TÉCNICO LISBOA

Data-Streaming Engine Architecture for the Unlimited Vector Extension

Xavier Batista Fernandes

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

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

Examination Committee

Chairperson: Prof. António Manuel Raminhos Cordeiro Grilo
Supervisor: Prof. Nuno Filipe Valentim Roma
Member of the Committee: Prof. Gabriel Falcão Paiva Fernandes

November 2023

Declaration

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

Acknowledgments

I am deeply grateful to my parents and family for granting me the incredible opportunity to pursue higher education and complete my master's degree. Your unwavering support, both emotionally and financially, has been the cornerstone of my academic journey. In moments of hardship, your motivation has kept me going, and in moments of triumph, your smiles shared my joy. Your belief in me has been my guiding light.

I extend my heartfelt gratitude to Nuno Roma, Pedro Tomás, and Nuno Neves, for your exceptional patience, invaluable guidance, and professionalism. Your mentorship and insightful advice were instrumental in shaping this thesis, and without you, it simply would not have been possible. I have learned immeasurable lessons from your expertise and dedication to my academic growth.

A special note of gratitude goes to the friends I had the privilege of meeting during my university years. Each one of you has played a significant role in making this chapter of my life rich with experiences and friendships that I will always carry with me. Thank you for being my pillars of strength and for making my university journey not just easier, but genuinely unforgettable and deeply meaningful.

I am profoundly thankful to each one of you for being a part of this significant journey.

This thesis was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT), under project 2022.06780.PTDC.

Abstract

The evolving demands in contemporary application domains, supported by General-Purpose Processors (GPPs), increasingly necessitate the adoption of specialized methodologies. While the primary objective is to enhance computational capabilities, concerns related to data access efficiency are also of growing importance, particularly in High-Performance Computing (HPC) applications. In these applications, consistent memory access patterns define many modern domains, setting the stage for data-streaming exploration. To address these challenges, a novel solution called Unlimited Vector Extension (UVE) has been recently introduced. UVE combines Single Instruction, Multiple Data (SIMD) extensions with a data streaming paradigm, necessitating the design of a dedicated Streaming Engine (SE). Its microarchitecture efficiently manages data flow, enabling data streaming within UVE-based processors. This thesis focuses on proposing and defining a comprehensive hardware description of the SE. Its microarchitecture comprises five interconnected modules and employs a descriptor hierarchy paradigm to generate various linear and non-linear memory access patterns. It autonomously communicates with memory to load and store data, maximizing data reuse to enhance data throughput. Experimental evaluations demonstrate the effectiveness of the proposed architecture in processing memory access patterns. It achieves an address generation rate of nearly one address per clock cycle for cache-friendly patterns, with data reutilization rates exceeding 11 times per memory load operation. Even with highly scattered patterns, the architecture maintains acceptable performance, generating new addresses every three cycles, with data reutilization rates surpassing 2 times.

Keywords

High-Performance Computing, Data-Streaming Computing, Processor Micro-architecture, Hardware Design.

Resumo

Os requisitos computacionais nos domínios de aplicação contemporâneos suportados por Processadores de Uso Geral (GPPs) estão permanentemente em crescimento e necessitam cada vez mais da adoção de metodologias inovadoras e especializadas. Muito embora o objetivo principal seja melhorar as capacidades de computação, as preocupações relacionadas com a eficiência no acesso aos dados estão também a tornar-se cada vez mais relevantes, sobretudo nas aplicações de Computação de Elevado Desempenho (HPC). Nestas aplicações, os padrões regulares de acesso à memória definem muitos domínios de aplicação modernos, e são particularmente adequados à exploração do paradigma de *streaming* ao nível dos dados. Para lidar com estes desafios, uma solução inovadora chamada de *Unlimited Vector Extension* (UVE) foi recentemente introduzida. A extensão UVE combina a paralelização ao nível dos dados do tipo *Single-Instruction Multiple-Data* (SIMD) com o conceito de *streaming*, requerendo a incorporação, na arquitectura do processador, de um módulo com a função de *Streaming Engine* (SE), dedicado a esta funcionalidade. A sua microarquitetura garante uma troca de dados de uma forma eficiente, permitindo o fluxo regular de dados entre a memória e os processadores baseados na UVE. Esta tese centra-se na proposta e definição de uma descrição detalhada do SE. A sua microarquitetura é composta por cinco módulos interligados e utiliza um paradigma de hierarquia de descritores para gerar vários padrões de acesso à memória, lineares e não lineares. Comunica autonomamente com a memória para ler e escrever dados, maximizando a reutilização de dados para aumentar o débito dos mesmos. A avaliação experimental realizada demonstra um elevado nível de eficiência da arquitetura proposta no processamento de padrões de acesso à memória. Em particular, quando o padrão de acessos é regular, alcança uma taxa de geração de endereços de quase um endereço por ciclo de relógio, com taxas de reutilização de dados superiores a 11 vezes. Com padrões irregulares e dispersos, a arquitetura mantém um desempenho aceitável, gerando um novo endereço a cada três ciclos, com taxas de reutilização de dados superiores a 2 vezes.

Palavras Chave

Computação de Alto Desempenho, Computação em Fluxo de Dados, Microarquitetura de Processador, Descrição de Hardware.

Contents

List of Figures

List of Tables

Acronyms

AAGR	Average Address Generation Rate
AGEDIR	Average Address Efficiency with Descriptor Iteration Ratio
AGU	Address Generation Unit
ALU	Arithmetic and Logic Unit
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
AVX	Advanced Vector eXtension
AXI4	Advanced eXtensible Interface 4
CPU	Central Processing Unit
CSR	Control and Status Register
DLP	Data-Level Parallelism
DM	Data Memory
DRR	Data Re-utilization Rate
DSA	Domain-Specific Accelerators
FIFO	First-In First-Out
FPGA	Field Programmable Gate Array
FPU	Floating-Point Unit
FSM	Finite State Machine
FU	Functional Unit
GPP	General-Purpose Processor
GPU	Graphics Processing Unit
HDL	Hardware Description Language

HPC	High-Performance Computing
IM	Instruction Memory
ISA	Instruction Set Architecture
ISSR	Indirection Stream Semantic Register
LMMU	Load Memory Management Unit
LLB	Load Line Buffer
LRQ	Load Request Queue
LFIFO	Load First-In First-Out
MMU	Memory Management Unit
OoO	Out-of-Order
PC	Program Counter
RISC	Reduced Instruction Set Computer
RF	Register File
RTL	Register Transfer Level
RVV	RISC-V 'V' Vector Extension
SAXPY	Single Precision A*X Plus Y
SC	Stream Configuration
SE	Streaming Engine
SI	Stream Iterator
SIMD	Single Instruction Multiple Data
SMMU	Store Memory Management Unit
SP	Streaming Processor
SS	Stream State
SSE	Streaming SIMD Extensions
SSR	Stream Semantic Registers
SVE	Scalable Vector Extension
TDP	Thermal Dissipation Power
UVE	Unlimited Vector Extension
VLA	Vector-Length Agnostic

1

Introduction

Contents

1.1 Motivation

With the breakdown of Moore's law, computer systems began shifting to a multi-core approach to increase their processing capabilities. However, with the end of Dennard's scaling, the power consumption of each core increases nearly at the same rate as they are added to a chip, which ultimately limits the clock frequency and forces cores to be deactivated when idle, thus resulting in systems that are limited by Thermal Dissipation Power (TDP).

Moreover, recent advances in prominent application domains such as artificial intelligence, machine and deep learning, and image and video processing are increasingly requiring computational resources and more advanced capabilities [4, 5]. The workloads of these domains place additional strain on the system by aggravating the Von Neumann bottleneck. The large amounts of data required in these application domains exacerbate the efficiency gap between the Central Processing Unit (CPU)'s processing capabilities and the rate at which data is communicated. Additionally, this bottleneck is often aggravated by wasted data transfers on strided and scatter-gather memory accesses, which occur due to the division of data transfers into fixed-size blocks determined by the cache structure. These issues are further limiting the effectiveness of multi-core approaches for General-Purpose Processor (GPP).

Common methodologies adopted to reduce the impact of memory access latency and low throughput include the detection of memory access patterns of applications followed by prefetching the corresponding data before it is requested. Several hardware and software prefetchers have been developed in the past years. However, although these have increased the efficiency of data access, bad timeliness of these prefetchers can lead to cache pollution. Another approach is data streaming, which takes advantage of well-defined memory access patterns, which are often associated with regular code structures. This deterministic pattern implies that computation and memory access components of an application can be specified independently, thus permitting the decoupling of memory access from the CPU through streaming functional units that hide the data transfers overhead by using buffering memories.

Moreover, these workloads are often highly parallelizable, as they often involve matrix and array-like computations on large datasets. Having this in mind, parallel execution techniques like Single Instruction Multiple Data (SIMD) can be used to increase the computational efficiency of applications, but even then computational bottlenecks may still be an issue. One potential solution is to design dedicated accelerators or Application-Specific Integrated Circuits (ASICs), but these approaches have limitations due to being inflexible and unable to adapt to different workloads, making them inefficient for supporting a wide range of applications. Consequently, these are solutions useful for specific, targeted applications, but may not be practical for more general-purpose computing needs.

An alternative approach is to create accelerators that have a processor core and specialized, closely coupled hardware modules that are controlled through custom Instruction Set Architecture (ISA) extensions [1, 3, 6, 7]. These modules, which may or may not be tailored to a specific application, become an

integral part of the processor's datapath. This goal can be more easily achieved by using a Reduced Instruction Set Computer (RISC) approach, due to its minimal instruction set, which provides a base ISA suitable for microcontrollers. An example is RISC-V, an open-source ISA that allows designers to adapt to a variety of application and processor domains. Moreover, RISC-V-based systems are highly customizable as more advanced features in architectures can be employed through the use of extensions provided by the ISA on top of its base one.

In response to the evolving demands of contemporary application domains, GPPs must embrace specialization methodologies to enhance computation capabilities while simultaneously optimizing data access efficiency, especially in the context of High-Performance Computing (HPC) general-purpose applications. An illustrative example is Unlimited Vector Extension (UVE), which combines SIMD extensions with data streaming. To support data streaming, a dedicated micro-architecture must be developed, specifically requiring an engine to manage the data flow.

This dissertation aims to investigate the infrastructure requirements of UVE to design an efficient hardware description for this engine.

1.2 Objectives & Contributions

This work is motivated by the challenges presented in modern computational systems concerning performance and energy efficiency, particularly in the domains of data access and computational capabilities. While the solutions employed in this context represent significant advancements, there is still substantial room for improvement. The primary objectives are as follows:

1. To investigate the latest data streaming methodologies utilized in contemporary designs, assessing their performance, and evaluating their impact on overall system efficiency. We also aim to identify and understand their current limitations and bottlenecks.

2. To develop a hardware implementation of the micro-architecture support used by UVE for the efficient utilization of data streaming mechanisms. In doing so, one aspires to translate the functionalities it offers into an Register Transfer Level (RTL) hardware description that supports integration into more complex systems.

By comprehending the existing limitations of state-of-the-art data streaming implementations and the requirements of UVE's micro-architecture, one can plan and design a solution that tackles both problems. These objectives culminate in the following contributions of this thesis:

- **Hardware Design Implementation**: This work involves creating hardware designs for the data streaming infrastructure, drawing inspiration from cutting-edge data streaming extensions, such

as UVE. A central aspect is the development of a configurable engine module with numerous adjustable parameters, capable of generating addresses and performing memory operations. Additionally, a minimal processing core is designed to evaluate this engine's performance in data streaming applications.

- **Efficiency Evaluation with Testbenches**: Comprehensive testbenches are developed and used to evaluate the efficiency and performance of the hardware modifications and designs. The process involves data collection and assessment of various performance metrics. Additionally, the work delves into understanding the limitations of the developed hardware description.

1.3 Outline

This document is structured as follows. In Chapter 2, fundamental concepts are introduced, serving as a foundation for the thesis's context. Topics covered include data parallelism and data streaming, with a focus on a work that combines both research areas. Chapter 3 delves into research domains aiming to improve data transfer bandwidth between memory and processing units. This includes discussions on scalable vectorization, data prefetching, data streaming architectures, and memory access decoupling. Furthermore, it highlights state-of-the-art designs, with a specific emphasis on data streaming implementations within general-purpose processors. Chapter 4 offers a detailed explanation of the hardware designed, along with insights into its features and the decisions made during its development. This chapter forms the core focus of this work. Chapter 5 is dedicated to assessing how the designed hardware integrates with other components and how testing is carried out. Here, the proposed engine module is integrated within a small processing core with minimal functionality to assess the efficiency of the design. Chapter 6 aims to explore the efficiency and limitations of the developed work. The hardware is subjected to various testbenches with different parameterizations, to extract performance-related insights. The chapter presents the results and engages in discussions regarding their implications. Finally, Chapter 7 concludes this document by offering a reflective section on the work undertaken, followed by an analysis of potential future directions within the thesis's context.

2

Background on Data-Streaming

Contents

This chapter serves as the foundational backdrop for the topics explored in this dissertation. In the realm of HPC research, Data-Level Parallelism (DLP) is a widely examined area. This chapter elucidates how DLP methodologies are commonly harnessed within processing units.

Moreover, the chapter delves into the essential domain of data streaming within applications, shedding light on the identification and exploration of memory access patterns. It introduces the most prevalent memory access patterns and presents a descriptor paradigm for their characterization.

To conclude, this chapter converges towards UVE, providing a comprehensive understanding of its composition, features, and functionalities.

2.1 Exploring Data Level Parallelism

Most modern processors try to explore DLP on several application domains through SIMD operations. By definition, it is a type of parallel processing where the same operation is simultaneously performed on different pieces of data. Its use is often associated with operations that are applied to different elements of an array, within a computational loop, which often incurs several operation repetitions. This processing method is possible in architectures with wide Arithmetic and Logic Units (ALUs), where data elements can be simultaneously processed. Naturally, since more processing is being performed at the same time, the throughput generally increases when using such methods.

However, SIMD has some limitations and is not always supported by the application. In particular, data dependencies between operations must be avoided, as parallel computation can produce incorrect results in these cases. Additionally, the potential for data-level parallelism may be limited by memory access bandwidth bottlenecks, particularly when there are complex memory patterns or conditional execution.

Modern ISAs already provide vector extensions that exploit the SIMD paradigm, such as the ARM NEON vector extension [8] or Intel's x86 vector extensions, such as Streaming SIMD Extensions (SSE) [9] and Advanced Vector eXtension (AVX) [10]. These extensions include special vector registers and SIMD operations are performed by distributing its elements through the functional units of the CPU, which are then executed in parallel.

The ARM RISC-based ISA provides the NEON [8] extension. It is an extension that adds 32 registers that can be accessed either as 64-bit or 128-bit registers. The NEON vector extension provides SIMD operations such as data processing (additions, multiplications, among others), memory accesses (load, store), data moving between vector register and scalar registers, and datatype conversion (width extension, integer to floating-point conversion). These instructions operate on vectors that support 8-bit, 16-bit, 32-bit, and 64-bit signed and unsigned integer elements, and 32-bit single-precision and 16-bit half-precision when the elements are in floating-point format. However, it does not support scatter/-

gather memory access patterns and does not have any predicate mechanisms to deactivate operating in certain elements.

In the x86 ISA, the SSE and AVX vector extensions were developed, and many versions were developed over time. The AVX-512 [10] extension adds 32 512-bit vector registers. It also adds 8 mask registers, which may be used to restrict operations to specific parts of a vector register, opening possibilities for conditional execution and processing gather/scatter patterns. Intel's AVX-512 has many variations which determine the element types supported.

One concern with these ISA extensions is that they are often associated with a fixed vector length. This is problematic because execution binaries are vector-length specific, and thus creating new extensions for larger vector lengths requires generating and compiling new code, meaning binaries aren't reusable across architectures with different characteristics. To overcome this, state-of-the-art scalable vector extensions were designed to diminish application incompatibilities, and will be explained in Chapter 3.

2.2 Data Streaming - Descriptor Paradigm

To incorporate streaming mechanisms in computing systems, it is crucial to define methods for representing memory access patterns. While traditional ISAs uses standard instructions to describe memory accesses, alternative solutions with lower overhead are emerging, enabling more efficient implementations of access patterns and effective memory optimization.

Efficiently characterizing memory access patterns is essential, especially for complex but regular patterns. Research on memory access patterns demonstrates that the use of memory access descriptors improves memory transfers by aligning data movement with hardware capabilities, yielding significant performance gains [11]. These benefits also extend to Graphics Processing Units (GPUs), where memory access pattern analysis enhances performance. This highlights the importance of efficient memory access pattern descriptors.

2.2.1 Memory Access Patterns

A memory access pattern refers to a specific sequence in which a program accesses and manipulates data stored in memory. This pattern is unique to each application and significantly influences a processor's efficiency in accessing data. The efficiency of this procedure is contingent on both the processor's architecture and the characteristics of the access pattern itself.

For instance, a memory access pattern characterized by large and sparse data access (resulting in reduced data locality) can lead to an increased number of cache misses. This is particularly true

when the cache capacity is limited. These cache misses, in turn, result in larger latency of the memory requests, forcing the processor to stall while awaiting the required data. Consequently, the overall performance of the system is adversely affected.

Memory access patterns can generally be categorized into three main types:

1. **Affine Data Patterns**: These patterns often arise in the context of nested loops, where each element of an affine sequence of addresses corresponds to a linear combination of loop iterator variables. This linear combination yields an accumulated offset added to the base offset of the pattern. Affine patterns can be one-dimensional or they can extend to multiple dimensions, involving more complex patterns.

2. **Indirect Data Patterns**: In contrast to affine patterns, indirect data patterns are generated not through direct loop iterator variables but via an offset stored in another variable. This offset is used to index the memory access. Indirect patterns can involve nested indirection or modifications to the stride field as the iterator variable progresses. These patterns are irregular and unknown before runtime but are well-defined due to the existence of an implicit affine pattern used to access the indirection variable.

3. **Pointer-chasing Dependencies**: Pointer-chasing patterns represent highly dependent sequences of memory accesses, where each access depends on the outcome of the preceding one. These patterns are commonly associated with traversing complex data structures such as linked lists, trees, or graphs, where parallelization poses significant challenges.

Understanding memory access patterns is vital for optimizing memory performance and overall system efficiency. These patterns are well-defined, allowing for the separation of the memory request from the processing part of the application. This separation enables the pro-active loading of pattern elements, effectively streaming the required data into the core. Memory access pattern descriptors are introduced as a structured methodology for articulating and analyzing these patterns, while also enabling compilers to organize memory transfers more efficiently [12].

2.2.2 Descriptors

A memory pattern descriptor can be defined as a tuple of parameters whose values parameterize the characteristics of the memory access pattern it is describing.

Although there are many other descriptor representations, the solution developed by Neves et al [13] allows the representation of a large number of access sequences in workloads. According to their work, the memory access modeling follows the affine function below:

$$\mathrm{y}(X) = \mathrm{y}_{\texttt{base}} + \sum_{k=0}^{dim_{\mathrm{y}}} x_k \times S_k \;\; \textbf{with} \;\; x_k \in \{x_0, \ldots, x_{dim_{\mathrm{y}}}\} \;\; \textbf{and} \;\; x_k \in [O_k, E_k + O_k] \tag{2.1}$$

According to this equation, each element of a memory access pattern $\mathrm{y}(X)$ can be computed by considering dim_{y} pairs of indexing variables (x_k) and stride multiplication factors (S_k) added to the base address of a multi-dimensional variable ($\mathrm{y}_{\texttt{base}}$). For this reason, to represent linear and affine patterns, they propose the use of a hierarchy of descriptors where each descriptor is associated with a single dimension.

Dimensional Descriptors

Each dimensional descriptor is associated with a given dimension (k) and holds three fields: `offset` (O_k), which indicates the base address of the dimension it is attached to; `stride` (S_k), for the interval between two consecutive accesses; and `size` (E_k) for the number of elements in that dimension. Additionally, the data type needs to be known for memory alignment coherence between memory accesses. Hence, a single-dimensional descriptor is capable of describing a linear, strided or not, memory access pattern.

Naturally, `offset` and `size` parameters must be positive, but negative `stride` fields can be configured to configure certain specific patterns. Figure 2.1 illustrates some examples of patterns that can be defined with a hierarchy of dimensional-only descriptors.

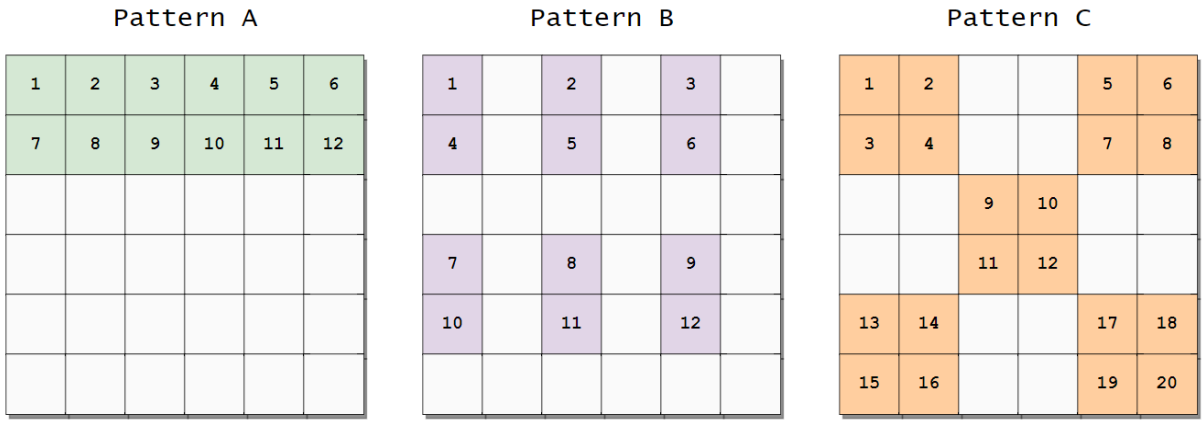

Figure 2.1: Memory Access Patterns employing only Dimensional Descriptors

Utilizing the notation {`Offset, Stride, Size`} for dimension descriptors, and considering that consecutive rows in matrices represent contiguous memory addresses with a base address of zero, it is possible to represent the following memory access patterns:

- **Pattern A -** Can be represented through {0, 1, 12} as its only descriptor, and therefore is a pattern with 12 elements where all elements are contiguous in memory, due to stride being unitary.

- **Pattern B -** Requires two descriptors: {0, 2, 6} for the first dimension and {0, 18, 2} for the second (outer) dimension. The first descriptor introduces a sequence of 6 memory accesses where the elements are separated with a stride of 2. The second descriptor, when it is iterated, accumulates a stride of 18 to the base address. Since it has 2 iterations, the pattern of accesses described by the first descriptor occurs twice.

- **Pattern C -** Uses three descriptors: {0, 1, 2} for the first dimension and {0, 6, 2} for the second and {0, 4, 5} for the third and most outer dimension. The first descriptor describes two consecutive accesses. The second descriptor introduces the block access behavior, by repeating the first descriptor once. The third descriptor is introduced to repeat the block accesses a total of 5 times.

One thing to note is that each descriptor contributes a unique behavior to the memory access pattern. This is noticeable as there are groups of access patterns that appear repeated along the pattern. Attempting to describe the same pattern using fewer descriptors would be impossible. For example, Pattern B, which is characterized by a large gap between some of its accesses, could not be described without the inclusion of a second descriptor. In an analogy to programming loops, a dimension descriptor is a loop that iteratively applies its inner descriptors, mirroring the way loops repeat their actions each time they iterate.

However, one can think of more complex patterns where the use of simple dimensional descriptors would not be sufficient.

Static Modifiers Descriptors

Non-linear modifications to the dimension descriptors can be applied by configuring a different type of descriptor, the static modifiers. These have four fields: `target`, to indicate the parameter of the dimensional descriptor getting modified; `behavior`, to indicate whether the target is getting incremented or decremented; `displacement`, which indicates the amount the target parameter is getting modified; and `size`, which indicates how many times the parameter should get modified.

Each static modifier descriptor is also associated with a given dimension and is implicitly coupled to the dimensional descriptor of the inner dimension, modifying one of its parameters. Although a modifier can only modify one field, it is possible to configure for a particular dimension up to three modifiers, one for each field it has. The introduction of these descriptors extends even further the amount of patterns that can be configured. Two different pattern examples are displayed in Figure 2.2.

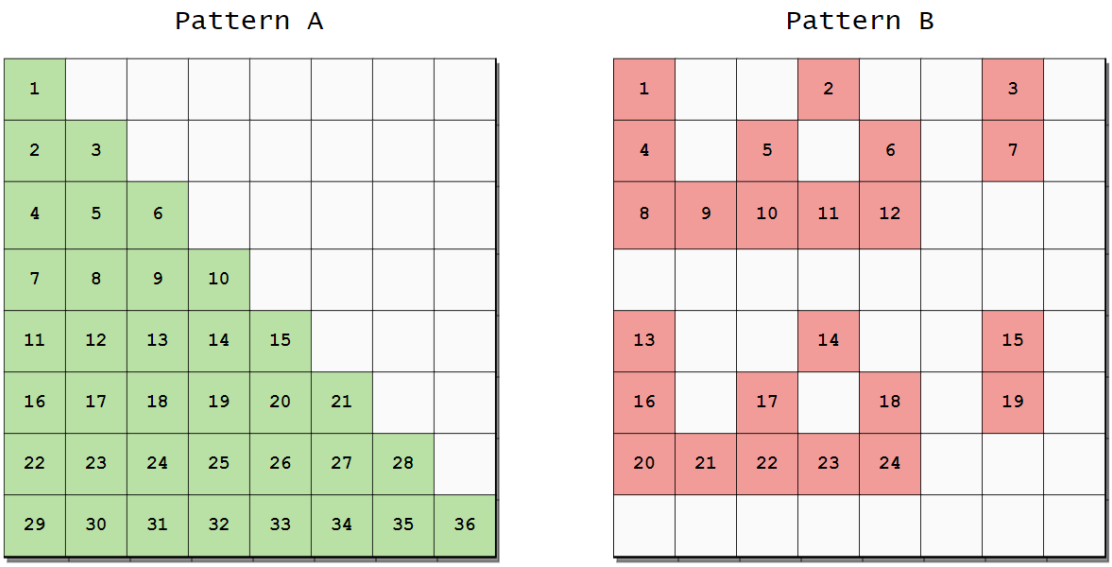

Figure 2.2: Memory Access Patterns employing Static Modifier Descriptors

Utilizing the notation {`Offset, Stride, Size`} for dimension descriptors and {`Target, Behaviour, Displacement, Size`} for static modifier descriptors, and considering that consecutive rows in matrices represent contiguous memory addresses with a base address of zero, it is possible to represent the following memory access patterns:

- **Pattern A -** Characterizes a triangular access pattern, and it can be represented with a total of three descriptors: two-dimensional descriptors and one static modifier descriptor. The first dimension descriptor is {`0, 1, 1`}, and for the second dimension, there is a dimensional descriptor {`0, 8, 8`} along with a static modifier {`size, add, 1, 8`}. The static modifier descriptor plays a unique role in enhancing the pattern representation. Specifically, it increments the size of the first-dimensional descriptor, ensuring that each iteration of the second dimension results in an increased number of iterations within the lower dimension.

- **Pattern B -** A more complex, requiring a total of three dimensions. The most inner and first dimension is described by the dimensional descriptor {`0, 2, 3`}. The second dimension uses the descriptor {`0, 8, 3`} and two modifiers: {`size, add, 1, 3`} and {`stride, sub, 1, 3`}. The third and most outer dimension is characterized by the descriptor {`0, 32, 2`}, without any modifiers. The descriptors configured present a more intricate pattern characterized by multiple adjustments in descriptor fields of the first dimension. The base descriptor, which corresponds to the first dimension, serves as a reference point, with its size and stride fields subject to alteration during the progression of the second dimension's iterations. Notably, each iteration of the second dimension

triggers changes in the base descriptor. As it advances, the size of the first dimension increases, while the stride between elements diminishes. Furthermore, the pattern has a third dimension that, as it undergoes iterations, repeats the sub-pattern defined by the adjustments in the first two dimensions.

Incorporating static modifier descriptors into the representation of a memory access pattern enables the specification of more intricate patterns with non-linear behavior without the requirement for an excessive number of simpler descriptors.

Multi-Descriptor Patterns - Iteration Process

In the context of the used descriptor paradigm, it is essential to iterate through all the descriptors in a precise order. Moreover, given the existence of a hierarchical structure comprising diverse descriptor types, distinct operations have to be performed during the pattern processing.

These descriptors are interconnected, following specific rules that govern their connections and the impact of each on the parameters of others. In essence, a configuration of multiple descriptors can be likened to nested for loops, with the innermost loops corresponding to the lowest dimensions and the outer loops representing the higher dimensions within the descriptor hierarchy. Much like how nested loops operate, when an inner loop completes its iterations, the next iteration takes place in an outer loop, and this cyclic process continues. Hence, when the lowest dimension completes its iterations, the next dimension to be iterated is the first one that still has remaining iterations, considering the prescribed order of the descriptors. For instance, the third dimension of a given memory access pattern will only be iterated if the second dimension has completed all its iterations.

Iterating a dimension entails more than simply iterating the dimensional descriptor itself. It also involves iterating all the associated modifiers configured for that dimension. When modifiers are employed in a pattern, the dimension they are associated with should be the one right above the dimension that is targeted to be modified. This is practical in the sense that the modifications occur only when the above dimension is iterated. For this reason, the root dimension (the first or innermost dimension) does not have modifier descriptors.

As depicted in Figure 2.3, the schemes illustrate which descriptor patterns are permitted within the described paradigm.

Within the allowed configurations, only Configurations A and B are considered valid. Configuration C exhibits descriptor relationships that deviate from the paradigm's guidelines, rendering it invalid for two distinct reasons. Firstly, it is invalid because a modifier cannot be configured in the first dimension; modifiers should interact with the dimension below, making their configuration in the first dimension nonsensical. Secondly, a modifier cannot exist in a dimension without the corresponding dimensional

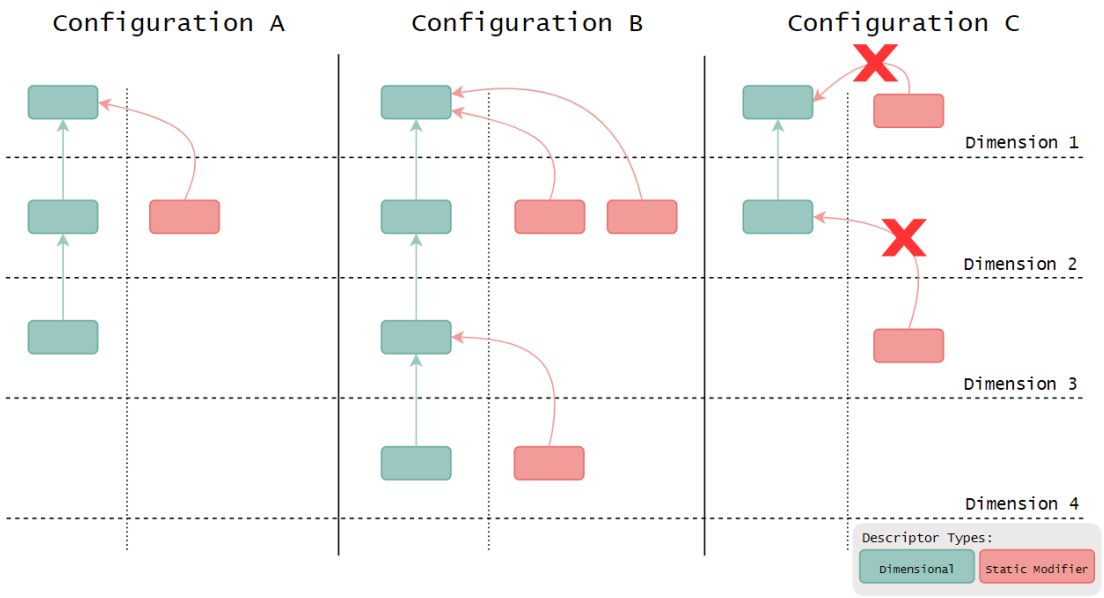

Figure 2.3: Examples of possible Descriptor Configurations

descriptor. The first two configurations satisfy these conditions, with Configuration B demonstrating that it is feasible to define multiple modifiers within the same dimension.

Throughout the iteration of the memory access pattern, all dimensions simultaneously accumulate an offset value. These accumulated offsets correspond to each dimension's contribution as the pattern is processed, reflecting the intricate interplay of descriptors and modifiers in shaping the memory access pattern's behavior.

An important rule to note is that when one dimension completes its iterations, any changes induced by the modifiers are reset, ensuring a clean slate for the next dimension.

2.3 Unlimited Vector Extension

Domingos et al. [1] have pointed out significant limitations in state-of-the-art scalable vector extensions, namely Scalable Vector Extension (SVE) [14] and RISC-V 'V' Vector Extension (RVV) [15]. These limitations arise from the substantial overhead that is required to implement the processing of essential data elements. This overhead constitutes a significant portion of the application loop code, diminishing processing throughput and leading to inefficient utilization of processor resources.

Their proposal solution introduces a novel Vector-Length Agnostic (VLA) extension employing a hierarchical set of descriptors, similar to those detailed in Section 2.2.2, to specify memory access patterns. By combining this descriptor-based approach with streaming methodologies, they effectively decouple memory access operations from the processing core.

The proposed extension introduces some micro-architectural adjustments in the processor. These adjustments include the incorporation of new stream registers tailored for stream processing and the introduction of instructions capable of configuring and managing data streams. Each stream configured will be associated with one of the new stream registers. The required micro-architectural support for this extension is realized through core adaptations and the creation of a dedicated Streaming Engine (SE) equipped with streaming buffers to facilitate the ISA extension.

2.3.1 Architectural State & Micro-architecture Support

Architectural State - Registers

The UVE extension, being a vector extension, necessitates the inclusion of 32 vector registers. Each of these vector registers is divided into multiple vector elements, which can be configured to be one of the elementary data types: byte (8 bits), half-word (16 bits), word (32 bits), and double-word (64 bits). This configuration sets the minimum register length to 64 bits.

Considering the scalable nature of the extension and the potential existence of edge cases or stream terminations, not all elements within the registers may be valid. To address this, meta-information is added to each register. This includes a `vector width` field, indicating the bit-width of each vector register, and a `valid index` field that maps the extent of valid elements. This mapping is possible because all elements are contiguous within the registers.

Additionally, the extension introduces 16 predicate registers, enabling control over lane execution. However, 8 of these registers are allocated for memory and arithmetic instructions, while the remaining registers can be used to configure the first 8 or support context saving. These predicate registers work as masks, determining which lanes of the vector registers should be processed.

Naturally, each stream is associated with a specific vector register. Thus, a stream interface is designed to ensure that when an instruction reads from or writes to these registers, it transparently consumes from or produces data to the corresponding stream.

Micro-architecture Support - Streaming Engine

The implementation of UVE necessitates adjustments to the processing core. In their work, Domingos et al. [1] introduced architectural modifications in an out-of-order processor. These modifications not only entail changes within the core but also entail the creation of a dedicated SE and associated buffering queues. The core underwent minor modifications in the decoder, register file, and execution units. Furthermore, a stream rename unit design was introduced to enable speculative stream configuration. Given the allowance for speculative stream operations, the SE can be signaled to commit and, if necessary, terminate any stream-related operations.

Figure 2.4 provides a visual representation of Domingos et al.'s [1] proposed SE, which serves as the base design for this work.

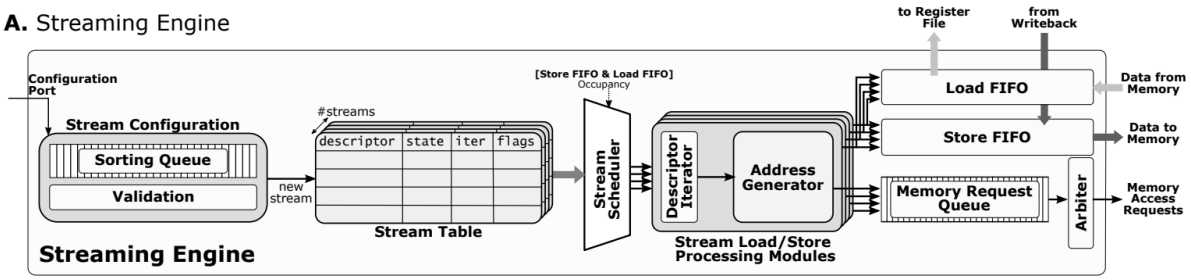

Figure 2.4: Streaming Engine proposed in Unlimited Vector Extension [1]

The core functionality of the SE can be described as follows. The SE enables the configuration of new data streams through a configuration port linked to the processor core. As the core decodes instructions related to stream configuration, it must temporarily store the corresponding descriptor parameters, given that stream configuration typically involves multiple instructions. A stream configuration unit is responsible for holding all incomplete configurations while also validating them. Once a configuration is finalized, the SE can allocate the required resources and initiate the processing. This entails maintaining an architectural internal state concerning streams. One approach is to establish a dedicated stream table for each stream, containing not only various descriptor parameters but also the iteration values for each, along with pertinent control and meta-information.

Stream processing involves the gradual generation of access pattern addresses. Consequently, the corresponding streaming processing modules must also be implemented. These modules can vary in design, but they must be capable of iterating through the descriptors and generating addresses accordingly. This is the pattern processing process of the SE, which needs Finite State Machines (FSMs) to correctly manage address generation.

If the SE permits more stream tables than the available stream processing modules, a stream scheduler is required to periodically select streams for processing. While a simple round-robin approach suffices for this selection, more advanced strategies may be adopted.

Address generation is identical for both load and store streams. However, load and store buffering queues function differently. In the case of load streams, as the SE generates addresses, it forwards requests to the memory system. The resulting data is loaded into the load queue associated with the corresponding stream. For store streams, the SE needs both an address and data to issue a store request to the memory system. This necessitates an address queue to temporarily store the results of pattern processing for that stream, while the core has yet to write the data into the store queue.

2.3.2 Instruction Set Architecture

The UVE architecture incorporates a diverse set of new instructions [1], encompassing 26 integer, 15 floating-point, and 19 memory-related instructions, including streaming instructions. These instructions can be categorized as follows:

- **Stream Configuration**: These instructions enable users to configure the streams by encoding the respective descriptors. Simple one-dimensional patterns can be configured using a single instruction, such as `ss.ld.{b|h|w|d}` for load streams or `ss.st.{b|h|w|d}` for store streams. To specify more intricate patterns, a hierarchy of descriptors must be defined. The initial dimension is defined with a `st.sta` operation, and additional descriptors are appended using `st.app`. If a descriptor serves as a modifier, the keywords `mod` or `ind` must be included. To conclude a configuration, a `st.end` instruction is used.
- **Stream Control**: While a stream is actively processing, it can be suspended via `ss.suspend` instructions. To resume execution, `ss.resume` instructions are employed. UVE also provides `ss.stop` instructions to prematurely terminate execution.
- **Advanced Control**: Instructions like `ss.getvl` and `ss.setvl` offer control over the vector length of each stream, while `ss.cfg.mem`**x** grants a stream access to the $L_{\mathbf{x}}$ cache level.
- **Loop Control**: The architecture offers two types of loop control instructions in the form of conditional branches. Instructions like `so.b.[n]c` and `so.b.[n]dc` initiate branch operations based on whether the stream or a stream descriptor specified by a register has (or has not - `[n]`) completed all its iterations, with optional negation variants.

In addition to these categories, there are instructions for **vector manipulation**, such as shift operations, and **scalar processing**, which are relevant for scalar-to-vector or vector-to-scalar operations.

More recently, Neves et al. [16] introduced a compilation workflow designed to generate code for UVE. This process involves the automatic extraction and encoding of memory access patterns and computation data-flow graphs, utilizing streaming representations.

2.4 Summary

This chapter begins by discussing DLP and the utilization of SIMD operations in modern processors. SIMD is a processing technique that enhances application performance by executing the same instruction on multiple data elements in parallel. However, SIMD processing can suffer from inefficiency due to bandwidth bottlenecks. Some contemporary ISAs include ARM NEON, Intel AVX, and SSE, but they also have limitations, particularly their fixed vector length, which makes them architecture-specific.

The chapter then introduces data streaming concepts, specifically explaining various memory access patterns. These patterns can be as simple as one-dimensional affine memory patterns or more complex, indirect, or pointer-chasing ones. When an application's memory addresses can be determined during compile time, it opens the possibility of streaming data to the processing core. This necessitates a methodology or paradigm to describe the pattern, often accomplished through descriptors. Various types of descriptors, when organized hierarchically, enable the description of a wide range of access patterns.

Finally, the UVE is introduced as a VLA extension that exploits data streaming mechanisms within processing cores. It provides an interface in its ISA for configuring, manipulating, and performing operations between streams. The employed streaming mechanisms relieve the processing core of address calculation-related workloads while increasing the bandwidth of data transfers. Implementation requires the addition of vector and predicate registers to handle vector processing, with data streaming features primarily managed by the introduction of a SE in the core's datapath.

3

Related Work

Contents

This chapter explores research areas in computer architecture that address inefficiencies in memory access and enhancements in computational power. It focuses on existing methodologies for improving processor architecture performance, particularly in terms of data processing throughput and efficiency.

Given the close alignment of this thesis with UVE, the chapter extensively references topics related to this extension. For instance, as a vector extension, it discusses vector-length agnostic extensions, emphasizing their impact on the computational aspect of the extension. In the context of memory, the extension incorporates mechanisms for proactive data retrieval, leading to discussions on data prefetching and memory access decoupling. These mechanisms collectively facilitate data streaming. Furthermore, the chapter evaluates state-of-the-art streaming mechanisms designed to enhance overall performance.

3.1 Vector-Length Agnostic SIMD

To exploit DLP, computer architectures have been extended to perform SIMD operations. Traditionally, SIMD extensions existing in modern ISAs, such as SSE and AVX in Intel x86 and NEON in ARM, assume a fixed vector length that must be known during compilation. This imposes limitations with code reuse across platforms with different characteristics, since modifying the length of the vector registers explicitly requires changing the code. To overcome this issue, VLA vectorization has been recently proposed, whose principle is for vector length to be determined during execution time. In principle, this allows the same code to be executed in architectures with different SIMD capabilities in terms of vector length.

One of these extensions, the ARM SVE [14], provides flexibility in the length of vector registers, which varies between 128 and 2048 bits in multiples of 128 bits. This extension does not require additional vector registers, but instead extends on top of the 32 registers provided by the NEON vector extension. Nonetheless, it adds 16 predicate registers which are used to control the execution of different lanes operating in vector elements. In particular, it relies on loop predication to enable vector length scalability, meaning that at each loop iteration, it verifies if the elements that are going to be processed are out-of-bounds. Through predicated instructions, it can eliminate loop tails.

Another example is the RVV extension [15], which introduces 32 vector registers and inherits the integer and floating point data types of the base RISC-V ISA [17]. This extension requires the specification of the vector size through the definition of several elements and the size of each element. This definition results in a vector-length value, which is compared with the implemented vector length, and the minimum between the two is chosen to configure all the vectors with that length. Having this comparison allows the same code to be compatible between different architectures.

Finally, it is also worth mentioning the UVE [1], which takes advantage of data-streaming mechanisms to reduce the loop control instructions overhead, while also improving vectorization opportunities. The streaming of data allows for only the necessary elements to be loaded into vector registers, thus

transparently making all accesses seem contiguous from the core's perspective. Similarly to SVE, it allows register predication to deactivate the lanes that are out-of-bounds, making the control logic simpler. This extension will be explained in greater detail further in this chapter.

3.2 Data Prefetching

One of the most prominent dedicated techniques to mitigate the inefficiency of memory access latency involves prefetching data to the cache before it is requested by the processor and both hardware and software approaches have been considered. Hardware prefetchers monitor the cache of the CPU, by tracking the history of cache misses to detect the memory access pattern [18–21]. Software prefetchers focus on identifying and inferring the memory accesses of the application by analyzing the code structure. This is possible by upgrading and developing compiler tools to detect memory patterns and change the code accordingly to minimize the memory access overheads [22–24].

However, hiding the latency of such operations may be difficult for some applications and processor architectures. This challenge becomes particularly evident when the sequence of requested memory addresses lacks good cache locality, leading to frequent high-latency memory requests. Moreover, when the dataset surpasses the cache's capacity, it can trigger cache thrashing, which occurs when the cache is constantly evicting useful data. This results in poor cache performance and a high number of cache misses and can significantly degrade the performance of applications.

Modern prefetchers have demonstrated almost perfect accuracy in detecting these particular access patterns, but have the downfall of not having the timeliness to pre-load the data to the cache. In other words, many prefetchers are fetching the right data to cache, but not when it is required by the processors. This timeliness leads to cache pollution by erasing useful data. Some designs involve more complex techniques such as having multiple hardware modules or fetching data with different granularities in an attempt to mitigate the impacts of this problem [21, 24–27], but the rate of improvement is decreasing. Not only that, hardware prefetchers are usually energy costly and software prefetchers impose overhead by adding instructions to the instruction stream [24].

3.3 Memory Access Decoupling

Decoupling memory access from the processing core is a useful method for reducing the impacts of memory operation overheads. One approach to achieving this decoupling is through the use of decoupled access-execute architectures, which physically separate memory-accessing and memory-consuming instructions. This separation promotes the decoupling of the mechanisms of memory access from the processing ones, which can help to reduce the overhead associated with memory operations.

Outrider [28] and DeSC [29] are two approaches that aim to increase the throughput of processors.

Outrider focuses on decoupling the memory-access streams from the rest of the computation by using multiple in-order threads with minimal additional cost and without adding additional thread contexts. Despite relying on a simple in-order pipeline and a lower number of explicit software threads, Outrider can reduce the latency effects of memory access, even when dealing with applications that have indirection and a data-dependent program flow.

DeSC proposes a hardware-based design to attack memory bottlenecks automatically, by decoupling data memory access from the computation. An out-of-order core is responsible for pre-loading data which then is communicated to the accelerator responsible for value computation.

3.4 Data Streaming Architectures

Data streaming research consists of providing detailed descriptions of the memory access pattern of applications to improve the efficiency of data acquisition.

Research following this approach is often associated with the design of ISA extensions and descriptor-based mechanisms to describe access patterns. This approach was first explored in Domain-Specific Acceleratorss (DSAs) [30] and in many-core systems or systems with multiple processing elements [31], and the access patterns of applications had to be modeled by hand by the programmer through an Application Programming Interface (API). It is the case of HotStream [32], a data streaming accelerator framework designed for multi-core accelerated-based systems. Later research has taken the advantage that most patterns can be detected at compile-time and thus automated the stream modeling process by designing compiler tools that detect memory access patterns and configure them [16, 33].

Furthermore, several solutions have been proposed to reduce the impact of memory accesses and optimize data traffic in many-core systems and GPPs [2, 3]. Contributions have also been made for DSAs, one example being the hardware-software interface streaming solution proposed by Nowatzki et al. [30]. Neves et al. [31] designed a stream management engine to minimize unnecessary data traffic and maximize throughput while reducing energy consumption in a multiple processing elements system. Wang et al. [2] explored the possibility of empowering out-of-order GPPs with data streaming capabilities by programming them to prefetch a well-known access pattern to load/store buffers. Schuiki et al. [3] modified single-issue in-order GPPs to support a register file extension that provided certain registers with stream semantics, allowing them to perform memory operations when read from or written to.

3.5 Streaming-Mechanisms in General Purpose Processors

3.5.1 Stream Specialized Processors

Wang et al. [2] studied the Cortex-Suite [34] and SPEC CPU 2017 [35] benchmarks and found that most memory accesses follow a small number of simple patterns, denoting the affine patterns as the most common. They argue that the predictability of these addresses is high enough to justify decoupling their processing from core execution, which would result in reduced overhead in the loop bodies of the benchmark's instruction streams and reduced latency of memory access operations.

They highlight opportunities that emerge with streams, such as prefetching, data decoupling, and specialized cache policies. A stream-based prefetcher can proactively pre-load the addresses of streams, while a streaming engine, following the principle of decoupled access execute [36], can generate memory requests that would be advantageous since the core could focus on processing instructions without having to compute memory addresses. Then, knowing the access patterns and the characteristics of the cache, bypass policies can be implemented to avoid polluting it.

To explore such opportunities, Wang et al. [2] proposed stream-ISA extensions and potential micro-architecture support to enable the exploration of these optimization opportunities. These ISA extensions provide the ability to configure and control the streams, and include the following instructions to perform various tasks related to streams:

1. **stream_cfg:** Configures a new stream, by defining its pattern, which consists of its data type, pattern, dependencies, and starting address. After configuration is complete, the hardware is ready to begin fetching data ahead of the core's requests to pseudo-registers (registers buffering stream data).

2. **stream_step:** Advances the position in the pseudo-register of a certain stream. This implies updating the induction variable (the iterator) of the stream itself and the dependent streams, avoiding redundant step instructions. This means that a stream datum can be read multiple times as long as the iterator doesn't get updated.

3. **stream_end:** Deallocates a stream from the corresponding pseudo-registers, which is often applied when a stream has terminated.

In addition to the ISA extensions, some micro-architecture adaptations had to be installed. This involves changes inside the core and the development of a stream engine and pseudo-register for reading and writing data. Figure 3.1 provides a high-level view of the changes required in the architecture of a general-purpose processor.

The changes inside the core involve adding an iteration map responsible for holding the state of the induction variables of each stream. Essentially, it uses counters that increment each variable as

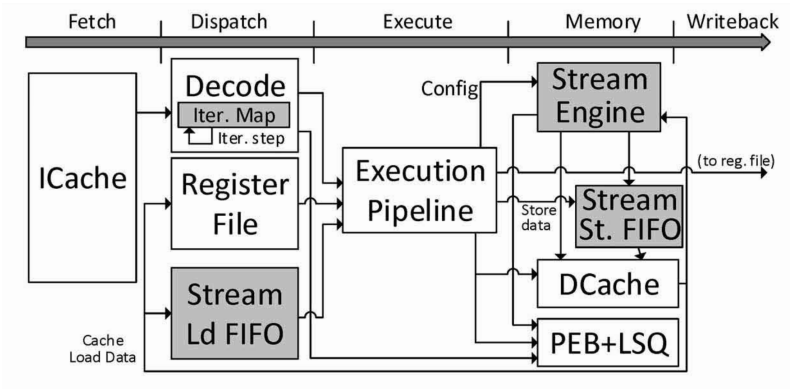

Figure 3.1: Overview of the SSP modifications to the CPU [2]

stream data is being processed. The stream engine is the most important component of the stream mechanism being implemented. It repeatedly needs to access the stream parameters to update its states, meaning that it has to have several tables to hold such information. It is separated into two sub-engines: a load and a store engine, respectively for load and store streams. Depending on the state and parameters of a stream, both compute the required addresses to issue the memory operations. Additionally, data resulting from such operations is buffered in pseudo-registers implemented as stream First-In First-Out (FIFO) queues to connect the core and the stream engine, allowing both to read or write to them.

The proposed work resulted in a 1.67x speedup compared to a baseline Out-of-Order (OoO) core. When the same baseline OoO core was evaluated with a stride prefetcher, the core achieved a 1.22x speedup. This further highlights the potential of empowering computing systems with data streaming mechanisms.

Stream Floating

In another work, Wang et al. [37] investigated how floating streams could enable proactive and decentralized optimizations, ultimately enabling higher efficiency in many-core systems. They consider a many-core system where each core has a bank of L3 shared cache and they propose allowing decoupled streams to float between banks of that cache. Streams are managed with SEs at three places in the core/cache hierarchy: the L3 cache bank (SE_{L3}), the private L2 (SE_{L2}), and within the core (SE_{core}). All stream interactions are performed in the SEs of caches. While processing a stream, the tile's SE_{L2} which is buffering the stream data, requests data to the appropriate L3 cache banks, which may be the L3 bank of any other core. The respective SE_{L3} will handle the request by loading the data and sending them back to the requesting tile, storing them in the SE_{L2}. This way the stream floats between tiles of the multi-core system, reducing data traffic and cache pollution.

Near Stream Computing

In addition to Stream Floating [37], Wang et al [38] proposed modifications to the hierarchy of stream engines to allow performing computation outside the core, adopting this way a near stream computation approach. If the SE of the core detects that offloading a stream is beneficial, it sends a request to a remote SE_{L3}, which then requests the data to the L3 cache. Then, it can decide if the computations are simple enough to be executed on a scalar unit installed inside the SE_{L3}.

3.5.2 Stream Semantic Registers

Schuiki et al. [3] proposed the Stream Semantic Registers (SSR), a lightweight extension targeting single-issue load-store architectures. This extension faces efficiency challenges since data transfers from and to memory have to be encoded explicitly as instructions and thus only a small portion of the instructions are performing computations. They note that the predictability of address patterns can be explored by intercepting, transparently, the accesses to certain registers at the register file and routing those accesses out of the core and into the memory system. Their solution consists of empowering a few registers with stream semantics, in the sense that reading or writing operations to these registers implicitly issue a load or store operation to a memory address.

One beneficial consequence of their work is that explicit load and store instructions are replaced with instructions that encode sequences of regular data accesses (memory access patterns), which efficiently mitigate some of the core's pressure of having to compute effective addresses for memory operations. These benefits are achieved by deviating the address computation to an Address Generation Unit (AGU), which is configured through the encoded instructions, and is responsible for pro-actively generating the addresses whose access will be either requested for reading or writing data. In their work, Schuiki et al. extended the RI5CY core [39], a RISC-V core, and the PULP cluster [40] with little modifications to the core and the addition of a Data Mover unit.

The SSR are intercepted through multiplexing logic that maps data out of the core. In the RI5CY core, there are 2 read ports and 3 write ports, meaning there are a total of 5 ports to the stream interface, which need to be added to the core's port list. Whether the access is or not deviated depends if the stream semantics are enabled, which is possible through Control and Status Registers (CSRs), more specifically a 1-bit `ssrcfg` CSR. The SSR extension is disabled by default. This ensures that code that does not benefit from the use of streams has access to all registers. Additionally, control signals must be introduced to ensure that each instruction accesses its registers precisely once. This becomes crucial in multi-cycle instructions to avoid erroneous data being read or written to memory. These control signals serve the purpose of allowing the register file to exert back pressure on the pipeline. Figure 3.2 presents an overview of the necessary modifications to the RI5CY core for supporting SSR.

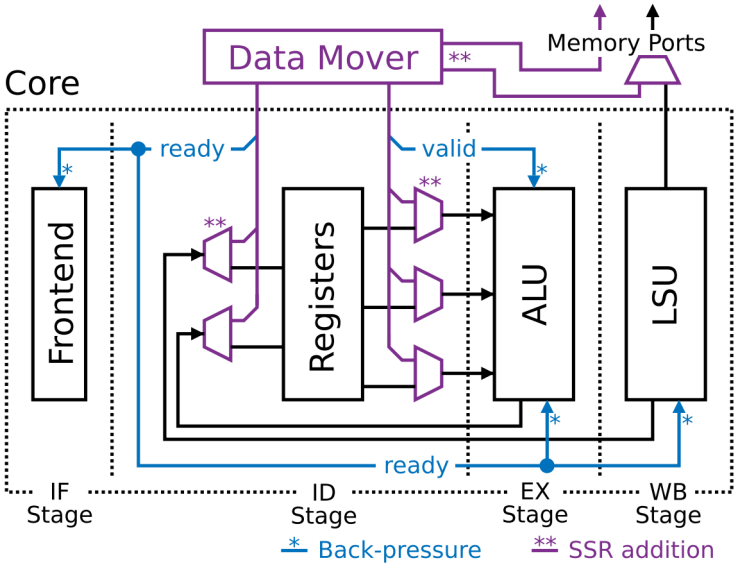

Figure 3.2: Overview of the SSR additions to the CPU [3]

The diverted accesses are processed inside a Data Mover, whose architecture is displayed in Figure 3.3. It consists of a set of lanes (FIFO queues) to buffer stream data and operate on read or write mode. A switch unit maps the deviated access to a lane according to the register's address. An AGU based in the works of Schuiki et al. [41] and Conti et al. [42] is pro-actively computing the addresses of the stream pattern. In read mode, the AGU requests data from memory and stores it in a lane, which is later read from the registers. In write mode, the lanes are written by the registers and the AGU tags each element with an address before writing it to the memory system.

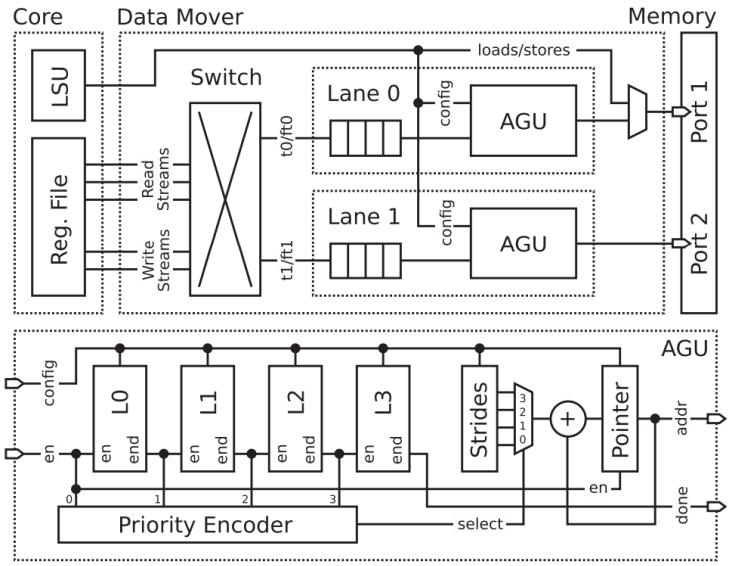

Figure 3.3: Data Mover and Address Generation Unit of SSR [3]

The SSR demonstrated promising outcomes, leading to a substantial performance boost ranging from 2x to 5x across diverse kernels. This improvement was achieved with only an 11% increase in core area. Notably, single-core sequential code exhibited a remarkable 3x increase in speed, thanks to the SSR integration. Furthermore, the extension made a significant impact by reducing instruction fetches by up to 3.5x and cutting instruction cache power consumption by up to 5.6x. These results underscore the potential effectiveness of streaming mechanisms.

One limitation of the AGU in the data mover, as presented in the work of Schuiki et al. [3], is that it is designed to process and generate affine memory access patterns, limiting its applicability to a smaller number of scenarios. To support indirection memory access patterns, Scheffler et al. [43] proposed the Indirection Stream Semantic Register (ISSR) extension, which was also designed for use within single-issue, in-order processing cores. Zaruba et al. also proposed Manticore [44], a general-purpose and ultra-efficient chiplet-based architecture for data-parallel floating-point workloads and Snitch [45], a tiny pseudo-dual-issue RISC-V processor paired with a powerful double-precision Floating-Point Unit (FPU). Both these architectures have been extended to support the SSR extension, building upon their previous research.

3.6 Summary

This chapter started by explaining state-of-the-art VLA extensions, namely SVE and RVV. These extensions allow for the code to be reused across architectures with different vector lengths. However, they still contribute to overheads as they introduce instructions in the instruction stream and are limited on the access patterns of applications, which may be too sparse and with dependencies.

Data prefetching techniques were also approached. Both state-of-the-art hardware and software prefetchers have capabilities of detecting and prefetching with high precision the memory accesses, but they lack timeliness in the process, and even the most complex designs are struggling to increase the provided gains.

When the patterns of workloads can be perfectly described, data can be processed without the intervention of the processing units, a process named data streaming. Some designs require manual configuration of the access pattern, but state-of-the-art solutions have developed compiler tools that detect them automatically. Since the memory accesses can be deviated from the CPU, the latter can dedicate its resources to actual computation, thus improving efficiency. Data streaming and data decoupling have been integrated both in DSAs and GPPs, but these are often limited by the access patterns.

4

Proposed Streaming Engine

Contents

Data streaming mechanisms represent an active research area within HPC systems and current state-of-the-art streaming mechanisms in GPPs introduce hardware infrastructure aimed at efficiently streaming data from the memory system. However, these solutions often exhibit inefficiencies in reusing data fetched from memory. Specifically, existing solutions tend to issue a memory request for each address within a given data stream, resulting in bandwidth bottlenecks.

This chapter provides an in-depth description of the proposed hardware infrastructure that was developed to align with the requirements of the UVE micro-architecture, while simultaneously introducing new mechanisms that mitigate the overall inefficiencies of existing solutions in HPC systems.

Taking into account the ISA defined in UVE and its vector processing-related capabilities, a high-level schema of the designed SE is depicted in Figure 4.1.

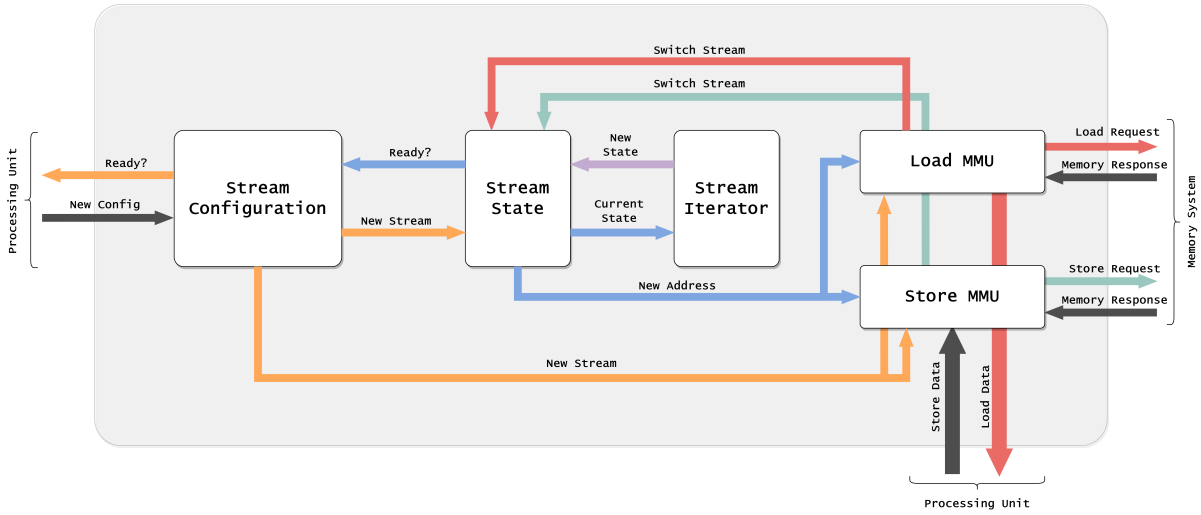

Figure 4.1: Diagram of the Proposed Streaming Engine's Architecture

The design bears some resemblance to the one put forth in UVE, but some modifications were incorporated to enhance its capabilities. It is organized into five discrete, interconnected modules. These modules encompass a Stream Configuration (SC), to receive incoming stream configurations; a Stream State (SS), to hold the state of a given stream; a Stream Iterator (SI), to update the state and generate new address values; and two Memory Management Units (MMUs), designated for handling both load and store memory requests. This chapter will provide a comprehensive exploration of the structure and functionality of these sub-modules, while also defining how they are connected and their inter-dependencies.

4.1 Stream Configuration

According to the streaming configuration instructions of UVE, the process of configuring memory access patterns can range from a single instruction to multiples of them, depending on the complexity of the pattern. According to its ISA, streams are configured incrementally, with each instruction adding a new descriptor. For this reason, configuring streams is very often a process that requires many cycles, and therefore a module specialized for handling configuration instructions is employed.

Channels & Functionalities

The SC is composed of two main communications channels, associated each one with two distinct functionalities. The first channel is dedicated to exchanging signals with an external processing unit, primarily responsible for receiving stream configuration information. This information comes as signals resulting from the decoding process of streaming instruction and should be handled by the processing unit. The second channel has the purpose of feeding the SE with the configuration signals that will initialize a new stream within the whole SE and start all the stream processing operations, namely address generation and memory requests.

This module acts as a separation layer between the processing unit and the remaining modules of the SE, being responsible for receiving multiple configuration instructions regarding a single stream, processing them, and translating them into new signals that will initialize that stream in the SE.

To ensure consistency in these data transfers, the module implements a handshake mechanism. Each channel is equipped with both `ready` and `valid` signals. The SC serves as the master when providing data to the SE but operates as a subordinate when receiving data from the processing unit.

Internal Registers

Since many cycles may be required to configure a stream, the module's architecture necessitates the inclusion of storage registers, which have the essential role of preserving the parameters of diverse descriptors and other meta-information that define a stream's memory access pattern.

Adhering to the descriptor paradigm for memory access patterns, the module must retain crucial details such as the stream type (load/store), the stream identifier, element width, and the distinction of scalar or vector-bound, which are stored in dedicated registers.

Additionally, given that it has to handle two descriptor types, specifically dimensional and static modifiers, the module also houses two register tables to manage these descriptors. In this case, the tables consist of groups of registers that hold the parameters of the descriptor. So, the dimensional descriptor table has registers to hold the `offset`, `stride` and `size` values. The static modifier descriptor table has

registers to hold `target`, `displacement`, `behavior`, `size` and `dimension` values, being the latter the dimension the modifier is attached to. They are tables because there are multiple entries of these groups of values, and each new descriptor provided corresponds to filling one row in the tables.

These tables are accessed through pointers, so when a new descriptor is provided, it is sequentially stored in the table, implicitly capturing the hierarchical information. The pointers are adjusted accordingly to maintain data organization. For example, the first descriptor will be stored in the table in the index 0, and if a second is received then it will be stored in the index 1 of the table.

The current design only has registers for the configuration of a single stream, meaning only one stream can be configured simultaneously.

An overall diagram of the architecture of the SC module is displayed in Figure 4.2.

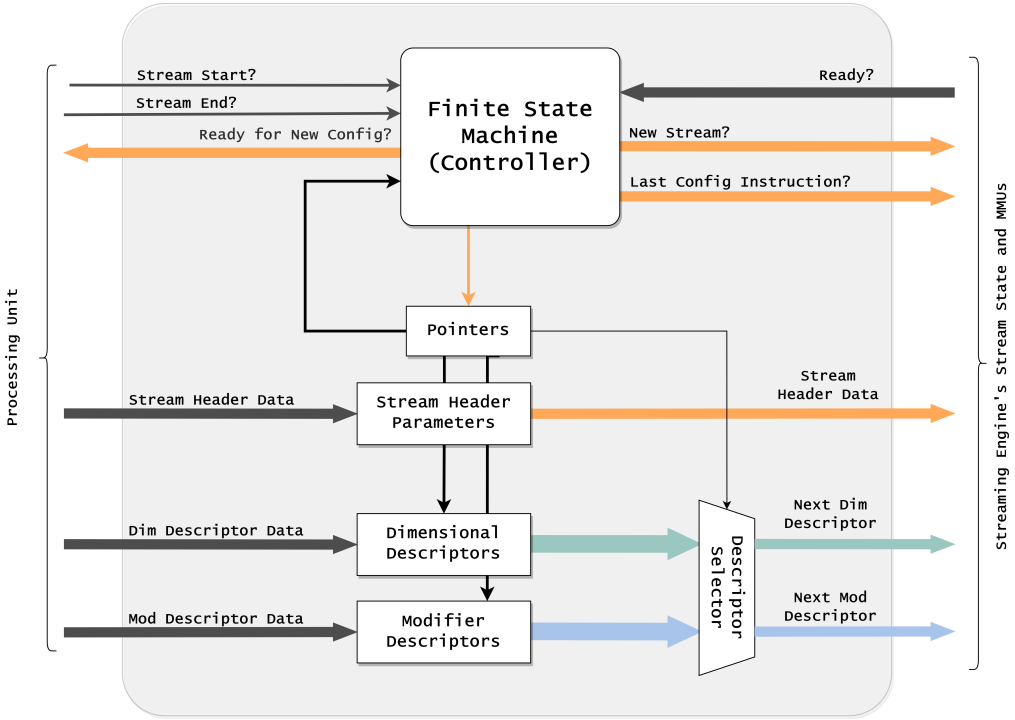

Figure 4.2: Diagram of the Stream Configuration's Architecture

From the processing unit, the SC receives decoded signals of the stream configuration instructions. These include the stream header and a dimensional or modifier descriptor. Also, control signals including if the stream configuration is starting or ended are received. The module contains registers to store the header data as well as the descriptors. The stored data is organized and redirected iteratively to the SE, through signals managed by a FSM.

Stream Configuration Life Cycle

The whole life cycle of a stream in the SC, from receiving the configuration start instruction from the processing unit to feeding the last descriptor to the SE is managed through a FSM. The configuration process is separated into three different stages, each stage corresponding to one state in the FSM.

In the first state, information is received from the processing unit, in the second the SC waits for the SE to be ready to receive a new stream and in the third state the SE receives signals with the stream initialization. A diagram of the FSM is displayed in Figure 4.3.

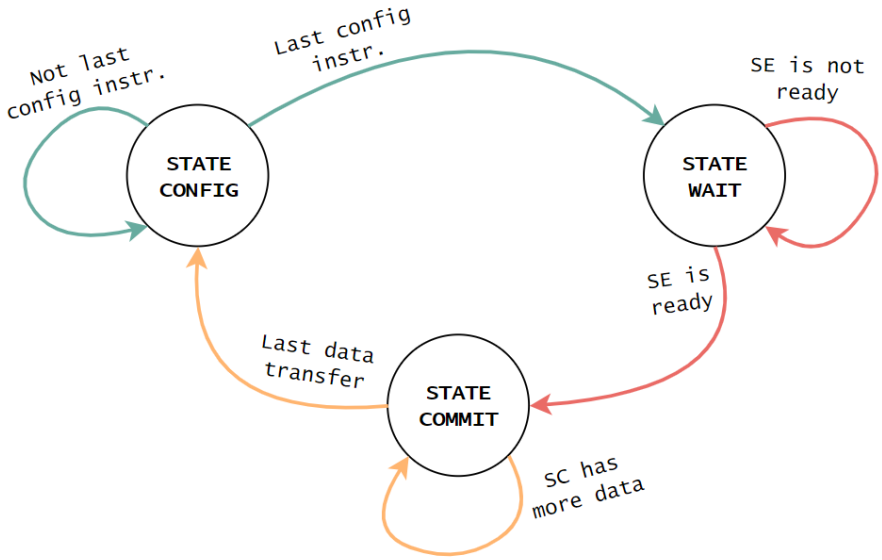

Figure 4.3: Scheme of the Stream Configuration's Finite State Machine

STATE_CONFIG: In this initial state, the internal registers are empty, and the module awaits decoded configuration signals from an external processing unit. The SC signals its readiness to receive new data by setting a `ready` signal to `HIGH`. The FSM will remain in this state from the start configuration instruction until the last one is received, meaning that this state will be active while the processing unit is providing configuration instructions. A new stream is configured with instructions like `ss.sta.<ld/st>.<width>`. The SC detects the start of a new stream with a dedicated input signal. When this signal is `HIGH`, a load operation is triggered in the internal registers, loading the header information, including stream identifier, direction (load/store), and element width. When a new descriptor is appended, such as with `ss.<app/end>.mod`, the SC uses signals to indicate the addition of a descriptor, whether it's dimensional or a modifier descriptor. The module stores this descriptor data in the internal descriptor table registers. The order of dimensional descriptors and the dimension each static modifier is attached to are maintained by using two pointers. These pointers are represented in a one-hot format for hardware optimization, avoiding the need for adders when updating. Adding a new dimensional descriptor means that a new dimension is implicitly being added to the access pattern. However, when a modifier is added, the SC determines its dimension by checking the last configured dimension, which will be the one the modifiers will be associated with. Instructions like `ss.cfg.vec` define the nature of the stream (scalar or vector). The channel includes signals indicating vectorization requirements. The end of a stream con-

figuration is marked by receiving an instruction with the `ss.end` format. The SC detects this instruction and transitions to the STATE_WAIT state. Until this point the stream has not yet been configured in the SE, it simply has gathered the required information to initialize a new stream.

STATE_WAIT: During this state, the SC module holds the accumulated data until the SE signals its readiness to receive new configuration data. Separate `ready` signals for load and store streams are used. The SC waits for the corresponding `ready` signal to be asserted, depending on the direction of the stream. For example, if the internal registers hold a load stream configuration and the SE signals readiness for load streams, the FSM transitions to the STATE_COMMIT state, where data is transferred to other modules of the SE. The same process is applied to store streams. The SC is designed to handle one stream configuration at a time to optimize resource allocation. This means that, while in this state, the SC signals the processing unit that it is not ready to receive new configurations. In the handshake cycle (when the SE is ready to receive the data), the header information of the stream is passed to the SS and one of the MMUs, depending on the stream's direction. This initiates a resource allocation process in those modules, and all the descriptor data will be transmitted in the next state.

STATE_COMMIT: In the final state, the SC prepares the SE for a new stream by initializing all the required modules. The SC copies gathered descriptors to the SS. It can copy up to one dimensional and one modifier descriptor per cycle. In this state, the pointers of the SC serve as reading offsets, in contrast to the first state where they were used as writing indicators. As descriptor information is transferred to the SS, the pointers shift once, updating descriptors for the next cycle. Eventually, the pointers will point to invalid data, indicating that no more dimensional and/or modifier descriptors remain. The SC detects the final data processing by monitoring the pointer values. It sends an acknowledge signal to the SS, triggering it to start processing. This action also prompts a state transition in the FSM, returning to the STATE_CONFIG state, where new configurations can be provided by the external processing unit.

4.2 Stream State

The SS can be considered one of the core modules of the whole SE. It is the module responsible for storing all the data related to the address generation of streams. Naturally, this includes not only the parameters that define the stream's memory access pattern but also the state in which each stream is at a given cycle.

Channels & Functionalities

The SS has three channel ports, each used to communicate with a different module of the SE and manage a different functionality:

- **Configuration Channel**: Serves the purpose of receiving new stream configurations. Contains all the signals related to the configuration of a new stream in the SS. Naturally, most of these signals come directly from the SC and include header information about the stream and descriptor information. Configuring a new stream requires initialization in both the SS and MMUs modules. For this reason, this channel also receives pointers to the location in the MMUs where this new stream is being configured.
- **Iteration Channel**: Used to generate the next state of a given selected stream. This includes the calculation of new addresses, updating the iteration counters of descriptors, or triggering dimension transitions in the stream processing. The operations that it performs and the results that it updates depend on the level of progress of a stream. The SS redirects the required signals to the SI module (see Section 4.3) to compute the next set of results.
- **Memory Controller Channel**: Responsible for redirecting the generated addresses of a given stream to the corresponding buffers in the MMUs. Depending on the stream direction, the addresses can be transferred to the Load Memory Management Unit (LMMU) (see Section 4.4) or to the Store Memory Management Unit (SMMU) (see Section 4.5). Accompanied by the generated addresses, this channel also has signals containing the state of a stream (e.g., which dimensions have completed all its iterations). This information is required as it impacts the vector generation in the MMUs.

In short, the configuration channel allows new streams to be added to the SS. The iteration channel serves as the interface to the SI, allowing for streams to progress and for new addresses to be generated. The memory controller channel allows the generated addresses and other meta information to be passed onto the memory controllers, which will handle memory operations.

Internal Registers

When dealing with stream representation, the SS module necessitates various levels of storage. To begin with, it is important to note that the module can manage multiple streams concurrently. For this reason, the architecture is structured into a set of stream tables, with each table containing the necessary data to represent a single stream. Details on the values that are kept in each table for a given stream are displayed in Figure 4.4.

A single stream is defined by its set of dimensional and static modifier descriptors, as well as other header information. This information must be stored in the SS to be able to correctly process the stream. For this reason, each stream table is composed of multiple registers containing header information, such as the `element width` of data, `stream identifier`, and `stream direction` (load or store streams). In addition to these registers, each stream table is composed of two sub-tables (see Figure 4.4): a

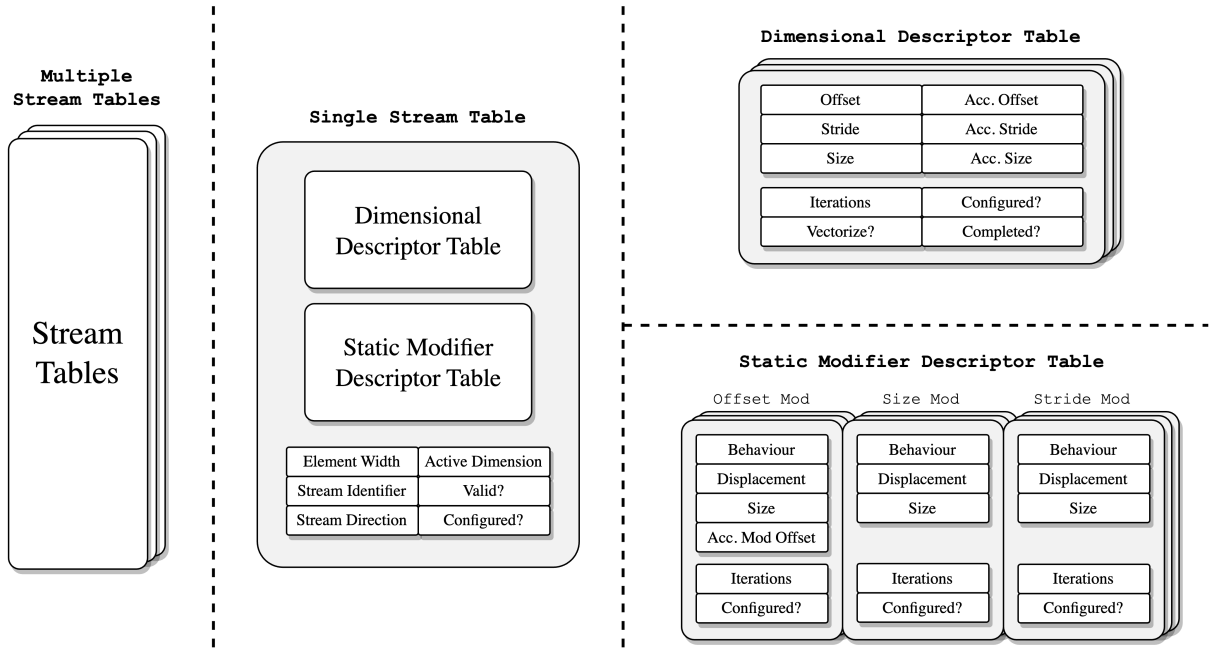

Figure 4.4: Different Depth Levels of the Stream Tables of the Stream State

dimensional descriptor table, which will hold the dimensional descriptor parameters, and a static modifier descriptor table, to hold the modifier parameters. Each one of these descriptor tables is composed of a set of registers, each storing a specific value important for the descriptor. Since a stream may have many descriptors, each one of these tables is composed of multiple ways, working as a full-associative descriptor table storage.

Regarding the dimensional descriptors, its tables include the `offset`, `stride`, and `size` base fields for each dimension. Due to the descriptor paradigm adopted (see Section 2.2.2), these values will suffer modifications as the stream gets iterated. For this reason, registers are also required to store accumulated values for each one these base parameters. The base parameters still need to be retained in registers because, throughout stream processing, these accumulated values will need to be reset to their original values.

Regarding the static modifier descriptors, its tables are divided into three specific tables. More specifically, each modifier table has a sub-table for each possible modifier target. As mentioned, each dimension can have multiple modifiers, and the SS supports the configuration of up to three distinct modifiers, one for each target (i.e., it is possible to target `offset`, `size` and `stride` simultaneously). For this reason, the `target` field of a modifier is implicitly defined by indexing the modifier table (see Figure 4.4), and each modifier table has registers for the remaining parameters, namely the `displacement`, `behavior` and `size`. One thing to note is that the offset modifier has an additional register, specifically to store the accumulated offset value that it is targeting. This is required to avoid concurrency conflicts as both

dimensional descriptors and offset modifiers write a new offset value.

Moreover, each descriptor requires an iteration counter, to keep track of how many times it has been iterated. This is achieved by employing a register for all descriptor types. This information is accompanied by a `completed`, to mark a descriptor as completed. Note that this information only exists in dimensional descriptors. Modifiers do not have it because when its iterations are complete it gets immediately reset, something that does not happen to the dimensional descriptors. Since not all streams use all the possible descriptors supported, each descriptor also has `configured` flags to mask the valid tables. Note also that each stream table has a `configured` and a `valid` registers. The reason for this is that the process of initializing a new stream takes more than one cycle, and thus it is possible for a stream to be configured (resources are allocated) but its data is not yet valid (it is still being transferred).

Other registers are utilized to monitor the active states of the FSMs within the SS module. These registers will be further elaborated on in this chapter.

Stream Address Generation Life Cycle

Having defined the internal registers of the SS, diverse operations occur internally in this module from the moment a stream is configured until its last address is generated. To better understand these operations, a visual representation of the SS's architecture is displayed in Figure 4.5. The whole process of processing a stream configured in the SS will be detailed in the following sub-sections.

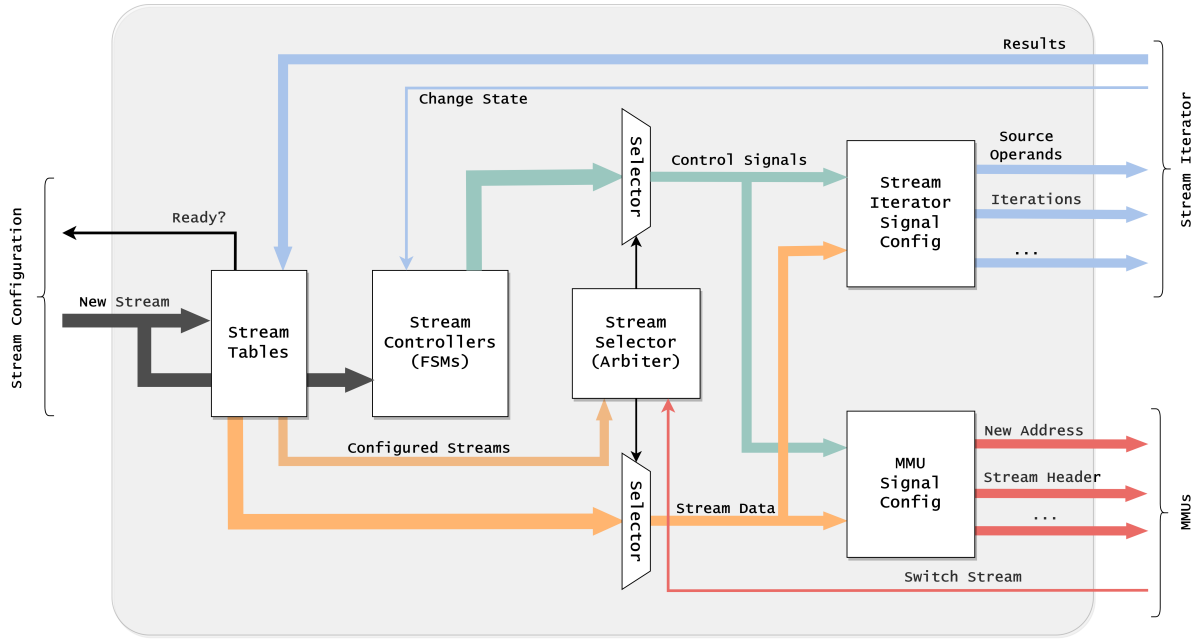

Figure 4.5: Diagram of the Stream State's Architecture

Initializing Stream Tables

The SS and the SC modules maintain direct communication to facilitate the detection of when a new stream can be loaded into the stream tables of the SS. The configuration channel within the SS needs to signal its readiness to accept new stream configurations to the SC, and this is achieved by identifying which stream tables are not yet `configured`. If there is at least one unallocated table, the SS asserts its readiness to receive new configurations by setting a `HIGH` signal. When the SC possesses a valid configuration, the handshake is initiated, resulting in a transaction that configures one of the stream tables. For the sake of hardware simplicity, the first available non-configured table is selected, although different criteria could have been implemented.

However, even after the handshake, a newly configured stream table is not immediately prepared to commence processing in most cases. The SC can only provide two descriptors of each type at a time. Consequently, more complex patterns requiring multiple descriptors require a fixed number of cycles to fully initialize the stream table. Bearing this in mind, all stream tables possess a `configured` flag, activated during the handshake, and a `ready` flag, which is set when the SC has supplied all the necessary information via an acknowledge control signal. Only when both the `configured` and `ready` signals are `HIGH`, is the stream deemed valid and available for selection during iteration.

Selecting Streams for Iteration

The current architecture for the SE only allows one stream to be iterated per cycle. To alter this restriction, the number of SIs and signal buses would have to duplicate.

Since the SS can hold multiple streams simultaneously, there needs to be a criterion for selecting one valid stream table from the module for the iteration process. The SS achieves this through a round-robin arbiter, which receives as input a signal indicating which tables are valid.

Because it follows a round-robin criteria, the arbiter holds as an internal state the current stream table selected, as a way to follow the criteria correctly. The arbiter can be triggered to change the selected stream table, and the next selection is a combination of the input signal with the internal state of the arbiter.

Generating Addresses for the Selected Stream

Generating all the addresses that define the memory access pattern of a stream is a process that increases in complexity with the number of descriptors configured, either with dimension or static modifier ones. The whole life cycle of a stream in the SE is managed by the SS through a FSM.

Since each stream is configured in its stream table, it is natural that every stream table will have its own, independent, FSM. A diagram of its states and transition conditions is displayed in Figure 4.6.

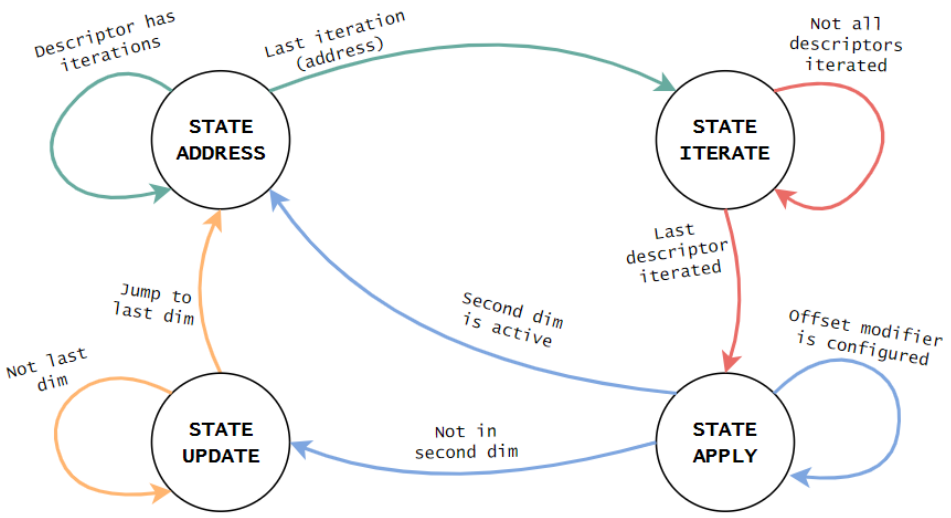

Figure 4.6: Scheme of the Stream State's Finite State Machine

STATE_UPDATE: As per the descriptor paradigm adopted (see section 2.2.2), the process of iterating through a stream begins from the higher dimensions and proceeds towards the base dimension by accumulating the offset values of each dimension. Consequently, when configuring a stream table, it initiates in the highest valid dimensions. The initial task is to accumulate all offset values down to the lowest dimension, and this task is carried out in the `STATE_UPDATE` state. In this state, several crucial parameters undergo alterations:

1. The accumulated offset of the lower dimension is updated with the cumulative offset value from the higher dimensions.
2. The iteration count of all descriptors in the lower dimension is reset to their base size parameter.
3. The accumulated size and stride fields of the lower dimension are reset to their respective base values.
4. The accumulated offset of the offset modifier in the current dimension is reset (if the modifier is configured).
5. The active dimension transitions to a lower dimension.

The FSM remains in this state until the active dimension transition leads to the lowest dimension. When this occurs, it triggers a state transition, and the next state becomes `STATE_ADDRESS`. This transition signifies the completion of the accumulation process and marks the beginning of address generation for the stream table.

STATE_ADDRESS: In this state, the active stream is at its lowest dimension. During each cycle in this state, a new address is generated by accumulating the current accumulated offset value with the accumulated stride of the lowest dimension (i.e., the active dimension). In this state, two crucial parameters undergo alterations:

1. The accumulated offset is updated to reflect the newest generated address.
2. The iteration count of the current dimension, which is the lowest, is decremented.

The FSM remains in this state until the number of iterations reaches zero. When this occurs, indicating that the descriptor has been fully iterated, the dimension is marked as completed. This triggers a state transition, and the next state is `STATE_ITERATE`, where a higher dimension will be iterated.

When the transition is triggered, preparations are made for the next state. In fact, during the transition cycle, the active dimension changes to the first dimension that has not yet been marked as completed. Consequently, when the state changes, so does the active dimension. Additionally, the configured descriptors (dimension and modifiers) are scheduled for iteration, providing information that will be processed by the SS in the `STATE_ITERATE` state.

STATE_ITERATE: During this state, the selected stream is updating descriptors of a higher dimension. All configured dimension and modifier descriptors are iterated, meaning there can happen up to four iterations: dimension, size modifier, stride modifier, and offset modifier. The goal is to advance the current dimension by iterating its parameters so they can be transferred back to the lowest dimension, where more addresses can again be generated.

In every cycle, when this state is active, the SS keeps track of which descriptors have already been iterated and selects one that has not for iteration. Depending on the descriptor selected for iteration, different parameters, and registers are updated. In this state, some arithmetic operations need to be performed to advance the stream. For this reason, the SS communicates with the SI by providing data through the iteration channel. The operands and other port values of this channel depend on the descriptor that is selected for iteration, and an overall scheme of these values is displayed in Table 4.1.

Every time one descriptor is iterated, it is removed from the queue of descriptors scheduled for iteration, allowing this way for a new descriptor to be iterated the next cycle. One thing to note is that the modifiers are being iterated in separate cycles for resource utilization optimization. The calculations are performed in the SI module, which only has two adders.

The SI consistently generates results for both iteration counts and accumulation operations. The specific registers that receive these values depend on the selected descriptor for iteration and the control flags generated by the SI. When iterating a dimension modifier, the SI loads the arithmetic result into the accumulated offset register of the currently active dimension. Simultaneously, it decrements the iteration count. If this count reaches zero, the dimension is marked as completed. In contrast, when

Table 4.1: Stream State - Input Ports of the SI channel

	Dimension	Size Mod	Stride Mod	Offset Mod
si_out_op1	acc_offset	acc_size (next)	acc_stride (next)	mod_acc_offset
si_out_op2	acc_stride	mod_size_disp	mod_stride_disp	mod_offset_disp
si_out_iterations	iter	mod_size_iter	mod_stride_iter	mod_offset_iter
si_out_behaviour	increment	mod_size_behav	mod_stride_behav	mod_offset_behav
si_out_size	acc_size	mod_size_size	mod_stride_size	mod_offset_size
si_out_mod	no	yes	yes	yes

The (next) notation indicates that the value comes from the lower dimension, otherwise it is a value of the active dimension. The acc keyword is short for 'accumulated'. The mod keyword is short for 'modifier' and its omission means it is a dimension descriptor field. The behav is short for 'behavior'. The disp is short for 'displacement'. The iter is short for 'iterations', a field common to all descriptors. Examples: mod_stride_size means that the port holds the size field of the stride modifier; acc_size (next) represents the accumulated size field of the dimension descriptor of the lower dimension.

iterating a static modifier, the registers updated follow a similar pattern across different modifier types but are tailored to each. For all modifiers, two distinct scenarios arise, contingent upon whether the iteration count has reached zero or not.

If the count hasn't reached zero, the iterations are simply decremented, and the arithmetic result is stored in the respective accumulated field associated with the modifier type. For example, a stride modifier stores the result in the accumulated stride register of the lower dimension. Conversely, if the iteration count has reached zero, it is reset to the default number of iterations (specified by the size field) of the modifier. Simultaneously, the accumulated field is reset to the default base parameter value. For instance, in the case of a stride modifier, it restores the accumulated stride register to the original stride value. However, there exists an exception—the offset modifier. Instead of storing the arithmetic result in the lower dimension, it preserves it in a local accumulated offset register that is specific to the offset modifier. When all descriptors are iterated, it transitions to STATE_APPLY.

STATE_APPLY: Iterating the dimension modifier and the offset modifier does not reflect the new values calculated to the lower dimension. For this reason, this state has the role of updating the accumulated offset for the lower dimension concerning the active one. There can happen up to two operations in this state, one for the dimension and the other for the offset modifier (if the active dimension has it configured), which happen in consecutive cycles. The result of this operation is that the accumulated offset of the lower dimension is now its base offset added with the accumulated offsets from the off-

set descriptors of the higher dimension, just like the descriptor paradigm describes. Depending on the active dimension when this state is active, the state transition can be either STATE_UPDATE when not in the second dimension, as these changes need to be carried until the first dimension, or STATE_ADDRESS when the active dimension is the second one, where the changes are already being transferred to the base dimension in the STATE_APPLY state.

Finite State Machine - Stream Life Cycle

When a stream initially becomes valid within a stream table, its FSM commences its operation in the STATE_UPDATE. The primary objective of this state is to ensure that the lower dimensions are updated to reflect any changes occurring in the higher dimensions. This update is critical for maintaining the consistency of generated memory addresses. Consequently, as the number of dimensions increases, the need for information transfer also rises. In this state, offset changes are propagated, iteration counters for lower dimensions are reset, and any modifications made by descriptors are cleared.

Upon reaching the lowest dimension, the FSM transitions to the STATE_ADDRESS, where the actual address generation process begins. This state remains active as long as the lower dimension has pending iterations. However, once the lower dimension has been completely iterated, the FSM must move on to iterate a higher dimension. The selection of the next dimension is determined by the completion status of each dimension, prioritizing the first incomplete dimension found. This transition leads to the STATE_ITERATE.

In the STATE_ITERATE, various modifiers may be present, and all corresponding fields are iterated as required. Notably, the offset-related descriptors do not immediately propagate their updates to the lower dimensions. Instead, the FSM progresses to the STATE_APPLY, where these updates are applied.

Depending on the selected dimension, the FSM can either transition back to the STATE_ADDRESS (when the higher dimension being iterated is the first one immediately above the lowest dimension) or to the STATE_UPDATE (when the dimension is higher, necessitating information to be cascaded down through multiple cycles until it reaches the lowest dimension again). These transitions continue until all dimension descriptors have completed their tasks, marking the conclusion of the address-generation process for the stream. At this point, resources are unallocated, and a new stream table becomes available.

Communicating with the MMUs

The SS solely generating addresses is insufficient since these addresses must be linked to memory requests within the memory system. Consequently, when a stream generates a new address, it necessitates reception and registration by the MMUs within the SE. For instance, if a valid stream is chosen by the arbiter during a particular clock cycle, the mmu_out_valid signal is raised HIGH. Further-

more, if the FSM is in the STATE_ADDRESS state, indicating that a new address is being computed by one of the streams, the mmu_out_addr_valid port is also activated. Simultaneously, information regarding the stream, including mmu_out_stream, mmu_out_mmu_idx, mmu_out_width, and mmu_out_type, is provided. Likewise, data regarding the address and the stream's state, such as mmu_out_addr, mmu_out_completed, mmu_out_vectorize, and mmu_out_last, is made available. In situations where the LMMU is unable to accept the address provided by the SS for load streams, or the SMMU for store streams faces the same issue, acknowledgment is conveyed by activating the mmu_in_trigger input port. This port is interconnected with the arbiter's trigger port, indicating that the MMUs instigates a shift in the selected stream for iteration when additional addresses cannot be introduced within the current cycle. Consequently, this allows another stream to progress while resolving the previous one.

4.3 Stream Iterator

The process of iterating through a given stream varies depending on the type of descriptor being used. Each descriptor type demands distinct arithmetic calculations, involving different operands. For this reason, the SE must be able to adapt the operations according to the specific descriptor in use.

To simplify the hardware complexity, the SI employs only two adders. The first adder is responsible for decrementing the iteration count of the selected descriptor. The second adder performs addition or subtraction on two stream parameters, such as an offset value and a stride. However, due to the simplicity of this hardware module, it can process only one descriptor per cycle. One thing to note is that this module is entirely comprised of combinational logic, as the results it generates are stored in different modules of the SE.

A visual representation of the SI's architecture is displayed in Figure 4.7.

Input and Output Ports

The SI's input and output ports contain the information required to perform the different calculations needed by the different descriptors, either dimension or modifiers.

The number of iterations of a descriptor is provided through the val_iterations port, which the SI decrements and returns the result through the res_iterations. In addition, there is also combinational logic to detect when the last iteration is being processed, which the module detects by asserting HIGH the val_last output port.

The SI detects, for a given descriptor, when the number of iterations or its size field is zero. In response, it activates the load_ena port which prompts the stream to transition to a higher dimension. Simultaneously, the SI calculates the load_dim output port, which provides a one-hot representation of the dimension to which the stream should switch in this case. This calculation is based on a combination

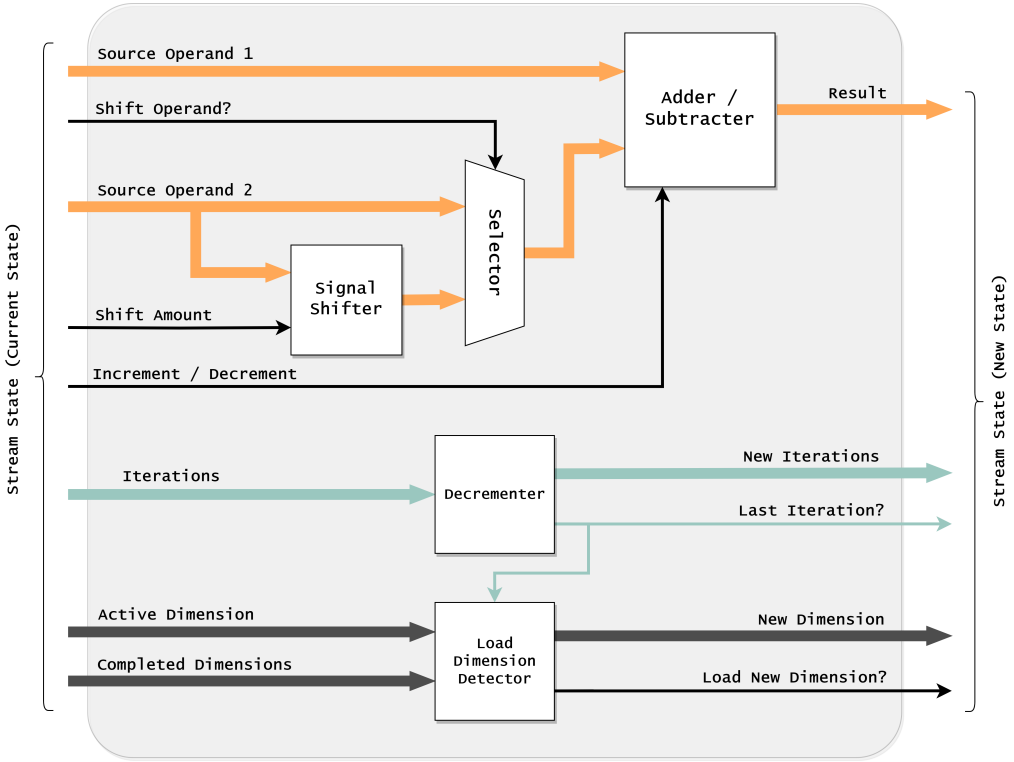

Figure 4.7: Diagram of the Stream Iterator's Architecture

of factors, including the dimensions that have been configured and completed within the stream, and information accessible via the `val_configured` and `val_completed` ports, respectively. Additionally, it takes into account the current active dimension (e.g., if the current dimension is the second, it will not switch to the first).

The SI computes new parameters (offsets, strides, or sizes) by reading two source operands, `src_op1` and `src_op2`. Depending on the value of `val_behaviour`, an addition or subtraction can be performed between the two signals. The values that are provided through these input operand depends on the descriptor type.

One thing to note is that when the descriptor is not a modifier (which occurs when `val_mod` is deactivated), the second source operand is a stride, and its value is shifted according to the element's width (provided through `val_width`). This guarantees that there is always memory alignment of the data elements, as long as the base offset already respects the memory alignment.

4.4 Load Memory Management Unit

When designing a MMU, one needs to consider the pitfalls of memory access operations. They generally have high latency, meaning accessing the memory system takes many clock cycles. Since the streams

are producing at most one address per cycle, one can think about serializing the memory operations, one for each address provided by the AGU units, but this would hurt severely the system's performance. Nevertheless, due to the nature of streams, many times the memory access patterns they are describing allow for data re-usage and open the possibility for parallel processing.

For this reason, the LMMU's architecture is composed of three main modules that communicate with each other, to maximize the throughput of the AGU responsible units (i.e., the SS and SI) while also reducing the number of times memory operations are issued to the memory system.

So, instead of performing one load operation for each address generated by the SE, it can be beneficial to design a more complex MMU that handles different tasks in parallel. Given a stream, the following opportunities arise:

1. As new addresses are generated, they can be scheduled as outstanding requests, which will be resolved later. These requests can be organized according to their memory tag.
2. Since different memory addresses can be mapped to the same memory row, a single memory load instruction is necessary to fetch the data.
3. The data retrieved by the load operation is sufficient to resolve all the outstanding requests of the same memory tag.

From these opportunities arise the three modules that together make the LMMU and its functioning is as follows, which is illustrated in Figure 4.8.

4.4.1 Load FIFO Queues

The Load First-In First-Out (LFIFO) queues are collections of registers that hold the data meant to be read by the processing unit that configured a given stream. It has logic to guarantee that the memory access pattern is respected, i.e., that the data has the same order as the respective memory addresses of the stream. The main feature of this module is that, as the data is obtained from the memory, the elements are joined together (coalesced) in the registers, no matter how irregular the pattern might be. From the processing unit's perspective, it is as if the memory accesses were all consecutive.

A scheme of the architecture of this module is displayed in Figure 4.9.

Internal Registers

The SE can support multiple streams and therefore there are also multiple LFIFOs. Each is described by a stream's meta-information, such as the identifier and width. Each can also hold multiple vector elements, and thus each FIFO is described by a table of vectors (or buffering queues), where the data will be loaded onto.

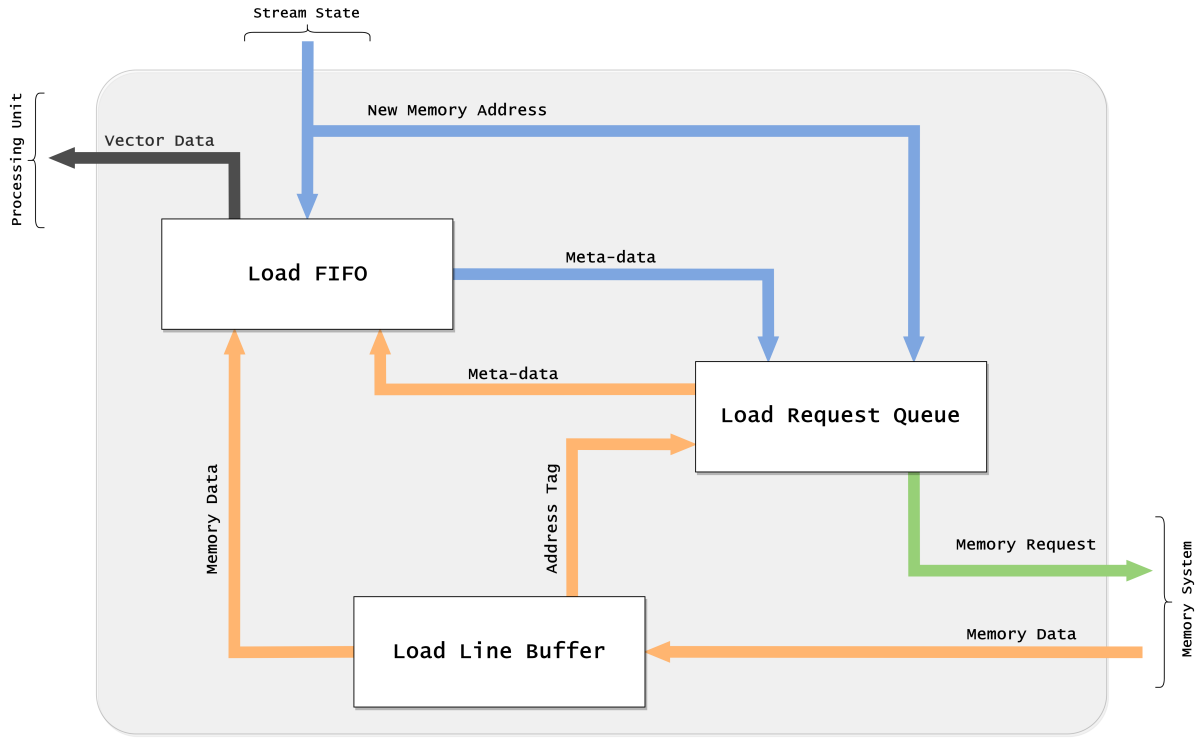

Figure 4.8: Diagram of the Load MMU's Architecture

For each vector, the elements are indexed by the byte, which is the smallest width possible. Each byte has `predicate` and `valid` flags, to inform if the byte within the vector is reserved and if the data is valid, respectively. Each vector also has a `reserved` flag, which will be used to indicate whether or not the vector is valid for reading.

This means that two writing pointers are required: the vector offset, which indicates which vector of the table of vectors is being at that moment written, and the byte offset, which indicates which byte is the next to be written in a given vector. Since reading occurs in vector batches, only a single read pointer is necessary. Each LFIFO also has a flag register to indicate whether or not all addresses have been generated by the SS and SI.

Populating Vector Registers

When the AGU components of the SE provide a new address, one of the LFIFOs is updated so that new vector elements are marked as reserved. The FIFOs index is provided by the SS, which was saved during the configuration process, which avoids unnecessary comparisons. The write pointers of the targeted queue are used to determine the next position of the address that is being generated.

Depending on the width of the elements, different numbers of consecutive bytes are reserved (which is achieved by activating the corresponding `predicate` registers). At this point, the `valid` registers are

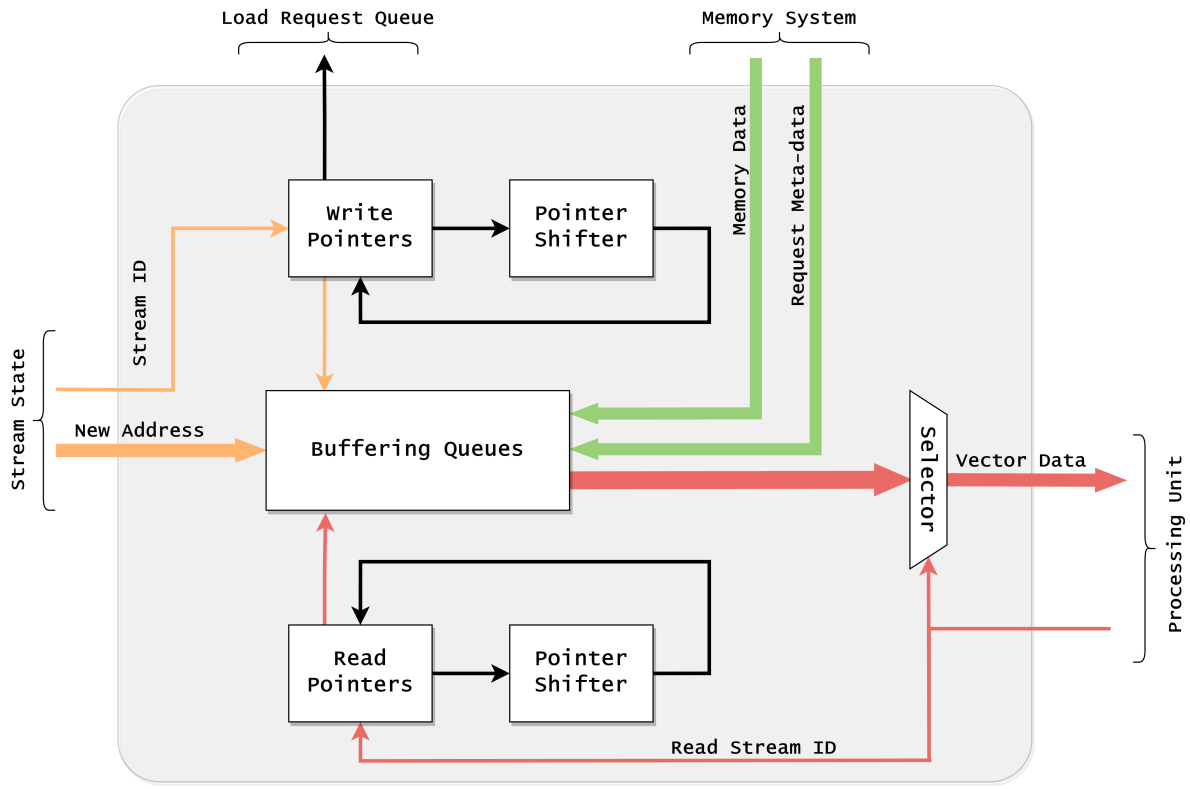

Figure 4.9: Diagram of the Load FIFO's Architecture

deactivated, as the data bytes currently on those positions do not hold correct information. The write pointers get updated accordingly: the pointer within the vector is incremented according to the element width; the vector pointer is incremented only when the pointer within the vector overflows, meaning the vector is full. When the vector pointer is incremented, the current vector is marked as `reserved`, meaning the bytes predicated will not change and, therefore, only the data needs to be provided for the vector to be ready.

The other modules of the LMMU will work to provide and map the correct data to the respective reserved (or predicated) bytes. When correct data is loaded to a given byte of the LFIFO queue, it is marked as `valid`. Finally, a vector is valid for consumption when all the `predicated` bytes are also `valid`.

Consuming Vector Registers

The LMMU provides access to the vector registers that are being built in the LFIFO queues. The read pointer is used to feed the data to the processing unit in the correct order. The consumption of a data vector can only occur when the vector pointed by the read pointer is valid, and this consumption is handled through a handshake protocol: the LMMU asserts if the vector is valid, while the processing

unit informs the readiness to receive the data. When the condition is met, the data is transferred and the vector resources in the LFIFO are freed, allowing for more addresses to be reserved and for new data to be mapped (i.e., for more vectors to be built).

4.4.2 Load Request Queue

The Load Request Queue (LRQ) module serves the purpose of accepting the addresses provided by the AGU units and transforming them into necessary memory requests. As mentioned, in traditional data memory, many different addresses are mapped onto the same row of bytes, and the LRQ organizes these requests in a way that addresses with the same memory tag are packed together. For each group of requests with the same memory tag, only one load operation is demanded, an optimization that the module takes advantage of. When the data is provided, the pending requests start to get processed.

A scheme of the architecture of this module is displayed in Figure 4.10.

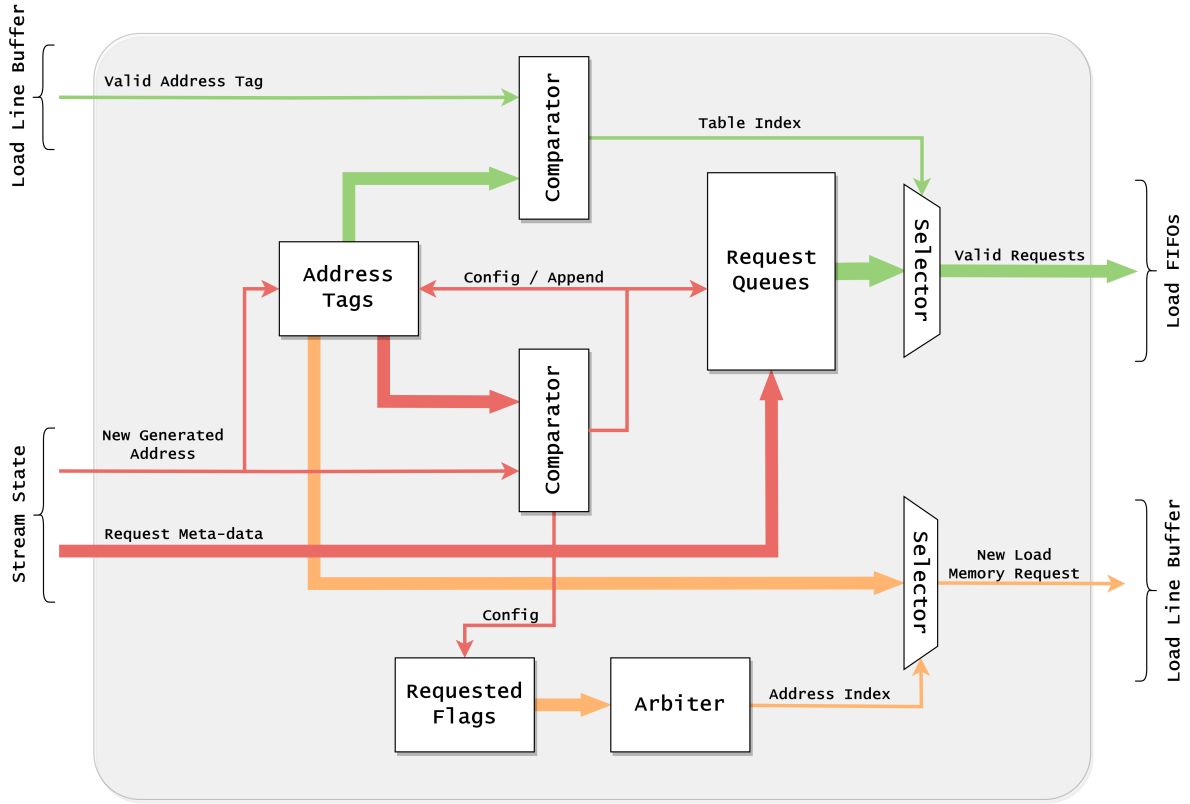

Figure 4.10: Diagram of the Load Request Queue's Architecture

Internal Registers

The LRQ stores every generated address into request bundles of data that can be processed further. Each request in the LRQ is defined by a stream identifier, essentially indicating which stream provided the address. Additionally, the request stores information about the desired data width, i.e., how many bytes are requested. Two essential pointers are associated with each request: one points to the offset for the complete row of data bytes linked to a memory tag, and the other indicates the offset within the LFIFO's buffering queues. This second pointer is vital for determining where to write the data within the data queue, which is sourced from the corresponding write pointer of the LFIFO. Finally, there's a validity bit that signifies whether the request is valid or not.

The LRQ's storage architecture is structured as a collection of tables, and each table is associated with a memory tag. Every table has a fixed number of requests that can be configured. This implies that a request's memory tag is implicitly encoded in the table to which it belongs. Furthermore, each memory tag is expected to generate a load memory request at least once, and this is tracked through a requested bit.

An essential distinction in the LRQ's organization is that, unlike other modules where each stream typically has its dedicated table of registers, the LRQ organizes requests based on matching tags. This means that if different streams provide addresses with matching tags, those requests can be grouped within the same tables in the LRQ. This approach allows for more efficient handling and management of requests, optimizing the use of resources in the LRQ.

Configuring New Requests

To add a new request to the LRQ, there are two primary methods.

1. **Configuration of New Tables**: When a new address with a previously unexplored memory tag arrives, it prompts the LRQ to set up a new table along with the new request. This table becomes associated with the new tag.

2. **Appending Requests to Existing Tables**: In cases where a table already exists for a specific memory tag, new requests with matching memory tags can simply be added to the existing table. The LRQ updates and configures the first request registers that are not yet valid within that table.

Depending on the memory access pattern, a table may not have enough capacity to store all the addresses linked to its corresponding memory tag. If a table becomes full and a new address with the same tag is received, a new table must be configured. This situation can result in multiple tables having identical memory tags. In this scenario, the new table inherits the reserved bit status from other existing tables with the same memory tag to maintain consistency.

An essential point to note is that when a particular stream submits a new memory address, it can only be accommodated or committed by the LMMU if, and only if, both the LRQ and the corresponding LFIFO queue possess available resources to configure this new address.

Triggering Load Operations

As previously mentioned, the LRQ ensures that only one load operation is issued for each memory tag. It achieves this by utilizing the reserved bits within its tables. Additionally, the LRQ is responsible for selecting memory address tags that have not yet been requested. This selection process is managed by an arbiter, which employs a round-robin policy.

The arbiter takes input from combinational logic, specifically, the tables that have been configured but not yet requested. It then outputs a single table, effectively choosing an address tag for a load operation. This chosen tag is subsequently processed by the Load Line Buffer (LLB). Communication between the LRQ and the LLB is conducted through a handshake protocol. If the LLB is capable of accepting the address tag, all tables with the same tag are marked as requested. Subsequently, the arbiter is triggered to select a new table for processing.

Solving Outstanding Requests

The LRQ and the LLB are tightly interconnected. The LRQ requests data associated with specific memory tags, and the LLB furnishes the corresponding data. Effectively, the LLB serves data bytes to the LRQ, which, in turn, utilizes this data to fulfill valid requests.

The LRQ can process one request per cycle, with ”processing” entailing the utilization of data provided by the LLB in conjunction with request information to populate the LFIFO queue with valid data. This action represents a significant step toward supplying data for the processing unit’s operations.

The request-solving process commences when the LLB signals that it is delivering a valid row of data bytes, along with the associated tag, to both the LRQ and the LFIFOs. Given that multiple tables may share the same tag, the LRQ compares the tag delivered by the LLB with the tags associated with its tables and selects the first match. Subsequently, it chooses a valid request from the selected table and forwards the request details to the LFIFO module. The LFIFO module then utilizes this information, along with the data bytes, to copy the correct data into the queue, effectively resolving the request and freeing up resources.

In scenarios where the LLB provides data for a memory tag absent in the LRQ tables (and the AGU modules are not either providing an address with that tag), the LRQ triggers the LLB to switch the row of data it is outputting.

As requests within a particular table are processed and resolved, the table gradually contains fewer valid requests. Ultimately, when no valid requests remain within a table, the table becomes available for

reconfiguration.

4.4.3 Load Line Buffer

The LLB is the module of the LMMU in charge of communications with the memory system. The LRQ communicates with the LLB providing which data from the memory system is required by the memory access pattern of a stream. The purpose of the LLB is to issue load operations to the memory and receive the data, which will be held by it also to solve all the pending requests. The architecture scheme of the LLB is illustrated in Figure 4.11.

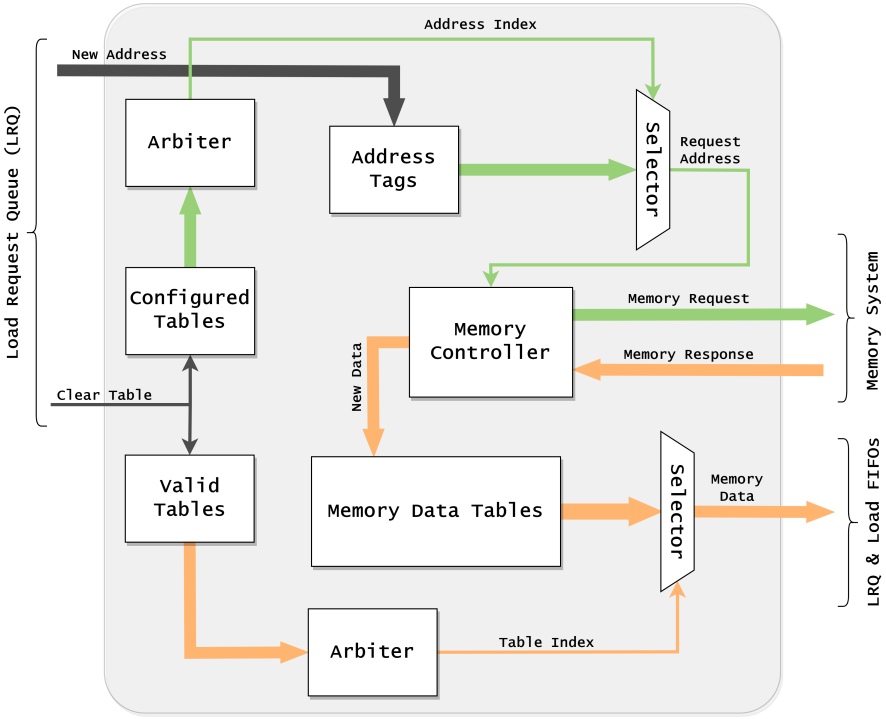

Figure 4.11: Diagram of the Load Line Buffer's Architecture

Internal Registers

This module is responsible for preserving the data retrieved from the memory system following a load operation. Similar to the storage architecture of other modules, the LLB is composed of multiple tables. However, in this case, each table is designed to store a memory word or row from the data memory. Consequently, the width of these tables matches the width of the memory word.

In addition to the data storage, each table in the LLB maintains an associated address tag to ensure proper request handling. Two control bits are associated with each table. The first, labeled as `config`, signifies the validity of the table's address tag. The second, labeled as `valid`, indicates whether the

data within the table is accurate. Data within an LLB table is considered valid for transfer to the other components of the LMMU only when both of these control flags are set to `HIGH`.

Beyond managing table information, the LLB employs registers to facilitate communication with the memory system.

Memory System Communication

The proposed architecture for the SE handles communication with the memory system through the Advanced eXtensible Interface 4 (AXI4) protocol [46].

As mentioned, this module is responsible for making load requests to the memory system. It receives addresses from the LRQ whose data has not been yet loaded. In fact, in this situation, the LLB accepts these new addresses by configuring one of the table registers. This table starts by having the `config` flag to `HIGH` but with the `valid`, which is a combination of states that indicate that this is an address that is scheduled for a load operation. Additionally, the LLB can have at a given time multiple tables in this state. The memory system can only handle one load operation at a time, meaning the LRQ selects one of these through an arbiter. This arbiter also follows a round-robin policy. To keep consistency with the AXI protocol, a FSM is used to control the different steps of a load request through this protocol. It is defined by only two states:

STATE_WAIT_ARREADY: The default state, which is active while no memory operation is being handled. In this state, the LLB waits for two conditions to be verified. The first is that the memory system must be ready to receive and initiate a new load operation. The second is that the arbiter must select a table with a valid address that does not hold yet valid data, meaning they are scheduled for a load request. These two being met triggers the handshake of the AXI4 protocol, which initiates a new load request. It also provokes a transition in the FSM to the `STATE_WAIT_ARREADY` state.

STATE_WAIT_RVALID: During this state, the LLB actively checks if the memory system is providing valid data. One thing important to note is that the width of the AXI data bus can be configured to be smaller than the width of the memory row. This implicitly means that the load operation is composed of multiple bursts of data transfers. For this reason, the LLB will incrementally ready segments of data and load them into the internal storage. When the last burst of data is provided, the LLB updates the `valid` flag, essentially granting a new row to solve pending requests in the remaining modules of the LMMU. When the last burst is received, the FSM changes back to the first state.

Providing Memory Data

When one of the tables within the LLB reaches a valid state, signifying that the corresponding load operation has been completed successfully and all data in the table is accurate, the module is ready to

transfer this data to the remaining modules of the LMMU to address any pending requests that align with the table's memory tag.

Given that the LRQ can only resolve a single request per cycle, the data must be retained in the LLB's internal storage registers until all requests associated with that specific memory tag have been processed. Consequently, the LLB can potentially hold multiple tables in a valid state concurrently. To address this, a round-robin arbiter is employed.

The arbiter's input consists of tables from the LLB that meet two criteria: they are both configured and contain valid data. The selected table can be changed via a trigger signal, which is activated by the LRQ when no further requests are linked to the designated memory tag.

it is important to note that when the LRQ initiates a table change, the LLB interprets this action as a request to release the memory row from its registers. This, in turn, allows for the configuration of another memory tag, which will lead to new data being requested from the memory system.

4.5 Store Memory Management Unit

Similarly to the LMMU, the SE makes use of a SMMU to control how write requests to the memory system are handled.

The architecture scheme of the SMMU is illustrated in Figure 4.12.

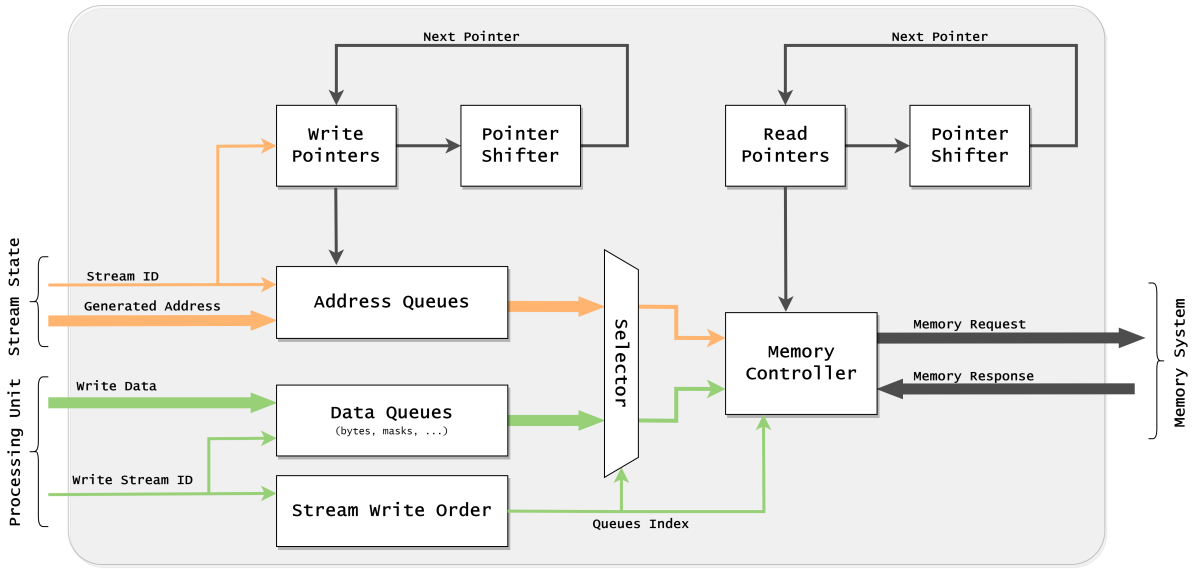

Figure 4.12: Diagram of the Store MMU's Architecture

Internal Registers

The SMMU utilizes various registers with specific functions. Organized in tables, each set of registers corresponds to a particular store stream. These tables consist of two buffering queues, one for addresses and another for data bytes. To facilitate control and tracking, pointer registers are also integral components, aiding in the detection of read and write pointers for both address and data buffers. Since each table is associated with a specific store stream, additional metadata such as the stream identifier, width, and valid flags are used. Each entry in the address and data queues is equipped with a valid bit, allowing for selective masking of valid addresses and data within the buffers. Additionally, because the SE communicates with the memory through the AXI4 protocol, buffering registers are employed to manage and configure the AXI4 write-related signals. Finally, to ensure the proper sequencing of write requests from different streams, a write order queue is maintained, guaranteeing that these requests are processed in the order in which data is delivered to the SE.

Populating the Address and Data Buffers

When a specific store stream is configured within the SE, it generates a sequence of memory access addresses. These addresses correspond to store operations, which entail sending data to the memory. As these addresses are generated, the SMMU sequentially stores them in the address buffers associated with the originating stream. This ensures the order of write operations within a stream. Concurrently, the processing unit performs operations and supplies the SE with the data bytes meant for memory storage. The data is provided with an accompanying mask vector, indicating which bytes are valid and should be written to memory. The SMMU possesses the capability to detect when a particular stream has a valid pair of address and data bytes (the number of which depends on the element width), which is when there is a valid address for every data element. Once such a request is ready for processing, the SMMU has logic that will attempt to issue the request.

Managing Store Requests

As multiple store streams can be processed simultaneously, the SMMU maintains an ordering queue where it registers the identifier of a stream that is providing new data bytes. Subsequently, using a pointer, the SMMU determines which stream should be the next to write to memory. It checks for the presence of a request ready for processing and employs logic to minimize the number of memory transactions. It attempts to buffer all requests that map to the same memory row before actually issuing a store request.

It is important to note that the SMMU communicates with the memory using the AXI4 protocol and utilizes a FSM to control the data flow. The states within this FSM are as follows:

STATE_WAIT_PAIR: This is the initial state where no memory transaction is being processed. Instead, the SMMU actively tries to accumulate valid requests in a buffer, all of which map to the same memory row. A state transition occurs only when the next request carries a different tag, indicating that the accumulated requests need to be issued before accepting the new one.

STATE_WAIT_AWREADY: In this state, the Address channel signals are appropriately set, and the FSM waits for the memory system to assert the ready signals, signifying that a new transaction can be accepted. A state transition occurs when this handshake takes place.

STATE_WAIT_WREADY: Here, the Write Data channel signals are updated, corresponding to the data bytes and their associated byte masks or strobes. The state transitions when the memory system is ready to receive the data, and the handshake occurs.

STATE_WAIT_BVALID: This state involves waiting for all memory response signals. Since transactions may occur in bursts, all response values are registered to detect any failures. Additionally, the FSM changes state only when the last response is received. If all bursts were successful, the FSM returns to the first state, where it resumes the accumulation of requests, now associated with a different memory row (or tag). In case of any failures, the entire request process is repeated.

5

Integration of the Streaming Engine in a Streaming Processor

Contents

As it was mentioned previously, the proposed SE was designed to be integrated with the datapath of a processing core, aiming to enhance its capabilities. Its primary objective is to harness data streaming mechanisms, thus relieving the core of some of its processing workload while simultaneously optimizing memory transfer bandwidth.

To assess the efficiency of the proposed SE, a minimalist processing unit, herein referred to as the Streaming Processor (SP), was designed. In particular, this module comprises a Functional Unit (FU) equipped with vector processing capabilities and the SE detailed in Chapter 4. It also includes two dedicated memories, one for instructions and the other for data. A visual representation of the SP diagram is presented in Figure 5.1.

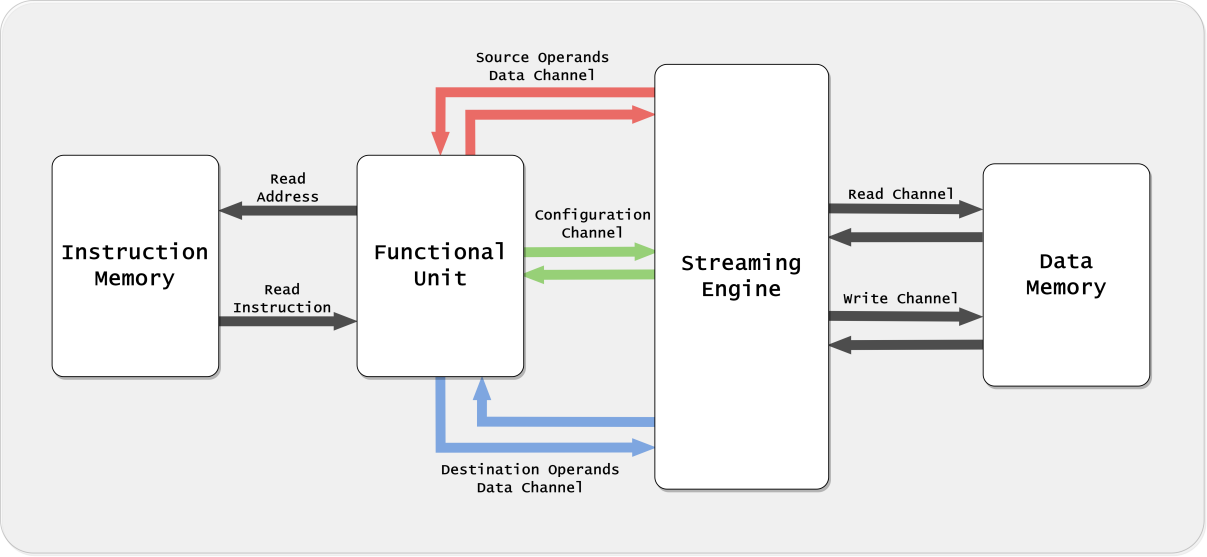

Figure 5.1: Diagram of the conceived Stream Processor's Architecture

The purpose of this chapter is to describe how the SE interfaces with a processing unit. To attain this objective, the SP was devised for testing the fundamental functionality of UVE when operating within a core. This serves as a means to understand how the performance of the SE varies with diverse streams, essentially testing distinct memory access patterns. Through the SP, several benchmark scenarios can be created to elucidate the advantages of the UVE and to unlock the potential of the SE. Despite its simplicity, this design was designed to be easily extended with extra functionalities.

5.1 Streaming Engine

5.1.1 Communication Interface Channels

The SE offers diverse communication channels that facilitate its integration into various systems, provided they adhere to the specified interface signals. A total of four channels can be defined, each with

its unique interface and dedicated purpose based on the capabilities of the SE. These channels include:

1. Configuration Channel: This channel serves as the primary interface for controlling and configuring the SE. It is used to set up and manage streams within the SE, control the stream behavior, and reset its architectural state when necessary.

2. Source Operands Data Channel: Functioning as the access channel for loading streams, this channel allows users to retrieve and consume vector data as source operands from the buffering queues (LFIFOs) within the SE. It is the conduit through which data streams leave the SE. The SE supports the concurrent access of multiple load streams, offering multiple access points to different streams. In the case of the SP, the SE provides two access points, enabling up to two vector streaming data transfers in parallel or, in other words, simultaneous data consumption by the processing unit.

3. Destination Operands Data Channel: Similar to the source operands channel, this channel serves as the access point for storing streams. It allows users to provide data intended for streaming or writing to the memory system, following the access patterns of the streams. This channel is where data enters into the SE. Since streaming operations involve a single destination operand, the SE provides a single access point or a set of signals for transferring vector data to the data buffers of the SMMU within the SE.

4. Memory System Channel: This channel encompasses the interface with the memory hierarchy, and it is used by the SE to execute all data transfers to and from the memory system under the memory access patterns of the streams.

Collectively, these channels enable seamless integration of the SE into a broader system. While it is currently integrated into a relatively simple processor unit, the SE can be integrated into more complex systems to leverage the advantages of data streaming capabilities.

5.1.2 Customizable Architectural Parameters

A key attribute of this architecture is its high degree of parameterization across various aspects. This extensive parameterization enhances the adaptability of the architecture in terms of the allocation of resources. One can choose to configure a SE with a more resource-efficient setup or assign greater importance to specific modules within the SE. Various parameters in the source files are defined, which are listed in Table 5.1.

Naturally, while some of these parameters are associated with the width of signals within the hardware architecture, others directly influence and have a significant impact on the available resources within the SE. These parameters can potentially affect the module's capabilities. Consider, for instance, parameters like `LRQ_NUM_TABLES` and `LRQ_NUM_REQUESTS`. When these values are set to larger numbers, the LRQ can accommodate more address requests associated with load streams. However, this expanded capability comes at a cost, as it requires additional registers and combinational logic, resulting

Table 5.1: Parameters of the Stream Processor and Streaming Engine modules

Parameter	Description
STREAM NUM DIM	Number of dimensions a stream can have
STREAM NUM MODS	Number of modifiers each dimension can have
STREAM OFFSET WIDTH	Width of the offset field of a stream
STREAM STRIDE WIDTH	Width of the stride field of a stream
STREAM SIZE WIDTH	Width of the size field of a stream
STREAM ID WIDTH	Width of the stream identifier signal
LRQ NUM TABLES	Number of LRQ tables
LRQ NUM REQUESTS	Number of requests each LRQ table has
LLB NUM TABLES	Number of LLB tables
LLB NUM BYTES	Number of data bytes each LLB table has
LMMU NUM VECS	Number of vectors each Load FIFO has
SMMU NUM ADDRESSES	Number of addresses each SMMU table has
ADDRESS WIDTH	Width of addresses produced by the SE
VEC WIDTH	Width of each vector in the Load FIFO
NUM SRC OPERANDS	Number of source operands the SE has
AXI R DATA WIDTH	Width of the AXI RDATA bus
AXI W DATA WIDTH	Width of the AXI WDATA bus
MAX NUM LOAD STREAMS	Number of load streams the SE has
MAX NUM STORE STREAMS	Number of store streams the SE has
AXI READ DELAY	The number of cycles a read request takes
AXI WRITE DELAY	The number of cycles a written request takes
FU NUM VECTOR REGISTERS	Number of vector registers
FU NUM SCALAR REGISTERS	Number of scalar registers
FU VECTOR RF WIDTH	Width of the vector register
FU SCALAR RF WIDTH	Width of the scalar register
FU NUM HW INSTRUCTIONS	Number of instructions the HW buffers have
IM DATA WIDTH	Width of the instruction memory output signal
IM ADDR WIDTH	Width of the address of the instruction memory
DM DATA WIDTH	Width of the data memory output signal
DM ADDR WIDTH	Width of the address of the data memory

in increased resource usage as the SE scales and grows.

5.2 Functional Unit

The FU is the processing module that is responsible for decoding the instructions obtained from the Instruction Memory (IM) and performing all the operations according to the instruction type fetched and executing them.

5.2.1 Instruction Set Architecture

The SP is tailored to process the following categories of instructions:

1. **Stream Configuration**: These instructions facilitate the configuration of streams within the SE.

Since the configuration ports are adapted to align with the configuration instructions of UVE, these instructions must be decoded and translated into the corresponding signals that can be interpreted by the SE. Having this in mind, these instructions encode the parameters of the descriptors and may also provide meta-information on the stream that is being configured, such as the width and the direction.

2. Arithmetic Vector Operations: To emulate the execution of UVE, which is a vector extension, some of the supported instructions correspond to vector operations between two vector registers. These instructions need to encode the source and destination operand, as well as other meta-information such as the operation type.

3. Hardware Loops: The SP is equipped to support hardware loops, which allow to specify the repetition of certain blocks of instructions without having to fill the IM with long and repetitive sets of instructions. Instead, these instructions specify that a sub-set of the instructions in the IM should be repeated a fixed number of times. The hardware loop instruction requires in its encoding two inputs: the number of instructions of the loop body and the number of times the loop should be iterated. Implicitly, the instructions of the loop body are the instructions that follow the hardware loop up in the IM.

4. End of Application: This is a single and straightforward instruction used to signify the conclusion of a benchmark. All programs or applications configured within this SP should culminate with this instruction, serving as a definitive endpoint.

Having defined a minimalist ISA for the FU, the architecture of the datapath of the FU was devised and a block diagram of it can be seen in Figure 5.2.

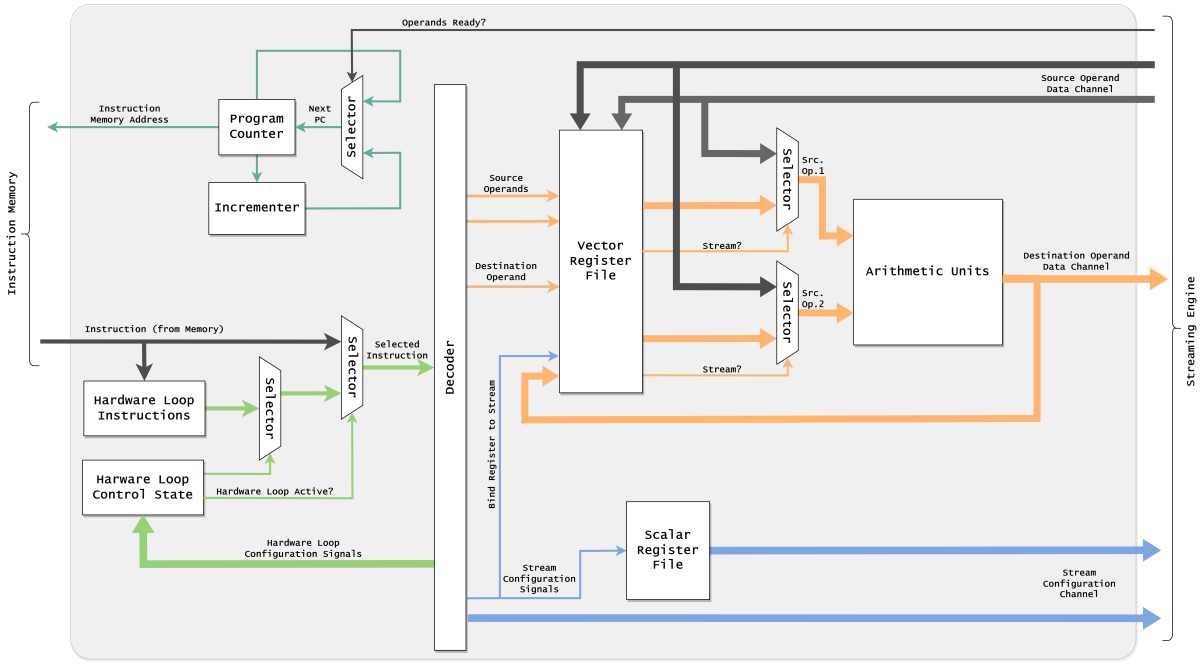

Figure 5.2: Diagram of the Functional Unit's Architecture

5.2.2 Register File

The Register File (RF) within the FU comprises both vector and scalar registers, primarily serving as temporary data storage.

Scalar registers are primarily used to store parameters for streams, hardware loop configurations, and other scalar values. Some of the instructions that the FU can decode contain fields indicating which scalar register in the RF should be accessed to read these parameters. As a result, these source registers must be pre-loaded to ensure the proper execution of the application.

In contrast, vector registers have a wider data width compared to scalar registers, and they are employed within the FU to hold data for vector operations. Each vector register can be associated with a stream processed by the SE. To determine whether a vector register is tied to a specific stream, each register contains a flag conveying this information. These registers enable arithmetic operations to be executed in a SIMD fashion.

Furthermore, additional registers are essential for managing the flow of the application. These include the Program Counter (PC) and other buffering registers related to hardware loop functionality and management.

5.2.3 Decoder

The application's instructions are encoded with fixed-length words. However, these instructions alone lack the needed information to directly control operations within the FU. To bridge this gap, the instructions are decoded in the decoder module.

This module takes an instruction word as its input and leverages the instruction's opcode to decide on how to translate the information into control signals. While some control signals, such as register operand identifiers, are directly mapped from the instruction, others may depend on the specific opcode. For instance, if the opcode corresponds to a stream configuration instruction, the decoder incorporates combinational logic to distinguish whether the newly configured descriptor is a static modifier or not.

5.2.4 Arithmetic Units

The FU is equipped with multiple arithmetic units, with each unit tailored to a specific element width. As a result, depending on the configured width of vector operands, the arithmetic results are multiplexed, ensuring that only the correct outcome is stored in the destination vector operand.

It's crucial to understand that the data used in arithmetic operations within the FU can vary depending on whether the operands are associated with a stream. When operands are not stream-bound, the FU straightforwardly employs the data stored in the source operands and directs it to the arithmetic units. Subsequently, the results are similarly written back to the destination register.

In contrast, when operands are stream-bound, the data flow follows a different path. For source operands, which correspond to load streams, the FU must consume data from the respective LFIFO queue of the SE. This process is executed through a handshake protocol over the source operand channel of the SE. If there is available vector data in the SE for consumption, the FU redirects that data to the arithmetic units while simultaneously writing it to the RF.

In the case of destination operands, associated with store streams, the FU must ensure that the results are not only saved in the RF but also transferred to the SMMU. This transfer is facilitated through a handshake protocol over the destination channel.

When a specific arithmetic instruction, decoded by the FU, involves stream operands that are not yet ready to transfer data, the FU is required to stall the program execution until the SE iterates on the running streams and the data flow becomes available.

5.2.5 Hardware Loops

When the application is executed on the SP and employs hardware loop instructions, the FU is responsible for interpreting the data within these instructions to configure and manage the hardware loop process. These instructions primarily provide information about the number of instructions and the number of iterations required for the loop. The FU must buffer this data in internal local buffers and subsequently read and process it repeatedly until the specified number of iterations are completed.

To set up a hardware loop, the FU begins by asserting a register that represents the activation of a hardware loop. The value of this register influences the control signals, which in turn affect the source of instruction data for the program. If the hardware loop is active, the instructions are fetched from the local buffers; otherwise, they are retrieved from the IM.

Initially, when the instruction buffers are empty, the FU continues fetching instructions from the IM. Simultaneously, it copies these instructions into the instruction buffers. Once the instructions for the loop body are exhausted, the FU switches to reading from the buffers, executing the loop until the hardware loop's specified number of iterations is complete. After this, it resumes the normal flow of instruction fetching.

5.3 Instruction and Data Memories

An IM is used to store the encoded instructions of the application to be executed in the SP. This memory is a straightforward dual-port memory whose word width matches the width of instruction encodings of the applications built to simulate the basic functionality of UVE. This memory, integrated into the SP, should only be configured with the ISA of the FU, as other encodings will not be interpreted by the SP.

The Data Memory (DM) serves as another crucial memory component within the SP. Unlike the IM designed for program instructions, the DM is dedicated to loading and storing data from applications running on the SP. Due to the SE architecture's reliance on the AXI4 protocol for communication with the memory system, the DM exhibits a higher level of complexity compared to the IM. Although it is also a dual-port memory, it includes a controller responsible for managing the AXI4 communication with the SE. In this context, the DM assumes a subordinate role while the SE acts as the master. Consequently, the memory module employs two FSM to oversee the subordinate side of the protocol. One FSM handles the read transactions, while the other is responsible for managing the write operations. As a result, applications configured with both load and store streams can concurrently read from and write to the memory, facilitating simultaneous data access and storage.

6

Experimental Evaluation

Contents

This chapter is dedicated to evaluating the performance of the designed SE when dealing with various memory access patterns. The SE module is highly configurable, and one objective is to gain insights into how its performance varies under different configurations.

The chapter begins with an overview of the development environment and the evaluation process, particularly how testing was conducted and how results were extracted.

Subsequently, it provides an in-depth analysis of the performance metrics collected from the SE's performance counters, which offer valuable insights into the operational efficiency of the module. With these metrics established, the SE, integrated within the SP, undergoes a series of tests covering diverse scenarios.

Following the performance metrics analysis, discussions aimed at exploring the implications of the results take place, focusing on understanding the factors contributing to the observed outcomes, and identifying potential paths for optimization and improvement.

This comprehensive exploration within the chapter aims to obtain a comprehensive perspective on the SE's capabilities, limitations, and potential avenues for future development.

6.1 Development and Evaluation Environment

The Hardware Description Language (HDL) that was used in the development of all the described hardware was Chisel, which is embedded in the Scala programming language. Chisel provides high-level constructs that allow the creation of reusable and parameterizable components. Ultimately, hardware developed in Chisel can be translated to Verilog, which can then be used to synthesize the circuit either in Field Programmable Gate Array (FPGA) technology or in ASIC technology. In other words, this means that designs written in Chisel can be directly synthesized to run on any technology process.

In the design process, testing was conducted using Chiseltest [47], a framework designed for the formal verification of Chisel designs. This framework was used to validate the designs through tests that integrate Verilator [48], a widely-used Verilog simulator that translates synthesizable Verilog code into C++ or SystemC, enabling efficient and accurate simulation of digital designs.

In practice, the hardware's simulation and result extraction were achieved by creating tests with Chiseltest, which would generate the corresponding Verilog hardware description. These descriptions were subsequently processed by Verilator to produce precise hardware behavior simulations.

6.2 Performance Counters and Metrics

As a way of measuring the efficiency, the SE is equipped with several performance counters. Then, when carrying out simulations through Verilator, these values can be extracted and processed.

6.2.1 Performance Counters

Throughout the SP there are a total of twelve performance counters, each holding different information. These are listed in Table 6.1, along with a brief description of each one. The performance counters are distributed across some of the sub-modules of the SE and they range from the number of stalls caused by each of the MMUs to the number of memory operations executed by each.

Table 6.1: Description of the Performance Counters

Performance Counter	Description
HIGHER ITERATIONS	Number of cycles iterating higher dimensions
LFIFO STALLS	Number stalls triggered by the LFIFO
LRQ STALLS	Number stalls triggered by the LRQ
LLB STALLS	Number stalls triggered by the LLB
LMMU STALLS	Number of stalls triggered by the LMMU
LMMU COMMITS	Number of accepted addresses by the LMMU
SMMU STALLS	Number of stalls triggered by the SMMU
SMMU COMMITS	Number of accepted addresses by the SMMU
FU COMMITS	Number of committed instructions in the FU
FU STALLS	Number of stalls in the FU
LOAD REQUESTS	Total number of load request operations
STORE REQUESTS	Total number of store request operations

6.2.2 Performance Metrics

From the implemented performance counters, some relevant metrics can be inferred, which bring even more relevant insights into the performance of the SE regarding how it processes different streams.

6.2.2.A Average Address Generation Rate

The Average Address Generation Rate (AAGR) is a performance metric that measures the rate at which memory addresses are generated by the AGU related modules. When an address is generated, the corresponding MMU must have resources to accept it. This metric also considers the stalls issued by the MMUs. It can be derived according to the following equation:

$$\texttt{Average Address Generation Rate} = \frac{\texttt{Address Enqueuing Cycles}}{\texttt{Total Number of Cycles}} \tag{6.1}$$

The `Address Enqueuing Cycles` is derived from the `LMMU COMMITS` or `SMMU COMMITS` performance counters, depending on whether the stream generating the addresses is a load or store stream, respectively. The `Total Number of Cycles` is the sum of `LMMU COMMITS` and `LMMU STALLS` for load streams and similarly for the `SMMU` performance counters for the store streams.

Hence, this metric quantifies how quickly the system can produce addresses of a stream and dispatch them to corresponding buffering queues. A higher average address generation rate indicates a more efficient address issuing process, meaning that little overhead exists in the address generation process and also few stalls are introduced from the MMUs. The higher this metric is, the more efficient the system is.

6.2.2.B Data Re-utilization Ratio

The Data Re-utilization Rate (DRR) is a performance metric that quantifies the efficiency of data reuse within the memory data buffers, namely the LLB, and is obtained through the following equation:

$$\texttt{Data Re-utilization Ratio} = \frac{\texttt{Number of Addresses Generated}}{\texttt{Number of Memory Operations Executed}} \tag{6.2}$$

The `Number of Addresses Generated` corresponds directly to the `LMMU COMMITS` or `SMMU COMMITS` performance counters, depending on the stream direction. The `Number of Memory Operations Executed` is directly inferred by the `LOAD REQUESTS` or `STORE REQUESTS` performance counters.

This metric is determined by the fraction of the number of addresses generated by the memory access pattern of the stream over the number of memory operations required to transfer data. In the worst-case scenario, for every address generated, a memory operation (load or store) needs to be executed, which means the higher this ratio is, the more efficient the hardware is.

This metric can be computed both in the perspective of load and store memory operations. A high Load DRR suggests that the data buffered in the internal registers of the LLB are frequently being used to solve distinct requests, meaning a single row of data from memory is sufficient to solve many data requests. A high Store DRR suggests that multiple store requests are first organized and grouped before initiating a memory request, buffering them instead of issuing a store operation for every address.

6.2.2.C Address Generation Efficiency with Descriptor Iteration Ratio

The Average Address Efficiency with Descriptor Iteration Ratio (AGEDIR) is a performance metric that evaluates how efficiently memory addresses are generated concerning the iteration of descriptors.

It is calculated as the fraction of the number of addresses generated over the total number of descriptor iteration cycles, as the following equation describes:

$$\begin{matrix}\texttt{Address Generation Efficiency} \\ \texttt{with Descriptor Iteration Ratio}\end{matrix} = \frac{\texttt{Number of Addresses Generated}}{\texttt{Number of Non-Stalling Cycles}} \tag{6.3}$$

Similarly to the DRR metric, the `Number of Addresses Generated` corresponds directly to the `LMMU COMMITS` or `SMMU COMMITS` performance counters, depending on the stream direction. The `Number of`

`Non-Stalling Cycles` is obtained by summing the `HIGHER ITERATIONS` and the `LMMU` or `SMMU COMMITS` performance counters.

The address generation is initiated during the iteration of the first descriptor, while the remaining descriptors must be iterated to maintain the desired memory access pattern. During the iteration of these additional descriptors, no new addresses are being generated, which can lead to cycles being potentially wasted.

This metric quantifies the effectiveness of this process by assessing the ratio of actual address generation cycles to the total number of descriptor iteration cycles. A higher ratio indicates a more efficient address-generation process, while a lower ratio suggests that a significant portion of cycles is spent iterating descriptors without generating new addresses. It helps identify potential inefficiencies in the address-generation process and provides insights into how effectively the system utilizes cycles when iterating descriptors.

One thing to note is that, in contrast with the AAGR, this metric does not take into consideration the stalls triggered by the MMUs, associated with the fact that they are not always ready to accept the generated addresses.

6.3 Performance Evaluation

With the performance metrics and counters in place, one can evaluate the proposed SE with a variety of benchmarks, extract the metrics, and elaborate possible explanations. Each one of the benchmarks that will be considered is an application loaded into the IM of the SP, which will make use of the SE to exploit data streaming while also performing DLP with SIMD arithmetic operations. This evaluation provides insights on how the SE would perform in a hardware system employing UVE.

6.3.1 Testbenches

As an initial phase in evaluating the architecture designed for the SE, this work exposes it to a series of testbenches where specific memory access patterns are configured. This procedure serves to assess particular hardware characteristics, primarily focusing on its efficiency in reusing data and generating addresses, both in isolation and in the presence of concurrent data streams. Then, it is tested how having different streams running concurrently impacts the performance of the streams involved. Given the high degree of customization allowed by the hardware's parameters, these testbenches explore various parameter configurations to gain insights into how scaling the hardware affects its overall performance.

6.3.1.A Single-Descriptor Applications

These applications involve the use of a single stream, which is defined by a single descriptor, to access memory positions. The key variation among these applications lies in the stride parameter within the descriptor, which will differ between applications. Essentially, these applications are designed to evaluate the hardware's efficiency in retrieving and reusing data, both for loading and storing data.

Depending on the direction of the stream (i.e., load or store), different MMUs modules will be used. The primary objective is to assess whether performance varies when the same memory access patterns are applied to streams with different stream directions. A pseudo-code representation of these applications is provided in Figure 6.1.

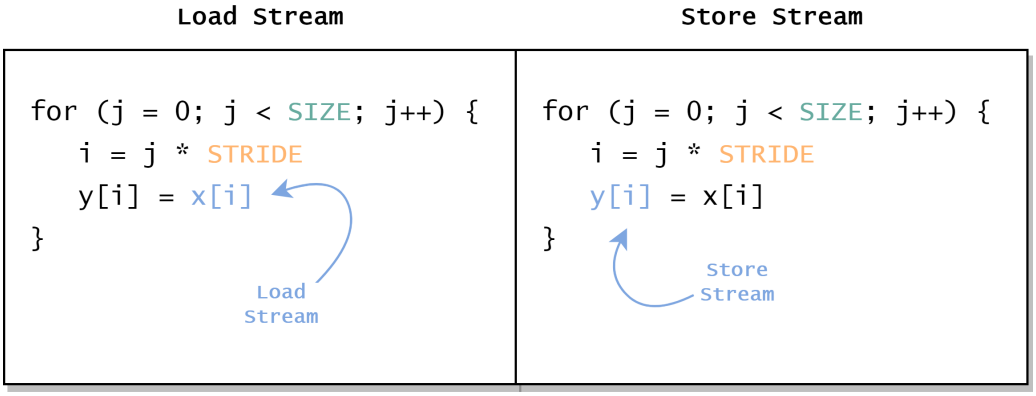

Figure 6.1: Pseudo-code for Single-Descriptor Testbenches

At this stage, the focus is not on running multiple data streams concurrently. Instead, each application configures only a single data stream. According to the pseudo-code, there are two arrays: `x[.]` and `y[.]` When examining the behavior of a load stream, the application exclusively configures a load stream on `x[.]`, while a store stream is configured for `y[.]` when assessing the store stream behavior. Although these arrays have different base offsets, the memory access patterns remain consistent for both scenarios. In both cases, the `stride` parameter varies, while the `size` remains constant across all tests.

6.3.1.B Multi-Descriptor Applications

As memory access patterns adhere to stricter rules, the number of descriptors required to describe them increases. This architectural characteristic of the SE may impact the system performance due to the overhead associated with the iteration of the different descriptors within a stream.

In contrast to the previous category of applications, multi-descriptor applications employ multiple descriptors to represent specific memory access patterns. These applications encompass different

memory access patterns, each described by a varying number of descriptors. While the number of addresses generated by each application may differ, fair comparison takes into account ratio metrics. Additionally, in these applications, only one stream continues to be configured, as this could influence the overall results and is beyond the scope of this evaluation stage.

The purpose of these tests is to evaluate the hardware's overhead when it comes to iterating through descriptors of multi-dimensional memory access patterns. This assessment provides insights into the system's performance when dealing with complex memory access requirements. A pseudo-code representation of these applications is provided in Figure 6.2.

Type A

```
for (i = 0; i < SIZE1; i+= STRIDE1) {
  for (j = 0; j < SIZE2; j+= STRIDE2) {
    for (k = 0; k < SIZE3; k+= STRIDE3) {

      y[i][j][k] = x[i][j][k]

    }
  }
}
```

Type B

```
for (i = 0; i < SIZE1; i+= STRIDE1) {
  SIZE2++
  for (j = 0; j < SIZE2; j+= STRIDE2) {

    y[i][j] = x[i][j]

  }
}
```

Type C

```
for (i = 0; i < SIZE1; i+= STRIDE1) {
  for (j = 0; j < SIZE2; j+= STRIDE2) {
    STRIDE3++; SIZE3++
    for (k = 0; k < SIZE3; k+= STRIDE3) {

      y[i][j][k] = x[i][j][k]

    }
  }
}
```

Figure 6.2: Pseudo-code for Multi-Descriptor Testbenches

Type A is described by only dimensional descriptors {`Offset, Stride, Size`} = {`0, STRIDE`**x**`, SIZE`**x**}, where **x** is the nested level of the loop. In addition to these descriptors, the second dimension of Type B testbenches has the static modifier {`Target, Behaviour, Displacement, Size`} = {`size, add, 1, SIZE1`}. Type C has three dimensional descriptors and two modifiers in the second dimension: {`stride, add, 1, SIZE2`} and {`size, add, 1, SIZE2`}.

As shown in the pseudo-code, the number of nested loops varies in the considered applications. Applications with a high number of nested loops are associated with the configuration and processing of a stream with the same number of dimensional descriptors. Each descriptor introduces new behaviors in the memory access pattern. These applications can be categorized into three types:

- **Type A**: Each for loop introduces a new `stride` field, as well as a new `size`, which can be mapped to the descriptor of that dimension. There are no static modifiers in this type of application.
- **Type B**: Corresponds to streams that require the use of static modifiers. As the pseudo-code demonstrates, the outermost loop introduces changes in the first inner loop's header parameters, namely the number of iterations. Translating this to descriptors, it means that each dimension has a static modifier descriptor attached, which increments to the first descriptor's `size` field.
- **Type C**: Similar to type B applications, but the middle loop introduces two changes in the first loop. In terms of the descriptor paradigm, when configuring the corresponding streams, this is equivalent to having dimensions with two static modifiers configured, where the `stride` and `size` fields of the first descriptor are incremented.

In summary, the number of descriptors in some dimensions increases from type A to type C applications. Type A applications consider different numbers of dimensions (i.e., nested for loops). Consequently, there will be testbenches of type A that have the same number of descriptors as other testbenches of type B or C, but the descriptor types will be different.

6.3.1.C Single vs. Multiple Stream Applications

The previous application categories introduced a diverse number of streams for the SE to process, involving more than one data source that can be configured as streams. In particular, Sections 6.3.1.A and 6.3.1.B describe applications that utilize two arrays, `x[.]` and `y[.]`.

The experimental evaluation proceeds in two stages. In the first stage, only one stream is configured, allowing for the testing of individual stream performance. The second stage evaluates the performance of the same applications with both array sources configured as streams. For this reason, the testbenches described in the sections above will also be tested having streams configured for both `x[.]` and `y[.]` arrays.

One important note is that the base address of the arrays of these applications is now relevant when considering data re-usage aspects. For this reason, different base addresses are also tested to assess their impact when multiple streams are employed concurrently.

6.3.2 SAXPY Benchmark

In addition to testing the designed SE architecture with specific testbenches tailored to certain characteristics, the hardware's behavior was also assessed using a real-world application, namely the Single Precision A*X Plus Y (SAXPY) benchmark.

SAXPY is a widely used benchmark in HPC research for evaluating system processing capabilities. It involves a simple linear algebra operation commonly employed in numerical computing and parallel programming. This operation combines two arrays and a scalar, storing the result in one of the source arrays. A pseudo-code representation of the SAXPY benchmark is provided in Figure 6.3.

SAXPY Benchmark

```
for (i = 0; i < SIZE; i++) {
   y[i] = A * x[i] + y[i]
}
```

Figure 6.3: Pseudo-code for the SAXPY Benchmark

In the context of data streaming, this benchmark can be conceptualized as having a one-dimensional memory access pattern but utilizing three streams: two load streams for computing results and a third store stream to write back the computations to memory. All streams can be described using a single-dimensional descriptor, but configured with different parameters. In this case, the dimensional descriptor {`Offset, Stride, Size`} for the load streams are defined as {`&x[0], 1, SIZE`} and {`&y[0], 1, SIZE`}, where only the base offset varies. The store stream descriptor is identical to the latter descriptor, the only difference being the direction defined for the stream.

6.3.3 Results and Discussion

The described testbenches and benchmarks were executed on the proposed architecture of the SE, and their corresponding results were gathered and compared. In this section, discussions and conclusions are made based on the obtained results.

Parameter Configurations

For testing purposes, various parameterizations of the hardware were evaluated to assess their impact on performance. All configurations share certain constant parameters, which are detailed in Table 6.2.

Table 6.2: Streaming Engine's Fixed Testing Parameter Values

Parameter	Value	Parameter	Value
STREAM_NUM_DIMS	4	MAX_NUM_LOAD_STREAMS	2
STREAM_NUM_MODS	3	MAX_NUM_STORE_STREAMS	2
STREAM_OFFSET_WIDTH	32	FU_NUM_VECTOR_REGISTERS	16
STREAM_STRIDE_WIDTH	32	FU_NUM_SCALAR_REGISTERS	32
STREAM_SIZE_WIDTH	32	FU_VECTOR_RF_WIDTH	128
LLB_NUM_BYTES	32	FU_SCALAR_RF_WIDTH	32
AXI_R_DATA_WIDTH	128	FU_NUM_HW_INSTRUCTIONS	8
AXI_W_DATA_WIDTH	128	IM_DATA_WIDTH	32
AXI_READ_DELAY	10	IM_ADDR_WIDTH	6
AXI_WRITE_DELAY	10	DM_DATA_WIDTH	256
SMMU_NUM_ADDRESSES	32	DM_ADDR_WIDTH	10

When evaluating the results, it is also crucial to take into account key metrics such as bus widths during communication with the memory system and the latency of read and write operations. In this experimental evaluation, the latency of memory operations was defined as 10 cycles and the bus widths were set to 128 bits, as seen in Table 6.2.

In certain configurations of the SE, specific parameters were altered to investigate their potential impact on performance. The SE configuration sets that were considered in the aforementioned tests are enumerated in Table 6.3.

In transitioning from Config A to Config B, all four parameters experience an increase in size. In

Table 6.3: Streaming Engine's Varying Testing Parameter Values

	Config A	Config B	Config C	Config D
LRQ_NUM_TABLES	2	4	16	4
LRQ_NUM_REQUESTS	2	4	4	16
LLB_NUM_TABLES	1	2	2	2
LMMU_NUM_VECTORS	2	4	4	4

contrast, Configs C and D exhibit changes in only one parameter compared to Config B. This deliberate setup aims to isolate and observe the individual impacts of specific parameters on the results. In Config C, the number of tables in the LRQ is augmented, and in Config D, it is the number of requests within each LRQ that sees an increase.

Increasing Stride between Memory Accesses

To evaluate the performance of the SE when configured with a single stream, the simplest one-dimensional testbenches of section 6.3.1.A were employed. For better performance comparison, the parameters specified in the Config D set from the group of existing configurations in Table 6.3 were adopted.

For this particular stream, the SE was subjected to various `stride` values, incrementing by powers of 2, totaling six different values. Additionally, different `size` values were considered, with the elements having a width of one. The performance counters registered upon its execution are presented in Table 6.4. With these performance counter measurements, it is possible to compute essential performance metrics, which are depicted in Figure 6.4.

Table 6.4: Relevant Performance Counters on Single Descriptor Testbenches

		Load Stream					Store Stream	
Size	Stride	Load Ops.	LFIFO Stalls	LRQ Stalls	Size	Stride	Store Ops.	SMMU Stalls
1000	1	32	16	0	1000	1	63	930
	2	63	43	0		2	125	1830
	4	125	0	832		4	250	3645
	8	250	0	2700		8	500	7275
	16	500	0	6446		16	1000	14535
	32	1000	0	13944		32	1000	14535
25000	1	782	766	0	5000	1	313	4930
	2	1563	1529	0		2	625	9580
	4	3125	0	21832		4	1250	18895
	8	6250	0	68700		8	2500	37525
	16	12500	0	162446		16	5000	74785
	32	25000	0	347582		32	5000	74785

Upon analyzing the results, it becomes evident that the SE effectively reuses data when issuing load requests, as the number of load operations is often lower than the total number of addresses generated. The performance counters and the DRR metric underscore that as the `stride` increases, the number of load operations also increases, even when the testbench size remains constant. This

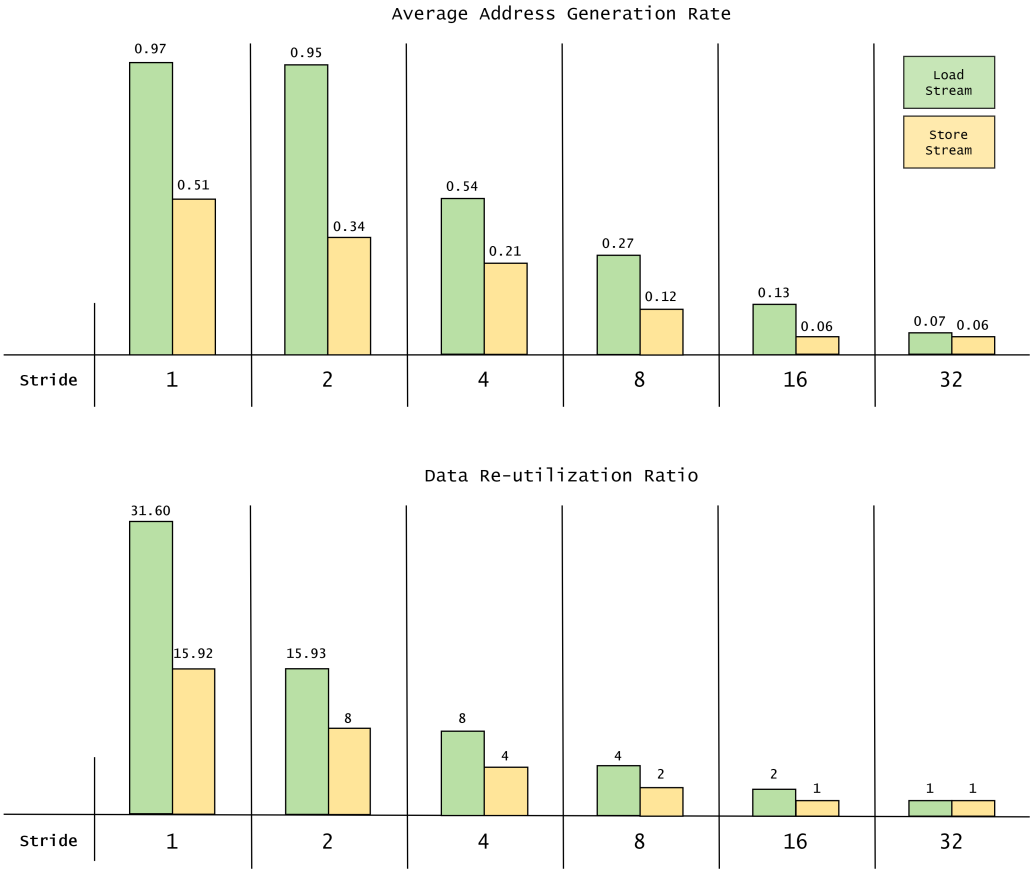

Figure 6.4: Performance Metrics on Single Descriptor Testbenches

phenomenon occurs because the memory rows transferred to the LLB's buffers contain progressively fewer elements from the memory access pattern. As the stride increases, it scatters the elements more, necessitating access to different memory regions. This shift in behavior also explains why stalls transition from occurring at the LFIFO to the LRQ: when elements are contiguous or very close, the LRQ tables can accommodate numerous requests, as many of them share a matching address tag. However, as the stride increases, the address tags change more rapidly, causing the table configurations to fill up faster.

A similar trend is observed with store streams. When consecutive elements map to the same memory row, the SMMU buffers these elements before issuing a store request, resulting in fewer store operations.

For both store and load streams, when the `stride` becomes large, the advantages of data re-utilization cannot be harnessed, and each address inevitably requires a memory operation. The latency associated with these operations leads to bottlenecks between the memory system and the MMU's buffers, ultimately diminishing the AAGR.

Increasing the Number of Descriptors

More intricate, multi-dimensional streams, were examined as well. The results were gathered while considering only one dimension, for reference, and extended to include up to three dimensions, corresponding to Type A testbenches (see Figure 6.2). In these tests, there are no static modifiers, and for each descriptor added, its parameters are defined as {`offset, stride, size`} = {`0, 0, 10`}. The results are presented in Table 6.5 for both load and store streams. The resulting performance metrics can be found in Figure 6.5.

Table 6.5: Performance Counters on Multiple Descriptor Testbenches (Without Static Modifiers)

			Load Stream			Store Stream	
Num Dims	Size 1st Dim	Num. Addresses	Load Ops.	LFIFO Stalls	LRQ Stalls	Store Ops.	SMMU Stalls
1	25	25	1	0	0	2	0
	50	50	2	0	0	4	0
	100	100	4	0	0	7	34
2	25	250	17	0	27	24	271
	50	500	26	18	32	40	527
	100	1000	40	36	11	70	1008
3	25	2500	175	22	490	250	3508
	50	5000	267	161	606	400	5929
	100	10000	400	415	154	700	10741

In the case of the simplest one-dimensional streams, the SE optimizes performance for both load and store streams, efficiently reusing data and generating one address per cycle without encountering any stalls. This success can be attributed to the stream's highly cache-friendly memory access pattern and the sufficiently large parameter dimensions, which allow all addresses to be buffered while memory requests are processed.

As additional dimensions are introduced, some irregularity is introduced into the pattern. Naturally, with the increase in the number of addresses, the number of load and store operations also increases. However, the SE continues to achieve favorable AAGR and DRR for both load and store streams.

In the case of load streams, when the first dimension has a larger `size` field, the SE can generate addresses more efficiently. This is reflected in the AAGR metrics and can be explained by the reduced transitions between descriptors. In fact, with 25 iterations, dimension transitions tend to occur more frequently, leading to less data re-utilization and more stalls. Furthermore, with a smaller number of iterations, the impact of iterating through higher descriptors becomes more noticeable, as evidenced by the increase in the AGEDIR metric, which also rises with the `size` of the first dimension.

In the case of store streams, adding more dimensions significantly reduces the performance of the SMMU. Notably, the SMMU can only buffer one vector of write data in its state registers for each store stream, whereas the LMMU allows multiple vectors of read data to be buffered. This difference leads to larger data flows and reduced bottlenecks. Consequently, store requests are processed more slowly, filling the address queues of the SMMU more quickly.

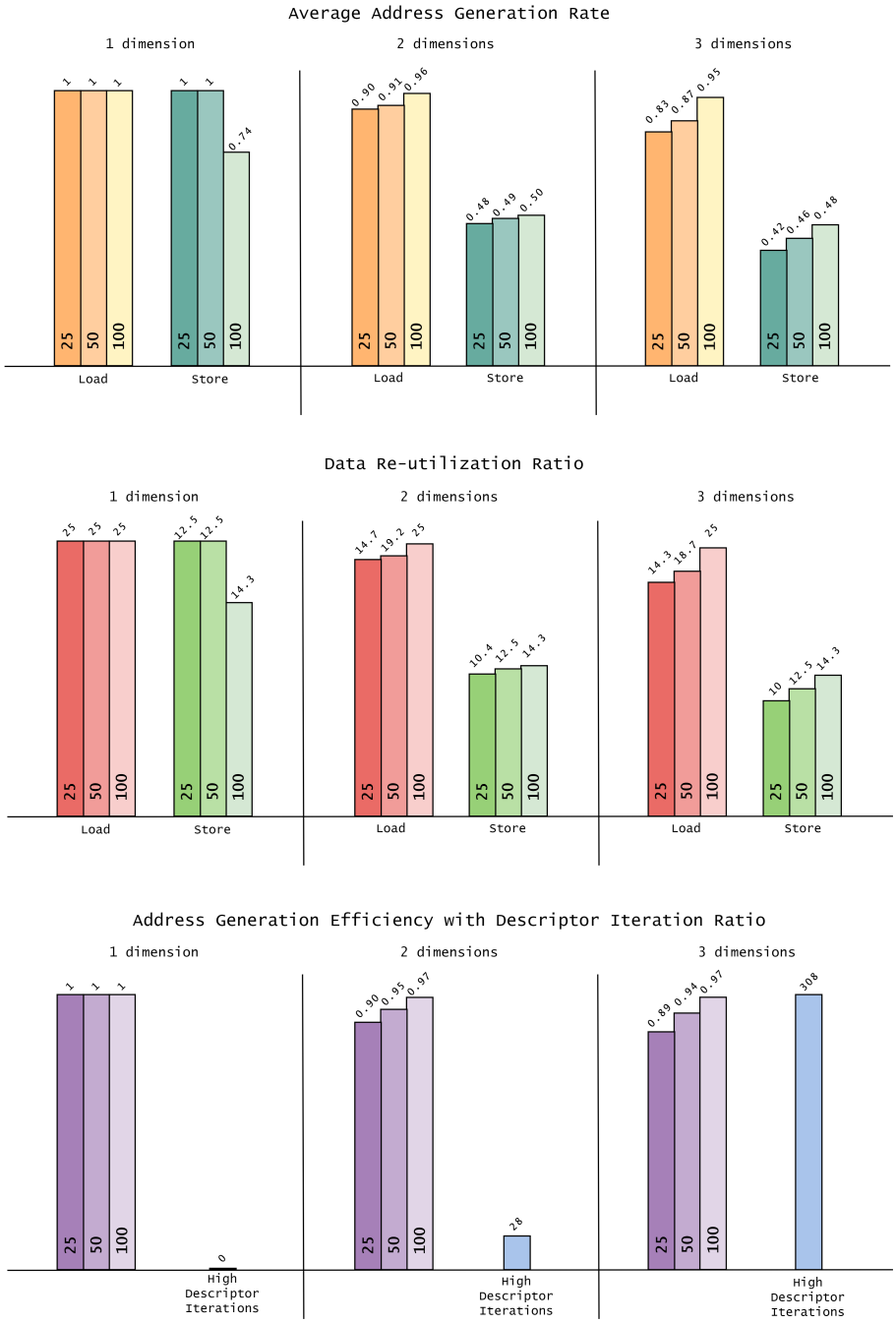

Figure 6.5: Performance Metrics on Multiple Descriptor Testbenches (Without Static Modifiers)

Incorporating Static Modifiers into Configured Dimensions

The introduction of modifiers leads to variations in the behavior of accesses during dimension iterations, consequently affecting system efficiency. To explore these effects, Type B and Type C (see Figure 6.2) configurations are also examined. Irrespective of the modifier's target, its `size` remains constant at 10,

matching the `size` of the dimension descriptor.

The performance counter measurements for each combination are provided in detail in Table 6.6. The resulting metrics are represented in the graph diagrams of Figure 6.6.

Table 6.6: Performance Counters on Modifier Testbenches with a Single Stream Configured

			Load Stream			Store Stream	
Testbench	Size 1st Dim	Num. Addresses	Load Ops.	LFIFO Stalls	LRQ Stalls	Store Ops.	SMMU Stalls
Type B	25	295	18	3	12	26	295
2 dimensions	50	545	25	3	4	42	551
1 modifier	100	1045	41	34	0	73	1047
Type C	25	2950	607	0	5722	1115	16076
3 dimensions	50	5450	1040	13	9869	1979	29207
2 modifiers	100	10450	1903	222	17955	3694	55245

The initial observation reveals that, as both testbenches incorporate `size` modifiers, the total number of addresses increases in all results. However, when examining the Type B testbench results, the metrics exhibit minimal variation and maintain a consistent pattern when compared to Type A. This outcome is expected since the modifier primarily increases the `size` field, but data re-utilization efficiency remains intact, as the new elements remain contiguous. In contrast, Type C testbench results tell a different story. In this case, the performance significantly deteriorates for both metrics. This decline can be attributed to the presence of both `stride` and `size` static modifiers combined within a dimension. As the stream progresses, consecutive elements become more sparse, leading to poor re-utilization of memory data.

Simultaneous Processing of Load and Store Streams

In a more realistic scenario, the application involves at least one load stream to retrieve data from memory and at least one store stream to write results back to memory. The SE can handle multiple streams simultaneously, and both Type B and Type C testbenches were configured with both load and store streams. By comparing these results to those of the single-stream scenarios, we can assess the impact of concurrent streams on performance. The performance counter measurements for these configurations are presented in Table 6.7, and the resulting performance metrics are depicted in the bar graphs of Figure 6.6.

Table 6.7: Performance Counters on Modifier Testbenches with Both Streams Configured

			Load Stream			Store Stream	
Testbench	Size 1st Dim	Num. Addresses	Load Ops.	LFIFO Stalls	LRQ Stalls	Store Ops.	SMMU Stalls
Type B	25	295	25	7	0	26	49
2 dimensions	50	545	38	13	0	42	41
1 modifier	100	1045	62	32	0	73	52
Type C	25	2950	727	5961	254	1115	6520
3 dimensions	50	5450	1246	11134	361	1979	11861
2 modifiers	100	10450	2406	21224	825	3694	22362

One notable difference in the results is the increased number of load operations, indicating a de-

crease in data re-utilization. The interaction between both streams influences how data is reused within the SE. The loaded data undergoes arithmetic operations in the FU, which is then written back as data for the store stream. Consequently, the load stream's efficiency is directly impacted by the bottlenecks associated with the store stream, and vice versa. The most significant bottlenecks are associated with the store stream since the SMMU only allows for one vector to be buffered concurrently. For this reason, the load stream often needs to pause address generation. This results in weaker data re-utilization, as the memory rows requested by the LMMU handle fewer requests due to the increased stalls. By modifying the structure of the SMMU so that each store stream can pre-load multiple vector elements of data, one can mitigate this bottleneck.

Conversely, because both streams are interconnected, the number of stalls issued by the SMMU also decreases. This occurs because only one stream is processed in a given cycle, yet both MMUs operate concurrently. As a result, while load stream addresses are being generated, the SMMU is dispatching data and freeing resources. Consequently, when it switches back to processing the store stream, the need for stalls is reduced.

Adjusting the Streaming Engine's Configurable Parameters

To assess the impact of varying the hardware parameters on performance, the SAXPY benchmark was executed for the four different configurations specified in Table 6.3. Additionally, for each configuration, various element widths were tested (i.e., the width of the elements within a vector). The results were recorded and are presented in Table 6.8.

Table 6.8: Performance Counters and Metrics for the SAXPY Benchmark

		Load Stream					Store Stream			
Width	Config	Load Ops.	LFIFO Stalls	LRQ Stalls	AAGR	DRR	Store Ops.	SMMU Stalls	AAGR	DRR
1	A	938	312	5319	0.985	21.32	625	2806	0.781	16
	B	626	0	1254	1	31.95	625	626	0.941	16
	C	629	414	0	0.980	31.80	625	207	0.980	16
	D	628	414	0	0.980	31.85	625	207	0.980	16
2	A	1250	625	7506	0.970	16.00	1250	4056	0.711	8
	B	1039	625	0	0.971	19.23	1250	309	0.973	8
	C	1041	624	0	0.970	19.21	1250	312	0.970	8
	D	1000	508	0	0.980	20.01	1250	254	0.975	8
4	A	2498	13729	10002	0.593	8.01	2500	11856	0.458	4
	B	1251	13326	1	0.601	15.99	2500	6645	0.601	4
	C	1251	13299	0	0.601	15.99	2500	6632	0.601	4
	D	1251	13299	0	0.601	15.99	2500	6632	0.601	4
8	A	2502	40000	19	0.333	7.99	5000	19972	0.334	2
	B	2500	39999	0	0.334	8.00	5000	19922	0.334	2
	C	2500	39999	0	0.334	8.00	5000	19922	0.334	2
	D	2500	39999	0	0.334	8.00	5000	19922	0.334	2

In general, Config A demonstrates the lowest performance due to its lower parameter values (see Table 6.3), quickly depleting the available resources and leading to more frequent stalls. However, when each element is 8 bytes wide, no differences between configurations are observed. This is because each

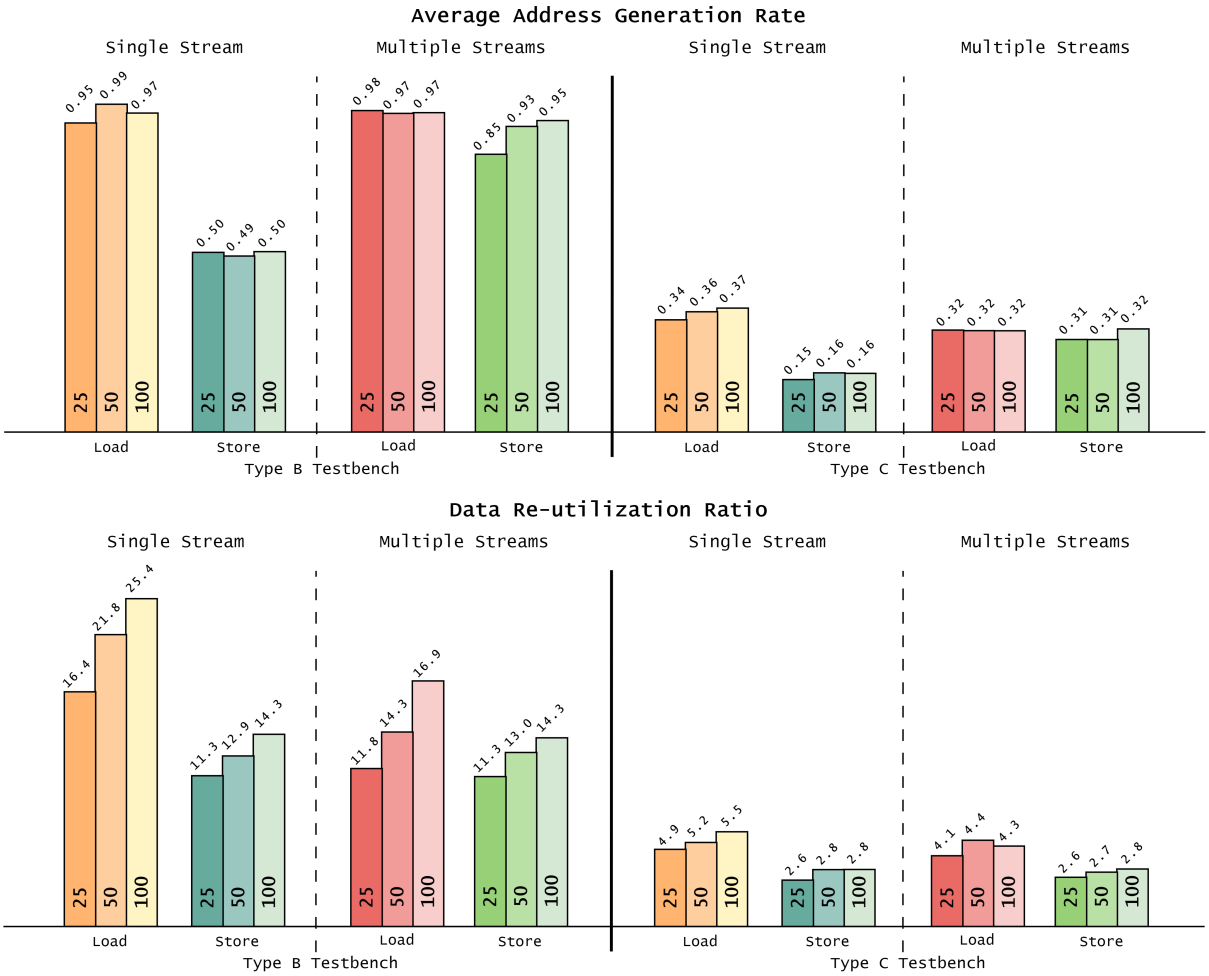

Figure 6.6: Performance Metrics on Multiple Descriptor Testbenches (With Static Modifiers)

element necessitates a separate memory row of data to be requested, resulting in a memory operation per generated address, regardless of the varying parameters in Table 6.2. The memory access pattern of SAXPY, characterized by highly contiguous elements, primarily causes stalls for the load streams at the LFIFO. This is expected since the resources of the LRQ are fully utilized. As the element width increases, performance counters for different configurations show fewer differences. This outcome is expected because wider elements mean that fewer data elements fit within the same memory row. In other words, when a memory row is retrieved, it contains more 1-byte elements than 8-byte elements. Consequently, the internal registers of the SE become frequently filled, leading to stalling in these circuit dimensions. In the case of SAXPY, having more tables in the LRQ (Config C) or a larger number of requests within a LRQ table (Config D) does not significantly impact performance. This is because the memory access pattern of SAXPY is already efficiently managed before exhausting all available resources. Therefore, adding extra registers provides minimal benefit in the SAXPY benchmark.

7

Conclusions & Future Work

Contents

7.1 Conclusions

The primary objective of this work was to develop a hardware architecture that implements the streaming mechanisms needed by any processor based on UVE, which is an extension for VLA designed to leverage streaming dataflows commonly found in many HPC applications.

To achieve this goal, a SE module was designed with a communication interface that can be integrated with a core implementing UVE. The SE is structured into various sub-modules, each serving a specific purpose. Among its capabilities, the SE can configure streams and generate memory access pattern addresses for the configured streams. The design employs a descriptor hierarchy organization to represent a wide variety of patterns. It can handle multi-dimensional affine-like streams using simple dimensional descriptors, and it incorporates static modifiers to enable the generation of more irregular accesses. All generated addresses, depending on the stream direction, are directed to the corresponding MMU, which efficiently seeks to reuse data, thereby reducing the need for high-latency memory operations.

To test the proposed design and implementation, additional hardware was developed to interface the SE with a simple processor known as the SP. It comprises just two memories and a FU, which is capable of performing basic vector arithmetic operations. These operations utilize vector registers, offering a stream-bound option for loading/storing data from the SE instead of the RF.

The experiments aimed to assess how well the SE performs when integrated into a processor's datapath. The results showed that the proposed architecture is highly efficient when the stream's access patterns are cache-friendly, achieving AAGR values above 0.95 and DRR values above 11. Even for highly scattered and irregular patterns, it consistently generates an address every 3 cycles, with AAGR surpassing 0.33 and DRR reaching 2 or higher. Concurrent stream processing enables the hiding of latency for individual streams, further enhancing data transfer bandwidth. However, it is important to note that the current implementation exhibits significant bottlenecks in store stream processing, which have a noticeable impact on the overall hardware system.

7.2 Future Work

The design introduced in this work for the SE represents an initial version primarily focusing on the fundamental functionalities associated with processing the entire lifecycle of a stream, starting from configuration and extending to the consumption of the final data elements. Although it can handle a diverse range of patterns, this design does not currently support memory access pattern indirection, a feature anticipated in UVE. Future work will involve expanding the proposed design to include support for indirection in streams.

At present, communication with the memory system is achieved via the AXI4 protocol. In projects

aimed at integrating this design into a hardware system, there may be a preference for an alternative protocol. Consequently, future work will encompass adapting the memory controllers to diverge from the AXI4 or to implement an alternative protocol.

Furthermore, it is essential to note that in this work, the hardware has exclusively undergone validation through simulations and testing employing Verilator. Subsequent phases of work will involve assessing the system's functionality post-synthesis and implementation. This evaluation will provide insights into timing constraints and critical paths.

Bibliography

[1] 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)*, 2021, pp. 209–222.

[2] 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.

[3] 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 Trans. Comput.*, vol. 70, no. 2, p. 212–227, feb 2021. [Online]. Available: https://doi.org/10.1109/TC.2020.2987314

[4] T. Hwang, "Computational power and the social impact of artificial intelligence," *CoRR*, vol. abs/1803.08971, 2018. [Online]. Available: http://arxiv.org/abs/1803.08971

[5] B. Li, C. Chenli, X. Xu, T. Jung, and Y. Shi, "Exploiting computation power of blockchain for biomedical image segmentation," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2019.

[6] N. Dao, A. Attwood, B. Healy, and D. Koch, "Flexbex: A risc-v with a reconfigurable instruction extension," in *2020 International Conference on Field-Programmable Technology (ICFPT)*, 2020, pp. 190–195.

[7] M. Tourres, C. Chavet, B. Le Gal, J. Crenne, and P. Coussy, "Extended risc-v hardware architecture for future digital communication systems," in *2021 IEEE 4th 5G World Forum (5GWF)*, 2021, pp. 224–229.

[8] "Technologies — NEON - Arm developer," 2022. [Online]. Available: https://developer.arm.com/technologies/neon

[9] S. Raman, V. Pentkovski, and J. Keshava, "Implementing streaming simd extensions on the pentium iii processor," *IEEE Micro*, vol. 20, no. 4, pp. 47–57, 2000.

[10] C. Lomont, “Introduction to Intel advanced vector extensions,” Intel, Tech. Rep., 2011. [Online]. Available: http://www.obpm.org/download/Intro_to_Intel_AVX.pdf

[11] A. Lopez-Lagunas and S. Chai, “Memory bandwidth optimization through stream descriptors,” *SIGARCH Computer Architecture News*, vol. 34, pp. 57–64, 03 2006.

[12] ——, “Compiler manipulation of stream descriptors for data access optimization,” in *2006 International Conference on Parallel Processing Workshops (ICPPW’06)*, 2006, pp. 8 pp.–344.

[13] 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.

[14] 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.

[15] A. Waterman and K. Asanovic, “RISC-V ’V’ Vector Extension,” 2019.

[16] N. Neves, J. M. Domingos, N. Roma, P. Tomás, and G. Falcao, “Compiling for Vector Extensions With Stream-Based Specialization,” *IEEE Micro*, vol. 42, no. 5, pp. 49–58, 2022.

[17] A. Waterman and K. Asanovic, “The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA Document Version 20190608-Base-Ratified,” 2019, [Online]. Available: https://riscv.org/specifications/.

[18] P. Michaud, “Best-offset hardware prefetching,” in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2016, pp. 469–480.

[19] L. Peled, S. Mannor, U. Weiser, and Y. Etsion, “Semantic locality and context-based prefetching using reinforcement learning,” in *Proc. 42nd Annu. Int. Symp. Comput. Archit.*, 2015, pp. 285–297.

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

[21] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad, “Bingo spatial data prefetcher,” in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 399–411.

[22] Z. Majo and T. Gross, “Matching memory access patterns and data placement for numa systems,” in *Proc. 10th Int. Symp. Code Gener. Optim.*, 2012, pp. 230–241.

[23] T. Grosser, H. Zheng, R. Aloor, A. Simburger, A. Grösslinger, and L.-N. Pouchet, “Polly-polyhedral optimization in llvm,” in *Proc. 1st Int. Workshop Polyhedral Compilation Techn.*, vol. 2011, 2011.

[24] S. Ainsworth and T. M. Jones, “Software prefetching for indirect memory accesses,” in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2017, pp. 305–317.

[25] S. Kondguli and M. Huang, “Division of labor: A more effective approach to prefetching,” in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit.*, 2018, pp. 83–95.

[26] H. e. a. Pugsley, “Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers,” in *Proc. IEEE 20th Int. Symp. High Perform. Comput. Archit.*, 2014, pp. 626–637.

[27] T. Hussain, O. Palomar, O. Unsal, A. Cristal, E. Ayguadé, and M. Valero, “Advanced pattern based memory controller for fpga based hpc applications,” in *2014 International Conference on High Performance Computing & Simulation (HPCS)*, 2014, pp. 287–294.

[28] N. C. Crago and S. J. Patel, “Outrider: Efficient memory latency tolerance with decoupled strands,” in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, 2011, pp. 117–128.

[29] T. J. Ham, J. L. Aragón, and M. Martonosi, “Desc: Decoupled supply-compute communication management for heterogeneous architectures,” in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015, pp. 191–203.

[30] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, “Stream-Dataflow Acceleration,” in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA ’17. New York, NY, USA: Association for Computing Machinery, 2017, p. 416–429. [Online]. Available: https://doi.org/10.1145/3079856.3080255

[31] N. Neves, P. Tomás, and N. Roma, “Efficient data-stream management for shared-memory many-core systems,” in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, 2015, pp. 1–8.

[32] S. Paiágua, F. Pratas, P. Tomás, N. Roma, and R. Chaves, “Hotstream: Efficient data streaming of complex patterns to multiple accelerating kernels,” in *2013 25th International Symposium on Computer Architecture and High Performance Computing*, 2013, pp. 17–24.

[33] N. Neves, P. Tomás, and N. Roma, “Compiler-assisted data streaming for regular code structures,” *IEEE Transactions on Computers*, vol. 70, no. 3, pp. 483–494, 2021.

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

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

[36] J. E. Smith, "Decoupled Access/Execute Computer Architectures," *SIGARCH Comput. Archit. News*, vol. 10, no. 3, p. 112–119, apr 1982. [Online]. Available: https://doi.org/10.1145/1067649.801719

[37] Z. Wang, J. Weng, J. Lowe-Power, J. Gaur, and T. Nowatzki, "Stream Floating: Enabling Proactive and Decentralized Cache Optimizations," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 640–653.

[38] Z. Wang, J. Weng, S. Liu, and T. Nowatzki, "Near-Stream Computing: General and Transparent Near-Cache Acceleration," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 331–345.

[39] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini, "Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2700–2713, 2017.

[40] D. Rossi, A. Pullini, I. Loi, M. Gautschi, F. K. Gürkaynak, A. Teman, J. Constantin, A. Burg, I. Miro-Panades, E. Beignè, F. Clermidy, P. Flatresse, and L. Benini, "Energy-efficient near-threshold parallel computing: The pulpv2 cluster," *IEEE Micro*, vol. 37, no. 5, pp. 20–31, 2017.

[41] F. Schuiki, M. Schaffner, F. Gurkaynak, and L. Benini, "A Scalable Near-Memory Architecture for Training Deep Neural Networks on Large In-Memory Datasets," *IEEE Transactions on Computers*, vol. 68, no. 4, pp. 484–497, Apr 2019.

[42] F. Conti, P. Schiavone, and L. Benini, "XNOR Neural Engine: a Hardware Accelerator IP for 21.6 fJ/op Binary Neural Network Inference," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2940–2951, Nov 2018.

[43] P. Scheffler, F. Zaruba, F. Schuiki, T. Hoefler, and L. Benini, “Indirection Stream Semantic Register Architecture for Efficient Sparse-Dense Linear Algebra,” in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 1787–1792.

[44] F. Zaruba, F. Schuiki, and L. Benini, “Manticore: A 4096-core RISC-V chiplet architecture for ultra-efficient floating-point computing,” *CoRR*, vol. abs/2008.06502, 2020. [Online]. Available: https://arxiv.org/abs/2008.06502

[45] F. Zaruba, F. Schuiki, T. Hoefler, and L. Benini, “Snitch: A 10 kGE Pseudo Dual-Issue Processor for Area and Energy Efficient Execution of Floating-Point Intensive Workloads,” *CoRR*, vol. abs/2002.10143, 2020. [Online]. Available: https://arxiv.org/abs/2002.10143

[46] *AMBA® AXI™ and ACE™ Protocol Specification*. ARM Limited, 2011, accessed: December 22, 2022]. [Online]. Available: https://www.arm.com/products/system-ip/amba-interface

[47] R. Lin and K. Laeufer, “Chiseltest,” https://github.com/ucb-bar/chiseltest, 2022.

[48] W. Snyder, “Verilator,” 2023, release Devel 5.017, 2023-10-29. [GitHub repository] https://github.com/verilator/verilator.