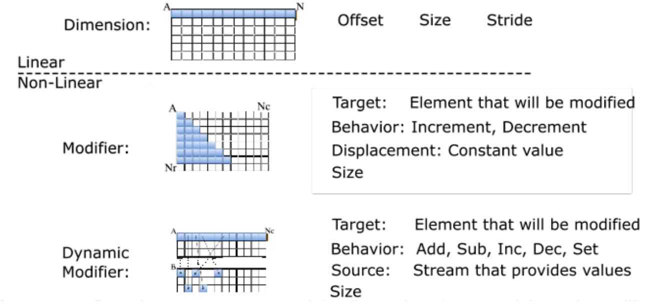

Fig. 4. Descriptors Overview.

1) Microarchitecture Overview: Streaming is a complex process that traditional Central Processing Unit (CPU) pipelines are not ready to support. Consequently, the CPU pipeline needs to be embedded with a streaming mechanism that handles all actions denoted by the stream paradigm. Figure 5 denotes a Microarchitecture overview of a CPU pipeline extended with a *Stream Unit*.

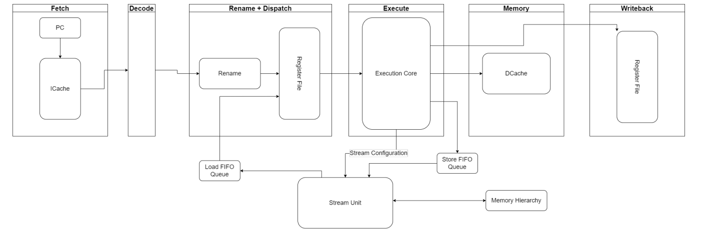

Fig. 5. Microarchitecture Overview with a Stream Unit.

To understand the behaviour of the *Stream Unit* denoted in Figure 5, the main actions in the operation of streams will be, in resume, described:

Stream Configuration: Whenever a new stream configuration instruction reaches rename, it is registered on a stream configuration reordering structure, which is part of the *Stream Unit*. This engine, similar to a re-order buffer, processes the configuration instructions in order, and as soon as the corresponding operands are available. Upon completion of the configuration, the *Stream Unit* immediately starts processing the stream, either by pre-loading data (for input streams), or by calculating store addresses (on output streams), and waiting for the commit of store data.

Stream Iteration: A stream iteration process is logically performed by reading/writing from/to an input/output stream. This is performed at rename, by signalling the *Stream Unit* to iterate the speculative stream state. For an output stream, this also implies reserving space in the Stream Store FIFO buffers and then waiting for both data and commit signals to arrive, to complete the operation. On input streams, the devised solution attempts to minimize the load-to-use latency,

by allocating the head of input streams to physical registers. As a consequence, when a stream-consuming instruction reaches rename, the operand is immediately read and a new data element is pre-loaded to a different physical register.

Stream Termination: The termination of a Stream is achieved at commit, either through an explicit termination instruction or by committing an instruction that signals the completion of the streaming pattern. When such an event occurs, all the structures in the *Stream Unit* are released, allowing the resources to be allocated to a new stream configuration.

Memory Coherence: On the core, stream and non-stream operations are kept coherent by matching the stream load/store state with the core load/store queues and by solving possible stream load/store dependencies through typical request delay, replay, or squash mechanisms. Hence, data written by the conventional pipeline can be immediately read by a newly configured input stream, and data written by an output stream can be loaded using a conventional load instruction. This ensures a reliable transition between sequential code and stream loops.

2) Available registers: The stream-based extension being defined is integrated with the rest of the subsets of the RISC-V ISA. Therefore, a set of new registers is not needed. The only addition needed is to integrate the rest of the RISC-V registers with mechanisms to identify streams.

When cooperating with the Standard RISC-V ISA, the proposed extension will make use of the scalar and floating point registers defined. As so, these registers will be integrated with mechanisms to identify whether a load/store stream is associated with them.

When cooperating with the RVV extension, the proposed extension will make use of the RVV vector registers. As so, these registers will also be integrated with mechanisms to identify whether a load/store stream is associated with them. Furthermore, in the specific case of the vector registers, these will also be integrated with mechanisms that will identify stream loop tails.

D. *Instructions Design*

This Section will describe in detail all the instructions present in the stream-based extension being defined. Furthermore, before going to the definition of the actual instructions, it is important to reference that since the goal of this work is to create an extension based on RISC-V, the names of the instructions pretend to follow this same label. Therefore, inspired by the instructions present on the RISC-V Vectors (RVV) extension, since this extension is mainly based on streaming mechanisms, all the instructions present on this extension have as a prefix the letter ”s” (Stream). Following the concept present in the RISC-V instructions, the rest of the instruction's name reflects the nature of the instruction.

After defining the memory pattern descriptors, it is now important to create the instructions that can efficiently yet simply represent those patterns.

First, let's start by describing single-dimension streams. To describe a simple pattern with one dimension, only a dimension descriptor denoting the offset, the size and the stride is needed. Therefore, to describe a one-dimension stream, the related configuration instruction must provide the offset, the size and the stride parameters. Additionally, this configuration instruction must also set the data width. Furthermore, the instruction also needs to set the transaction direction of the stream, differentiating whether it is a continuous flow of data meant to be loaded from memory or a continuous flow of data meant to be stored in memory. Hence, the proposed stream configuration instruction is denoted in Figure 6.

```
scrt.<dir>.<width>    Vd, Rs1, Rs2, Rs3
```

Fig. 6. Simple Stream configuration instruction.

The demagogic name ”scrt” references ”Stream Create”. The ”dir” parameter sets the transaction direction of the stream, where ”ld” is for a continuous flow of data meant to be loaded from memory and ”st” is for a continuous flow of data meant to be stored in memory. Furthermore, the ”width” parameter sets the data width, whether possibilities are ”d” (8 bytes), ”w” (4 bytes), ”s” (2 bytes) and ”b” (1 byte). Finally the actual instruction parameters ”Vd”, ”Rs1”, ”Rs2” and ”Rs3” denote respectively the destination register, the offset, the size and the stride.

One dimension can completely describe a stream, the previous stream configuration instruction starts and ends the stream description. However, as already explained, there are examples like 3-dimensional memory accesses that require a chain of descriptors to represent the whole of the 3 dimensions. Therefore, the starting stream configuration instruction can have the ”sta” field to denote that other dimension descriptors will be chained up.

To describe a 2-dimensional access pattern, 2 dimension descriptors are needed. To describe the first dimension of the stream, the previously explained configuration instruction is used. In this case, since the stream is multi-dimensional, the first stream configuration instruction will have the ”sta” field.

With the first dimension configuration done, to add a higher level configuration, it is necessary to describe the remaining configuration. Additionally, since it is a 2-dimensional stream, the complete description of the stream is supposed to terminate with the next stream configuration instruction. With this, the proposed configuration instruction that intends to add a dimension representation and terminate the stream description is denoted in Figure 7

```
send    Vd, Rs1, Rs2, Rs3
```

Fig. 7. final dimension descriptor Stream configuration instruction.

In this case, the source registers contain the configuration data of the second dimension. It is important to note that when adding a configuration, the direction of the transaction and the element width are not needed anymore, this information is only necessary for the first configuration.

Up to this point, with the two stream configuration instructions presented, the description of a 2-dimensional stream can be denoted. However, in other cases, it might be necessary to represent up to 8-dimensional streams. Consequently,

to represent the middle dimensions of the stream, another configuration instruction that at the same time is supposed to add a dimension description and not terminate the full stream description is needed. Therefore, the proposed stream configuration instruction that follows this behaviour is denoted in Figure 8

```
sapp    Vd, Rs1, Rs2, Rs3
```

Fig. 8. append dimension descriptor Stream configuration instruction.

In this case, the source registers contain the configuration data of the dimension that is being appended to the full stream description.

Figure 9 denotes all the instructions present in the defined extension and its corresponding behaviours.

Instruction	Description
scrt.ld.[width]	Simple Load Stream
scrt.st.[width]	Simple Store Stream
scrt.sta.ld.[width]	Start load stream configuration
scrt.sta.st.[width]	Start store stream configuration
sapp	Append dimension descriptor to stream in configuration
send	Add dimension descriptor and end configuration of stream
smod.[type].[target].[mode]	Append modifier descriptor to stream in configuration
sdmod.[type].[target].[mode]	Append dynamic modifier descriptor to stream in configuration
s.suspend	Momentarily disable the automatic iteration of a stream
s.resume	Enables again the automatic iteration of a stream
s.terminate	Cease completely the stream configuration
sb.nc	Branch if stream not finished
sb.c	Branch if stream finished
sb.ndc.x	Branch if dimension x not finished
sb.dc.x	Branch if dimension x finished
scfgvec	Configure vector-coupled Streaming

Fig. 9. Stream-based extension Instructions.

IV. Spike Simulator Extension

This section describes the implementation of the defined RISC-V stream-based extension on the Spike functional Simulator. Since the Spike simulator does not inherently support a streaming paradigm, to denote the stream-based extension's behaviour, it is necessary to first add simulated streaming mechanisms into the Spike architecture. To do so, first, the simulated components added to Spike will be described, that support the streaming mechanisms of the defined extension.

Since the majority of this section is related to the vast code implemented during the time of this work, only the main simulated components will be tackled.

To provide the streaming mechanisms needed for the support of the defined extension, a new simulated component, namely the **Stream Unit** was created and extended to the Spike main modules, previously denoted in Figure 3

Listing 2 denote the full code that represents the referenced simulated Stream Unit class.

Listing 2. Stream Unit Class

```
class streamUnit_t
{
public:
  processor_t* p;
  int stream_type[NSR];
  stream streams[NSR];
  bool scalar_cooperation;
  bool rvv_enabled;
public:
  streamUnit_t():
    p(0),
    stream_type{0},
    scalar_cooperation(true),
    rvv_enabled(true) {
  }
  void init_stream_unit();
};
```

In the previous code, the most important variable is the ”streams” array which will be composed of ”stream” structures. This array denotes all the possible streams that could be defined. Moreover, Listing 3 denotes the actual code of the ”stream” structure.

Listing 3. Stream Structure Code

```
struct stream
{
  std::vector<uint64_t> addresses;
  std::variant<
        std::vector<uint32_t>,
        std::vector<uint16_t>,
        std::vector<uint8_t>,
        std::vector<uint64_t>
    > values;
  dimension dimensions[8];
  modifier modifiers[8];
  int dimension_counter;
  int elements_counter;
};
```

Examining the Stream structure code, the defined ”addresses” array will hold the various memory positions that constitute a stream. Depending on which memory access pattern the stream is representing, this array will hold the respective necessary memory positions. Moreover, the defined ”values” array will hold the respective values of the memory positions of the stream.

Up to this point, in the stream representation, the simulated components that will denote the actual stream memory access positions and the respective values were already explained. However, there is still a need to simulate the various descriptors present in the defined extension, that allow to arrive at the memory positions that constitute the stream. As so, the ”dimensions” and the ”modifiers” array will respectively denote the dimensions and the modifiers descriptors present in the stream. Listing 4 and 5 denote respectively the actual code of the ”dimension” and the ”modifier” structure.

Listing 4. Dimension Structure Code

```
struct dimension
{
  uint64_t offset;
  uint64_t size;
  uint64_t stride;
  int dim_elements_counter;
  int total_dim_elements;
};
```

Listing 5. Modifier Structure Code

```
struct modifier
{
  uint64_t target;
  uint64_t behavior;
  uint64_t displacement;
  uint64_t size;
};
```

To define a stream dimension three elements are needed, the offset, the size and the stride. Therefore, in the dimension structure, these same elements are defined. To define a modifier four elements are needed, the target, the behaviour, the size and the displacement. Therefore, in the modifier structure, these same elements are defined.

V. Results Overview and Discussion

This section provides a comprehensive validation of the proposed RISC-V stream-based extension. First, it is described the validation methodology. Then it is presented a detailed breakdown and characterization of the set of benchmarks used in the validation steps.

Despite having the extension instructions code already in the Spike Simulator, a testing process was needed to validate the proposed extension. The testing protocol denoted in figure 10 was adopted.

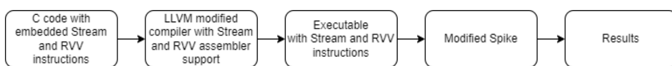

Fig. 10. Testing Protocol used to validate the modified Spike with the defined Stream-based Extension.

This testing protocol is defined by sequentially performing the following steps:

- Implement a C code denoting a benchmark. The benchmark is executed with the use of the defined stream-based extension's instructions and the RVV's instructions, the C code will be implemented by using the ASM [30] mechanism that allows reading and writing of C variables using assembly code.
- By using the modified version of the LLVM compiler that has assembly support for stream and RVV instructions, compile the benchmark created, which will give the correspondent benchmark executable.
- With the help of the RISC-V Proxy Kernel tool, execute the benchmark under the Spike simulator. By doing so, the respective benchmark will be executed by the simulated functional model of the proposed stream-based extension.
- Finally, check the outputted results by executing the benchmark with the stream-based instructions and confirm if it corresponds to the expected results. If it is confirmed, it can be assumed that the correspondent instructions used in the benchmark were successfully simulated in the Spike environment.

In order to test that all the instructions present in the defined stream-based extension were successfully added, the denoted testing process was repeated to a lot of benchmarks. Figure 11 denotes all the benchmarks that were tested, the correspondent behaviour being tested and if we were able to obtain the expected results.

Benchmarks	Behaviour	Spike Support
SimpleStream-WordExe	Memory Copy to test the correct configuration of simple single-dimension streams with Word data width	✓
SimpleStream-HalfExe	Memory Copy to test the correct configuration of simple single-dimension streams with Half-Word data width	✓
SimpleStream-DoubleExe	Memory Copy to test the correct configuration of simple single-dimension streams with Double-Word data width	✓
SimpleStream-ByteExe	Memory Copy to test the correct configuration of simple single-dimension streams with Byte data width	✓
1DimStream-StrideExe	Memory Copy to test the correct configuration of single-dimension streams with stride different than one	✓
SignedAddExe	Signed addition to test the configuration and iteration of single-dimension streams	✓
FloatingPointAdd-Exe	Floating point addition to test the configuration and iteration of single-dimension streams	✓
2DimStreamExe	Memory Copy to test the correct configuration of simple 2-Dimensional streams	✓
2DimStreamRect-ScatterExe	Memory Copy to test the correct configuration of 2-Dimensional streams denoting the complex Rectangular-Scattered Pattern	✓
3DimStreamExe	Memory Copy to test the correct configuration of 3-Dimensional streams	✓
SimpleDimCouple-Exe	Inner loop processing with outer loop memory access to test Vector-Coupled Streaming	✓
LowerTriangular-Exe	Memory Copy to test the correct configuration of modifiers denoting the complex Lower Triangular Pattern	✓
ComplexDim-CoupleExe	Inner loop processing with outer loop memory access in conjunct with Lower Triangular access pattern to test Vector-Coupled Streaming with modifier	✓
3mm	3 Matrix Multiplications using streams	✓
covariance	Covariance Computation using streams	✓
jacobi-1d	1-D Jacobi stencil computation using streams	✓
jacobi-2d	2-D Jacobi stencil computation using streams	✓
gemm	Matrix-multiply C=alpha.A.B+beta.C using streams	✓
gemver	Vector Multiplication and Matrix Addition using streams	✓
mvt	Matrix Vector Product and Transpose using streams	✓
saxpy	Single Precision A times X plus Y using streams	✓
seidel-2d	2-D Seidel stencil computation using streams	✓
trisolv	Triangular solver using streams	✓

Fig. 11. Benchmarks Executed with the defined Stream-based Extension.

Spike is deterministic in its executions, meaning that either the code is successfully executed and the outputted results match the expected, or the code is unsuccessfully executed and the final results do not match the expected. This characteristic was rather useful to prove that the defined stream-based extension was successfully extended on the Spike Simulator. However, to test the actual performance enhancements brought by the defined stream-based extension, Spike does not give enough details to work around. As so, the only inherent way to prove the improvements in using the defined extension is

to analyse the reduction of instructions needed to execute certain benchmarks. As mentioned, the goal of the defined extension is to increase performance in memory access, by reducing the processor's workload associated with memory addressing/indexing. Therefore, a simple analysis of the number of instructions needed to attain the total number of memory addressing/indexing actions in a respective benchmark already provides enough conclusions to measure performance enhancement.

Figure 12 denotes the real number of instructions executed in Spike when executing two of the studied benchmarks. In this case, the numbers show the difference between executing the benchmarks using only the RVV extension and using the RVV extension plus the defined stream-based extension. The numbers represented make reference to an execution where the vector length was 8 and the number of elements to be processed was 256.

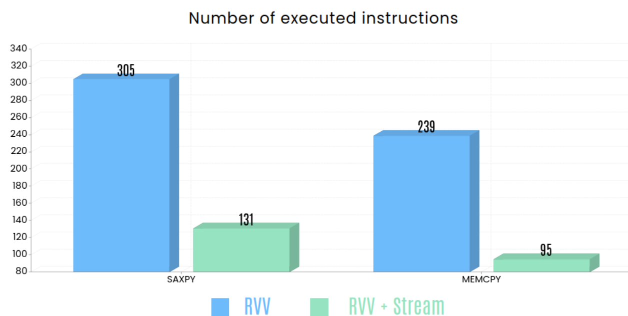

Fig. 12. RVV versus RVV + Stream.

Figure 13 denotes the real number of instructions executed in Spike when executing two of the studied benchmarks. In this case, the numbers show the difference between executing the benchmarks using only the scalar RISC-V instructions and using the scalar RISC-V instructions plus the defined stream-based extension. The numbers represented make reference to an execution where the number of elements to be processed was 64.

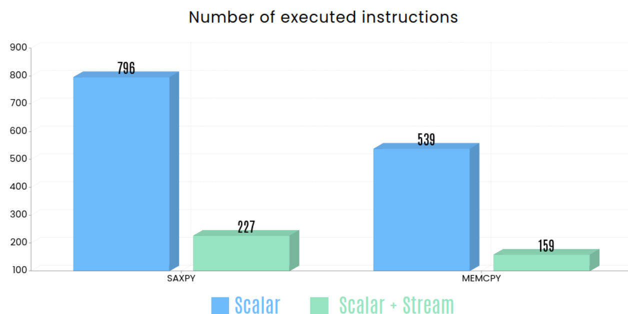

Fig. 13. Scalar versus Scalar + Stream.

Regarding the examples referenced, it is obvious that the defined stream-based extension delivers a real enhancement in terms of code efficiency, as the number of executed instructions in a correspondent benchmark is significantly reduced by the use of the defined extension. Moreover, it is also stated that with the increase in the number of elements to be processed, the delivered enhancement also increases. This statement is based on the fact that by using the stream-based extension, the instructions associated with memory addressing/indexing are successfully removed from the loop processing code.

VI. Conclusions and Future Work

A. Conclusions

Recently, the number of data streaming techniques implemented into general-purpose processors has increased considerably [16], [17]. This is due to the fact that the performance of memory devices has not improved at the same rate as that of processors, leading to an increase in the gap between Central Processing Unit (CPU) and memory speeds. 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 excellent opportunity to decouple the memory access procedures from the computational operations.

This work defines a new stream-based extension, whose goal is to increase performance in memory access. This extension counts with a set of specialized stream configuration and manipulation instructions. By decoupling the memory access from the core processing, the defined stream-based extension can remove the instructions associated with control and memory indexation.

Additionally, on the computational side, Single-Instruction-Multiple-Data (SIMD) extensions are becoming predominant in general-purpose processors [4], [5], [6], [7]. This is due to the proliferation of certain application domains, such as deep learning and artificial intelligence, with increasing computational performance demands, leaning on the need to further exploit Data-Level Parallelism (DLP). Furthermore, with the emergence of Vector-Length Agnostic (VLA) SIMD extensions [9], [8], each processor implementation can be tuned to achieve any desired SIMD behaviour. One example of these extensions is the recent RISC-V Vector (RVV) extension developed by the RISC-V organization.

The proposed stream-based extension is integrated with the subsets of the RISC-V Instruction Set Architecture (ISA), which means it can interact and interoperate with the RVV extension. By doing so, it is possible to combine the performance enhancement delivered by RVV, adding a way to decrease the memory access latency by using the defined stream-based extension. Furthermore, the defined stream-based extension can also cooperate with the scalar RISC-V instructions.

Attached with the RISC-V paradigm, Spike is a RISC-V ISA simulator, which implements a functional model of one or more RISC-V harts. Despite being recognized as the golden reference ISA simulator by RISC-V, Spike still does not support any extension that accommodates streaming techniques.

This work extended Spike with the defined stream-based extension. By doing so, the functional behaviour of the defined extension was validated.

B. Future Work

The defined stream-based extension and the correspondent extended support on the Spike simulator can already be used in the future to enhance certain applications' performance. However, due to the time constraints of the developed work, there were still some possible improvements that were not implemented and are kept for future work. Therefore, further improvements that can be developed into the base of the developed work are:

- To further improve the proposed stream-based extension, it is important to further tune essential characteristics such as the number of configurable descriptors and the available descriptors.
- To make the defined stream-based extension more reliable, it is important to create full compilation support for the extension.
- In the extended Spike Simulator, there were still a few instructions present in the defined extension that were not supported. One example is the support for the dynamic modifier descriptor. It is important to further extend Spike with such modifications to have a fully functional model of the defined extension.
- Spike is a deterministic tool, which does not provide enough details to measure the full enhancements delivery by the defined extension. It is important to further extend the defined extension in other simulation tools.

References

[1] G. E. Moore *et al.*, "Progress in digital integrated electronics," in *Electron devices meeting*, vol. 21. Washington, DC, 1975, pp. 11–13.

[2] R. H. Dennard, F. H. Gaensslen, H.-N. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of solid-state circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[3] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Communications of the ACM*, vol. 62, no. 2, pp. 48–60, 2019.

[4] A. Peleg and U. Weiser, "Mmx technology extension to the intel architecture," *IEEE micro*, vol. 16, no. 4, pp. 42–50, 1996.

[5] S. Thakkur and T. Huff, "Internet streaming simd extensions," *Computer*, vol. 32, no. 12, pp. 26–34, 1999.

[6] C. Lomont, "Introduction to intel advanced vector extensions," *Intel white paper*, vol. 23, 2011.

[7] ARM, "Neon programmer's guide."

[8] A. Waterman and K. Asanovic, ""risc-v "v" vector extension"," 2021.

[9] 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, 2017.

[10] N. C. Crago and S. J. Patel, "Outrider: Efficient memory latency tolerance with decoupled strands," in *Proceedings of the 38th annual international symposium on Computer architecture*, 2011, pp. 117–128.

[11] T. J. Ham, J. L. Aragón, and M. Martonosi, "Desc: Decoupled supply-compute communication management for heterogeneous architectures," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 191–203.

[12] 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.

[13] S. Ciricescu, R. Essick, B. Lucas, P. May, K. Moat, J. Norris, M. Schuette, and A. Saidi, "The reconfigurable streaming vector processor (rsvp/spl trade/)," in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* IEEE, 2003, pp. 141–150.

[14] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-dataflow acceleration," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 416–429.

[15] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," *ACM SIGARCH computer architecture news*, vol. 44, no. 3, pp. 367–379, 2016.

[16] Z. Wang and T. Nowatzki, "Stream-based memory access specialization for general purpose processors," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 736–749.

[17] F. Schuiki, F. Zaruba, T. Hoefler, and L. Benini, "Stream semantic registers: A lightweight risc-v isa extension achieving full compute utilization in single-issue cores," *IEEE Transactions on Computers*, vol. 70, no. 2, pp. 212–227, 2021.

[18] J. M. Domingos, N. Neves, N. Roma, and P. Tomás, "Unlimited vector extension with data streaming support," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA).* IEEE, 2021, pp. 209–222.

[19] R. Organization, "riscv-isa-sim," https://github.com/riscv-software-src/riscv-isa-sim, January 2023.

[20] I. Corporation, "Intel sse2 (streaming simd extensions 2)," Intel Corporation, Tech. Rep., 2001.

[21] ——, "Intel sse3 (streaming simd extensions 3)," Intel Corporation, Tech. Rep., 2004.

[22] ——, "Intel sse4 (streaming simd extensions 4)," Intel Corporation, Tech. Rep., 2007.

[23] 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.

[24] I. Corporation, "Intel avx2 (advanced vector extensions 2)," Intel Corporation, Tech. Rep., 2013.

[25] ——, "Intel avx-512 (advanced vector extensions 512)," Intel Corporation, Tech. Rep., 2016.

[26] A. Tanikawa, K. Yoshikawa, T. Okamoto, and K. Nitadori, "N-body simulation for self-gravitating collisional systems with a new simd instruction set extension to the x86 architecture, advanced vector extensions," *New Astronomy*, vol. 17, no. 2, pp. 82–92, 2012.

[27] D. Gregg and A. Forin, "Dynamically-sized vector operations on simd architectures," *ACM SIGARCH Computer Architecture News*, vol. 21, no. 1, pp. 267–275, 1993.

[28] A. Pohl, M. Greese, B. Cosenza, and B. Juurlink, "A performance analysis of vector length agnostic code," in *2019 International Conference on High Performance Computing & Simulation (HPCS).* IEEE, 2019, pp. 159–164.

[29] W. A. Wulf, "The memory wall," *ACM SIGARCH Computer Architecture News*, 1995.

[30] GNU, "Gcc assembler instructions with c expression operands," 2020.