TÉCNICO LISBOA

UVE Streaming Engine: Hardware Implementation and Evaluation

Gonçalo Pinto Midões

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisors: Prof. Nuno Filipe Valentim Roma
Prof. Nuno Filipe Simões Santos Moraes da Silva Neves

Examination Committee

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

November 2024

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

Gostaria de começar por agradecer à minha família por todo o apoio que me foi dado nesta minha jornada académica que chegou agora ao fim com este trabalho. Em particular, gostava de agradecer aos meus pais que ao longo da minha vida como estudante, desde o primeiro dia de escola ao último dia de faculdade, sempre me deram tudo aquilo que eu precisava para conseguir dar o meu melhor, quer fosse material para a escola, tempo e paciência para me ajudar em algo que não entendesse, quer através de colo e mimos nos momentos mais frustrantes e complicados. Foi um percurso cheio de emoções, desafios e stresse, não só para mim, mas também para aqueles que me rodeiam e por isso agradeço toda a força que me deram e que tiveram.

Não posso esquecer de agradecer também à minha segunda família que construí quando entrei na faculdade, vocês, que conheci ao longo destes 5 anos, foram também uma peça essencial no meu sucesso. Foram aqueles que sem qualquer motivo aparente escolheram-me para criar uma irmandade nesta etapa tão importante nas nossas vidas. Não podia pedir por um grupo de amigos mais espetacular. Sempre pude contar com a vossa ajuda e por isso agradeço-vos. Por fim, um agradecimento especial aos meus orientadores que me apoiaram imenso ao longo deste último ano e que me desafiaram e puxaram por mim, para dar sempre o meu melhor.

Já fechei mais um capítulo da vida que certamente não seria possível sem a presença de todos na minha vida.

Obrigado a todos.

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) under project UNIFY: 2022.06780.PTDC (DOI: 10.54499/2022.06780.PTDC).

Abstract

Technology is continuously advancing, both through specific application domains like Artificial Intelligence (AI), Machine Learning (ML), Big Data, and video and image processing, and through improvements in computational infrastructure, such as General Purpose Processors (GPP), Graphics Processing Units (GPU), and accelerators. However, it has become a race for modern computational systems to keep pace with the demands of these data-centric applications.

This thesis highlights the critical role of memory access efficiency in computational systems, with a particular focus on how data fetching and manipulation impact overall performance. Techniques such as prefetching, data streaming, and vector processing are explored as potential methods to enhance throughput and energy efficiency. This study quantitatively assesses the effectiveness of integrating streaming engines and Single Instruction Multiple Data (SIMD) operations into CPU architectures, offering insights into the viability and benefits of adopting these methods for improved computational performance.

Keywords

Data Streaming; Scalable SIMD/Vector Processing; Data Manipulation Processor Microarchitecture; Unlimited Vector Extension; Energy Efficient High Performance Computing; CVA6; RISC-V.

Resumo

A tecnologia tem vindo a evoluir ao longo do tempo, quer a nível de aplicações específicas como a Inteligência Artifical (IA), *Machine Learning (ML)*, *Big Data*, e processsamento de vídeo e imagens, quer a nível de melhorias na infraestrutura computational, como *General Purpose Processors (GPP)*, *Graphics Processing Units (GPU)*, e Aceleradores. Contudo, apesar do progresso dos sistemas computacionais, existe uma certa dificuldade em estes acompanharem a evolução e necessidades destas *data-centric applications*.

Esta tese realça a importância do acesso à memória de forma eficiente nos sistemas computacionais, com destaque no impacto que a transferencia e manipulação de dados tem no desempenho do sistema. Técnicas como *prefetching*, *stream* de dados, processamento vectorial são exploradas como possíveis métodos que levam a maior *throughput* e eficiência energética. Este trabalho avalia quantitivamente a eficácia da integração de um *streaming engine* e de operações do tipo *Single Instruction Multiple Data (SIMD)*, na arquitetura de um processador, obtendo-se resultados de viabilidade e de mais valias na adopção destas técnicas na melhoria do desempenho computacional.

Palavras Chave

Streaming de Dados; Scalable SIMD/Vector Processing; Manipulação de Dados na Microarquitetura do Processador; Eficiência Energética na Computação de Alto Desempenho; CVA6; RISC-V.

Contents

List of Figures

List of Tables

Acronyms

AI	Artificial Intelligence
ILP	Instruction-Level Parallelism
GPU	Graphics Processing Units
GPP	General Purpose Processors
SIMD	Single Instruction Multiple Data
ALUs	Arithmetic and Logic Units
ISA	Instruction Set Architecture
ISAs	Instruction Set Architectures
DFG	Data Flow Graph
CGRA	Coarse Grained Reconfigurable Architecture
VLA	Vector Length Agnostic
SVE	Arm Scalable Vector Extension
UVE	Unlimited Vector Extension
CPU	Central Processing Unit
IPC	Instructions Per Cycle
ROB	Reorder Buffer
PC	Program Counter
SCROB	Stream Configuration Reorder Buffer
RVV	RISC-V Vector Extension
SV	System Verilog
HDL	Hardware Description Language
FU	Functional Unit
MMU	Memory Management Unit
MMUs	Memory Management Units
ITLB	Instruction Translation Lookaside Buffer

HPC	High Performance Computing
SC	Stream Configurator
SS	Stream State
LMMU	Load Memory Management Unit
SMMU	Store Memory Management Unit
SI	Stream Iterator
ST	Stream Tables
LF	Load FIFO Unit
LRQ	Load Request Queue
AGU	Address Generation Units
LLB	Load Line Buffer
SE	Streaming Engine
AXI4	Advanced eXtensible Interface 4
FSM	Finite State Machine
DUT	Design Under Test
ALU	Arithmetic Logic Unit
IM	Instruction Memory
GUI	Graphical User Interface
PE	Processing Elements
SoC	System-on-Chip

1

Introduction

Contents

1.1 Motivation

Technology is constantly evolving and requires faster and more efficient computing systems. With the increased attraction to data-centric applications like machine learning, deep learning, AI, and image and video processing, improvements to current computational infrastructures like General Purpose Processors (GPP), Graphics Processing Units (GPU) and Accelerators are becoming increasingly important, causing the need for the adoption of more capable methodologies, while at the same time embracing the need for better data efficiency.

Currently a major bottleneck of computer architectures ends up being the latency and insufficient throughput associated with the communication between the processing units and the memory system. This inefficiency appears from the diferent speed at which both technologies have evolved, creating a performance gap between the two systems, sometimes referred as memory wall [1, 2]. While mitigation of this problem is possible, through the implementation of strategies such as data caching and prefetching [3–5], there has been a decrease in performance improvement causing a plateau [6].

A common way to enhance a system performance could be made by increasing the number of processing cores or by elevating the processing frequency. However, this brings with it the challenge of managing the additional energy consumption, resource utilization, and, ultimately does not address the inefficiency associated with data access, but rather increase the existing performance gap.

For this reason, another strategy to achieve the goal of improving system performance lies in enhancing the efficiency of its surrounding components, more specifically the parts related to memory access and data transfer. In the previously mentioned application domains, where substantial amounts of data are fetched from memory and processed by the Central Processing Unit (CPU), the time and energy cost associated with data transfers can be significant, leading to undesired CPU stalls and to an overall slowdown in performance. Addressing these challenges in data manipulation efficiency becomes pivotal for optimising system performance.

There has been a lot of active research on the topic of reducing memory access latency, namely through the use of prefetching [3–5]. In prefetching, the system analyses the memory access patterns and learns the behaviour of memory access to fetch data before the CPU requires it for processing. There are various implementations of prefetching in both software and hardware. However, prefetching can cause cache pollution which, despite improving data access efficiency, can cause problems at cache level if not properly accounted for through the use of suited cache sizes or specialised techniques.

Another approach to reducing memory access latency, and that has gained a renewed interest, is through the use of data-streaming structures [7–11]. This paradigm utilizes special functional units and dedicated data structures called stream which leverage memory access patterns and their advantages. The implementation of these systems, referred as streaming engines, create a decoupled access-execute computing model, allowing it the use of specialized hardware optimizations. These streaming engines mitigate the data fetching overhead through the use of buffering memories, allowing a seamless fetching of data.

Another capability inherited by most stream-based solutions is the hability to exploit data paralleliz-

able operations. By obtaining data preemptively through the use of pattern analysis, streams, and memory buffers, the data becomes available for computation earlier. This means that for repeatable workloads or tasks, such as loop operations across arrays or matrices, the data can becomes available in *"chunks"* and so it can be manipulated as a vector through the user of Single Instruction Multiple Data (SIMD) operations, significantly enhance the system's throughput and overall processing efficiency.

Previously developed Instruction Set Architecture (ISA) provide extensions that allow for the use of SIMD operations. Examples of such extensions are Intel's Advanced Vector eXtension (AVX) [12] and, the Streaming SIMD Extensions (SSE), and the ARM NEON vector extension [13]. These extensions utilize special vector registers to distribute the workload of said registers throughout several paralell Arithmetic and Logic Units (ALUs).

One of the drawbacks of traditional SIMD ISA extensions is that the operations performed on vector registers are specific to a fixed vector size. This poses a problem because the compiled binaries become architecture dependent. Therefore, using the same binary on different systems with different vector lengths becomes impossible without recompiling the program. Another problem that arises from the use of traditional SIMD ISA extensions comes from necessity to handle loop tails. Loop tails refer to the remaining data elements that do not complete a full SIMD vector, in such situations this elements end up being processed as scalar values.

To address this limitations, the Arm Scalable Vector Extension (SVE) [14] and the RISC-V Vector Extension (RVV) ISA [15] have emerged, adopting a distinct approach to vector length and elevating the abstraction of such hardware constraints by only requiring knowledge of it at runtime. However, these implementations are not without challenges. In order to allow a dynamic vector length the system needs the ability of disabling specific SIMD vector entries and this is achieved through the use of auxiliary signals called predicates. The most notable issue with this approach is the necessity for control instructions within loop iterations to manipulate the vector predicate. This can result in an increased number of instructions, possibly leading to significant overhead [11]

The recently proposed Unlimited Vector Extension (UVE) [11] ISA also aims to address the previously mentioned challenge associated with vector size. It adopts the strategy of vector-length agnostic code, akin to ARM SVE [14] and RISC-V Vector Extension [15]. However, its approach to implementing the required instructions differs from other solutions, since it allows the configuration before the program loops, eliminating the need for control commands within the loop iteration and enhancing instruction efficiency. Moreover, UVE introduces a streaming engine model to efficiently retrieve data by configuring complex access patterns. Through the configuration of the access patterns and the use of prefetching and buffering technic, the UVE streaming engine model allows a high data transfer to the processing unit enabling SIMD operations to achieve higher throughput, contributing to overall performance improvements.

1.2 Thesis Objectives

Despite the significant progress that was made through the study, development, and implementation of some of the aforementioned solutions, there is still room for improvement when talking about the perfor-

mance and energy efficiency of modern computational systems. For this reason, continued research in these areas is crucial. Propelled by the need for more progress in the solution of data use bottleneck, the objectives of this work are as follows:

- Study of the current approaches to data streaming implementations and their capabilities;
- Study the effect of access pattern configurability on data availability, latency and throughput of streaming engines for complex access patterns.
- Perform synthesis analysis of a Streaming Engine focusing on clock frequency, footprint dimension, and power consumption.
- Study and evaluate the requirements necessary for the integration of the streaming engine micro-architecture proposed by the Unlimited Vector Extension with Data Streaming Support on GPP.

1.3 Contributions

With the previously enumerated objectives in mind the work of this thesis can be resumed to the following contributions:

- **Re-implementation of the Streaming Engine (SE)**: This tasked involved the implementation of a more consolidated architecture for further testing and validation.
- **Validation and Evaluation of the SE**: This task involved the creation of a robust test environment, with the design of specific test-benches and hardware synthesis.
- **Profiling of access patterns on the SE Design** The conducted analysis allowed the evaluation of the efficiency of the streaming engine when confronted with diferent access patterns.
- **Validation of the SE in a case study**: Validation of the SE through its implementation on a stream-based dataflow accelerator, focused on expanding a GPP stream vectorization model into a fully data-flow-driven computing module, leveraging spatial computation and a stream-based co-acceleration structures.

1.4 Thesis Outline

This document is structured as follows:

Chapter 2 State of the Art presents several state-of-the-art solutions aimed at improving data transfer between memory and processing units, including discussions on data streaming architectures, data prefetching, memory access, and scalable vectorization.

Chapter 3 Background explores the UVE ISA, covering its features, micro-architecture, and ISA extention. It also presents the characteristics, architecture, features, and execution flow of the CVA6 processor.

Chapter 4 UVE Streaming Engine Implementation provides a detailed description of the composition of the streaming engine studied in this thesis.

Chapter 5 Development and Evaluation Environment describes the environment used in the validation, profiling and improvement of the evaluate the streaming engine.

Chapter 6 Experimental Evaluation covers the analysis of the results obtained from validation and a performed case study.

Chapter 7 Conclusion concludes the document with a reflection on the work developed addressing potential future research.

2

State of the Art

Contents

In recent years, there has been a rise in research and interest in creating powerful, data-centric applications, including video and image processing, machine learning, deep learning, Artificial Intelligence (AI), and cryptography. While advancements continue in processing power to meet the high demands of these applications, the performance improvements of single-core processors have stagnated. This slowdown is largely due to the impending end of Moore's Law [16] and Dennard Scaling [17], as well as the limits of Instruction-Level Parallelism (ILP) and the inability to further increase clock speeds due to the operating temperatures it would reach, refereed sometimes by power wall. These limitations underscore the urgent need for new processing paradigms.

One promising approach to improving computational systems involves addressing a major bottleneck in computer architecture: the inefficiency of data transfer. Latency and insufficient throughput between processing units and memory systems are primary causes of degraded performance. Techniques such as data caching and prefetching have been employed to mitigate this issue [3–5], but their performance gains have gradually diminished, leading to a performance plateau [6].

This stagnation has caused a renewed interest in alternative techniques, such as data streaming mechanisms [7–11]. By implementing these mechanisms, systems can leverage access pattern information to decouple data fetching and transfer from the core processing unit. This decoupling enables advanced data prefetching and buffering, as well as improved data reuse, contributing to more efficient memory access. The following sections of this chapter provide an in-depth analysis of several streaming approaches, examining their advantages and limitations.

2.1 Data Streaming

Specializing the core computation part of a processor and relying only on traditional memory abstractions is not sufficient for the necessary improvements on modern-day CPUs [7]. As a matter of fact an area where specialization has not advanced at the same pace as other components is memory primitives. Exposure to more memory information at the ISA level and utilizing unique micro-architecture mechanics would allow for tackling multiple shortcomings with memory use in the core. Some of these shortcomings are the added overhead in the pipeline on data indexing and memory address generation, the overlapping memory requests, and transfers from the memory system.

The solution for some of these problems might lie in using dedicated structures of memory access such as data streams [7]. Data streams are easily constructed through the analysis of repeated patterns of memory accesses, most of the time associated with loops and nested loops of programs.

The following subsections present a brief overview of some recent state-of-the-art proposals to address data streaming in accelerators and general purpose processors.

2.1.1 Stream Data-flow Accelerators

Stream Dataflow denotes an implementation of an accelerator proposed by Nowatzki *et al.* [7] with the sole purpose of managing streamed memory accesses. According to its authors, the proposed

architecture can serve as the basis of configurable memory accelerators.

The acceleration unit proposed in this project was designed with the objective of achieving as good of performance as an application or domain-specific system while trying to maintain efficiency and utilization abstraction. The approach taken to achieve such goals was through the use of data stream memory structures.

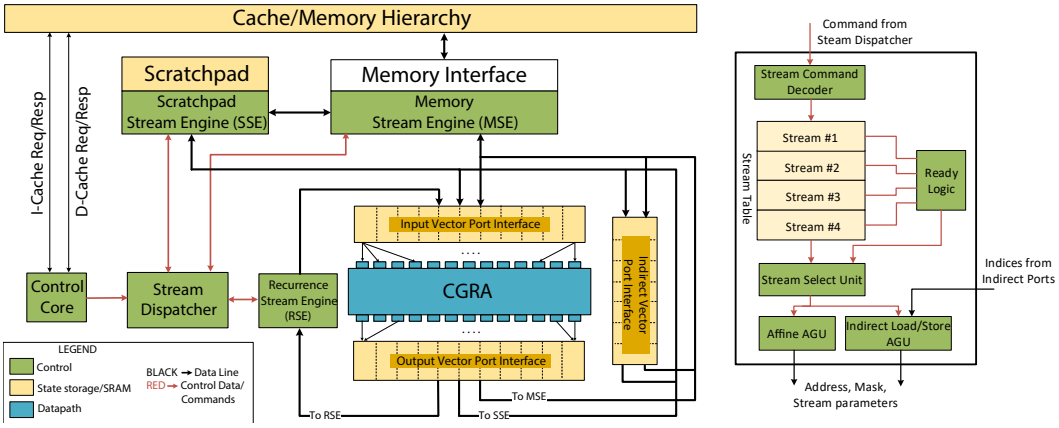

Figure 2.1: Stream Data Flow micro-architecture (Softbrain) (left); Stream engine pipeline (right) - Figure from [7]

The proposed micro-architecture, called *Softbrain*, can be seen in Figure 2.1 (Left), and it is composed of five primary components:

- **Control Core** - Tasked with the generation of stream-dataflow commands;
- **Stream Dispatcher** - Unit that tracks stream resource dependencies, issues commands to stream engines, and manages concurrent stream execution.
- **Stream Engines** - 3 engines that carry out the data access: one for memory (facilitating wide access to memory), one for scratchpad (efficient data reuse), and one for DFG recurrences (for immediate reuse). The use of each engine is dependent on the streams that are assigned to them.
- **Vector Ports** - The main interface between the computation that is carried out in the CGRA and the streams. A set of isolated vector ports allows for indirect memory loads/stores.
- **Main Acceleration Unit (CGRA)** - The coarse grained reconfigurable pipeline where the main processing of the dataflow graphs is conducted.

The Stream-Dataflow accelerator operates by executing precompiled graphs known as DataFlow Graphs (DFG). The compilation of these graphs occurs alongside program compilation and consists of multiple instructions organized in a network of operations and dependencies.

During runtime, the main acceleration unit is reconfigured according to the DFG to correctly map the input and output of the vector ports and define the appropriate vector length for compatibility with the memory streams. Additionally, this functional unit is set up for the operations specified by the flow of the graph, creating the advantage of smooth execution of the operations without the need for control inside the CGRA unit.

2.1.1.A Streams

The stream data structures utilized in the Dataflow Graph Accelerator are defined by 3 parameters: a source location, a destination, and an access pattern. Both the source and destination are associated with a memory address, programmable scratchpads, or a named vector port.

Port locations are associated with the CGRA unit, making the associated streams serve as inputs and outputs of the DFG.

DFG ports support simple first-in-first-out patterns, while memory and scratchpad access patterns can be more intricate, incorporating features such as strides, repetitions, indirect access, scatter-gather, etc.

Streams typically operate concurrently, but those associated with the same DFG port must execute sequentially.

2.1.1.B Streaming Engines

In order to be able to utilize stream data structures and take advantage of the high parallelism and low power overheads [7], streaming engines need to be implemented into the micro-architecture. The proposed design for each of the two implemented engines is driven from the diagram illustrated in Figure 2.1 (Right) and works as a pipeline system.

The engine operates by decoding instructions issued by the dispatcher, keeping track of the number of active streams, and maintaining associated data for each stream. On each clock cycle, the engine selects one of the active streams and generates the required address and data transfer commands. Additionally, it monitors back-pressure signals to ascertain if a stream is considered ready. This readiness condition implies that the destination port is not full, and data can be fetched from the source.

As was referred before, the memory system comprises two engines: the memory stream and the scratch-pad stream. The memory stream engine is tasked with facilitating the transfer of data to and from the primary memory cache, utilizing streams, and incorporating memory buffering. On the other hand, the scratch-pad stream engine bears resemblance to the memory stream engine. However, it is distinct in that it features its dedicated ported SRAM, supporting single-read and single-write operations.

2.1.1.C Stream-Dataflow ISA

For the correct operation of the stream-dataflow micro-architecture, three main types of instructions were defined:

> **T1 DFG Specification** Given that the physical architecture may impose limitations on the number of instructions for each type in a graph, several constraints on the size of the input and output vectors, and the number of inputs and outputs, the configuration of such parameters could become cumbersome. For that reason, Nowatzki *et al.* [7] have chosen to simplify the matter by introducing a single instruction that loads a compiled configuration directly from memory.

T2 Stream Specification The provided instructions enable the configuration of streams for access with various patterns. These instructions are versatile and can be set up for the most common type of memory access pattern, namely, two-dimensional affine accesses. The definition of such patterns involves specifying the access size, the stride value (size between consecutive accesses), and the number of strides. Additionally, indirect streams can be configured by designating one stream's output as input for another stream.

T3 Barrier Specification The proposed ISA includes three barrier instructions. Two of them synchronize the reading and writing of the scratchpad memory, while the third ensures the completion of the Data Flow Graph (DFG) and guarantees that the correct data is stored in the host memory.

2.1.1.D Result Analysys

To assess the goal of designing an accelerator that could replace a domain-specific implementation while maintaining generality, Nowatzki *et al.* [7] conducted a series of workload tests from the MachSuite. Each test was run by a *Softbrain* implementation, configured to a max of 20 DFG instructions, and a customized Aladdin ASIC with its implementation generated for each application. The comparison focused on their power consumption, performance, energy efficiency, and area.

Both the Softbrain and the ASIC implementations showed 1-7x speedup gains across all workloads when compared with a SandyBridge OOO Quad-Core, keeping on par with one another. In terms of power and energy efficiency, the *Softbrain* accelerator showed worse results than the ASICs implementations, however not far behind it.

On area comparison, the *Softbrain* showed a bigger area use however, since *Softbrain* is programmable on runtime and able to run all workloads, it is possible to advocate that the *Softbrain* implementation is more area efficient than the ASIC implementation since designing a system for the full set of benchmarks performed would require 2.4x the area of *Softbrain*.

Nowatzki *et al.* [7] conclude by stating that the developed work demonstrates the potential of designing programmable/configurable architecture that leverage the advantages of streaming structures.

2.1.2 Stream-based Memory Access Specialisation for General Purpose Processors

In a follow-up research to the work developed by Nowatzki *et al.* in Stream-Dataflow Acceleration [7], Wang and Nowatzki (2019) [8] set out to apply some other same streaming principles on General Purpose Out-of-Order Processors, with the main objective of enhancing and study the data fetching capabilities of such systems.

Through the use of stream-decoupling, prefetching, and specialized cache policies, the authors aim to reduce the instruction pressure on the processor's pipeline while also making the delivery of data to the processor faster.

2.1.2.A Microachiteture Extension

In order to keep modifications to the existing microarchitecture to a minimum, most of the frontend structure of the core was maintained unaltered, with the exception of the addition of an iteration map, that tracks the position of each stream index by mapping it to a counter table, the introduction of FIFOs that store the accessed data of the streams, and the implementation of a streaming engine. The location of these units is shown on the left side of Figure 2.2.

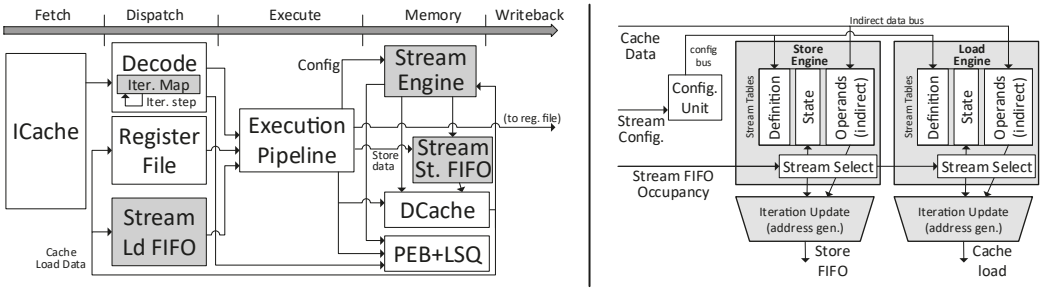

Figure 2.2: Stream Specialized Pipeline (Left); Stream Engine (Right) - Figure adapted from [8]

The streaming engine comprises two identical units, one designated for loading from memory and the other for storing in memory. These units incorporate a set of tables employed to describe the state of any configured stream. The **definition table** encompasses the pattern (affine, indirect, linked) along with the parameters of the access (stride, width, size). The second table is the **state table** , responsible for storing the memory view of the induction variables, indicating the current address. Lastly, the **operands table** contains information on any stream dependencies. Through the use of this stored information, the streaming engine generates, on each clock cycle, the memory address of the following reads/stores caused by the execution pipeline.

The FIFOs of the streaming engine have the purpose of holding the decoupled state either for storing or loading. Whenever a core instruction makes a read to a stream, it accesses the data on the load FIFO. Whenever a core instruction performs a store, the value to store is combined with a previously generated address kept in the store FIFO.

2.1.2.B Decoupled-Stream ISA

The stream-specialized interface proposed by Wang and Nowatzki (2019) [8], tries to maintain the stream data structure implementation separated from the rest of the cores' microarchitecture maintaining a higher level of abstraction. This is made through the use of special pseudo-registers to which the execution pipeline has access and are mapped to specific data streams. The streaming part of the system applies a similar principle to the Stream Dataflow Accelerator in the sense that it is configurable through an initial instruction.

The execution of code compiled for this stream interface contains such instruction at the beginning of code sections that will benefit from data streaming (loop portions of code). It is responsible for: **setting**

up the necessary **pseudo-registers** employed by the core pipeline and mapped to the **data streams**, configuring the **streams type** (induction/memory) and **pattern** (stride, width, and optional length), the dependencies and the starting address. The configuration of streams can also be made to address another stream, thereby establishing a dependency through indirect memory access.

During the execution of the code in the pipeline, a ”step” instruction is employed to advance the position of the stream by modifying the induction variables related to the streams and providing the core with abstract information on the stream’s progress. At the end of the loop iterations, the configured streams are deallocated and released for the next utilization.

The proposed Instruction Set Architecture (ISA) incorporates a set of instructions that enable fundamental stream operations, like pattern definitions and data width configuration, supporting complex accesses to data and coalescing of data structures.

Figure 2.3 depicts four examples of the decoupled-stream pseudo-code, along with its stream dependence graph.

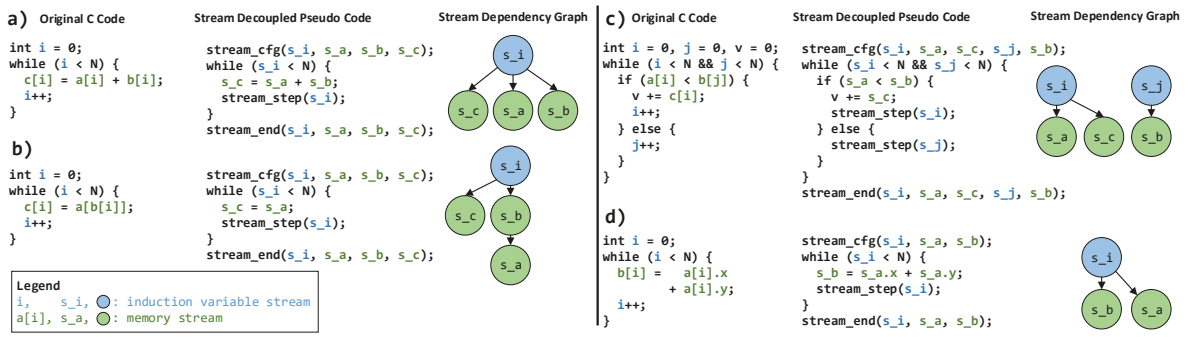

Figure 2.3: Pseudo Code Examples - Figure adapted from [8]

stream_cfg is the instruction used for the configuration, defining all the streams within the loop. After configuration, the stream engine can start fetching data based on the knowledge previously defined.

The use of ***stream_step*** inside loop iterations advance the induction variable stream, altering the stream pseudo-register position and resulting in a cascade on all dependent streams. This behavior is evident in the Stream Dependency Graph, where the advancement of *s_i* causes all dependent streams (*s_a, s_b, s_c, ...*) also to advance.

The ***stream_end*** instruction, as the name implies, the utilization of streams concludes by deallocating and releasing the mapped pseudo-registers.

The proposed ISA comes with its limitations since the decoupling nature of the implementations becomes limited whenever the access patterns cannot be determined at the configuration step. Such cases are conditionally used data and unknown length for an access pattern.

2.1.2.C Using Stream Information

Wang and Nowatzki (2019) [8] also propose the exploitation of the stream information provided by the decoupled design of the presented implementation. Accordingly a prefetching technique and stream-aware cache bypass policies were implemented leveraging such stream information.

In what concerns the prefetching, the distance in which prefetching is made can highly impact the smooth operation of a cores' pipeline. In an ideal prefetcher, the required data would be provided whenever the corresponding load instruction is ready to be issued. This means that a sign of poor performance of the prefetcher could be falling behind on the data requests. For the monitoring and correct utilization of the resources of the processor, Wang and Nowatzki (2019) [8] took the approach of implementing a set of counters that track the number of stalls that occur with each stream pseudo-register, entitling it as the dynamic throttling prefetching system. These counters enable the pipeline to dynamically readjust the size distribution of the Load FIFO across stream pseudo-registers. This adjustment provides streams with increased available space and subsequently allow the prefetching distance of specific streams to be incremented based on the rate at which data is being consumed.

In the proposed implementation, the initial value of each stream pseudo-register length is defined in the configuration step with a default value. However, this could be further improved by defining a more personalized value during compilation.

Another technique implemented by Wang and Nowatzki (2019) [8] that leverages the information given by the patterns defined by streams, is cache bypassing. In memory requests with low temporal locality, the utilization of cache can become a problem and hurt the overall execution performance. Through the stream information provided by the system (memory footprint, stream length, reuse distance etc), it is possible to decide whether a memory request should be made to the cache or bypass it completely and request the information directly to the memory system. This technique would help avoid polluting the cache with data that would not be reused, as well as avoid unnecessary time wasted with tag lookup and MSHR allocation.

2.1.2.D Result Analysis

Wang and Nowatzki (2019) [8] concluded through a series of workloads that their proposed decoupled-stream ISA extension show very promising results in terms of performance and energy consumption when compared with the baseline simulated 8-Way OOO processor. The tests conducted were made to evaluate the impact of the improvements suggested by the authors, like the inclusion of streaming with prefetching, the utilization of a dynamic throttle of the prefetching distance, and the utilization of stream access information in cache bypassing policies.

The implementation of just the streaming engine resulted in an overall gain of 1.20x in performance. Utilizing both the dynamic throttle (prefetching stall counters) and stream-aware cache policies pushed the gain value to 1.67x when compared with the baseline out-of-order core. In terms of energy efficiency, the implementation of the proposed full system resulted in an improvement of 1.53x compared to the baseline.

Wang and Nowatzki (2019) [8] concluded that utilizing the benefits of stream-based prefetching, address computation decoupling, and stream-aware cache policies can lead to a faster, more specialized access and communication within the cache and memory hierarchy, albeit with certain limitations like the problem of unknown length of patterns during stream configuration.

Some further research topics like vector data manipulation could also be seen as a possible improve-

ment to the described system leveraging parallel computation of data.

2.2 Stream Semantic Registers

Stream Semantic Registers (SSR) [10] introduces an innovative approach to address the von Neumann bottleneck in single-issue processor cores. This bottleneck arises from the explicit fetch and issue of load/store instructions for data to reach the Arithmetic Logic Units (ALUs) and Functional Units (FUs). In order to mitigate this bottleneck, Schuiki *et al.* [10] presents a solution in the form of a straightforward and lightweight RISC-V ISA extension. This extension implicitly encodes memory accesses, thereby reducing the number of instructions associated with memory accesses and enhancing the overall energy efficiency of the system.

To enable this capability, SSR gives some registers stream semantics, meaning that reading from or writing to such registers triggers a direct read or write operation in the memory. This innovative approach streamlines the execution of instructions and contributes to energy efficiency.

2.2.0.A Micro-achiteture

The following modifications to the processor's core are necessary to implement the SSR system, and allow the mapping of register access to the stream interfaces:

C1 **Register File Extension:** Expanding the register file to facilitate the interception and rerouting of accesses to a subset of registers.

C2 **Stream Interfaces:** Incorporating stream interfaces to each register file port to allow stream semantics registers.

C3 **Control:** Implementing a Control and Status Registers (CSR) to enable or disable stream semantics.

C4 **Data Paths:** Adding the paths needed for the communication of stall conditions and back-pressure path to the core's main controller.

Schuiki *et al.* [10] designed the needed modifications with the added intent of minimizing the impact on the complexity of the underlying architecture. A high-level schematic of the architectural changes needed for this implementation can be seen in Figure. 2.4.

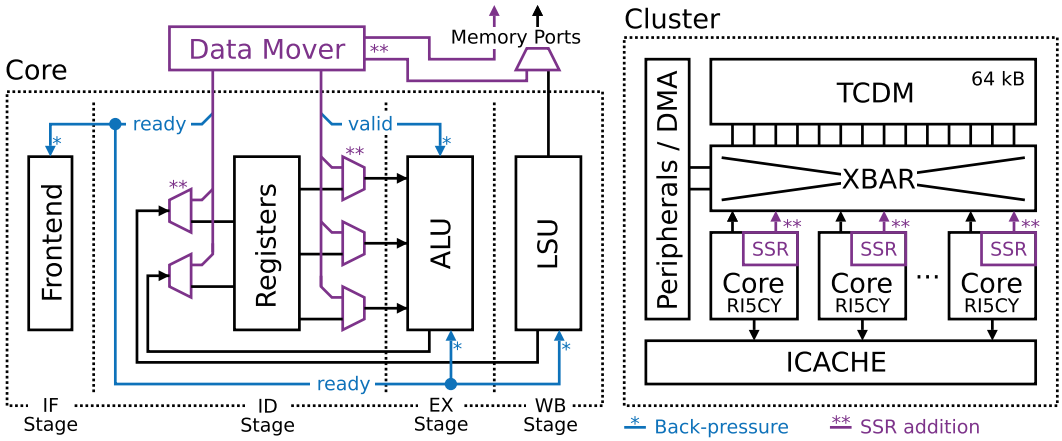

Figure 2.4: High-level data flow of the SSR extension in a single core - Figure from [10]

2.2.0.B Register Files

The primary architectural modification introduced by the SSR extension is in the processor register file. This register file is structured as a default three read and two write port configuration. However, each port is customized to incorporate the needed circuitry (refer to Fig. 2.5) to intercept read/write accesses and detect whether the specified register has stream semantics enabled.

The process of a register access unfolds as follows: the access is intercepted, the presence of stream semantics is determined, and if confirmed, the access is rerouted to the respective external stream interface connected to the data mover unit.

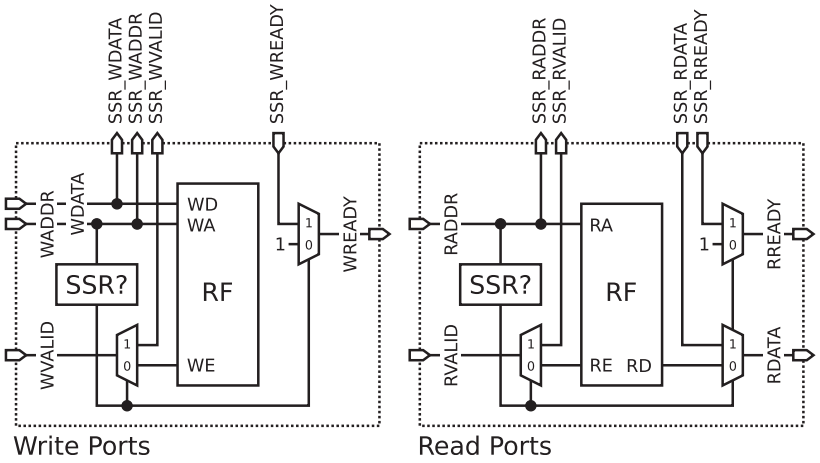

Figure 2.5: Required circuits for stream semantic enabled verification - Figure from [10]

The confirmation of the active status of stream semantics is carried out in the *SSR* section of the circuit, as illustrated in Fig. 2.5. This verification involves checking the following conditions:

1 The register address connected to the WADDR or the RADDR must correspond to one of the registers allowed to utilize stream semantics, on the write ports circuit or read ports circuit, respectively;

2 The stream semantics flag must be enabled in the core's CSR;

In case both conditions are true then the transaction is routed out of the core through a stream interface to the data mover.

2.2.0.C Data Mover

The accesses initiated within the core and directed outward through the stream interfaces reach the data mover unit. This segment of the architecture, visible in Figure 2.6 is tasked with accurately mapping the incoming accesses to the memory system. To execute this function, the data mover incorporates a set of lanes, each comprising a *First-In-First-Out (FIFO)* queue buffer and an address generator unit (AGU). The AGU is responsible for associating each streamed access with the corresponding memory address, whether for reading or writing. Offloading the memory address generation outside of the main pipeline reduces the pressure caused by this type of function.

The internal diagram of the AGU can be seen on the right of Figure 2.6 and is based on AGU presented by Schuiki *et al.* [18]. It works through the configuration of the hardware loops ($L0$ to $L3$), which are composed of 16-bit counters that iterate over the parameters of the stream access pattern and provide the values necessary for the address generation.

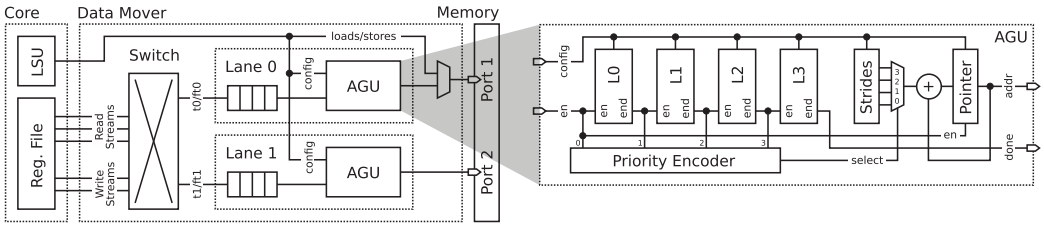

Figure 2.6: Data mover interfaces and inner structure (Left); Internal design of Address Generator Unit (Right) - Figure adapted from [10]

For each processed stream, the AGU is configured in either reading mode, generating memory addresses, or writing mode, producing tags for each datum to be written. It's noteworthy that the processing of streams cannot be interleaved; thus, the same mode has to persists per AGU until the conclusion of the address pattern.

A positive aspect arising from the previously mentioned constraint is the proactive capability of the data mover to perform memory reads using the defined configuration. This behavior facilitates quicker access to data, as whenever the processor decides to read from an SSR, the data is already available. This stands in contrast to normal data accesses, where each request needs to be individually made and awaited.

Unfortunately, due to the design of the data mover and the pro-active read, memory incoherence could occur since a lane could have read a value from memory and placed it in the *FIFO* before the other lane could have updated the same memory location with a new value, for this reason, to avoid data races write operation cannot initiate on a section of memory that has been set to read from.

2.2.0.D Execution Flow

The execution of a program compiled for this solution starts with the setup of the configuration registers present in the AGUs of the data mover. These registers define the loop dimensions of a stream, allowing for a total of four dimension loops. After the activation of stream semantics in the cores' CSR the iteration of the computations occurs with the AGU loop counters iterating over the preconfigured access patterns and providing the necessary data to the main pipeline of the core.

At the end of the code section optimized for the use of stream semantics, the program executes an instruction to deactivate the use of stream semantics, allowing the core to utilize every register as normal registers.

2.2.0.E Result Analysis

Several tests were conducted with a focus on evaluating power consumption, performance, efficiency, and the implementation area. In their research, Schuiki *et al.* [10] observed a performance increase ranging from 2x to 5x, attributed to the overall reduction in instructions. This improvement was achieved with only a minor increase in the core area (11%), accompanied by a notable decrease in cache energy consumption by up to 5.6x. These findings substantiate the belief in the potential effectiveness of streaming mechanisms.

While the outcomes achieved for the compiled workloads were favorable, there is room for improvement in the compilation method. The current compiler approach applies SSR indiscriminately to every encountered loop, this leads to the utilization of SSR in smaller loops that do not derive significant benefits, mainly due to the higher setup time required. This observation underscores a potential drawback and emphasizes the necessity for additional research and enhancements.

As also highlighted by Schuiki *et al.* [10], the design of the Address Generator Unit (AGU) in the data mover restricts the applicability and gains of the implementation, primarily due to the limitations in the types of memory access patterns that can be configured. Other research has explored similar principles to those used in SSR but with the capability to accommodate more intricate data patterns, like the Indirection Stream Semantic Register (ISSR) extension [9].

2.3 Vector Extensions and Streaming

As it was previously discussed, enhancing data load and store operations contributes to the improvement of the system's memory access latency and overall performance. However, individual computations on this data still occur sequentially. This is where vector extensions prove beneficial. Vector extensions employ Single Instruction, Multiple Data (SIMD) instructions across a vector of data in a single execution step, enabling increased system throughput.

2.3.1 ARM Scalable Vector Extention

The ARM approach to a vector-length-agnostic programming model was the definition of SVE [14]. SVE brings a new architectural state with the inclusion of:

- 32 new scalable vector registers, *Z0 to Z31*, with a total width ranging from 128 to 2,048 bits depending on the implementation;
- 32 128-bit wide Advance SIMD registers, *V0 to V31*, allowing for 64-, 32-, 16- and 8-bit scalable containers
- 16 scalable predicate registers *P0 to P15* and a set of control registers *ZCR_EL1 to ZCR_EL3*.

2.3.1.A Predication

The SVE solution allows programmers to think at a higher level of abstraction regarding vector length. This is possible through predicate registers that dynamically modify the vector length during execution. This technique allows the computation of only those lanes that matter, as opposed to the usual SIMD strategy of computing the whole vector and discarding unwanted values through masking.

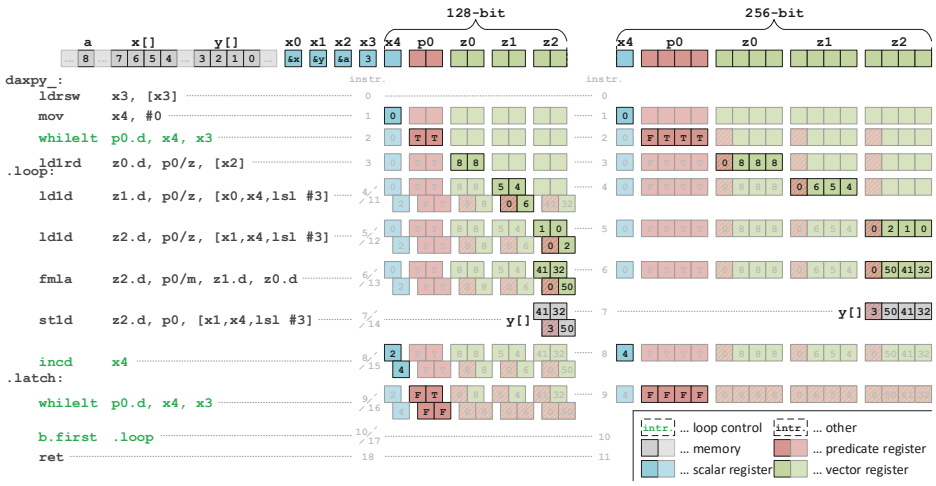

Figure 2.7: Cycle by cycle example of daxpy with n = 3 and hardware vector lengths of 128- and 256-bit - Figure from [14]

Figure 2.7 illustrates a step-by-step example of a Daxpy program for a vector length of 128 and 256 bits. The use of a predicate variable is demonstrated in the 128-bit configuration, with both its values set to true on the first comparison, utilizing the full dimension of the vector. This state of the predicate signifies that the load and computation of values occur in both positions of the variables *z0, z1, z2*. After the loop iteration, the variable tracking the number of processed elements is incremented (in this case, by the total number of float elements in the vector), and compared with the total number of values to be processed. From this comparison, another predicate is generated, and, since two elements have been processed and the total is three, the new predicate indicates the execution of instructions over only a single entry of *z0, z1, z2* in the next iteration.

Something similar happens in the 256-bit vector configuration. However, the initial comparison generates a predicate with the last position being false. This means the vector won't be utilized in full, and the next predicated instructions to execute should only compute three values.

SVE also includes in its extended instruction set a family of *while* instructions that operate by populating a new predicate through the use of a scalar counter. The value of this stride can be incremented not only by the value of the total number of elements in the system vector but also through more complex increment patterns like the number of active lanes, or the largest multiple of a data element on a vector, allowing for a powerful data access and manipulation.

Due to this nature of vector-length-agnostic programming, situations that require vector initialization can become more difficult. To address this complexity, the SVE extension incorporates a set of instructions capable of dynamically generating vector initialization during runtime. This dynamic approach is particularly useful as it adapts to pattern initializations, allowing for flexibility and efficiency in vector initialization processes.

Some examples of specific vector initialization are as follows:

- Initializing a vector with a set of indexes that start on 1 and increment by 4 can be achieved using *INDEX Zd.S, #1, #4* resulting in **Zd = [1, 5, 9, 13, 17, 21, 25, 29]** for a vector size of 8 elements.

- Initializing a predicate vector that is supposed to contain as many triples of the true value as will fit in the vector:*PTRUE Pd.S, MUL3* will result in **Pd = [(T,T,T), (T,T,T), F, F]** for a vector size of 8 elements.

2.3.1.B Horizontal Reductions

Another implemented feature in ARM SVE is the ability to do operations across lanes of a vector, essentially taking an operation and applying it to every element of the vector, for example, a sum, a bitwise-and, etc.

Designing a program with these intrinsics in mind allows for better performance. Fig. 2.8 shows the compiled code of a sum of a multiplication of two vectors with and without intrinsics. Both compiled versions of code execute, at the beginning of each loop iteration, the load of the values of a and b (green instructions), the differences occur on the computation part of each iteration. In the ARM SVE version without intrinsics (Fig. 2.8.C), the loaded vectors are multiplied, and the result is stored in a vector (purple instruction) following that instruction, a sum of values occurs by adding each entry of the vector to the register $d0$ this leads to the increase of the number of operations that occur per iteration.

In the ARM SVE version with intrinsics (Fig. 2.8.D), a horizontal reduction instruction is used at the end, meaning that during the loop iterations, it is more advantageous to multiply the vectors and add the result to third temporary vector (yellow instruction - fmla). The advantage comes from the fact that the instruction can happen across the vector entries simultaneously, resulting in a lighter computational pressure when compared with the previously compiled code. To achieve the same final result at the end of the execution, the horizontal reduction sums the entries of the final resulting vector.

Even though both Fig. 2.8.C and 2.8.D appear to have similarly sized loops, the executed instructions differ a lot in terms of execution time, with the ARM SVE version with intrinsic reaching the final result faster.

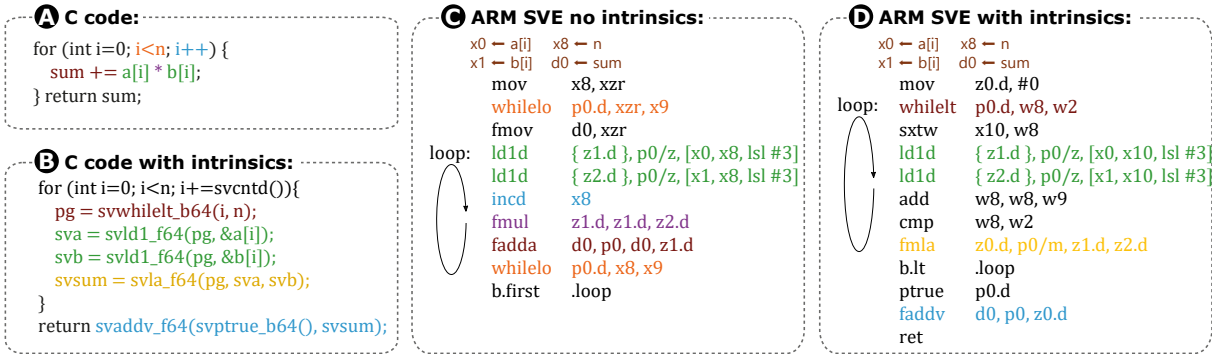

Figure 2.8: Comparison of assembly code between ARM SVE without and with horizontal intrinsics

2.3.1.C Handling unknown data vector size

In earlier instances of ARM SVE examples, predicates have consistently played a crucial role in the loading, storing, and manipulation of data. Nevertheless, scenarios may arise where determining the size of the vector becomes impractical before the data load.

One approach in such situations could be to load as much data as can be fitted into a vector. However, this approach introduces the risk of loading invalid or protected values from memory, potentially triggering a fault.

To avoid the aforementioned problem, speculative vectorization can be utilized. SVE incorporates a distinctive predicate, the First Fault Register (FFR), generated when employing a memory-probing instruction. This instruction essentially examines the memory entries, starting at a base address, and return a predicate that indicates the regions in memory corresponding to the desired data.

This feature facilitates the vectorization of uncounted loops with break conditions without the need for explicit size specification. Figure 2.9 illustrates the utilization of a memory-probing instruction, the *LDFF1D*, within an uncounted loop, and will be explained in the following paragraphs.

During the initial iteration of the probing instruction, all memory entries are examined, as the P1 predicate is initialized as entirely true. The resulting Forwarding Flag Register (FFR) signifies that data on the initial 2 entries of the memory can be loaded, given the presence of valid data.

In the subsequent iteration, the P1 predicate register is modified to turn each entry false where the FFR indicates true. In this specific example, this adjustment ensures that the next probing instruction will disregard the first two entries of the vector. The probing instruction then generates a new FFR, and since its first entry is false, it triggers an operating system trap, indicating that the end of the vector has been reached.

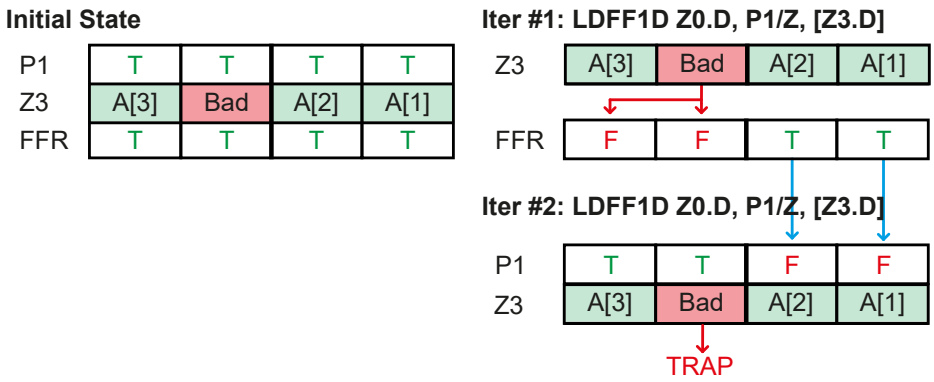

Figure 2.9: Simplified example of loop iteration and vector partition with FFR

Nevertheless, while this feature effectively addresses the challenge posed by an unknown data size, it introduces a potential downside. The increase in the number of instructions within the loop may contribute to heightened pressure on the processor pipeline. This, in turn, could impact the overall efficiency of the vectorization process. Balancing the advantages of dynamic size handling with the potential impact on pipeline performance becomes a crucial consideration in optimizing the execution of SVE instructions.

2.3.1.D Result Analysis

The authors of ARM SVE [14] conducted a series of workloads in 4 different configurations of the same generic medium-sized out-of-order microprocessor, one without the implementation of SVE, and three implementations with SVE and where the vector size was changed to 128-, 256- or 512-bit vector length. The overall results of the SVE implementations showed significant speedups, with workloads achieving 3x, 5x and 7.5x gains for the SVE 128-bit, SVE 256-bit, and SVE 512-bit implementation, respectively, when compared with the non-SVE implementation. Some workload tests presented no speedup due to the nature of the test since they do not benefit from vectorization, so performance was very similar between the four configurations.

2.3.2 RISC-V Vector Extention (RVV)

Just like the ARM Scalable Vector Extension (SVE) [14], the RVV [15] adopts an ISA that centers around a vector-length-agnostic programming model. RVV operations distinguish themselves solely based on vector-vector and vector-scalar distinctions, considering signed and unsigned types, without specifying element lengths. Consequently, the element length becomes a dynamic parameter in this context. Moreover, RVV allows the utilization of only a segment of a vector register or even a combination of multiple vector registers, introducing the necessity for dynamic parameters in such scenarios.

The RVV introduces a collection of 32 variable-length vector registers and 7 context and status registers (CSRs). These additional CSRs are integrated into the base scalar RISC-V ISA and serve the purpose of storing the vector configuration of the currently executing application. Among the config-

urations maintained by these CSRs are the vector length, data type, and the starting position of the vector.

2.3.2.A Operations and Execution

The manipulation of vectors is facilitated through the implementation of a new set of instructions introduced with the RVV extension. This extension supports typical arithmetic operations for both fixed and floating-point data, along with reduction operations such as sum, max, and min. Additionally, certain instructions can be customized through the use of masking, allowing the disabling or enabling of specific vector elements during processing.

In Fig. 3.1, an example of a simple saxpy application compiled with the RVV extension is illustrated. Throughout the loop iteration, vector registers are used to store data and vector-specific instructions to manipulate the vectors. At the beginning of the loop, an instruction ($vsertvli$) is utilized to define the characteristics of the vector, including its length and data type. Afterward, in the loop section, all instructions that manipulate vectorized data, do so with the information of the current vector configuration defined in the beginning.

2.3.3 Unlimited Vector Extension

The UVE [11] introduces a novel solution aimed at combining faster data access with increased system throughput by incorporating a data streaming engine that seamlessly integrates streaming with SIMD processing, thereby maximizing data parallelism. The envisioned ISA addresses common latency challenges associated with loop control, memory access indexing, and memory access by introducing specialized instructions for pre-configuring data stream and memory access patterns. This innovative access control reconfiguration feature enables precise, expedited prefetching, even in complex scenarios involving multidimensional arrays or indirect memory accesses.

Departing from conventional approaches that rely on predication or similar mechanisms, this method offloads such tasks to dedicated hardware units that can operate concurrently with the main processor pipeline.

This approach, as demonstrated by the UVE [11], shows promising results compared to previously discussed methods in this chapter. By combining SIMD operations with a dedicated data stream, this solution enhances the synergy of patterned computation without compromising gains with the increase of control instruction. Following this promising results, a foundational implementation of the envisioned SE was developed, setting out the basis for further consolidation and validation of the design. This thesis contributes to this consolidation effort, laying the groundwork for the future implementation and validation of the system on a GPP.

Accordingly, the following chapter provides a detailed explanation of the UVE instructions and architecture, establishing a foundational background for the work developed.

2.4 Summary

This chapter is divided into two main state-of-the-art concepts: Data Streaming and Vector Extensions. It begins by discussing Data Streaming, focusing on how the implementation of dedicated memory access structures, such as data streams, helps mitigate memory access issues. These structures reduce processing overhead caused by challenges like data indexing, memory address generation, and overlapping memory requests.

The first implementation of data streams is introduced as part of a hardware accelerator, functioning through a set of preconfigured DFG. These graphs are critical to the stream configuration in the processor, as they define the source and destination locations, as well as the access patterns.

The chapter then shifts focus to the use of streaming memory structures within GPP. It starts by detailing the necessary micro-architectural changes that offer the same benefits as a decoupled memory access system. This section also delves into the implementation of a new ISA that supports stream operations, such as pattern definition and data width configuration, to handle complex memory access patterns efficiently.

Before transitioning to the topic of Vector Extensions, the chapter presents an innovative solution to the von Neumann bottleneck in single-issue processors. This section outlines the micro-architecture and use of stream semantic registers to provide direct, encoded memory access, thereby reducing the number of instructions required for memory access operations. It also describes the individual units that compose this system and their execution flow.

The chapter concludes with an overview of state-of-the-art Vector Length Agnostic (VLA) extensions, namely the RVV and SVE. These extensions enable code to be executed across architectures with diferent vector lengths but introduce additional configuration instructions, which contribute to processing overhead. The section discusses the key design decisions of each extension, highlighting their access pattern limitations and prefetching techniques. Finally, the chapter offers a brief introduction to the UVE, providing insight into the content of the next chapter and an overview of the design choices that aim to overcome some of the limitations present in other Instruction Set Architectures (ISAs).

3

Background

Contents

This chapter serves as the technical foundation of the work developed in this thesis. The chapter starts with a detailed description of the UVE ISA, going over its capabilities and features, its defined architecture extension and its approach to stream's configuration and description.

The chapter then presents the CVA6 processor. This CPU is described along the chapter going over its execution pipeline as well as its memory subsystem. It is presented here as the chosen CPU for further development and integration of the SE with GPP.

3.1 Unlimited Vector Extension with Data Streaming Support

As previously introduced in section 2.3.3, this section further explains the features, architecture and behaviour of the UVE.

3.1.1 UVE Features

Figure 3.1 illustrates a comparison of the compiled code for a saxpy application targeting ARM SVE, RVV, and UVE. Initial observations suggest that all three vector extensions yield instruction codes of similar sizes. However, upon closer examination, the loop section of each implementation varies, with UVE exhibiting the smallest number of instructions per loop iteration.

Both ARM SVE and RVV need the inclusion of configuration instructions within their loop iterations, along with memory fetching instructions. This results in a higher number of executed instructions compared to UVE. UVE mitigates this issue by utilizing preconfigured stream registers, requiring only an initial configuration outside the loop iteration. This approach minimizes the number of instructions, leaving the loop solely for the actual computations.

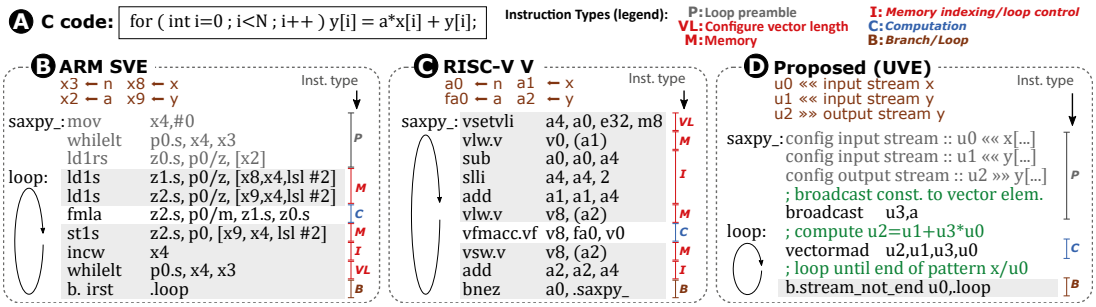

Figure 3.1: Assembly code comparison between ARM SVE, RVV, and UVE - Figure from [11]

The main difference of this solution from other recently developed SIMD solutions (ARM SVE and RISC-V Vector extension (RVV)) comes from features like:

F1 Decoupled Memory Access The ability to decouple the memory accesses from the computation part by streaming the data directly to the register, allowing the occurrence of loads/stores in parallel with the data manipulation, thus reducing the memory access latency.

F2 Indexing-free Loops By describing the load/store patterns in advance, it allows minimizing the number of instructions related to memory indexing, reducing the pressure on the processor

pipeline.

F3 Simplified Vectorization The use of memory access pattern descriptors allows the Streaming Engine to anticipate all non-coalesced accesses as linear patterns and deliver them to the processing unit. The examples shown in Figure 3.2 illustrate this principle since the code to run any of the access patterns will be the same from the execution pipeline point of view. The differentiating factors occurs only in the configuration part of the code.

F4 Implicit Load/Store Since the streams are described in the loop preamble, the indexing instructions can be removed, meaning that all explicit loads and stores are associated with active streams and different vector registers.

Besides the mentioned features, UVE is also register-size agnostic, similar to SVE and RVV. However, in UVE, there is no need for size control instruction since the Streaming Engine automatically disables all vector elements that fall out of bounds, making the loops simpler with a minimal set of control functions.

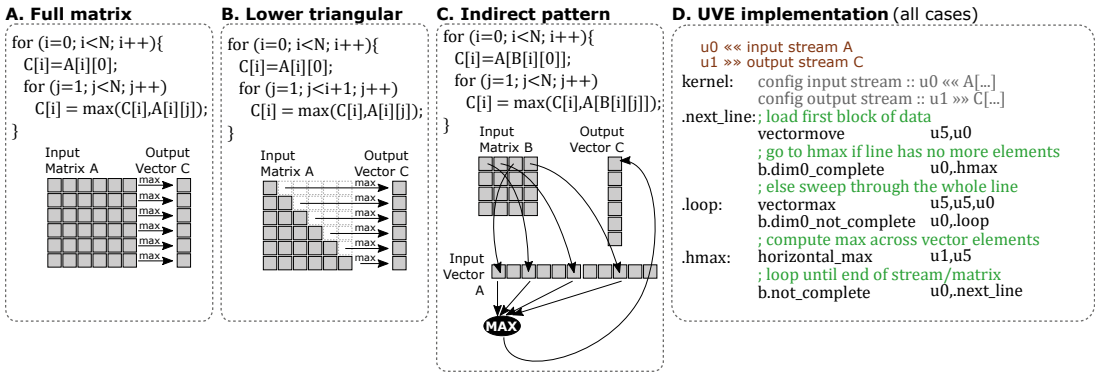

Figure 3.2: UVE pseudo-code implementation for the computation of the maximum across the rows on three possible matrices: (A) full matrix, (B) lower triangular matrix, and (C) full matrix with pointers to an array - Figure adapted from [11]

3.1.2 Microarchiteture Extension

In order to decouple the memory accesses the UVE implementation proposed by, Domingos *et al.* [11] assumed the incorporation of a streaming engine in the microarchitecture of a CPU core. Adding such a feature requires the modification of other preexisting components, like the instruction decoder (to support SIMD operations as well as the streams configuration, control, and manipulation instructions), modifying the reorder buffer to enable the renaming of vector registers and streams, allowing for speculative stream configuration, and lastly, add a new execution unit on the execution stage to process all the stream manipulating instructions. The proposed changes and their placement can be seen in Figure 3.3.

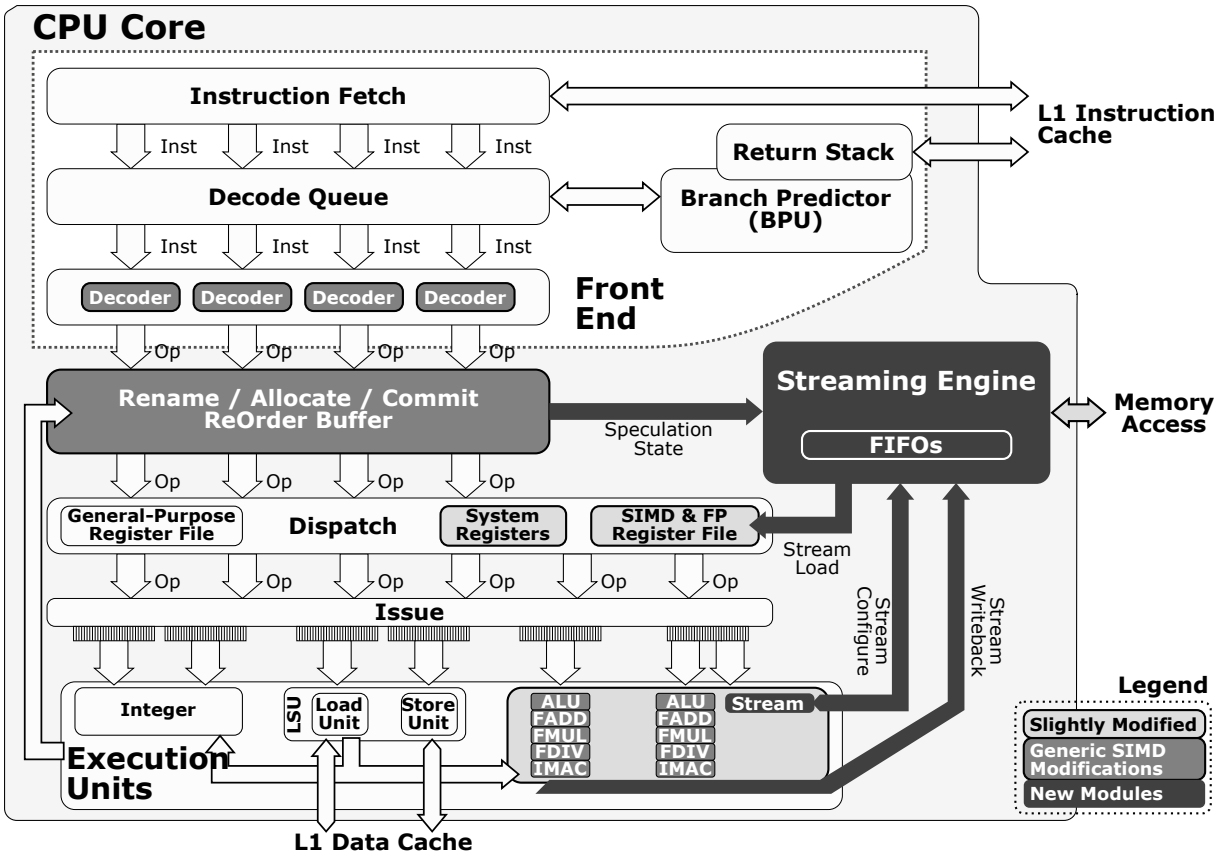

Figure 3.3: Modifications overview for UVE support

The implemented streaming engine has the responsibility of managing all the input and output streams. Its inner structure is depicted in Figure 3.4 and is composed of 4 main modules: the Stream Configuration Reorder Buffer (SCROB) (unit corresponding to the sorting queue and validation on the left of Figure 3.4), a stream table, a stream scheduler and an address generator, and a set of load/store FIFO queues.

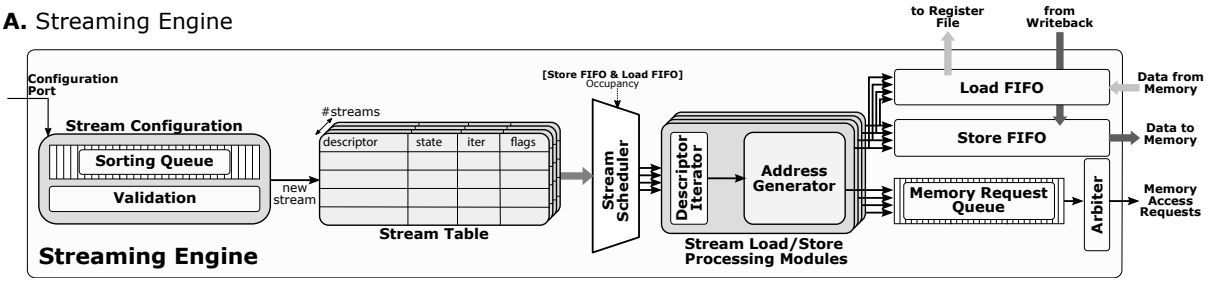

Figure 3.4: Streaming engine structure

3.1.3 Stream Configuration

The Stream processing starts with the configuration of the streams. Whenever a configuration instruction reaches the rename stage of the pipeline, it is moved to the first unit of the streaming engine, the stream configuration queue, where it will wait until all operands of the configuration descriptor are avail-

able. This approach of using a dedicated buffer unit is used since complex access patterns might require multiple instruction and holding such instructions on the pipeline commit buffer would prevent the speculative configuration of streams, impacting performance. Following the validation of the configuration, the configured stream is moved to the stream table, allowing the stream scheduler to start processing the stream by either pre-loading data for inputs (storing these requests in the memory requests queue and data on the Load FIFO) or calculating addresses for outputs (storing the generated addresses into the Store FIFO).

The iteration of the chosen streams occurs at each clock cycle and is decided through prioritization of lower FIFO occupancy, by the stream scheduler. Each stream processing model receives a stream and processes it according to their descriptor configuration values, iterating a single dimension and corresponding modifier at a time (this allows for a lower hardware complexity and size). On each iteration, an update is made to the stream table, keeping the table entries synchronized with the stage of each processed stream.

3.1.3.A Data Streaming Descriptors

As mentioned in the previous section, the configuration of streams is encoded in a descriptor-based representation. These descriptors provide the necessary values/parameters utilized on the proposed address modeling equation, defined by Eq. 3.1.

$$y(X) = y_{\text{base}} + \sum_{k=0}^{\text{dim}_y} x_k \times S_k \quad \text{with } X = \{x_0, \ldots, x_{\text{dim}_y}\} \text{ and } x_k \in [O_k, E_k + O_k] \tag{3.1}$$

Stream descriptors can be classified into two types: *Base Stream Descriptor* and *Descriptor Modifiers*.

Base Stream Descriptors define a uni-dimensional access pattern. It is represented by three-parameters $\{O, E, S\}$ corresponding to the offset (O_k), the size (E_k) and the stride (S_k). These patterns configure the base address of the memory access through the offset value (O_k), the total size of the stream to be loaded/stored with (E_k), and the value of the interleaving space between memory accesses with (S_k). Examples of simple patterns supported by descriptors of this type can be seen in Figs. 3.5.B1, B2 and B3.

Static Descriptors Modifiers are descriptors that can be used in conjunction with parameters of Base Stream Descriptors to modify the access pattern, allowing for loop-conditioned accesses, like, for example, a lower triangular memory access as depicted in Figure 3.5.B4. This descriptor is represented as the tuple T, B, D, E where T corresponds to the target parameter to modify, B is the modification operator (add or sub), D the value applied with the operator, E the number of times the modifier is applied

The **Indirect Descriptor Modifier** allows for the content of a stream to modify the values of another stream, creating indirect and indexed scatter-gather patterns. This modifier is represented by the tuple T, B, P, with T being the target parameter, B the behavior parameter (supporting adding a dynamic displacement, subtracting a dynamic displacement, and setting a value directly) and P corresponding to

the pointer of the origin data stream. Figure 3.5.B5 shows the use of this description in order to create indirect memory access on data of vector B by indexing it from values of vector A

As depicted by Figure 3.5.A1, A2, A3, all three descriptors can be used in multidimensional access patterns as well as in a combination of each other, creating complex memory access patterns.

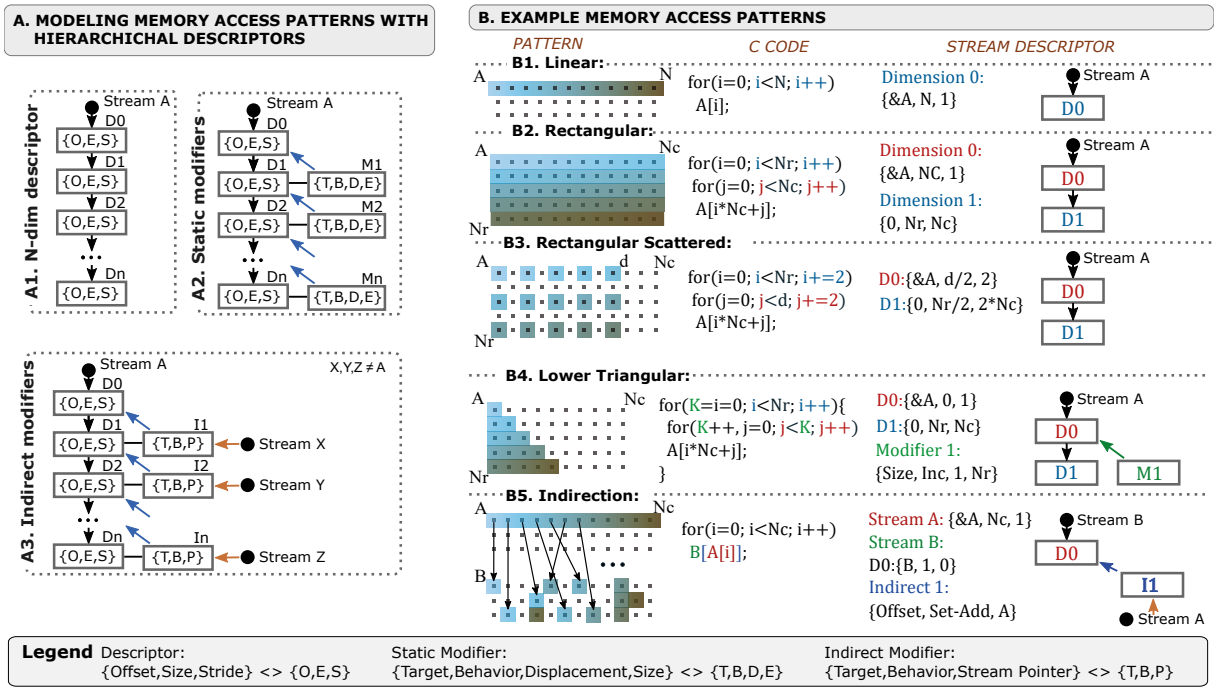

Figure 3.5: Examples of memory access patterns with UVE descriptors - Figure adapted from [11]

3.1.3.B ISA Extension

To support the added functionalities of the streaming data structures and the use/manipulation of SIMD vector operations, the ISA needs to be extended with the following type of instruction:

- **Stream configurations** instructions are responsible for encoding the descriptors/modifiers described above, controlling the behavior of the streaming engine.

- **Stream control** instructions enable the control of streams and allow vector registers to stop/restore momentarily. This behavior control enables the execution of multiple processes without interfering with stream configurations.

- **Predication and masking** allow the control of execution of SIMD operations by disabling selected lanes. The proposed UVE provides instructions for predicate configuration based on comparisons and valid vector register elements.

- **Loop control** is assured through three conditional branch instructions: predicate-based (condition tested on a specific predicate register), end-of-stream (a condition on the end of a stream), and end-of-dimension (a condition on a stream dimension end).

- **Vector manipulation** instructions allow the common arithmetic, logic, and shift operations to be applied to the vector registers.

3.1.3.C Experimental Results

All workload tests were conducted using a CPU simulation modeled after an ARM Cortex A76. The result analysis involved comparing the UVE streaming engine implementation with two ARM cores: one featuring only the ARM NEON Extension and the other supporting the ARM SVE.

The overall performance results demonstrated a significant advantage of 2.4x over the ARM SVE. The UVE's speedup was attributed to the reduction in compiled code and the streaming infrastructure, which effectively decreased load-to-use latency and enhanced memory utilization.

3.2 CVA6 Processor

The choice of the CVA6 processor [19] was predefined by the supervisors of this thesis and based on its production-quality attributes and open-source design, utilizing RISC-V as its underlying ISA and System Verilog (SV) as its Hardware Description Language (HDL). The decision was motivated by the processor's high extensibility, making it an ideal candidate for implementing the new streaming engine and its associated instruction set architecture extension.

Partially influenced by the processor choice, SV was also selected as the HDL for the consolidation of the streaming engine design. Although previous version of the streaming engine was implemented in Chisel, SV was ultimately chosen. This decision was driven by the need to evaluate timing, synthesis results, and the potential integration and testing on the CVA6 processor. Since Chisel hardware description capabilities come from computer generated code, correcting, optimizing, and analyzing the streaming engine would have been more complex, making SV the more practical option for hardware analysis and manipulation.

3.2.1 Architecture and Features

The CVA6 processor is a simple 6-stage, single-issue CPU that implements the RISC-V instruction set and can be configured to run in either 32-bit or 64-bit with support for floating-point operations [19]. It fully implements I, M, and C extensions of RISC-V ISA as well as a three-level privilege system. Its pipeline architecture was designed with the reduction of critical path length in mind, allowing for a operating frequency that ranges from around $400\,\mathrm{MHz}$ up to $1\,\mathrm{GHz}$ depending on the processor configuration. The processor architecture, its components and units can be observed in Figure 3.6.

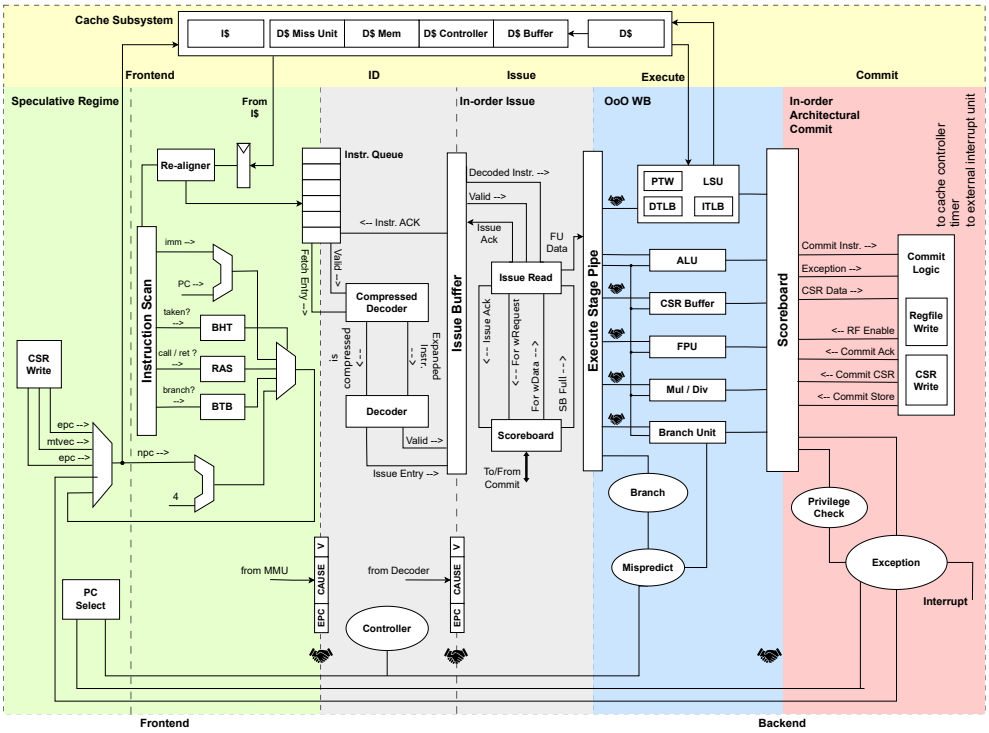

Figure 3.6: Overview of the 6-stage pipeline of CVA6

To enhance the Instructions Per Cycle (IPC) rate, the CPU incorporates, in its Issue Stage, a scoreboard mechanism, which aids in mitigating the latency of the data RAM (cache) by concurrently executing data-independent instructions, and committing the data for the load store unit (LSU).

3.2.1.A Issue Stage

The issue stage controls the execution flow of the processor pipeline, by receiving the decoded instructions and issuing them to the various Functional Unit (FU)s (FUs) on the execution stage. Furthermore the issue stage also keeps track of all issued instructions, the FUs status and receives the write-back data from the execute stage. This tracking process is achieved trough the use of a special data-structured called Scoreboard.

The flow of execution starts with the delivery of a new decoded instruction to the issue stage. As a first step, the FU required by the instruction is checked in order to evaluate if is free or will be free on the next cycle. Following the readiness of the FU the issue stage check if the source operands are available and if no other, currently issued, instruction will write the same destination register. This write destination constrain allows the simplification of the design of the forwarding pathing for read operands.

The ability to forward results from previous executions allows the scoreboard to provide source operands from instruction that would run back to back with no delay in between.

Although the write-back of each FU can happen out-of-order, the program flow is maintained in its natural order since the issue of instructions happens in-order. To allow this behaviour an ID is created, assigned to each issued instruction and stored in the Scoreboard. At the end of execution the issue

stage can then create the correlation between the finished operation and its results with the destination registers.

The Scoreboard is implemented as a FIFO with one read and one write port with valid and acknowledge signals, providing further control to the processors pipeline, mainly between the issue and commit stage.

3.2.1.B Memory Management Unit and Cache

The Memory Management Unit (MMU) of the CVA6 processor is comprised of two cache modules: the data cache on the execution state (*D$*) and the instruction cache on the frontend part (*I$*). Both of these units have fully set-associative caches and are configurable in size (before synthesis), ranging from *0 Kbyte (no cache)* to *32 Kbytes*.

Although very similar in purpose, the access to the instruction cache and the data cache, execute fundamentally different paths through the MMU.

In order to respond to an instruction fetch request the MMU will, depending on whether the address translation is enabled, either transparently let the request directly go to the *I$* or do address translation. In case address translation is activated, and theres a Instruction Translation Lookaside Buffer (ITLB) miss then, the cache pipeline is stalled until the translation is valid. The ITLB is implemented as a cache made out of flops and translation is purely combinational path, meaning that the request path of instructions can become quite long and critical.

The interface on the data memory side of the MMU is composed of a simple request-response handshake system. Either the load unit or the store unit requests the MMU to perform an address translation. In this case the translation process is not combinatorial, having an additional bank of registers, this change adds an additional cycle to the response, however reduces the probability of creating a critical path. Since the data fetch process would start with the address generation from the Load or Store units, this would certainly cause a critical path since address generation is a costly time process.

In addition to the cache memory, the CVA6 processor contains 2 Register File units, one for reading present on the issue stage of the pipeline and one for writing located in the commit stage, each with 32 registers. These units will be crucial components to alter and adapt for the implementation of the streaming engine.

3.2.2 Execution Pipeline

The flow of this pipeline starts in the frontend section where the Program Counter (PC) is updated according to the execution signals. These signals can indicate a normal advance across the executing process instructions or a modification according to branch prediction information, exceptions, or pipeline flushes. The instructions are then translated from the PC in the instruction fetch section of the frontend, and stored on the instruction queue. In the instruction decoding stage, an instruction is taken from the queue. It realigns, decompresses, and decodes the instruction, passing it to the issue stage, where it will await the availability of its operands. Execution is made out-of-order in the execution stage, which is

possible thanks to the Reorder Buffer (ROB) and scoreboard in the commit stage. This last stage uses the information stored in the ROB to commit the instructions and any possible produced data in the order in which they were issued.

The expandable nature and simple design of this 6-stage CPU make it the perfect candidate for the implementation of the proposed streaming engine on a hardware description, enhancing its capabilities.

3.3 Summary

This chapter begins with an introduction to the UVE, building on the concepts discussed earlier in this thesis, followed by an overview of the CVA6 processor. The description of UVE highlights its enhanced vector processing capabilities, achieved through an efficient data streaming process combined with SIMD operations. The proposed UVE implementation addresses some of the shortcomings found in other ISAs, such as ARM SVE and RVV, by incorporating features like decoupled memory access, indexing-free loops, simplified vectorization, and implicit Load/Stores. These improvements lead to significant reductions in the total number of executed instructions and memory access time.

The chapter then delves into the UVE microarchitecture, explaining the necessary changes required to integrate a streaming engine into a CPU. Modifications to components such as the instruction decoder, reorder buffer, and execution units are discussed, with a focus on supporting stream management and vector processing. The UVE section concludes with a description of the stream pattern configuration instructions, outlining their requirements and capabilities for the streaming engine.

The chapter concludes by presenting the CVA6 processor, an open-source, 6-stage single-issue CPU. This section provides an overview of the processor architecture, with particular emphasis on the ISSUE stage, which details the in-order issue and out-of-order write-back processes. The MMU and associated cache are also described, along with its internal address translation buffer. Finally, the execution pipeline is reviewed, completing the explanation of the execution flow of an issued instruction.

4

UVE Streaming Engine Implementation

Contents

The use of data streaming mechanism in High Performance Computing (HPC) has become a very prominent area of research with the implementation of different streaming mechanisms.

The following chapter does a detailed overview of the different modules Streaming Engine developed in accordance with the needs of the UVE micro-architecture.

The description of the streaming engine was divided in two main sections: the Stream Manipulation 4.1, focused on describing the configuration of streams and consequent iteration and state tracking, and the Data Handling 4.2, explaining some of the implemented mechanisms that improve on data requesting inefficiencies that exist on other data streaming solutions for HPC, like reusing data fetched from memory.

The Figure 4.1 below shows a pipeline style view of the two previously mentioned sections and their units. The operation flow of the streaming engine resembles a processor pipeline due to its configurable nature and its almost linear stream manipulation, skewing from it on the data load and store stage.

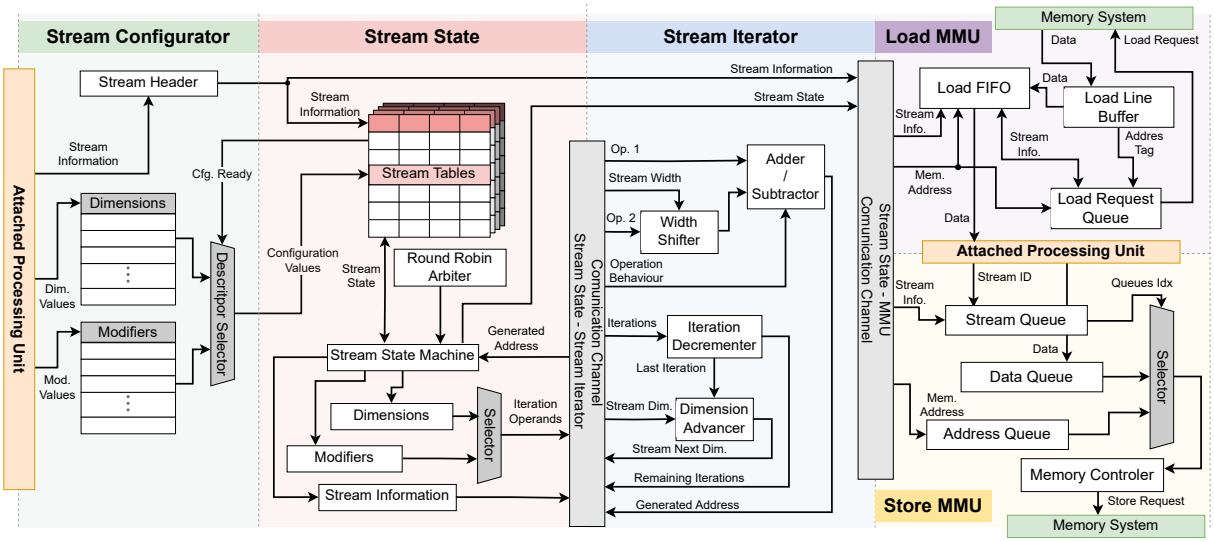

Figure 4.1: High level overview of the Streaming Engine's Architecture

The presented streaming engine encompasses an architecture very similar to the one presented on the UVE paper [11], with the added modifications proposed by Xavier Fernandes in is thesis work [20], as well as the ones to be presented on this thesis. The initial implementation of the SE by Xavier Fernandes served as a foundational prototype, establishing the groundwork for future research on the UVE streaming engine. Building on this foundational work, this thesis consolidates the SE's design, providing an in-depth analysis and evaluation of its architecture, along with a critical assessment of its design limitations. This work aims to provide the base for the next step of development, which corresponds to the integration of the SE into a GPP.

On the Stream Manipulation side the engine is composed by: a Stream Configuration Unit, tasked with receiving the new stream configuration parameters; a Stream State Unit, used to store each configured stream iteration state on the Stream Tables module; a Stream Iteration Unit, to generate the new memory address values for the selected stream. On the Data Handling side of the streaming engine two Memory Management Units were implemented, one for memory Load request, which utilises FIFO's,

queues and buffers, and one for memory Store Requests.

Throughout this chapter a detailed description of the behaviour of each unit will be made, covering from their internal structure to its functionality.

4.1 Stream Manipulation

4.1.1 Stream Configuration

In the UVE version, on which this streaming engine is based, streams are described to the system as a series of instructions, each defining either the dimensional parameters or the modifiers of the stream pattern. The complexity of the stream access pattern determines the number of dimensions and modifiers, meaning a stream configuration can be as simple as a single instruction or as complex as ten or more.

Due to the design of the ISA, stream configurations are processed sequentially, which requires an initial set of cycles to load the complete stream pattern. This process is managed by the Stream Configurator (SC) unit. This module serves as the communication handler between the processing unit and the other components of the streaming engine. It is responsible for receiving the configuration values of new streams and generating the corresponding stream signals used by the streaming engine.

4.1.1.A Communication Interfaces

The SC operates through two main communication interfaces, shown in Figure 4.2. The first interface is used for communication with the processor pipeline, receiving external information that describes the stream access pattern. This information is extracted from the configuration instructions by the CPU's decoding unit, which then passes the stream data to the streaming engine in the form of a set of values. The second interface connects to the other units within the streaming engine and is used to transmit the configured stream information. This enables the initialization of the new stream and triggers the address generation and memory requests associated with it.

Positioned between the two systems (the CPU and the streaming engine), the SC acts as the master of the streaming engine while serving as a subordinate when receiving stream configurations from the CPU. To manage communication across both interfaces, a set of handshake signals is implemented, which informs all parties—the processing unit, the SC, and the other streaming engine units—of their readiness and availability states.

4.1.1.B Internal Architecture

As shown in Figure 4.2, the SC logic is relatively simple, consisting of a state machine that manages the reception and delivery of the stream pattern from the CPU to the rest of the streaming engine.

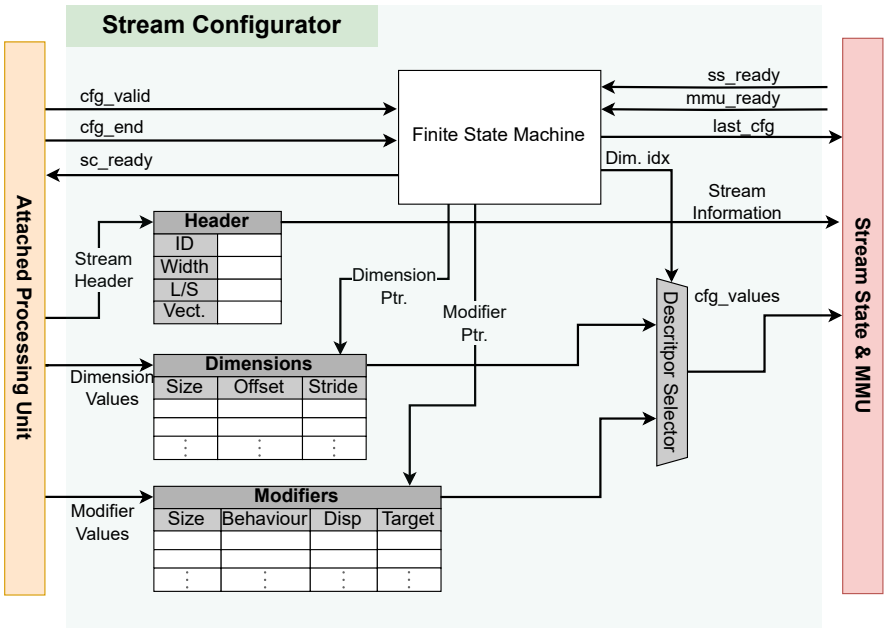

Figure 4.2: Diagram of the Stream Configuration Unit and its connection with the CPU, the Stream State Memory Management Units (MMUs)

Since the stream configuration is received over multiple clock cycles, the SC must retain the information until it receives an acknowledgment that all data has been delivered and can be passed to the rest of the streaming engine. The SC utilizes two distinct register tables: one for dimension descriptor information, which stores the size, offset, and stride values for each dimension, and another for static modifier descriptors, which hold the target, behaviour, displacement, and size values. Both tables are accessed via pointers, with each entry corresponding to a single descriptor.

In addition to storing dimension and modifier information, the SC registers also hold the stream header, which includes the stream type (Load/Store), stream identifier, and element width. This information is crucial for configuring the behaviour of the address generation unit and the memory management units within the streaming engine.

4.1.1.C Execution Flow

As described in the beginning of the Internal Architecture section 4.1.1.B, the SC operates using a state machine with three states that correspond to the three phases of stream pattern configuration. The state machine diagram and its transitions are illustrated in Fig. Figure 4.3.

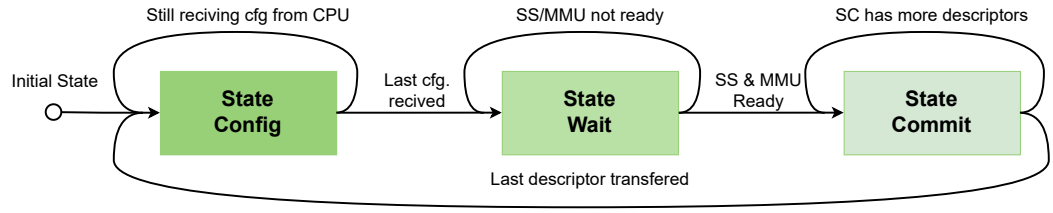

Figure 4.3: Stream Configuration Unit - State Machine and Transition

At the start of execution, upon receiving a reset signal, the SC clears all internal registers and enters the **STATE_CONFIG**, where it waits for decoded configuration instructions from the CPU. The SC remains in this state while receiving the streams information from the processor, until an end signal (*cpu_end_i*) is sent by the CPU.

The start of a new stream configuration is signalled by the CPU, at which point the stream header information, including the *stream ID*, *type* (load/store), and *element width*, is stored. During this phase, all valid dimension and modifier configurations received are processed and stored in the corresponding register tables. As noted earlier, pointers are used to correctly map the incoming pattern information to the tables. For the modifier, the pointer value of the last stored dimension is also saved to ensure the modifier is associated with the correct dimension.

When the stream configuration is fully delivered, the CPU signals the end of the configuration, prompting the SC to transition to the next state: **STATE_WAIT**. With the stream pattern information loaded, the SC waits for a readiness signal from the rest of the streaming engine, during this stage, the SC notifies the processing unit that no further stream configurations can be accepted until the current one is fully passed onto the streaming engine.

To initiate the transfer of the stream pattern, the SC receives ready signals from the Stream State (SS) Unit, the Load Memory Management Unit (LMMU), and the Store Memory Management Unit (SMMU). These signals are then asserted based on the type of stream currently stored: for a load-type stream, only the SS and LMMU signals are relevant, for a store-type stream, the SMMU signal would be used in place of the LMMU.

After completing the handshake, the stream header information is passed to the SS Unit and the corresponding MMU. This initiates the configuration of the streaming engine modules and transitions the SC to the **STATE_COMMIT**, where the descriptors data is transferred to the rest of the streaming engine.

In this final state, the SC signals the rest of the streaming engine modules to prepare to receive a new stream and initialize the necessary register structures. The SC then transfers the descriptors to the SS Unit, doing so at a maximum rate of one dimension descriptor and one modifier descriptor per clock cycle. During this phase, the pointers previously used for writing to the register tables become read pointers.

Upon transferring the last descriptor, the SC sends an acknowledgment signal to the streaming engine, triggering the processing of the stream. After this transfer it returns to the initial **STATE_CONFIG**,

informing the CPU that it is ready to receive the next stream configuration.

4.1.2 Stream State

The SS module can be considered the next step in the lifecycle of a stream. It is responsible not only for storing the values related to the stream's access pattern and address generation, but also for managing the state of the stream. As such, the SS is a core component of the streaming engine.

4.1.2.A Communication Interfaces

As a central element of the Streaming Engine, the SS has a total of four communication interfaces, each serving different modules and purposes:

C1 Configuration Channel Used to receive new stream configurations, and connected directly to the SC, this interface handles the reception of the stream's header information, a dimensional descriptor values and a modifier descriptor values. In addition to the data from the SC, two additional signals — related to the configuration but not directly connected to the SC — are the pointer values that reference the configured stream within the MMUs register structures.

C2 Iteration Channel Responsible for communicating the selected stream's dimension and modifier values to the Stream Iterator (SI), this interface provides the necessary data for calculating each memory address of the selected stream. It also handles the reception of the calculated address generated by the SI (see Section 4.1.3).

C3 Memory Controller Channel This interface is used to route the received memory addresses of the selected stream to the corresponding MMUs. It includes two sets of signal channels, one for each stream type. Based on the stream type, the addresses are transferred either to the LMMU (see Section 4.2.1) or the SMMU (see Section 4.2.5). In addition to the generated addresses, this channel also communicates the stream state to the MMUs, enabling them to generate data vectors according to the stream dimensions that have been completed.

C4 Stream Tables Channel This communication channel links the SC with the Stream Tables (ST), a submodule responsible for storing stream states in a set of register tables (see Section 4.1.2.C). In addition to the stream configuration signals, table entry information is exchanged between both modules.

4.1.2.B Internal Architecture

Similar to the SC, the Stream State module also includes a finite state machine that manages the transfer of stream state values to the Stream Iteration Unit, updates the stream state, and receives the generated addresses, which are then passed to the MMUs. Within its internal structure, as illustrated in Figure 4.4, the SS also includes a submodule called the Stream Tables, which stores the table registers for the configured streams and their current states.

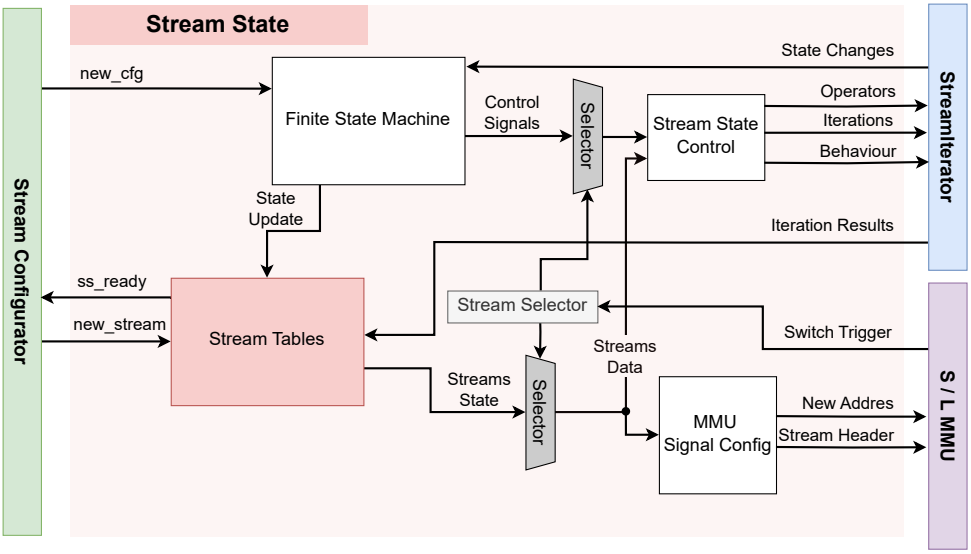

Figure 4.4: Stream State Architectural Overview

4.1.2.C Internal Architecture - Stream Tables

Since the Stream State module is capable of handling multiple streams concurrently, a lot of stream information and states needed to be stored, to simplify the design, the Stream Tables submodule was developed containing a set of multi-dimensional register tables. Figure 4.5 illustrates the composition of the stream table registers.

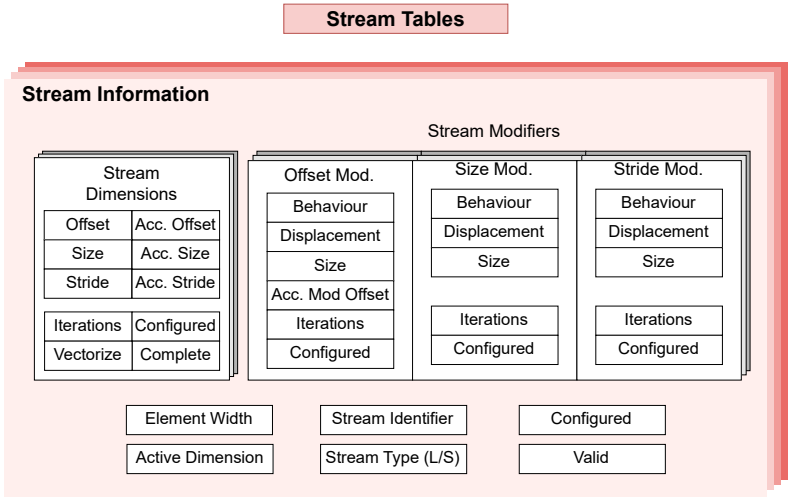

Figure 4.5: Stream Tables Overview

A stream is defined by its dimensional and static modifier descriptors, along with other header information, all of which must be stored in the ST to ensure proper processing. Each stream table consists of registers that store key header data, such as element width, stream identifier, and stream direction

(load or store).

In addition to these registers, each stream table has two sub-tables (see Figure 4.5): one for dimensional descriptors, which contains parameters like offset, stride, and size, and another for static modifier descriptors, which holds modifier parameters. Each table includes multiple registers, as streams may have numerous descriptors.

For the dimensional descriptors, in addition to storing the initial values for size, stride, and offset of each dimension, each entry also retains the modified values of these parameters. This is necessary because, during stream processing and address generation, modifiers may be applied, altering the base values of any of the parameters that define a stream dimension.

Since each dimension can have up to three distinct modifiers, the modifier descriptor registers must be capable of storing all three types of modifiers simultaneously. To achieve this, the table is divided into three smaller tables, each containing the necessary registers for each type. This structure implicitly defines the modifier type by requiring the tables to be indexed by type.

Both descriptor sub-tables include a register in each entry for tracking the number of iterations. This value is used to determine whether the descriptor has reached the desired size — signalling completion in the case of a dimensional descriptor, or signalling a reset in the case of a modifier descriptor.

4.1.2.D Execution Flow

Diferent sets of processes occur in the Stream State module, from the configuration of new streams, to the consequent iteration and address generation. In order to better analyse the execution flow, it will be divided in the major tasks:

Stream Table Initialization:

To begin processing a stream, an initial state must be created and recorded in the SS unit, which occurs via the Configuration Channel. Once the SC has finished receiving a new stream configuration from the processing unit, it signals to the SS that the stream is ready to be configured in the engine. The SS then checks its register tables for available space. If at least one unallocated table entry is found, it sets its ready state to *HIGH*, initiating a handshake between the SC and the SS.

Upon the handshake, the newly configured stream does not immediately begin processing. Instead, it starts receiving stream information, with a maximum transfer rate of one dimensional descriptor and one modifier descriptor per clock cycle. As a result, more complex stream patterns will require additional cycles before the SS has all the necessary information to begin processing. During this stream-loading phase, the table entry being configured will keep its ready flag set to *LOW*, indicating that it is not yet ready for processing.

Stream Selection:

Due to the SS's ability to hold the state of multiple streams simultaneously, a mechanism was developed to select a stream for the iteration process. The chosen method is a simple Round-Robin unit, which takes as input the set of tables that are configured and valid for iteration.

The selection changes based on a trigger that switches to the next valid and configured stream. This trigger is asserted based on the processing state of other streaming engine units and activates whenever

processing the currently selected stream would cause a stall. This ensures that the Streaming Engine optimizes processing time by avoiding unnecessary stops.

Address Iteration: Although the calculation of the address value does not occur directly within the SS unit, its role in the process is crucial. The complete lifecycle of a stream is managed by the SS through a state machine. Each stream when being processed enforce its current stage to the SS, allowing multiple streams to be in different states of their lifecycle simultaneously.

To better understand the SS's role in stream address generation, on the next sub-section a description of each state that a stream can occupy will be made.

4.1.2.E SS State Machine

As previously mention each stream maintains its own state, starting in **STATE_UPDATE**, as indicated in Figure 4.6.

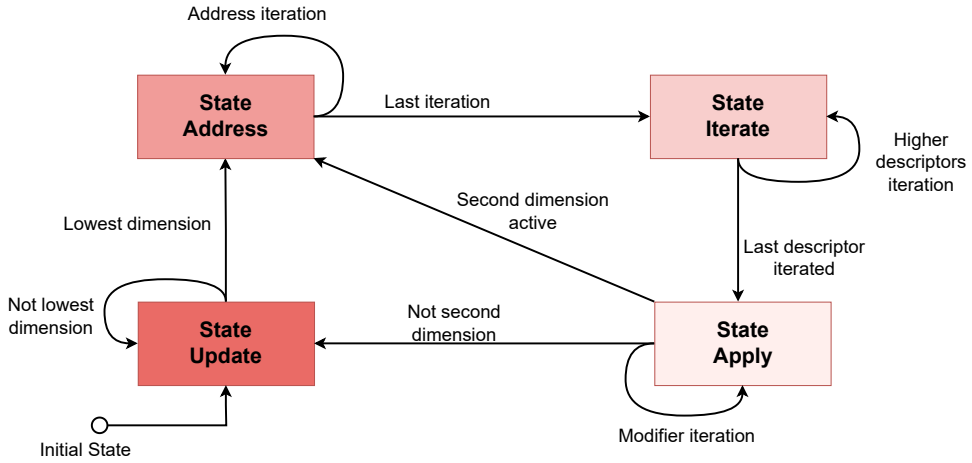

Figure 4.6: Stream State Unit - State Machine and Transition

STATE_UPDATE: This is the initial state of the stream. Following the behaviour defined by the UVE (see Section 3.1.3), stream iteration begins with the highest configured dimension and proceeds toward the lowest.

During this phase, the offset values for each dimension are accumulated, ensuring that by the time the lowest dimension is reached, the correct initial offset is applied for iteration. As the stream dimensions are traversed, all accumulated size and stride fields are reset to their initial values, clearing any changes caused by stream modifiers. This reset only occurs when transitioning into **STATE_UPDATE** from another state, not at the very beginning of the process.

The stream's state machine remains in this state until it reaches the lowest dimension. Once this is achieved, the system transitions to the **STATE_ADDRESS** state, initiating a round of address generation for the selected stream.

STATE_ADDRESS: Considered the second step in address generation, in this state the lowest

stream dimension is active, generating a new address for the stream in each cycle. This process occurs in the SI by adding the accumulated offset and accumulated stride values of the lowest dimension.

During this state, two stream values are updated: the accumulated offset, which reflects the generated address, and the iteration count for the dimension, which is decremented with each iteration.

The stream's state machine remains in this state until the iteration count reaches zero, indicating that the lowest dimension has been fully iterated. At this point, the next phase of address generation, which involves iterating over a higher dimension and its modifiers, is triggered, causing a state transition to **STATE_ITERATE**.

STATE_ITERATE: During this phase, the SS unit manages the iteration of the stream's higher-dimensional values. The configured descriptor values are passed to the SI for processing. Since each higher dimension can be iterated and may have up to three associated modifier descriptors (size, offset, and stride modifiers), a total of four iterations can occur per dimension. The results of these iterations will then affect the behaviour of the next lower dimension.

A queue is used during this phase to schedule the descriptors iteration. The values for iteration and the ST registers modified are determined by the type of descriptor being processed.

- **Iteration of Dimension** - In this case, the accumulated offset and stride are passed to the SI, and the result is stored as the next accumulated offset for the dimension.
- **Iteration of Size Modifier** - The SS adjusts the accumulated size of the immediately lower dimension by incrementing or decrementing it with the displacement value in the modifier descriptor. The result is stored in the accumulated size register of the lower dimension.
- **Iteration of Stride Modifier** - Similarly, the SS modifies the accumulated stride of the immediately lower dimension with the increment or decrement of it with the displacement value in the modifier descriptor. The result is stored in the accumulated stride register of the lower dimension.
- **Iteration of Offset Modifier** - Here, the SS modifies the accumulated offset of the current dimension by the value specified in the offset modifier descriptor. The result is then stored in the accumulated offset register of the current iterated dimension.

For each of the four types of iteration described above, the SI also provides the iteration count for each descriptor. This value is stored in the corresponding descriptor's register. Whenever the iteration count reaches zero, one of two scenarios occurs:

- If the descriptor is a dimensional descriptor, the dimension is marked as complete, and the SS schedules the iteration for the dimension immediately above, processing it accordingly.
- If the descriptor is a modifier descriptor, its iteration count is reset to the initial value, and the modified values are reverted to the base values of the dimension it influences.

At the conclusion of the iteration of the final scheduled descriptor, the SS transitions to the **STATE_APPLY**.

STATE_APPLY: Since the iteration of the offset related descriptors do not immediately propagate their updates to the lower dimensions, this state serves to accumulate the base offset of the lower dimension with the accumulated offset of the active dimension, along with the offset from the modifier descriptor calculated in the previous state. These operations occur sequentially and may take up to two clock cycles if both the dimensional offset and modifier offset are present.

Depending on the active dimension, the state machine can transition to either **STATE_UPDATE** - if the current active dimension is not the second-lowest dimension - or **STATE_ADDRESS** - if the current active dimension is the second-lowest dimension, proceeding to address generation again.

4.1.2.F Address and MMU information transfer

In order for the streaming to provide the CPU with the necessary data values, or to store incoming values from the CPU, the generated address of the configured stream must be processed by the appropriate MMU to handle the load/store request to the system's memory. For this reason whenever the SS unit has a valid stream selected for address generation and the state machine is in **STATE_ADDRESS**, the MMUs are informed that a valid address is being generated, along with the selected stream ID, width, type, vectorization, and completion state.

In response, the corresponding MMU (based on stream type) signals its readiness to accept new memory addresses into its table registers and queues. If, however, the MMU is unable to accept the new address - potentially causing a stall in stream iteration — it signals back to the SS that it is not ready. This prompts the SS round-robin arbiter to select a different configured stream for processing and iteration.

4.1.3 Stream Iterator

Although different types of descriptors use distinct iteration operands — such as size, strides, and offsets — the arithmetic calculations required are quite similar, making it easy for the SI to adapt. As a result, the hardware complexity of the SI is fairly simple, utilizing two adders and purely combinational logic. This means all generated results are directly stored in the SS, as mentioned previously.

The architecture of the SI is shown in Figure 4.7.

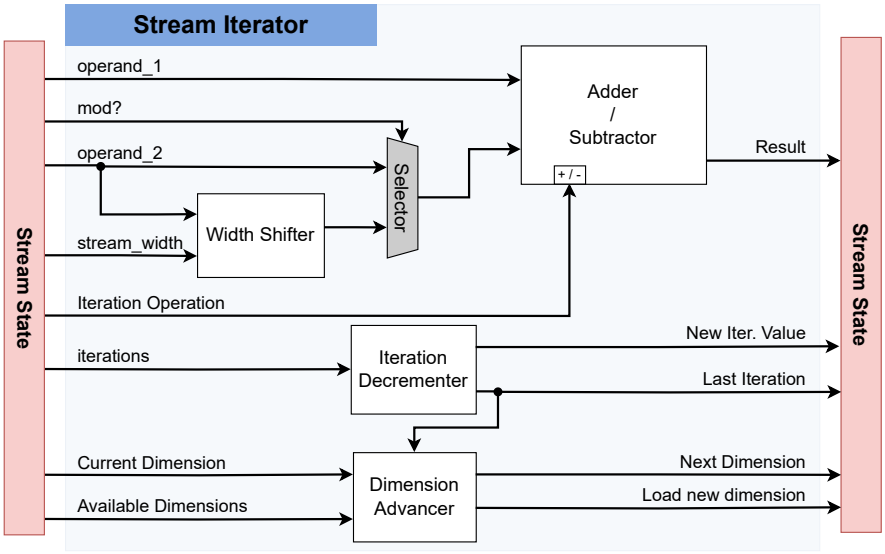

Figure 4.7: Stream Iterator's Architecture

To adapt the SI to different kinds of descriptors, the inputs and outputs of the module are kept as abstract as possible, allowing for essentially two operations: the addition or subtraction of two operands.

For normal address generation and dimension iteration, the signal *src_op1* represents the accumulated offset of the stream, while *src_op2* defines the stride between addresses. During dimension iteration, *src_op2* is shifted according to the element width, ensuring that the address is always aligned with the data width.

Similarly, the iteration of modifier descriptors can be handled by a single *Adder/Subtracter unit*. In these operations, the two source operands are the offset, stride, or size value to be modified and the amount to add or subtract, depending on the operation type (*mod_behaviour*).

In both types of iterations, the remaining number of iterations for the descriptor is also calculated, enabling the SI to signal when the next dimension should be loaded for iteration.

4.2 Data Handling

Building on the core logic of stream iteration and address generation in the streaming engine, this section will cover the approach taken for communication between the streaming engine and the memory system. It will also discuss the capabilities of the chosen design and the internal architecture of each module that constitutes both the Load Memory Management Unit and the Store Management Unit.

4.2.1 Load Memory Management Unit

When designing an efficient Memory Management Unit (MMU), it is essential to address the common challenges associated with data loading. One of the primary factors contributing to low system performance is the high latency involved in memory access, which often causes processing stalls across

various operations. Therefore, it is crucial to evaluate how a data streaming approach, as outlined in the previous section, can improve the loading process. While the proposed system is capable of handling multiple streams, it only generates one address per clock cycle. In a naive approach these addresses could be dispatched individually, but doing so would underutilize the advantages of streaming and the patterns inherent in data streams, like the possibility of data re utilization and parallel processing.

With these characteristics in mind, the design of the LMMU consists of three complex modules that facilitate communication between the memory system and the attached processing unit. The primary goals are to handle the maximum address generation throughput of the SE while minimizing both the number of memory operations and their execution time. Managing the generated addresses and dispatching requests to memory through these modules enables the exploration of more complex behaviours, particularly the parallel execution of tasks related to data loading. This approach has led to the development of advanced features such as:

F1 Requested Data Optimization Since different memory addresses can be mapped to the same memory row, a single fetch can retrieve data for multiple addresses, thereby reducing the total number of requests made.

F2 Outstanding Request Scheduling Generated addresses are grouped using a common address tag, allowing them to be scheduled as a set of outstanding requests.

F3 Data Re-Utilization By leveraging address grouping through address tags, multiple outstanding requests can be fulfilled with the data retrieved from a single load operation.

4.2.2 Load FIFO Queues

4.2.2.A Communication Interfaces

Acting as an intermediary for data transfer between the memory system and the attached processing unit, the Load FIFO Unit (LF) has two primary communication interfaces: the stream state channel, which connects to the Streaming Engine and communicates the stream state along with associated memory addresses; and the outgoing data channel, which connects to the processing unit and transmits the values fetched from memory. The LF also comunicates with the other two modules (Load Request Queue (LRQ), Load Line Buffer (LLB)) in order to synchronize, data and stream information exchanges. Figure 4.8 provides a visual overview of the modules that compose the LMMU and their connections.

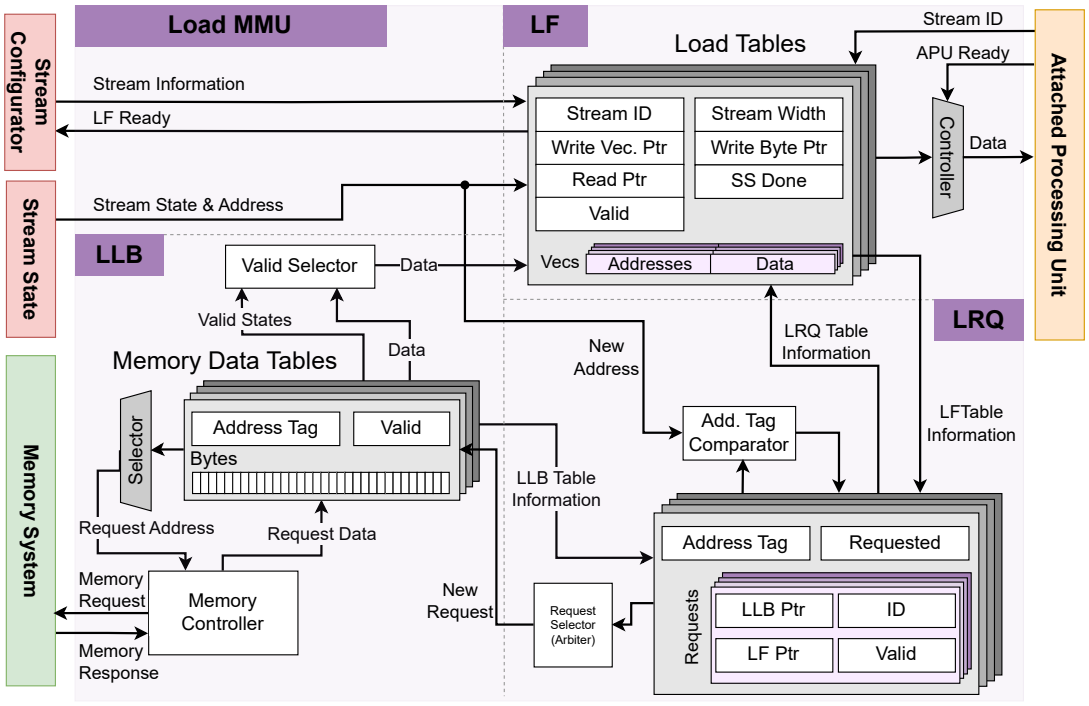

Figure 4.8: Overview of Load Memory Management Unit

4.2.2.B Internal Architecture

The LF unit is responsible for storing loaded data before delivering it to the attached processing unit. To achieve this, the unit implements a collection of register tables, each containing the associated stream metadata (stream identifier, and width) and a set of vectors (buffering queues) to temporarily hold the data before passing it to the processing unit.

Each table entry is equipped with a set of pointers that control the writing of data into the correct vector and correct byte, ensure proper data coalescing. Additionally, each vector entry also contains a **valid** flag and a **predicate** flag. The **valid** flag indicates whether the specific byte contains valid data, while the **predicate** flag determines whether the data should be processed or ignored.

Once a vector reaches its full capacity, a **reserved** flag is activated, advancing the vector pointer and signalling the availability of data to the attached processing unit.

Finally, each Load FIFO table entry includes a **done** flag, which indicates whether the Streaming Engine (SE) has finished generating addresses for the configured stream and it is only awiting for the data to be fetched from memory.

4.2.2.C Execution Flow

The execution of this unit can be described in two steps: data loading and data consumption.

During the initial configuration of a stream in the SS unit, an entry in the LF register table is set up. This process clears any values from previous configurations and sets the stream's characteristics,

such as its ID and width. The configuration of this entry also generates a corresponding index, which is passed to the SS and linked to the stream's state to facilitate easier address generation and delivery to the LMMU.

At this stage, the configured entry initializes its pointers to the start of the first vector, having each entry in the vector marked as invalid. As the address generation phase progresses, addresses are passed to the LF, advancing the byte pointer according to the defined stream width. Whenever the byte pointer reaches the last element of the vector, the vector pointer is advance, the vectormarked as reserved, preventing any changes to the vector's predicate and awaiting data availability.

In parallel, the LRQ and LLB handle the appropriate data requests from memory, ensuring that the correct data is mapped to the corresponding entry in the LF buffer queues. Once the vector contains all valid bytes, the data is ready for consumption.

Data consumption is initiated via a handshake protocol. The processing unit signals its readiness to receive data, while the LF unit verifies the validity of the vector currently pointed by the read pointer. This read pointer ensures the correct ordering of data to the processing unit.

Once these conditions are met, the data in the vector is transferred to the processing unit, and the vector's entries are cleared, allowing the allocation of new addresses for subsequent data mapping.

4.2.3 Load Request Queue

The LRQ unit serves as the planner for memory requests, managing the reception of memory addresses from the Address Generation Units (AGU) of the SE and accurately translating them into memory requests. Given that memory addresses in memory systems are organized by rows, the LRQ is designed to take advantage of this structure by grouping incoming memory requests with identical address tags. This organization enables the efficient processing of multiple memory addresses with a single memory load operation, thereby optimizing this costly action.

4.2.3.A Internal Architecture

As previously noted, the Load Request Queue (LRQ) stores the generated addresses for the configured streams using a register table. As shown in Figure 4.8, each entry in the register table consists of a grouping address tag, a request signal, and a list of requests. Each request within the queue table represents a single generated address, which includes the write pointers for both the Load FIFO Byte (LFB) pointer and the offset of the corresponding value in the full row of data obtained by the Load Logic Block (LLB). This pair of pointers enables the Load FIFO (LF) to coalesce data received from the LLB whenever each row of data is passed onto it.

The design of the Load Request Queue (LRQ) Tables allows the storage structure to handle complex address patterns during runtime, though the underlying resources remain fixed which can cause limitations. Each LRQ entry is allocated a set number of requests that can be grouped under a single address tag. Consequently, depending on the stream pattern and the specified maximum number of requests per entry, multiple addresses sharing the same address tag may exceed an entry's request capacity, neces-

sitating allocation in an additional table entry. A detailed explanation of the request allocation process will be provided in the subsection ”Execution Flow”.

4.2.3.B Execution Flow

After configuring a new stream in the SE, new addresses begin to be generated and sent to the LMMU. When a new address arrives at the LRQ, the address tag is checked against existing entries. Based on the result, one of two actions is taken:

B1 **New Request Entry** If the received address's tag does not match any existing tags in the request queue, or if the relevant entry lacks available request slots, a new request queue entry associated with the address tag is created.

B2 **Address Grouped to a Request** If the address tag corresponds to an existing request queue entry that has available request slots, the new address is added to that entry.

It is important to note that incoming addresses can only be processed if they can be registered in both the LF and LRQ modules, as this relies on the close coordination of these modules and their pointer references.

As previously mention since each table entry has a limited number of request slots, certain stream patterns that generate consecutive memory addresses may fill up an entry for a particular address tag, requiring the configuration of an additional request queue entry. This limitation could be mitigated at the expense of additional hardware resources, as analysed in Chapter 6, where the impact of various SE configurations is discussed.

Once a new address is registered in a request queue, the LRQ employs a Round Robin Arbiter to issue a load operation to the LLB. This arbiter takes as input the table entries that have been configured but are not yet requested and selects an address tag to process.

The request communication between the LRQ and the LLB is initiated through a handshake protocol between the two units. Upon a successful handshake, each LRQ table entry with the same address tag is marked as requested, enabling the resolution of multiple outstanding requests (this feature is further explained in the next subsection). Once the LLB indicates readiness to handle a new memory request, the LRQ arbiter is reactivated to proceed to the next configured address tag.

4.2.3.C Outstanding Memory Requests

As previously mentioned, memory addresses are grouped in the LRQ request table by their address tag. However, in cases where multiple addresses share the same tag and exceed the maximum requests per table entry, multiple entries may be allocated to the same address tag.

To minimize redundant load requests for identical address tags, the LMMU is designed to resolve multiple outstanding memory requests with a single load operation.

Through close communication between the LRQ and the LLB, when an address tag is selected for a memory fetch, the LRQ signals all relevant entries in its table that correspond to this tag as requested.

Once data is delivered by the LLB, the LRQ then instructs the LF on all data vectors that can be populated from that single load request.

4.2.4 Load Line Buffer

Tasked with direct communication with the memory system, the LLB is a key module within the LMMU. Its purpose is to issue load requests to the memory system, manage incoming response data, and hold it until the LMMU signals that all requests related to the fetched data have been resolved.

4.2.4.A Internal Architecture

As shown in Figure 4.8, the main architecture of the LLB comprises several buffering tables, as it is responsible for both issuing load requests and storing retrieved data for the LMMU. Each entry in this storage structure can hold a row from memory and is associated with a single address tag. To support this, each entry includes an address tag register, a valid signal (indicating data validity), and an array of bytes sized to match the elements in a memory row.

The LLB uses the Advanced eXtensible Interface 4 (AXI4) [21] protocol, and the module also includes a set of registers to assist in delivering load requests to the memory system.

4.2.4.B Execution Flow

As mentioned earlier, the LLB works closely with both the LRQ and the LF units. When a new address is registered and processed by the LRQ, the LLB initiates a new memory access request. This process begins with receiving an address tag from the LRQ. If there is available space in the LLB tables, the address tag is registered in a new entry and added to the queue. Since the LLB can only process one memory operation at a time, a Round Robin Arbiter is used to sequentially select configured table entries.

To manage the memory request, the LLB uses a Finite State Machine (FSM). The first stage, STATE_WAIT_ARREADY, is the default stage when no memory operation is active. In this stage, the LLB waits for the next address tag to be selected by the arbiter. Upon selection, the LLB initiates a handshake with the memory system; once this handshake is complete, the load request is issued, and the FSM transitions to STATE_WAIT_RVALID.

In the final stage, STATE_WAIT_RVALID, the LLB actively checks for valid data from the memory system. Given the flexibility of the AXI4 protocol, the communication bus width can be configured to be smaller than the memory row size. This characteristic means that data may be delivered in multiple bursts. The LLB is designed to handle this by incrementally loading segments of data into its buffer as they become available. Upon receiving the final burst, the LLB signals readiness for a new request and returns to the initial FSM state.

4.2.5 Data Store Management

Similarly to the LMMU, the SE includes a dedicated unit responsible for managing its write operations, the SMMU. This module was designed with a straightforward architecture, foregoing certain optimizations. Potential modifications to enhance its performance are discussed in Section 7.2 and left for future work.

The architecture of the SMMU is illustrated in Figure 4.9.

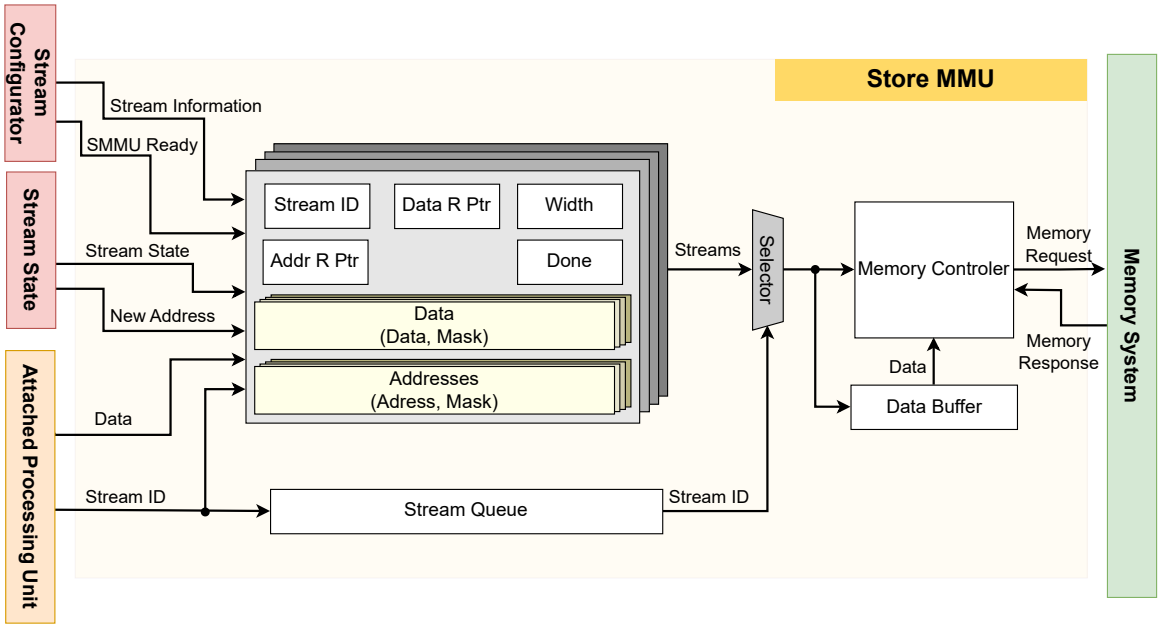

Figure 4.9: Overview of Store Management Unit

4.2.5.A Internal Architecture

The SMMU must store both the generated addresses and the data produced by the attached processing unit, making its dedicated register table a crucial component. Each entry in this table corresponds to a configured stream and consists of two buffering queues—one for addresses and another for data—along with stream information registers and pointers that control the read and write positions in both buffer queues. To enable selective masking of addresses and data, each entry in the buffer queues is equipped with a valid signal, allowing for predicate-based storing.

The SMMU also implements a FSM in order to control the store memory requests and all the necessary steps to communicate with the memory system trough the AXI4 protocol. A set of register for pointer tracking, and queue ordering are also utilised across the module.

4.2.5.B Execution Flow

Similarly to the LMMU the SMMU, starts by being configured according to the stream information. On this stage informations like the stream id and width are received a new entry on the register table is created, if space is available. During the processing of this stream addresses will be generated and

passed onto the SMMU, this addresses are then registered sequentially on the correct table entry. At the same the attached processing unit will be performing its computations and passing onto the SE the resulting values to be stored. With this values corresponding to a configured store stream, they are passed onto the SMMU where they are stored on the correct data buffer along side their address and masking vector.

The SMMU periodically verifies if a particular stream contains all the valid address and data pairs, issuing a store request if such is the case.

Although multiple streams can be configured and receiving data in the SMMU, only one can make a store request at a time, this is where the ordering queue enter in effect. The SMMU orders every configured stream in a queue selecting the first in it that contains a complete and valid data and address pairs. The transaction starts with the creation of the a buffer that corresponds to the same memory row, utilising for this the address tag information of a the address, after which the SMMU waits for a valid handshake with the memory system. The SMMU then transitions to the writing stage providing the information to the memory system, and finaly after reciving a completion signal form the memory system the SMMU FSM transitions back to the first stage where it waits for another valid set of data and addresses pair.

4.2.5.C Internal Architecture

The SMMU is responsible for storing both the generated addresses and the data produced by the attached processing unit, making its dedicated register table a critical component. Each entry in this table corresponds to a configured stream and includes two buffering queues—one for addresses and one for data—along with stream information registers and pointers that manage the read and write positions within both buffer queues. To support selective masking of addresses and data, each entry in the buffer queues is equipped with a valid signal, enabling predicate-based storage.

The SMMU also incorporates a FSM to manage the store memory requests and control the communication with the memory system via the AXI4 protocol. Additionally, a set of registers for pointer tracking and queue ordering is used by the module to ensure proper sequencing and data management.

4.2.5.D Execution Flow

Similar to the LMMU, the processing of the SMMU starts with the configuration of streams. At this stage, details such as the stream ID and width are received, and a new entry is created in the register table if space is available. During stream processing, addresses are generated and sequentially recorded in the appropriate table entry in the SMMU. Simultaneously, the attached processing unit performs computations and passes the resulting values and their masking to the SE, which transfers them to the SMMU for storage. These values, corresponding to a configured store stream, are stored in the correct data buffer alongside their associated addresses.

The SMMU periodically checks each stream to verify if it contains a complete set of valid addresses and data pairs. When such a condition is met, a store request is issued.

Although the SMMU can manage multiple streams simultaneously, only one store request can issued at any given time. This is where the ordering queue becomes essential. The SMMU arranges all configured streams in a queue, selecting the first one that contains a complete and valid pair of data and addresses. The transaction begins with the creation of a buffer that corresponds to a specific memory row, using the address tag information. The SMMU then waits for a valid handshake with the memory system. Once the handshake is confirmed, the SMMU proceeds to the writing stage, transferring the information to the memory system. Finally, after receiving a completion signal from the memory system, the SMMU's FSM transitions back to the initial stage, where it waits for the next valid set of data and address pairs.

4.3 Summary

This chapter details the architecture of the SE, organized into two main parts: Stream Manipulation and Data Handling. Each section addresses the communication interfaces, internal structures, and execution flow of all six modules. The design presented aligns with the UVE ISA specifications, ensuring full compliance with its requirements and functionalities.

The chapter begins with the Stream Manipulation, starting with the SC module, a dedicated configuration unit that bridges the attached processing unit with the rest of the SE system. The description of this module covers its communication channels and internal architecture, which includes a series of buffers for receiving stream configurations from the processing unit. The section then describes the execution flow, explaining that once a stream is fully configured, the SC module initiates its transfer to the SS module. The SS, the next module described, includes an overview of its communication channels and functionality, highlighting its central role in controlling the stream manipulation process. This chapter also presents the SS submodule and the developed data structures used to track stream iteration progress. Finally, it covers the SI module, detailing its architecture and role in address generation and dimension iteration within the system.

In the second section, the chapter describes the Data Handling part, covering both MMU modules. The LMMU, responsible for load memory operations, consists of three submodules, each with a detailed breakdown of internal architecture, and their buffers and queues registers. This section also explores execution flow and highlights data reuse features enabled by the buffering design. The chapter concludes with a description of the SMMU module, detailing its architecture and its function as the primary module for memory store operations.

5

Development and Evaluation Environment

Contents

To assess the performance of the consolidated SE implementation in System Verilog, a comprehensive testing environment was developed. This environment supports the entire evaluation workflow—from configuration creation via a Graphical User Interface (GUI) to an automated test execution script. The setup is similar to the one developed by Xavier Fernandes in his thesis work [20], but this new environment, fully implemented in System Verilog, enhances the original design by introducing a dynamic entry test-bench file which enables configuration through binary files. This new implementation eliminates the need to recompile the Design Under Test (DUT) for each diferent memory initialization. In addition to introducing the DUT of the test environment, this chapter covers the configuration parameters of the SE, key performance variables, the automation scripts developed for executing and generating test-benches as well as a brief description of several analysed test-benches.

5.1 Design Under Test

This DUT consists of the SE, a simple functional unit, and a memory system. The following sections provide a detailed description of each of these components. A visual overview of the DUT is presented in Figure 5.1.

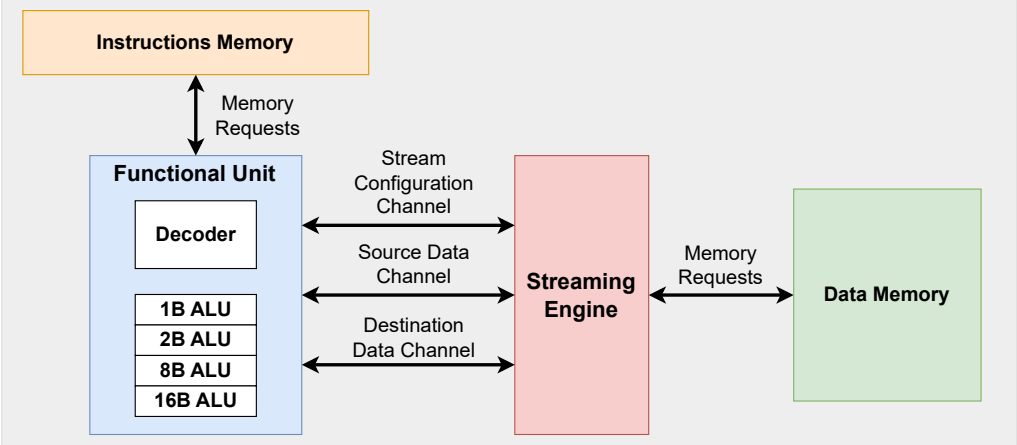

Figure 5.1: Design Under Test Architecture Overview

5.1.1 Streaming Engine

5.1.1.A Communication Channels

To facilitate the integration of the SE into other systems, it was designed with four primary communication channels: the configuration channel, the memory access channel, the source operands channel, and the destination operands channel.

- The **configuration channel** is used by the system to pass stream configuration values and control signals to the SE, allowing the configuration of stream patterns and their processing.
- The **memory access channel** handles all memory requests issued by the SE, managing the transfer of data between the memory system and the SE.

- The **source operands channel** and **destination operands channel** are responsible for transferring data between the SE and the attached processing unit. These channels connect directly to the Load MMU and Store MMU, respectively, allowing the system to fetch data based on the configured stream data pattern or store the resulting data back into the memory system after processing.

5.1.1.B Architecture Parameters

The proposed Streaming Engine offers a comprehensive set of architectural configuration parameters that can be adjusted to modify various aspects of its operation. This parametrization enables control over resource utilization, allowing for resource-efficient designs, compatibility with more complex stream pattern configurations, and improved data delivery efficiency. A complete list of configurable parameters is provided in Table 5.1

Table 5.1: Streaming Engine Parametrization Values

Parameter	Description
Stream Num Dim	Number of dimensions a stream can have
Stream Num Mods	Number of mods each dimension can have
Stream Offset Width	Width of a dimension offset field
Stream Stride Width	Width of a dimension stride field
Stream Size Width	Width of a dimension size field
LRQ Num Tables	Max number of scheduled address tags
LRQ Num Requests	Max number of addresses stored per single address tag
LLB Num Tables	Max number of loaded memory rows concurrently
LMMU Num Vecs	Number of address vectors per Load FIFO table
SMMU Num Addresses	Number of addresses per SMMU table
AXI R Data Width	Width of the AXI read data bus
AXI W Data Width	Width of the AXI write data bus
Max Num Load Streams	Max number of concorrently configured Load Streams
Max Num Store Streams	Max number of concorrently configured Store Streams

5.1.2 Functional Unit

The FU in the DUT is designed to be simple, with the sole purpose of decoding the configuration instructions from the UVE, passing configuration values to the SE, and managing the consumption of data values utilized by the SE. As a result, the FU includes a decoder unit, an ALU, a set of vector and scalar registers, and control signals.

The scalar registers are primarily used to store values referenced by the decoded instructions and must be pre-loaded at the beginning of the test-bench. These registers simplify access to stream parameters and hardware loop configurations, allowing the FU to obtain these values without communicating with the system's data memory unit.

The vector registers in the FU serve as buffers for holding values passed by the SE for use in the ALU. These vector registers are directly associated with the configured streams and are managed through a set of control flags.

Additionally, the FU contains extra registers, such as program counters, performance counters, and hardware loop-related signals, utilized for managing its internal operation.

5.1.2.A Decoder

To enable stream configuration and processing, the decoder unit in the FU must be able to decode different types of instructions:

T1 UVE Configuration Instruction Adhering to the UVE ISA, the decoder unit is able to interpret diferent configuration instructions, allowing it the extraction of values from dimensional and modifier descriptors. These instructions also directly encode the corresponding meta-data of a stream like: stream ID, width, and type.

T2 Operational Instructions To consume the data provided by the SE, the decoder interprets arithmetic instructions, guiding the FU on how to manipulate the data. These instructions define the operation to be executed and reference the two vector registers involved in the operation.

T3 Hardware Loop Instructions To reduce the number of instructions stored in the Instruction Memory, the system uses hardware loop instructions for repeatable operations. These instructions configure the FU to execute a loop for a specified number of iterations. In this scenario, these looped instructions operate the ALU.

Each instruction has a fixed length, leading to the fact that most instructions lack the actual values required for execution. For example, configuration instructions for descriptors do not contain the actual values for size, stride, offset, or displacement but instead reference scalar registers that hold these parameters. Similarly, hardware loop instructions do not specify the number of instructions to repeat or the number of iterations but reference the relevant scalar registers.

The decoder unit also provides additional signals to the FU depending on the decoded opcode. For example, stream configuration instructions require the decoder to distinguish between dimensional and modifier descriptors.

5.1.2.B Arithmetic Logic Units (ALUs)

Although result-accurate operations on loaded data are not strictly necessary for evaluating the performance of the SE, incorporating such functionality, allows for validation of the system and simplifies the debugging and optimization process by allowing both the loaded data and stored results to be analysed. For this reason, the FU includes a set of Arithmetic Logic Unit (ALU)s capable of performing basic arithmetic operations, such as addition and subtraction. Each of the four possible stream widths is assigned a dedicated ALU, and these are utilized accordingly.

During the execution of operations, the operands used by the ALU are drawn from the vector registers within the FU. This means that the data can either remain static, matching the values present in the vector register, or be sourced from a vector register associated with a load stream. Similarly, the results produced by the ALU are stored in a vector register that may be linked to a store stream.

When an arithmetic operation is issued by the FU, the availability of the required operands is verified. If the operands are not ready, the FU pipeline stalls until the operation can be completed.

5.1.3 Memories

To facilitate the creation of testbenches and validate the stream load and store operations, two memory structures were implemented: an instruction memory and a data memory.

5.1.3.A Instruction Memory (IM)

The instruction memory is a simple dual-port memory, with each word sized to match the instruction encoding. This memory is loaded at the beginning of the testbench with instructions encoded according to the decoding capabilities of the FU.

5.1.3.B Data Memory

The data memory module is more complex and designed to interface correctly with the AXI4 protocol. While it shares a dual-port architecture with the Instruction Memory (IM), it also includes the necessary logic to handle AXI4 requests. The memory control system allows for two concurrent operations—one load and one store—by utilizing two dedicated FSMs.

5.2 Testbench Automation

To evaluate the performance impact of the SE parameters and access patterns on data-fetching efficiency, a set of tools was developed to simplify the testing process. These tools include a dynamic test-bench, a stream configuration generator, and an automatic test evaluation script.

Since System Verilog was used to describe both the SE and the complete system, the dynamic test-bench was also developed using System Verilog. This test-bench includes the full DUT along with the necessary logic to initialize the instruction memory, data memory, and the scalar and vector registers of the FU. It also presents the resulting values from the executed operations, as well as performance counter values and metrics. The compilation and execution of this test-bench were performed using the Verilator [22] compiler and simulator.

Thanks to the flexible design of the test-bench, different sets of instructions and scalar values can be initialized via files provided as command line arguments. This functionality streamlines the testing of multiple stream patterns without requiring any modifications to the hardware description in the test-bench file.

5.2.1 Configuration Generator

As previously mentioned, initializing the DUT requires configuring both a set of instructions for the IM and scalar values for the FU. Since these instructions must be delivered in an encoded format (32-bit strings with appropriate scalar register indices), manually creating multiple stream configurations with complex patterns and processing behaviors can quickly become a cumbersome task.

To address this, a GUI was developed with a simple stream configuration system. This tool enables the configuration of up to seven streams, each with a maximum of six dimensions, and allows users to define the stream type (load/store) and stream width (1B/2B/4B/8B). The platform, shown in Figure 5.2, automates the conversion of scalar values into binary and assembles the corresponding encoded instructions for the stream configuration. The resulting binary values, along with a quick command to generate the necessary files on the test machine, is then provided by the interface.

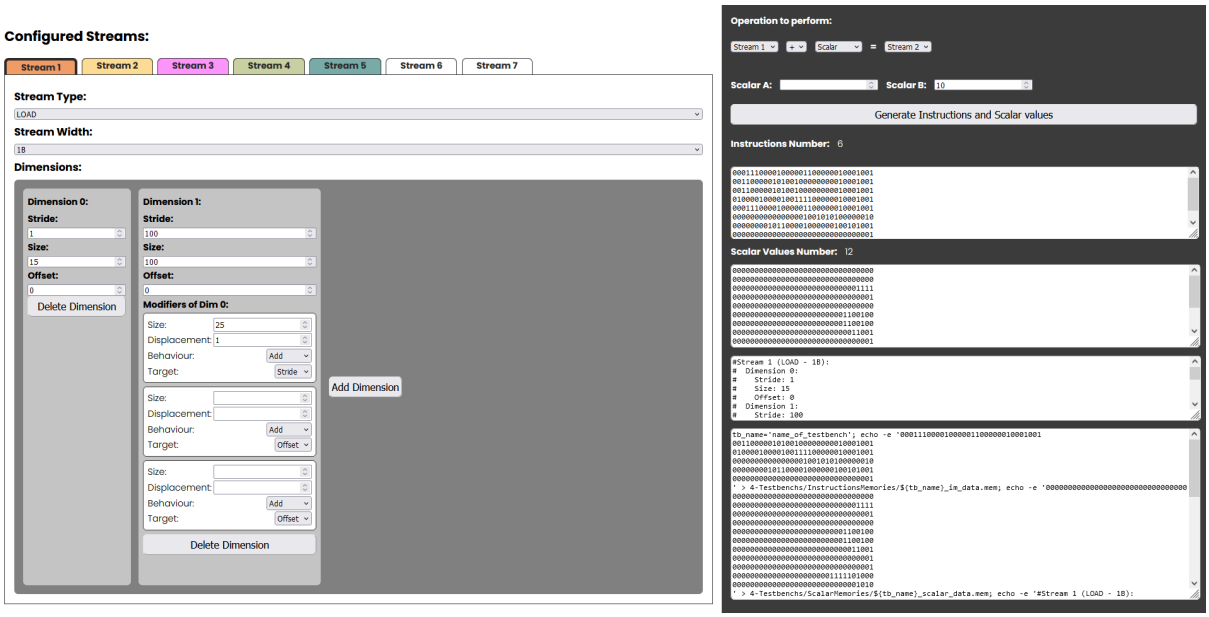

Figure 5.2: Web Interface of the instructions and scalar value generator

5.2.2 Regression Testing

With the development of tools that simplify the creation of tests, the process of evaluating and validating them became time-consuming. To streamline this task, an automatic script was created to handle compilation, execution, and validation.

The script requires a structured directory containing the necessary folders for evaluation steps, as well as configuration files that enable the script to locate the hardware description files of the DUT.

The script provides three main commands:

- The **setup** command guides the user in specifying the locations of the required files.
- The **testbench** command, which includes an optional argument for either a specific test or the keyword ”all,” triggers the execution of either a single test or all defined test-benches.
- The **clean** command removes old compilation files from the project folder.

Before each test-bench execution, the script compiles the DUT, however, if all test-benches are being executed, compilation only occurs before the first test-bench.

At the end of each test-bench run, the output is temporarily stored and compared with results from previous runs. If it’s the first execution of that test-bench, the result is automatically saved in the results

folder, and the user is prompted to manually review the output. In cases where it is not the first run, the results are compared, and if discrepancies are found, the script notifies the user and stores a differences file for further investigation.

5.3 Performance Variables

5.3.1 Counters

To conduct the intended performance evaluations of the developed SE, a set of performance signals were implemented in various modules of the SE. These signals allow for the quantification of several key metrics, including the number of operations executed by the engine, the number of memory access stalls, the number of address generation operations, and overall stream iteration metrics. Below is a table containing the list of available Performance Counters of the SE.

Table 5.2: Streaming Engine Performance Counters

Performance Counter	Description
Higher_Dimensions_Iterations	Number of cycles iterating higher dimensions and modifiers
Lower_Dimension_Iteration	Number of cycles outputing address
Loaf_FIFO_Stalls	Number of stalling cycles caused by the lack of register space for new addresses
Load_Request_Queue_Stalls	Number of stalling cycles caused by the lack of register space for new addresses
Load Line Buffer Stalls	Number of stalling cycles caused by the lack of register space for new requests
LMMU_Stalls	Total number of stalls of the LMMU
LMMU_Commits	Total number of load cycles in which the LMMU was available to receive addresses
SMMU_Stalls	Total number of stalls of the SMMU
SMMU_Commits	Total number of store cycles in which the SMMU was available to receive addresses
Load_Request_Operations	Total number of load operations
Store_Request_Operations	Total number of store operations

5.4 Summary

This chapter establishes a comprehensive environment to assess the performance and functionality of the consolidated SE implemented in SV. To facilitate testing, a DUT was created, composed of three key components: the SE, a FU, and a memory subsystem. A detailed breakdown of each component was provided, including their communication channels, configuration parameters, and architectural details.

The automated testing process was also outlined. A description of the implementation robust test-bench automation suite was made, complete with a dynamic test-bench, configuration generator, and a regression testing script. This suite streamlined configuration, initialization, and performance evaluation, allowing for consistent and repeatable test scenarios. Metrics such as memory stall cycles, address generation efficiency, and operation throughput were calculated from performance counters integrated within the SE. These metrics provided quantifiable measures of the SE performance under various workload conditions.

In summary, this chapter provided a detailed look at the environment, configurations, and testing methodologies developed to assess the SE efficiency and adaptability.

6

Experimental Evaluation

Contents

This chapter is dedicated to the analysis of the developed SE and its performance towards diferent types of memory access patterns, the analysis of the synthesis of the SE and CVA6 processor, and the case study of the integration of the SE in a Stream-based Dataflow Accelerator.

In order to analyse the performance of the SE, several key performance counters will be utilized, including *LOAD_REQUEST_QUEUE_STALLS* (LRQ Stalls), *LOAD_FIFO_STALLS* (LF Stalls). This counters, associated with the LMMU, will provide valuable insights into the behaviour of the SE architecture, revealing correlations between access patterns and the LMMU. The test-benches have been specifically designed to stress test individual aspects of the LMMU, ensuring a thorough performance evaluation.

Additionally, this thesis analysis will also focus on AGU efficiency. This evaluation will be made through the use of the *HIGHER_DIMENSIONS_ITERATIONS*, and the *LOWER_DIMENSION_ITERATION* performance counter in the creation of a quantifiable metric called **Address Generation Efficiency**. This test will leverage carefully configured stream patterns to analyse the relationship between stream complexity and its processing within the SE.

6.1 Metrics

As previously introduced, a metric calculation will be created through the use of multiple performance counters of the SE, allowing the analysis of other behaviours.

6.1.1 Address Generation Efficiency

An important metric to analyse in more complex stream access patterns is the efficiency of address generation. Due to the design choices of the AGU in the SE, only one dimension or modifier can be iterated at a time. This means that, during stream iteration, there are intervals where no memory addresses are generated, instead, the system iterates through higher dimensions and modifiers to progress the state of the stream.

To analyse this behaviour the following equation was used:

$$\text{Address Generation Efficiency} = \frac{\text{Number of Lower Dimension Iteration}}{\text{Total Number of Iterations}} \tag{6.1}$$

This equation divides the total amount of cycles spent iterating the lower dimension of stream which corresponds to the *Lower_Dimension_Iteration* counter, by the total number of valid iterations which corresponds to sum of the *Higher Dimensions Iterations* counter and the *Lower Dimension Iteration* counter.

This metric quantifies the efficiency of the system in generating address. A higher ratio indicates a efficient memory generating process, while a lower ration suggests a lower efficiency.

6.2 Testbench Definitions

This section defines the stream configurations used for the pattern analysis impact and the rationale behind them. All the developed test-benches are configured with Load Streams and Store Stream of width equal to 1B.

6.2.1 Linear Stream Patterns Testbench

This test-bench is designed to analyse the LMMU of the SE and establish correlations between the sequence of generated addresses and system stalls.

It consists of three stream configurations: two for load and one for store. The load streams pattern features two dimensions: a first dimension with a stride of 1, a size of 32, and an offset of zero, the second dimension has a stride of 32, a size of 32, and an offset of zero. The store stream pattern follows the same configuration as the load stream. The FU performs a simple addition between both load streams and stores the resulting values in the the vecotr associated with the store stream.

This test-bench enables an analysis of the LMMU's ability to handle multiple sequential addresses and provides a basis for comparison with less sequential stream patterns. This test will be executed utilizing diferent SE configurations, modifying in each the number of LRQ requests per table. Based on the design of the LMMU, a correlation between this pattern and the number of stalls of the LRQ unit is expected, as the number of acceptable addresses per address tag is limited by the number of requests the LRQ table can handle.

This test-bench will also be utilized in a second analysis where the SE configuration will be altered to diferent numbers of LRQ tables, while maintaining the number of request.

6.2.2 High-Stride Stream Patterns Testbench

Similar to the previous test-bench, this one further analyses the LMMU, but this time stresses the unit by providing sparse memory addresses rather than sequential ones.

It consists of three configured streams. The first load stream has an access pattern featuring two dimensions: a first dimension with a stride equal to the number of elements per memory row (32), a size of 32, and an offset of zero. The second load stream features a similar configuration as the first however its first dimension offset value is equal to 160 in order to force the non re-utilization of memory fetches. The FU performs a simple addition between the load streams and stores its results through a store stream. The store stream pattern follows a linear configuration with two dimensions very similar to the configuration of the load streams of the linear test-bench.

This test-bench complements the analysis of the LMMU by evaluating its ability to handle sparse memory addresses. The pattern will be analysed under the same configuration environments as the simple linear stream pattern. An opposite stall behaviour is expected compared to the linear pattern, as this pattern is highly dependent on the number of tables rather than the number of requests.

6.2.3 High Repetition Stream Pattern

This test-bench is designed to stress-test the loading of sequential memory addresses multiple times, loading a total of 512 data elements twice, resulting in 1024 elements being loaded.

The test-bench consists of three stream configurations. The load streams are configured with an access pattern featuring two dimensions: the first dimension has a stride of 1, a size of 32, and an offset of 0; the second dimension has a stride of 16, a size of 32, and an offset of 0. The store stream pattern follows a linear configuration with two dimensions very similar to the configuration of the load streams of the linear test-bench.

This test-bench will place significant pressure on the LMMU by generating multiple sequential and repeatable addresses. Since the test-bench has very linear load behaviour, the expected results should be similar to those obtained from the Linear Stream Pattern Testbench.

6.2.4 High-Dimensional and Modified Stream Patterns Testbench

To evaluate the efficiency of the AGU in the SE, this test-bench is designed to stress-test the iteration module. Address generation efficiency is quantified by comparing the number of iteration cycles that produce valid addresses, to the number of cycles spent iterating through higher dimensions and modifiers. To assess different scenarios, this test-bench will use a set of stream configurations with a varying number of descriptors.

The test is focused on analysing the effects of dimensions and modifiers on address generation, for this reason the relevant metrics will be the performance counters of lower and higher descriptor iterations, meaning that the type of stream is not critical, as it would only influence the MMU performance counter, which is not being evaluated in this test. For this reason the defined streams will be a single Load stream with an access pattern based on the following configuration:

- **Dimension 1** - Stride: 1, Size: 32, Offset: 0;
- **Dimension 2** - Stride 1, Size: 10, Offset: 0, Modifiers:
  - Stride Modifier - Displacement: 1, Size: 10, Behaviour: Add;
  - Size Modifier - Displacement: 1, Size: 10, Behaviour: Add;
  - Offset Modifier - Displacement: 1, Size: 10, Behaviour: Add;
- **Dimension 3** - Stride 1, Size: 10, Offset: 0, Modifiers: 0

The evaluated descriptor combinations are shown in Table 6.1. A dimension count of one refers to using only the first dimension from the base configuration; two dimensions correspond to using both the first and second dimensions; and three dimensions include all three defined dimensions of the base configuration. The varying number of modifiers is applied sequentially and always associated with the second dimension.

Table 6.1: Stream Pattern Combinations

Combination	Nb. Dims	Nb. Modifiers
A	1	0
B	2	0
C	2	1
D	2	2
E	2	3
F	3	0
G	3	1
H	3	2
I	3	3

6.3 Performance Results

6.3.1 Pattern Impact on LMMU Stalls

The first analysis explores the existing correlation between LMMU stalls caused by a linear stream pattern and the LMMU configuration parameter, specifically the number of requests per LRQ table entry. Following this, the correlation between stalls from a high-stride memory access pattern and the number of LRQ table entries was analysed, confirming the expected behaviour.

For these tests, a base configuration of the SE was set with the following values:

Table 6.2: List of Streaming Engine configuration parameters

Parameter	Value
Stream Num Dim	5
Stream Num Mods	3
Stream Offset Width	32
Stream Stride Width	32
Stream Size Width	32
LLB Num Tables	4
LMMU Num Vecs	4
SMMU Num Addresses	32
AXI R Data Width	64
AXI W Data Width	64
Max Num Load Streams	2
Max Num Store Streams	2

The LRQ parameters, specifically the number of tables and the number of requests per table, were adjusted according to the requirements of each analysis.

As shown in Figure 6.1 and Figure 6.2, system performance is significantly influenced by both the access pattern type and the SE configuration. For this first test, the SE was configured with four LRQ tables and a variable number of requests per table. This configuration limits the total number of distinct memory address tags to four, while altering the maximum number of groupable memory addresses per address tag.

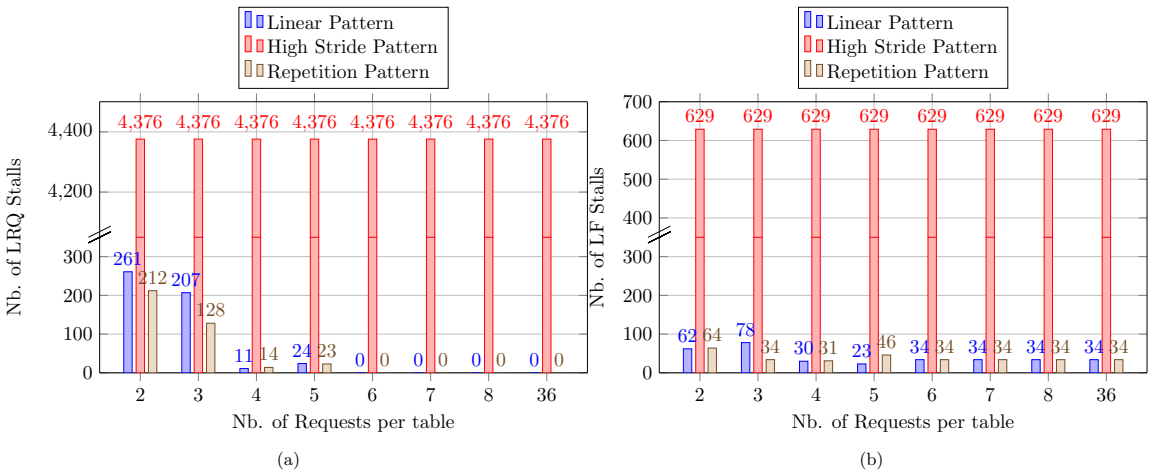

Figure 6.1: LRQ and LF stalls per number of LRQ Requests

The results confirm the previously anticipated behaviour. For the **Linear Pattern** in Figure 6.1(a), we observe that the number of stalls caused by the LRQ decreases as the number of requests per table increases. This occurs because the generated addresses are sequential, allowing them to be grouped under the same address tag. Notably, in Figure 6.1(b), the number of LF stalls fluctuates across tests until the number of requests is sufficiently high to prevent LRQ stalls, sifting at this point, the bottleneck, to the number of LF vectors, as indicated by the stabilization of the stall values.

The **Repetition Pattern** exhibits similar behaviour to the **Linear Pattern**, with a slight reduction in LRQ stalls. This improvement is due to the data re-utilization feature provided by the LMMU: since the generated addresses repeat every 16 values, the buffered data in the LLB unit can automatically resolve quicker the repeated outstanding requests.

The stall values for the **High Stride Pattern** exhibit the expected behaviour as well. Due to the rigid construction of this pattern, the generated addresses all correspond to distinct address tags, preventing it the benefit of the increased number of requests per LRQ table. Consequently, the number of both types of stalls remains constant across configurations.

Recognizing that, increasing the number of requests, does not improve system performance for High Stride Patterns, a second test was conducted, this time altering the number of LRQ tables while maintaining two requests per table. In theory, this configuration would allow the LRQ unit to store more ungrouped memory addresses and lower the number of LRQ stalls caused in the **High Stride Pattern**.

Figure 6.2 illustrates the results of this test. For the **Linear Pattern**, it's unsurprising that stall numbers decrease as the number of tables increases. Since the system is designed to allocate a new table whenever a new address tag is received or if the current tag lacks available request slots within the existing table, this test causes exhaustion of the two available requests per table leading to the assignment of the same address tag to another table, allowing it to still take advantage of data re-utilization mechanism.

For the **High Stride Pattern**, while LRQ stall values do seem to lower across configurations, the changes are not substantial until a higher number of tables. This behaviour arises because each address

must be assigned to a unique table either way, and the output of generated addresses still exceeds the number of available tables.

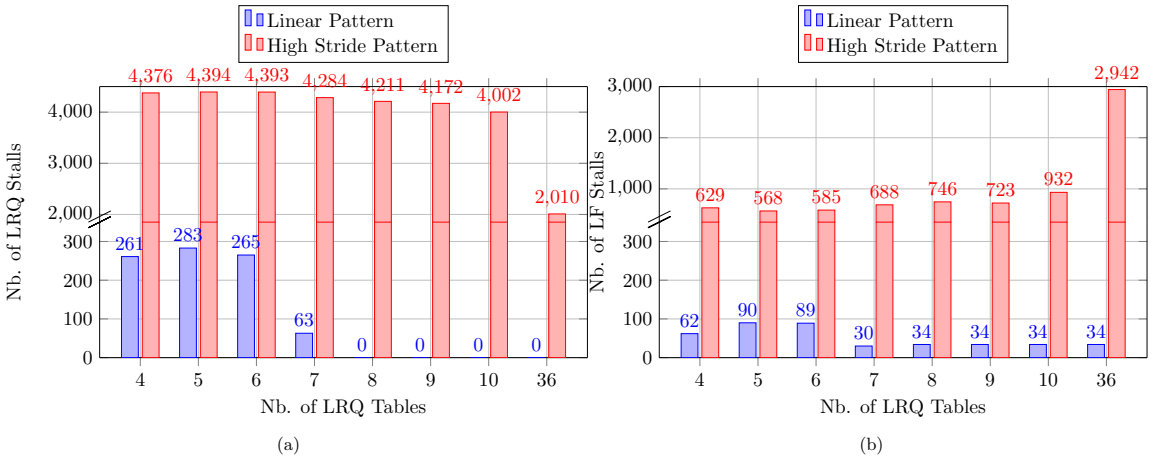

Figure 6.2: LRQ and LF stalls per number of LRQ Tables

A closer analysis of both plots in Figure 6.2 shows that the total number of stalls caused by the High Stride Pattern consistently approaches a value close to 4900 stalls across configurations. This behaviour can be explained by the fact that, although increasing the number of tables provides more storage, the bottleneck shifts to the number of tables that can be requested simultaneously. Additionally, since the addresses were designed to belong to different memory rows, no outstanding requests can reuse fetched data, causing the LF to wait for each individual request, triggering stalls as it attempts to resolve the currently allocated vectors.

6.3.2 Multi-Descriptor Access Pattern Impact on Address Generation Efficiency

Shifting from the LMMU stall analysis, this subsection analyses the impact of memory access patterns on the AGU, specifically the SI module of the SE. As described in Section 6.2.4, for this test-bench a range of stream configurations with varying numbers of dimensions and modifiers, where utilized.

The results of this analysis, presented in Table 6.3, report metrics related to address generation and the calculated Address Generation Efficiency.

Table 6.3: Address Generation Efficiency based on access pattern complexity

Pattern Type	1st Dim. Iteration	Other Iterations	Address Generation Efficiency (%)
A	32	0	100
B	320	28	91.95
C	320	37	89.64
D	365	46	88.81
E	365	64	85.08
F	3200	308	91.22
G	3200	398	88.94
H	3650	488	88.21
I	3650	668	84.53

Examining the results, we observe an expected behaviour: address generation efficiency decreases

as the number of dimensions and modifiers increases. This metric illustrates that, in high-dimensional and modified access patterns, the SE experiences reduced efficiency in address generation due to an increased number of iteration cycles in which no address is generated.

An interesting trend emerges when comparing the address generation efficiency among patterns C/D, D/E, and G/H, H/I. Although the difference between patterns C and D or D and E corresponds to adding a modifier, with the same number of iterations, the type of modifier itself influenced the generation efficiency.

This effect occurs because the modifier added in pattern D is a size modifier, which increases the size of the first dimension. This change results in more first-dimension iterations, thereby countering other descriptor iterations, mitigating their influence in address generation efficiency. Contrarily, the third modifier added in pattern E does not increase first-dimension iterations, so the rise of non-first-dimension iterations leads to a significant reduction in address generation efficiency. A similar behaviour occurs between patterns G, H, and I.

These results clearly demonstrate that not only the number of dimensions and modifiers in a pattern affect the address generation efficiency of the SE, but also the type of modifier. Since the Size Modifier can increase or decrease the size of the first dimension, then it directly impacts efficiency by altering the total number of first-dimension iterations.

This metric, however, does not account for the MMU's availability to accept new addresses which would affect the practical efficiency of the AGU. In situations where the address generation rate exceeds the MMU's address utilization rate, cycles that iterate higher dimensions or modifiers might not be able to be accepted by the MMU even if they produce a valid address. Essentially, higher-dimension or modifier iterations could be "hidden" behind MMU's stalls, taking advantage from the fact that if the first dimension were to be iterating then the SI would be in a stalled state and no new address would be generated.

6.4 Synthesis Results

For the synthesis analysis, a separate synthesis of the SE and the CVA6 processor were conducted using the Genus Synthesis tool from Cadence [23]. The synthesis was conducted using 28nm technology, providing both area and power consumption metrics.

For the CVA6 processor the configuration utilized for its synthesis was of the core hardware design with RVV extension, 64 bit AXI bus width and 32bit words. The synthesis of the complete SE was made with the following configuration:

Table 6.4: List of Streaming Engine configuration parameters

Parameter	Value
Stream Num Dim	5
Stream Num Mods	3
Stream Offset Width	32
Stream Stride Width	32
Stream Size Width	32
LRQ Num Tables	4
LRQ Num Requests	12
LLB Num Tables	4
LMMU Num Vecs	4
SMMU Num Addresses	32
AXI R Data Width	64
AXI W Data Width	64
Max Num Load Streams	2
Max Num Store Streams	1

After synthesizing both the SE and the CVA6 processor, the resulting data is presented in Table 6.5

Table 6.5: Hardware Resources Breakdown

	Component	Area (mm^2)	Power (mW)
Streaming Engine	SI	0.001	-
	SC	0.005	-
	SS	0.045	-
	LMMU	0.021	-
	SMMU	0.013	-
	TOTAL	0.133	105
CVA6 CPU	TOTAL	0.229	231

A detailed analysis of the area results of the SE reveals that the largest component is the Stream State module, which occupies approximately 34% of the total area. This substantial footprint is largely due to the Stream Tables submodule, which comprises complex register tables used to store stream information and their current states. These tables consume significant area, as they are designed to hold the maximum configurable number of streams, along with all possible dimensions and modifiers required for each stream.

The second and third largest components of the SE are the LMMU and SMMU, respectively, primarily due to their buffer and data queue implementations. Configuration parameters such as the number of LRQ tables, LRQ requests, and LF vectors significantly impact the area of these modules, as they determine the size of the required registers.

Comparing the area of the Streaming Engine to that of the CVA6 CPU reveals that implementing the current SE configuration within the processor architecture would result in an approximate 58% increase in the area of the CVA6 processor core.

6.5 Case Study: Streaming Engine Integration in an Stream-based Dataflow Accelerator

As an additional analysis and validation of the developed SE, a case study was conducted. This study focused on expanding a GPP stream vectorization model into a fully data-flow-driven computing module, leveraging spatial computation and a stream-based co-acceleration structures.

This case study introduced a 2D array of general-purpose Coarse Grained Reconfigurable Architecture (CGRA)-like Processing Elements (PE) powered by the previously presented UVE-compliant SE [11], deployed in a host RISC-V GPP within a full System-on-Chip (SoC) infrastructure, as per illustrated on Figure 6.3.

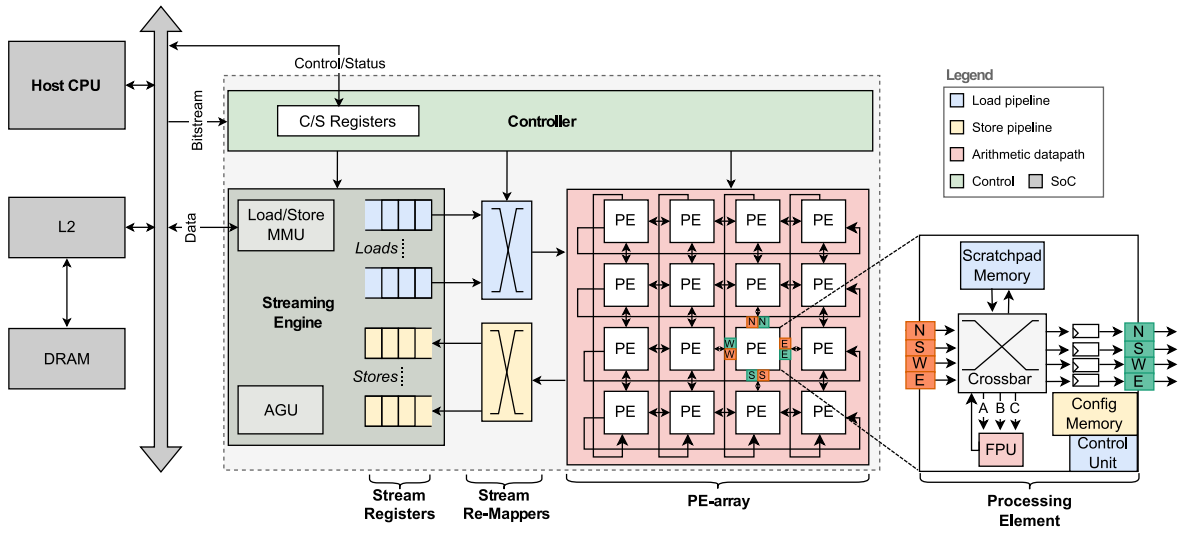

Figure 6.3: Overall SoC Hardware Architecture (with PE detail)

This case study contributes with: an analysis and validation of this the developed Stream-based Accelerator Architecture capable of spatially and temporally distributing general computational tasks. When integrated within a complete RISC-V SoC and supported by the UVE-compliant SE [11], the proposed accelerator implements a fully decoupled access-execute scheme. This approach allows memory operations to proceed in parallel with computation, reducing memory-access-induced overheads such as load-to-use latency and enhancing data throughput.

The validation of the case study accelerator was conducted through its integration on a Chipyard Rocketchip SoC [24, 25] composed by a Rocket Core RISC-V CPU, a 4-bank 8-way set-associative L2 cache (totalling 512 KiB), quad-channel DRAM, and a 256-bit system bus.

The accelerator's throughput was evaluated for a 4×4 PE-Array, with a SE configuration of 32-entry Load Request Queue and a 4-row Load Line Buffer.

6.5.0.A Performance

The presented SoC was evaluate through a set of benchmarks selected to evaluate the response of the accelerator to different memory access pattern and usage of the PE-Array. The RTL simulation where

executed using the Synopsys VCS 2022.0 simulator, and the results compared with a scalar Rocket Core RISC-V CPU and a vector Arm Cortex-A53 NEON CPU.

To achieve comparable performance metrics, benchmark results were collected in terms of clock cycle counts for each of the following CPU: a scalar Rocket Core RISC-V CPU, an Arm Cortex-A53 NEON CPU, and the modified Rocket Core RISC-V CPU. This approach enabled the normalization of performance differences that could exist due to differences in operating frequency. Accordingly, Figure 6.4 illustrates the performance gains of the accelerator relative to the other two CPUs.

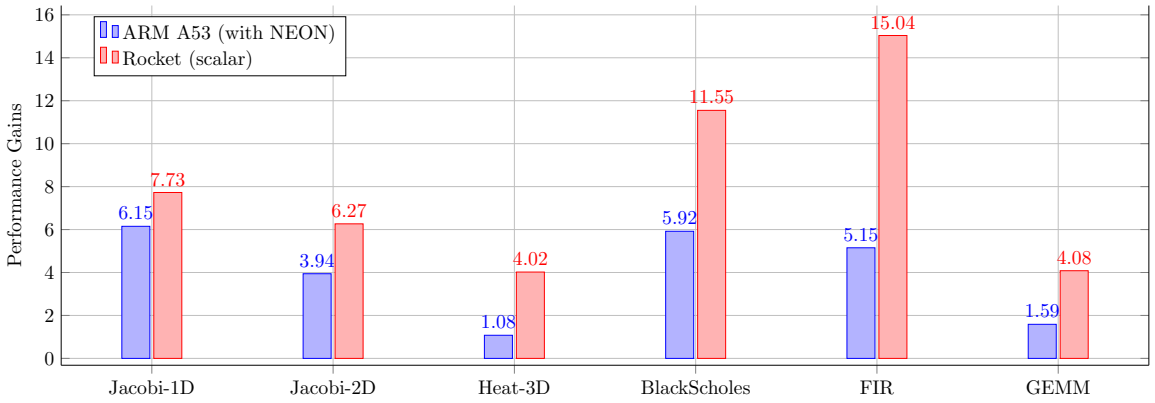

Figure 6.4: Clock cycle performance gains provided by the studied accelerator

The obtained results revealed a clear advantage of the developed accelerator, providing performance gains between 1.08x and 6.15x compared to the Arm A53 CPU (with NEON), and between 4.02x and 15.04x when compared to the scalar Rocket RISC-V CPU.

6.5.0.B Hardware Resources

For the hardware analysis of the accelerator, the synthesis were obtained through the Cadence Genus 21.15 tool, targeting a 7nm ASAP7 [26] technology process.

The base Rocket CPU was also synthesised, allowing a comparison of results between the area foot print of the accelerator and the area of the Rocket CPU. This synthesis also allowed the evaluation of the energy efficiency of the combined accelerator + Rocket Core *vs* the scalar Rocket CPU.

The following table presents the area occupied and the power consumption associated with the designed accelerator as well as the Rocket Core CPU.

Table 6.6: Hardware Resources Breakdown

	Component	Area (mm^2)	Power (mW)
Accelerator	PE-Array (4x4)	0.027	11.10
	SE	0.079	20.37
	ReMapper	0.004	1.81
	Controller	0.006	2.11
	Total	0.122	37.02
SoC	Rocket Tile	0.033	4.98
	L2 Cache (512 KiB)	0.232	6.26
Total		0.433	56.79

The resulting values indicate that the largest footprint in the design is attributed to the SE, primarily because this component implements buffer data structures that allow the mitigation of memory access latency and the handling of the out-of-order nature of memory responses. The second-largest area impact comes from the PE array, accounting for approximately 22% of the accelerator's total area.

The overall area of the accelerator can be seen as small when compared with complete SoC, since its area corresponds to just 28%. The area of this implementation boosts its merits when analysed in comparison with the total area of a simple scalar Rocket Core CPU. Although the accelerator occupies the equivalent of 39% of the scalar Rocket Core CPU's area, it incorporates a complete streaming engine and a PE array with 16 functional units, providing substantial performance improvements over the single ALU of the scalar Rocket Core CPU.

When analysing the power results of the developed accelerator, energy values were calculated for the executed benchmarks, both with and without accelerator integration. Figure 6.5 illustrates the energy savings achieved by using the accelerator, with energy efficiency improvements of up to 4.32x compared to the Rocket Core without the accelerator integration.

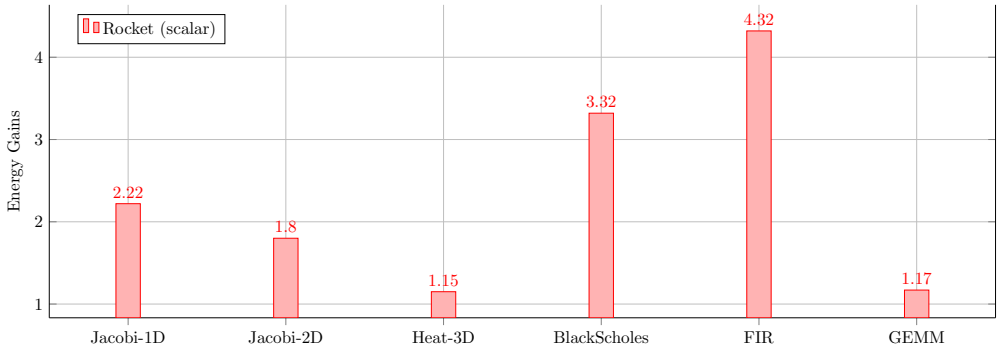

Figure 6.5: Energy improvement provided by the designed accelerator

These results indicate that integrating the accelerator into the SoC introduces minimal power overhead to the overall system resulting in an efficient combination of effective data streaming and an adaptable acceleration structure.

6.6 Result Analysis

The obtained results of the performed evaluation can be divided in three categories: the performance of the SE when confronted with diferent types of memory access patterns, the synthesis report of area and power, an the results of the utilization of the SE integrated on a Stream-based Dataflow Accelerator.

The obtained performance results highlight how the SE performance is influenced by access patterns and configuration parameters. In tests with a **Linear Pattern**, LRQ stalls decreased as requests per table increased, responding directly to the expected behaviour of grouping sequential addresses effectively under the same address tag. This showed the reduction of bottlenecks in the LRQ, with LF stalls becoming the limiting factor once LRQ stalls were minimized.

The **Repetition Pattern** showed similar stall behaviour to the Linear Pattern, though with slightly fewer LRQ stalls due to the LMMU's ability to quickly handle repeating addresses through the reuse of data. In contrast, the **High Stride Pattern** produced consistently high stall values across configurations because its non-sequential addresses limited grouping potential in the LRQ, resulting in no performance gain from increasing requests per table.

Further testing, this time with variable LRQ table counts, revealed a predictable decrease in stalls for the Linear Pattern, as more tables allowed higher address tag allocation, even with tag repetition. However, only a large number of tables showed any impact on the High Stride Pattern, indicating that increasing the tables alone does not substantially reduce stalls if the stride is high enough to cause a unique address tag per address generated.

Still on the performance results, the **Address Generation Efficiency** analysis revealed that as the number of dimensions and modifiers increases, address generation efficiency declines. This trend indicates that higher-dimensional and modified patterns require additional iteration cycles where no address is generated, reducing overall efficiency. Notably, comparisons among patterns (e.g., C/D, D/E, G/H, and H/I) highlight that the specific type of modifier also influences efficiency. For instance, a size modifier that expands the first dimension counteracts other iterations, slightly improving efficiency.

Analysis of the synthesis results for the SE shows that the Stream State module occupies the largest area, accounting for approximately 34% of the total footprint. This is mainly due to the Stream Tables submodule, which contains complex register tables used to store the state of each stream.

The next largest components are the LMMU and SMMU, whose area requirements are driven by the buffers and data queues within these modules. Key configuration parameters—including the number of LRQ tables, LRQ requests, LF vectors, LLB vectors, and total address capacity of the SMMU—directly influence register sizes and thus the area of these components. Integrating the current SE configuration into the CVA6 processor would increase the processor core's area by approximately 58%.

Lastly the analysis of the **case study** revealed very positive results, both in terms of performance, as well as hardware resource utilization. **Performance gains** were observed across the set of executed test-benches reaching gains of up to **6.15x** when compared with the Arm A53 NEON and up to **15.04x** when against the scalar Rocket RISC-V CPU. The implementation of the accelerator onto the choose SoC revealed substantial **energy efficiency gains** against the scalar Rocket RISC-V at the **expense** of an increase of **39%** to the original **area**. This performance and energy gains contribute to the validation of implementation of an UVE compliant SE on GPP.

7

Conclusion

Contents

7.1 Summary of Thesis Achievements

The primary focus of this thesis was the re-implementation of the SE based on the UVE ISA and the consolidation of its hardware architecture through extensive validation and testing.

The developed SE operates with specialized modules that handle distinct stages of stream manipulation, efficiently delivering data to the attached processing unit. Designed according to the UVE ISA, the SE interprets complex memory access patterns using configurable dimension and modifier descriptors. This capability allows the SE to generate memory addresses for a variety of multi-descriptor access patterns, ranging from simple single-dimensional patterns to complex, multi-dimensional configurations. To capitalize on known stream patterns, generated addresses are directed to a dedicated MMU of the SE, composed by data buffers queues, enabling data pre-fetching and reuse, effectively reducing data operation latency.

To validate the SE, a DUT was created, simulating a simplified processing core with a FU and separate memory units for data and instructions, which, in turn, managed the SE. A series of test benches were developed to evaluate the system's response to various stream patterns, focusing primarily on LMMU stalls and Address Generation Efficiency. This testing demonstrated that linear patterns adapt well to the MMUs regardless of the SE configuration, while high-stride patterns are more sensitive to system configurations. Additionally, patterns with a greater number of descriptors resulted in lower address generation efficiency.

A case study further implemented the SE in an Stream-based Dataflow Accelerator. This study showed promising results in both performance and energy efficiency, with performance gains reaching up to 15x compared to the baseline scalar Rocket RISC-V CPU and up to 6x over the ARM Cortex-A53 CPU (with NEON). In terms of energy, the accelerator achieved up to 5x improvements in efficiency compared to the base scalar Rocket RISC-V CPU, with these gains directly linked to the efficient data streaming facilitated by the SE.

7.2 Future Work

While the validation process in this work provides a detailed analysis of memory access patterns and their influence on system performance, the current SE design does not fully support all stream patterns outlined by the UVE ISA. Although multi-dimensional irregular access patterns can be configured using dimensional and modifier descriptors, indirect memory access patterns are not yet supported. Extending the SE to accommodate these indirect patterns and validating this capability represent key areas for future work. Additionally, the SMMU in the SE currently lacks memory access improvements similar to the ones present on the LMMU due to its simpler design. Future improvements to the SMMU should explore the redesign of its buffering structures in order to leverage burst write capabilities inherited from the AXI4 protocol.

The hardware analysis conducted on the SE and CVA6 processor has also indicated potential for further integration work. Completing the integration of the SE into the CVA6 will allow for a thorough

evaluation of performance gains and energy efficiency relative to the base CVA6 system and comparable architectures. These results would further validate the SE design and potentially uncover additional optimization opportunities. This integration will require modifications to the existing ISA of the CVA6 to support UVE instructions, adjustments to the renaming and commit stages in the CVA6 pipeline to handle vector registers and streams, and the development of an interface to facilitate communication between the SE and CVA6 systems.

Bibliography

[1] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *Proceedings of the 37th annual international symposium on Computer architecture*, 2010, pp. 37–47.

[2] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *ACM SIGARCH computer architecture news*, vol. 23, no. 1, pp. 20–24, 1995.

[3] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, "Efficiently prefetching complex address patterns," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 141–152.

[4] S. Mostofi, H. Falahati, N. Mahani, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Snake: A variable-length chain-based prefetching for GPUs," in *56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 728–741.

[5] I. Hadade, T. M. Jones, F. Wang, and L. d. Mare, "Software prefetching for unstructured mesh applications," *ACM Transactions on Parallel Computing (TOPC)*, vol. 7, no. 1, pp. 1–23, 2020.

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

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

[8] 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.

[9] 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)*, 2021, pp. 1787–1792.

[10] F. Schuiki, F. Zaruba, T. Hoefler, and L. Benini, "Stream Semantic Registers: A Lightweight RISC-V ISA Extension Achieving Full Compute Utilization in Single-Issue Cores," *IEEE Transactions on Computers*, vol. 70, no. 2, pp. 212–227, 2021.

[11] 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.

[12] "Intel advance vetor extension (avx)," https://www.intel.com/content/www/us/en/developer/tools/isa-extensions/overview.html, 2024, [Online; accessed 10-January-2024].

[13] A. Ltd., "Neon – arm®." [Online]. Available: https://www.arm.com/technologies/neon

[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] "Risc-v vector extension," https://github.com/riscv/riscv-v-spec/blob/master/v-spec.adoc, 2024, [Online; accessed 10-January-2024].

[16] T. N. Theis and H.-S. P. Wong, "The end of moore's law: A new beginning for information technology," *Computing in Science & Engineering*, vol. 19, no. 2, pp. 41–50, Mar 2017.

[17] M. Bohr, "A 30 year retrospective on dennard's mosfet scaling paper," *IEEE Solid-State Circuits Society Newsletter*, vol. 12, no. 1, pp. 11–13, Winter 2007.

[18] F. Schuiki, M. Schaffner, F. K. Gürkaynak, 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, 2019.

[19] "Cva6: An application class risc-v cpu core," https://cva6.readthedocs.io/en/latest/, 2024.

[20] X. Fernandes, "Data-streaming engine architecture for the unlimited vector extension," 2024, [Thesis Report - 2023].

[21] "AMBA AXI4 Interface Protocol — xilinx.com," https://www.xilinx.com/products/intellectual-property/axi.html, [Accessed 26-10-2024].

[22] W. Snyder, "Verilator simulator," https://www.veripool.org/verilator/, 2024, [Online; accessed 10-January-2024].

[23] "Genus Synthesis Solution — cadence.com," https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/genus-synthesis-solution.html, [Accessed 27-10-2024].

[24] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton *et al.*, "Chipyard: Integrated design, simulation, and implementation framework for custom socs," *IEEE Micro*, vol. 40, no. 4, pp. 10–21, 2020.

[25] Berkeley Architecture Research, *Chipyard Documentation, Release 1.8.1*, Oct. 2022. [Online]. Available: https://chipyard.readthedocs.io/_/downloads/en/1.8.1/pdf/

[26] L. T. Clark, V. Vashishtha, L. Shifren, A. Gujja, S. Sinha, B. Cline, C. Ramamurthy, and G. Yeric, "Asap7: A 7-nm finfet predictive process design kit," *Microelectronics Journal*, vol. 53, pp. 105–115, 2016.