FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Support for Streaming SIMD Paradigms Via MLIR

Manuel de Magalhães Carvalho Cerqueira da Silva

Mestrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Dr. Nuno Miguel Cardanha Paulino

Second Supervisor: Dr. João Carlos Viegas Martins Bispo

October 23, 2024

Resumo

Este trabalho aborda os desafios contemporâneos da computação, causados pelo abrandamento da Lei de Moore e do escalonamento de Dennard. A mudança para arquiteturas heterogéneas requer estratégias de compilação inovadoras, levando a iniciativas como o projeto Multi-Level Intermediate Representation (MLIR), onde baixar progressivamente o nível de abstração do código pode ser alcançada através do uso de *dialetos*. O nosso trabalho centra-se no desenvolvimento de um dialeto MLIR capaz de representar acessos de dados em fluxo contínuo à memória, e operações vectoriais Single Instruction Multiple Data (SIMD). Também propomos a nossa própria Structured Representation Language (SRL), uma Domain Specific Language (DSL) para servir como precursor da camada MLIR e subsequente interoperação entre dialetos novos e existentes. O SRL expõe os conceitos computacionais de "streaming" e computação vetorial a um nível superior e serve como passo intermédio para suportar a geração de código contendo o nosso dialeto proposto a partir de código de entrada arbitrário, que deixamos como trabalho futuro. Este trabalho apresenta as sintaxes do SRL DSL e do dialeto, e ilustra como pretendemos empregá-las para abranger tanto General-Purpose Processors (GPPs) com co-processadores SIMD como opções de "hardware" customizado, como Field-Programmable Gate Arrays (FPGAs) e Coarse-Grained Re-configurable Arrays (CGRAs) e também mostra resultados quantitativos que demonstram a eficácia das estratégias de compilação propostas, apresentando melhorias significativas de desempenho em vários benchmarks.

Abstract

This work addresses the contemporary challenges in computing, caused by the stagnation of Moore's Law and Dennard scaling. The shift towards heterogeneous architectures necessitates innovative compilation strategies, prompting initiatives like the Multi-Level Intermediate Representation (MLIR) project, where progressive code lowering can be achieved through the use of *dialects*. Our work focuses on developing an MLIR dialect capable of representing streaming data accesses to memory, and Single Instruction Multiple Data (SIMD) vector operations. We also propose our own Structured Representation Language (SRL), a Domain Specific Language (DSL) to serve as a precursor into the MLIR layer and subsequent inter-operation between new and existing dialects. The SRL exposes the streaming and vector computational concepts to a higher-level, and serves as intermediate step to supporting code generation containing our proposed dialect from arbitrary input code, which we leave as future work. This work presents the syntaxes of the SRL DSL and of the dialect, and illustrates how we aim to employ them to target both General-Purpose Processors (GPPs) with SIMD co-processors and custom hardware options such as Field-Programmable Gate Arrays (FPGAs) and Coarse-Grained Re-configurable Arrays (CGRAs) and also shows quantitative results demonstrate the efficacy of the proposed compilation strategies, showcasing significant performance improvements across various benchmarks.

Sustainable Development Goals

Introduction

The integration of Sustainable Development Goals (SDG) into engineering dissertations is key to contributing to global sustainability challenges. As articulated by the United Nations, the SDG are a universal call to action to end poverty, protect the planet, and ensure prosperity for all [1]. Each goal has specific targets to be achieved by 2030, requiring collaboration across various sectors, including engineering.

In this chapter, we explain the contributions of this dissertation in advancing the SDGs, with a focus on illustrate how SDGs can be integrated into this research, using the guidelines provided by the United Nations, that are illustrated in Figure 1.

Figure 1: List of Sustainable Development Goals (SDG) [1].

Relevant SDGs

In the context of the current research, the development of an Multi-Level Intermediate Representation (MLIR) dialect capable of representing streaming data accesses to memory and Single Instruction Multiple Data (SIMD) vector operations can be linked to several SDG illustrated on Table 1.

Table 1: Contributions to Sustainable Development Goals

SDG	**Goal**	**Contribution**	**Performance Indicators and Statistics**
9	Industry, Innovation, and Infrastructure	Advancing computational efficiency and promoting innovative hardware solutions	Processing speed.
12	Responsible Consumption and Production	Optimising resource usage in data processing	Computer energy usage.

Agradecimentos

Aos meus orientadores, o Prof. Doutor Nuno Paulino e o Prof. Doutor João Bispo, cuja orientação revelou-se um farol incansável neste último ano.

Aos meus colegas de laboratório, Pedro Ramalho e ao Luis Sousa, assim como aos parceiros de projeto, Ana Fernandes e Luís Crespo. O qual os seus esforços e ideias deram vida às páginas que seguem.

Aos meus colegas, em especial, aos amigos do IEEE com a sua camaradagem e apoio, que partilharam as agruras e as epifanias deste caminho tortuoso.

Ao INESC-TEC, que apoiou este projeto através da atribuição das bolsas de investigação B-0124-22 2022.06780.PTDC (DOI: 10.54499/2022.06780.PTDC), financiadas pela Fundação para a Ciência e Tecnologia (FCT).

A todos, o meu agradecimento profundo.

Manuel Cerqueira da Silva

This work was funded by Fundação para a Ciência e a Tecnologia (FCT), under project 2022.06780.PTDC (DOI: 10.54499/2022.06780.PTDC).

"All truly great thoughts are conceived while walking."

Friedrich Wilhelm Nietzsche

Contents

List of Figures

List of Tables

Abbreviations

3MM 3 Matrix Multiplication

ANTLR ANother Tool for Language Recognition

ARM Advanced RISC Machines

ASIC Application-Specific Integrated Circuit

AST Abstract Syntax Tree

CFG Controlflow Graph

CGRA Coarse-Grained Re-configurable Array

CIRCT Circuit IR Compilers and Tools

CPU Central Processing Unit

DSE Domain Space Exploration

DFG Dataflow Graph

DSL Domain Specific Language

FMOPA Floating-Point Outer Product

FPGA Field-Programmable Gate Array

GCC GNU Compiler Collection

GEMM General Matrix Multiplication

GPP General-Purpose Processor

HDL Hardware Description Language

HLS High-Level-Synthesis

IR Intermediate Representation

ISA Instruction Set Architecture

ISS Instruction Set Simulator

JAR Java ARchive

JSON JavaScript Object Notation

LLVM Low Level Virtual Machine

MLIR Multi-Level Intermediate Representation

OOP Object Oriented Programming

PCIe Peripheral Component Interconnect Express

RISC-V Reduced Instruction Set Computer Five

RVV RISC-V Vector Extension

SAXPY Single Precision A X Plus Y

SDG Sustainable Development Goals

SIMD Single Instruction Multiple Data

SME Scalable Matrix Extension

SRL Structured Representation Language

SSA Single Static Assignment

SVE Scalable Vector Extension

TRISOLV Triangular Solve

UC Use Case

UVE Unlimited Vector Extension

Chapter 1

Introduction

1.1 Context

In the past decade, there has been a noteworthy transformation in the field of computing, marked by the stagnation of Moore's Law and Dennard scaling. The spotlight has shifted towards heterogeneous architectures, which encompass diverse processing units and have the potential to leverage hardware specialisation [13]. However, with Moore's Law no longer driving consistent performance improvements, compiling applications for heterogeneous systems or application-specific computing engines has become more challenging. Traditional high-level source code struggles to efficiently harness the diverse capabilities of such specialized hardware. This makes it difficult to compile applications in a way that fully utilizes the specialized nature of different processing units while still maintaining programmability and portability.

In response, initiatives such as the Multi-Level Intermediate Representation (MLIR) project have emerged [14]. Nested within the Low Level Virtual Machine (LLVM) ecosystem, MLIR introduces a unified Intermediate Representation (IR), utilising the concept of dialects. These dialects can represent different functionalities or paradigms of data manipulation. The intention is to facilitate code generation for targets (i.e., hardware platforms or processors) containing heterogeneous components.

This work will focus on developing an MLIR dialect capable of representing data streaming and vector computations and will test its use by lowering it to platforms with such capabilities.

1.2 Motivation

The increasing use of specialized computing architectures aims to boost performance. However, deploying code across these heterogeneous systems is challenging, requiring different programming models, or specialised languages (e.g., CUDA). Traditional compilation approaches, designed for general-purpose processors, struggle to support this diversity, with each architecture demanding distinct compilation strategies or custom toolchains, making the process complex and time-consuming.

While the use of dedicated computing architectures, such as co-processors and accelerators, is motivated by the need to improve computing performance, the investment in the compilation is motivated by allowing for straightforward deployment of code onto these types of architectures.

In other words, the all-encompassing motivation of investing in compilation flows (e.g., through the MLIR ecosystem) is to streamline the compilation process for application developers, allowing them to target diverse back-ends without requiring compiler/assembler modification efforts on their part, and/or use of non-conventional programming languages.

1.3 Objectives

Given the motivation above, the primary objective of this work is to leverage MLIR as a middle ground to generate code for heterogeneous architectures. Specifically, the focus is on designing a dialect capable of representing data stream accesses and operations on vector data types (i.e., Single Instruction Multiple Data (SIMD)-like). In essence, this work can be summed up in two core research questions:

1. How can an MLIR dialect capable of expressing streaming operations and/or vector operations be defined?
2. In what ways can the MLIR dialect serve as a versatile bridge, allowing for effective code, targeting different SIMD extensions?

To validate the new dialect abstractions, the compilation targets will include an existing instruction set extension named the Unlimited Vector Extension (UVE), which extends a baseline Reduced Instruction Set Computer Five (RISC-V) architecture [15]. This extension is dedicated to streaming and vector operations. Moreover, the generated *UVE* will be compared with RISC-V code from other compilers. The validation process will also include a proposal for generating code to support other streaming and vector engines, such as RISC-V's own V extension for vector operations, RISC-V Vector Extension (RVV).

1.4 Contributions

This section outlines the key contributions of our work:

1. Development of an MLIR dialect and Domain Specific Language (DSL) to vector and data streaming operations.
2. Analysis of MLIR tooling for hardware generation.
3. Propose a how-to target SIMD instruction using either custom instruction sets or custom-generated hardware.
4. Evaluation of the performance of the generated codes with *UVE* when compared with code only containing regular RISC-V instructions.

5. Two published papers included in Appendix A.

1.5 Document Structure

Beyond this introduction, this document is organised into five more chapters and two appendices. In Chapter 2, the background knowledge of this work and relevant related studies are delineated. This includes a review of the background, as well as related work in domain-specific languages and compiler support for streaming computation. Chapter 3 details the use of MLIR flow for compiling into specific hardware targets and presents the preliminary work. This chapter also focuses on supporting streaming SIMD paradigms via MLIR. Chapter 4 highlights the compiling efficiency in vectorising diverse kernel and compares generated UVE assembly performance with standard RISC-V. Lastly, Chapter 5 provides concluding remarks and possible future research directions. Appendix A includes a list of related publications. Appendix B contains the generated code from the compiler transformations discussed in the document. Appendix C provides the *Structured Representation Language (SRL)* specification, including *SRL* MLIR operations and *SRL* DSL methods.

Chapter 2

State Of The Art

2.1 Background

2.1.1 Multi-Level Intermediate Representation

Multi-Level Intermediate Representation (MLIR) has been created as a compiler infrastructure crafted for reusability and extensibility. Its goal is to mitigate software fragmentation, elevate the efficiency of heterogeneous hardware compilation, significantly reduce the costs of constructing domain-specific compilers, and streamline the integration of pre-existing compiler systems. This infrastructure, initially intended for use in compilation flows of machine learning tasks, (e.g. TensorFlow or PyTorch) is currently, acting as the foundation for novel compilers.

MLIR aims to facilitate extensibility through dialects, serving as a mechanism to logically group operations, attributes, and types within a distinct namespace. Dialects themselves do not introduce different semantics but function as an organisational tool, allowing for dialect genericity. The categorisation of operations, types, and attributes into dialects is a conceptual separation, similar to structuring a programming language library. As an illustration, one dialect may encompass operations and types designed for operations on hardware vectors (e.g., shuffle, insert/extract element, mask). In contrast, another dialect may focus on operations and types for operations on algebraic vectors (e.g., absolute value, dot product, etc.). Decisions on sharing a vector type between dialects and determining where to define it are intentional choices that the developer must make when using this framework.

Consider the MLIR code snippet shown in Figure 2.1 as an example. This code illustrates a simple MLIR `@return_smallest_number` function that performs a conditional branch based on the comparison of two arguments. The code is highlighted to demonstrate the utilisation of various MLIR dialects, namely arithmetic (*arith*) and control flow (*cf*) followed by a period (.) and the operation name.

While it is technically possible to consolidate all operations, types, and attributes within a single dialect, such an approach could become unmanageable due to the simultaneous presence of various concepts and the potential for name conflicts, among other challenges. Despite each operation, type, and attribute being allocated to a specific dialect, MLIR deliberately supports

```
func @return_smallest_number(%arg0 : i32, %arg1 : i32) -> i32 {
    %0 = arith.cmpi %arg0, %arg1 : i32
    cf.cond_br %0, ^bb1, ^bb2
    ^bb1:
        return %arg0 : i32
    ^bb2:
        return %arg1 : i32
}
```

Figure 2.1: MLIR code example.

a diversity of dialects, enabling a step-by-step representation lowering. Figure 2.2 illustrates an example flow of MLIR, exemplifying the framework's strategy for gradual lowering by employing multiple dialects.

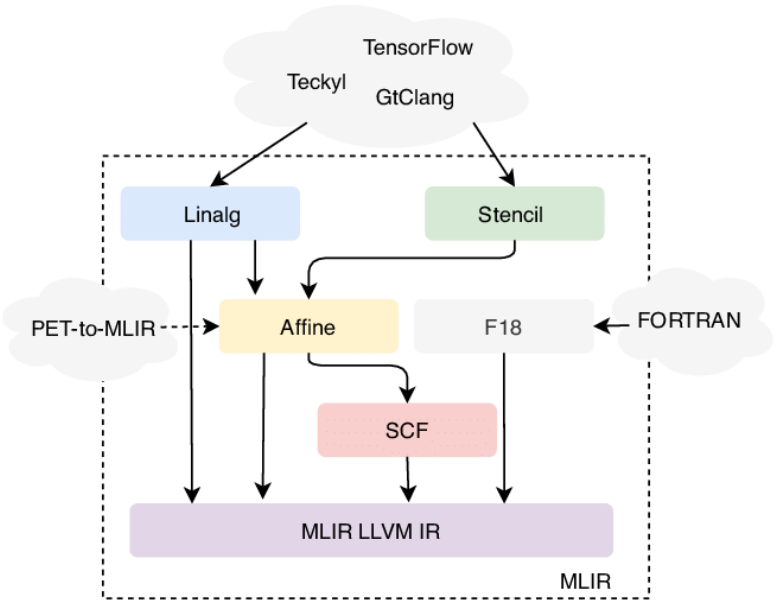

Figure 2.2: MLIR lowering flow example [2].

Operations originated from distinct dialects can coexist seamlessly at any level of the Intermediate Representation (IR) simultaneously, and they can leverage data types defined in different dialects. The integration of dialects has the purpose of enhanced reuse, extensibility, and flexibility, intending to eliminate the need for developers to employ non-composable workarounds [14].

2.1.2 Spike

Spike is an Instruction Set Simulator (ISS) recognised as the official reference simulator for the Reduced Instruction Set Computer Five (RISC-V) instruction set architecture Instruction Set Architecture (ISA) and is widely used to test various RISC-V extensions. It offers a straightforward and efficient simulation environment, speeding up the development process and ensuring the specification is accurate [16].

2.1.3 ANother Tool for Language Recognition

ANother Tool for Language Recognition (ANTLR) is a parser generator designed to read, process, and translate structured text or binary files. With this tool, the developer can accommodate different grammars defined by developers. This capability allows ANTLR to decouple grammars from application code, thereby enhancing the reusability of grammars across various applications. Figure 2.3 demonstrates a simple ANTLR grammar for parsing arithmetic expressions.

```
1
2    expr    : expr ('*'|'/') expr    # MulDiv
3            | expr ('+'|'-') expr    # AddSub
4            | INT                    # Int
5            ;
6
7    INT     : [0-9]+ ;
```

Figure 2.3: ANTLR grammar snippet for parsing arithmetic expressions.

In this example, the `expr` rule specifies an arithmetic expression that can be a multiplication or division (`MulDiv`), an addition or subtraction (`AddSub`), or an integer (`Int`). The `INT` rule is used to match numeric values. In this work, ANTLR was exploited to easily generate parsers for our defined Domain Specific Language (DSL) and MLIR dialect.

2.1.4 The LARA Framework and AnyWeaver

The LARA Framework [17], written in *Java*, serves as a robust platform for developing source-to-source compilers, also known as transpilers. Within LARA, compilation passes are implemented in *JavaScript*, functioning as libraries the compiler interprets. This modular design promotes flexibility and extensibility, allowing developers to compose various transformation passes seamlessly.

Typically, each instance of the LARA compiler is dedicated to supporting a single programming language. Creating a new compiler in LARA involves defining an Abstract Syntax Tree (AST) for the language and specifying points in the code that can be selected and modified. Updates or modifications to these components require revising the compiler, a process that can be resource-intensive and time-consuming.

To solve these issues, in contrast to previous LARA based compilers that focus on specific languages, AnyWeaver introduces a more adaptable approach to source-to-source compilation [18]. Rather than relying on predefined language-specific ASTs, AnyWeaver dynamically loads AST like structures and processes them without prior knowledge of their internal configuration. This capability enables AnyWeaver to operate effectively across diverse programming languages and evolving code bases, eliminating the need to create distinct compiler versions for each language.

AnyWeaver achieves this flexibility through its modular architecture, leveraging *JavaScript* or Java ARchive (JAR) files that can be included in its libraries. Developers can integrate and process

these components as needed, simplifying the development of code analysis and transformation tools. The Figure 2.4 illustrates AnyWeaver architecture when used with an ANTLR-based parser.

Although AnyWeaver is a recent development within the context of this work, it is not a direct contribution to this dissertation.

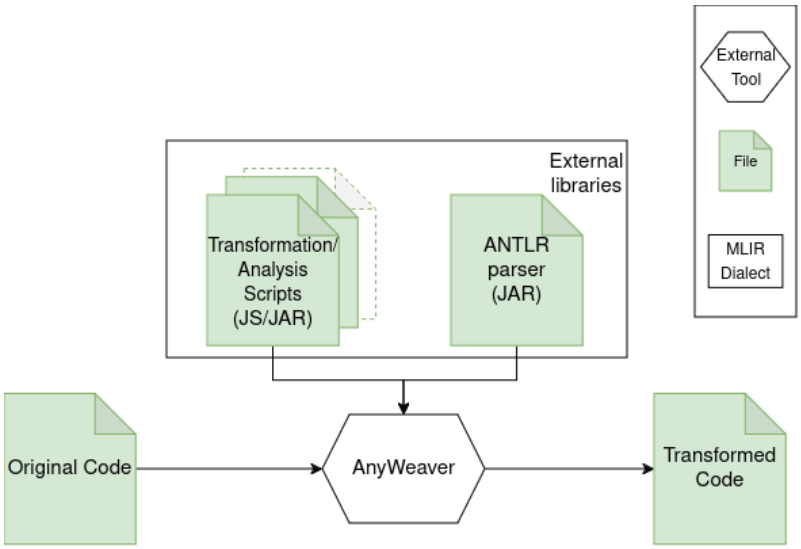

Figure 2.4: AnyWeaver use-case with ANTLR parser.

2.1.5 Streaming Vector & Computation

In the context of this work, streaming refers to a sequence of data flowing between the memory and the processor. This stream must efficiently manage different memory access patterns, align with hardware design considerations, and separate memory access from the processor pipeline. This separation ensures that indexing operations do not impact the overall computational loop.

This work delves into a particular streaming engine that implements the *Unlimited Vector Extension (UVE)* instruction set as its interface [15]. This design was tailored to address challenges inherent in Single Instruction Multiple Data (SIMD) extensions, specifically those optimised for fixed-size registers, such as ARM Scalable Vector Extension (SVE) and RISC-V Vector Extension (RVV).

Within the streaming paradigm employed by *UVE*, a direct streaming mechanism facilitates the seamless input flow directly into the register file. This architectural choice eliminates the need for separate memory access during computational processes, providing a streamlined and efficient computational pathway, as stated by *UVE* paper authors [15].

Articulating memory access patterns within *UVE* intentionally avoids explicit indexing. This strategic design choice optimises the streaming process's efficiency, thereby reducing the complexities associated with loop control. The dedicated Streaming Engine within *UVE* can execute scatter-gather operations, transforming non-coalesced accesses into linear patterns. This strategic approach ensures the sequential alignment of data in memory from the execution pipeline viewpoint, thereby increasing the overall efficacy of vectorisation.

A comprehensive description of all streams at the loop preamble is what allows *UVE* code to eliminate not only indexing instructions but also explicit loads and stores. This implicit handling adds a layer of efficiency to the computational process. In alignment with SVE and RVV, *UVE* exhibits register size agnosticism. However, it distinguishes itself by not requiring control instructions to define active vector elements. This adaptive approach contributes to the seamless handling of varying vector sizes, enhancing the computational versatility of the framework [15].

2.2 Related Work

Approaches for streaming and other computation paradigms, such as vectorisation or fully custom computing, exist at different abstraction levels. Some designs explicitly present the unconventional hardware capabilities to the programmer via DSLs, while compiler-level approaches delegate the task of identifying streaming opportunities during code generation. The following sections briefly summarise and exemplify both cases.

2.2.1 Domain Specific Language for Streaming Computation

Existing DSLs for streaming (also referred to as data-flow) proposes adding specific syntax to high-level languages to expose this computing paradigm. This DSLs serves as a wrapper to functionalities implemented by a middle layer, which may still require dedicated compilers to realise fully.

The StreamIt methodology is specifically designed to directly integrate support for streaming computation into high-level source code [19]. This involves the utilisation of various techniques to represent and manipulate abstractions that serve as representations of data streams and operations. This design choice provides the programmer with a notable degree of control over the configuration of the data-flow, enabling the exploration of diverse design variations. The implementation of StreamIt is done in *Java*, capitalising on the language's mature features. However, as acknowledged by the authors, it is worth noting that an opportunity exists to improve the clarity of the syntax employed in the language.

In the MaxCompiler flow for generating Field-Programmable Gate Array (FPGA) bitstreams [4], the developer must provide a host application in *C/C++* kernels and a manager encapsulating those kernels in *MaxJ*. Kernels delineate the computations chosen for hardware acceleration while the manager configures their interfaces for interaction with the host application.

The MaxCompiler flow for generating FPGA bitstreams, the developer must provide a host application in *C/C++*. The kernels and a manager encapsulating them are implemented in *MaxJ*, a high-level domain-specific language (DSL) based on Java with operator overloading. The FAST language, illustrated in Figure 2.5, serves as an imperative language for data flow models [3].

```
void Price_FPGA(float* p, float c_0_0_0, float c_p_0_0, float c_n_0_0, int n1, int ORDER) {

    in(p);

    float* i4 = count(1000, 1);
    float* i1 = countChain(n1, 1, i4);

    #pragma FAST DSPBalance:full
    int result = p[0] * c_0_0_0 + p[1] * c_p_0_0 + p[-1] * c_n_0_0;

    int up =(i1 >= ORDER) && (i1 < n1 - ORDER);
    out(up ? result : p);
}

void Price_CPU(...) {...}

int main() {
    #pragma FAST hw:Price_FPGA
    Price_CPU(...);
}
```

Figure 2.5: "FAST dataflow kernel for European Options pricing [3]."

It utilises the MaxCompiler as a back-end. Still, it provides support for *C/C++* by introducing a transformation step which generates *MaxJ* code, thus making it a type of source-to-source approach, or even an High-Level-Synthesis (HLS) approach given that it is intended for FPGA targets. The code is transformed based on user-supplied source annotations in pragmas. This approach does not require modifying the target *C/C++* code with custom DSL syntax but is only suitable for targeting FPGAs. Figure 2.6 presents the tool flow for development, highlighting the primary components of MaxCompiler.

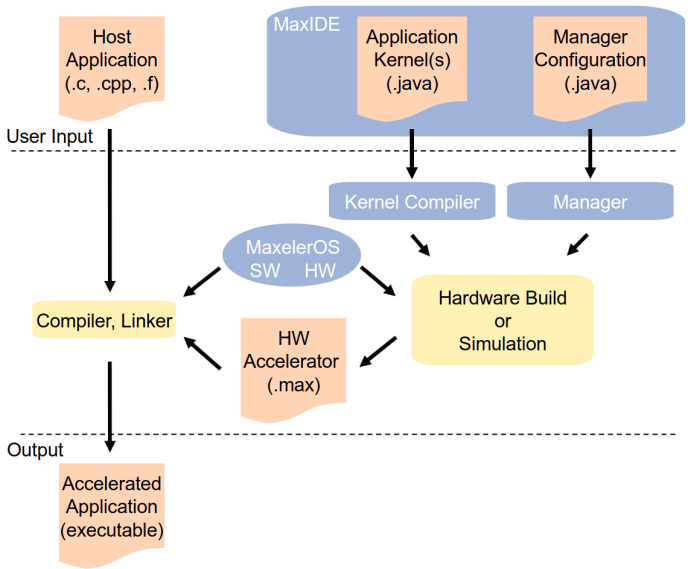

Figure 2.6: Maxeler development tool flow [4].

The SARA approach generates code for the Plasticine Coarse-Grained Re-configurable Array (CGRA) by using Spatial [5] [20] [21], a DSL for hardware accelerator specification, as a front-end, whereas SARA proper is the proposed back-end compiler. Using Spatial to represent computation kernels mandates that developers articulate the kernel code within the confines of the DSL, potentially impacting code portability. Therefore, the approach offers a structured flow for optimisation exploration, despite the trade-off of language-specific algorithm expression, and only specifically for CGRAs. Figure 2.7 shows SARA's two-phase process to target Plasticine.

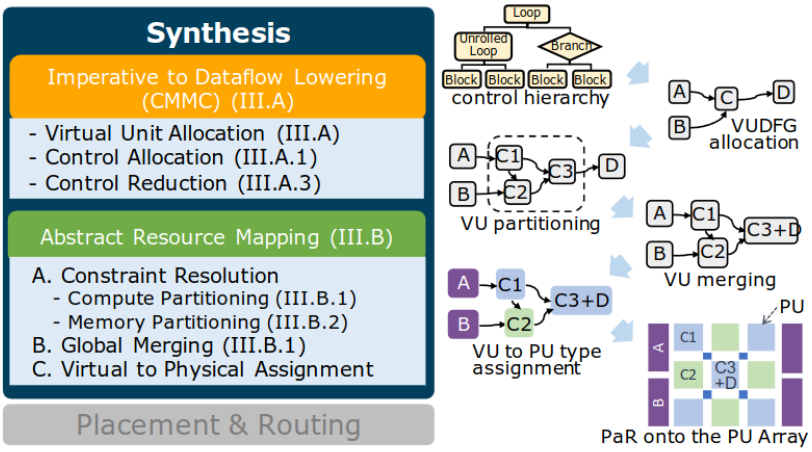

Figure 2.7: SARA's compilation flow [5].

2.2.2 Compiler Support for Streaming Computation

While previously presented approaches used some compiler support, these were custom compilers tailored for the respective DSLs or hardware targets. It was also up to the developer to exploit the features the DSLs exposed. In contrast, approaches presented in this section identify streaming opportunities during code generation by relying on existing compilation front-ends which produce *LLVM-IR* code (or MLIR dialects).

The OXiGen flow is based on extracting data flow graphs from *LLVM-IR* code generated from any front-end language compilable to this representation [6]. The targeted *LLVM-IR* code is restricted to a user-specified function name. OXiGen then infers streams, despite access pattern limitations, from memory accesses performed through any iteration variable in the outermost loop dimension. Vectorisation is inserted by modifying the inferred streams' input and output data types. Graphs are finally translated into *MaxJ* code for use with the MaxCompiler [4], i.e., only FPGAs are targeted. The front-end, back-end and overall design flow for OXiGen are shown in Figure 2.8.

SODA-OPT and Bambu emerge as a possible option in the domain of high-level compilation and hardware synthesis [7]. These tools harness MLIR to contribute to generating hardware accelerators. SODA-OPT utilises MLIR to generate finely-tuned *LLVM-IR* code optimised for

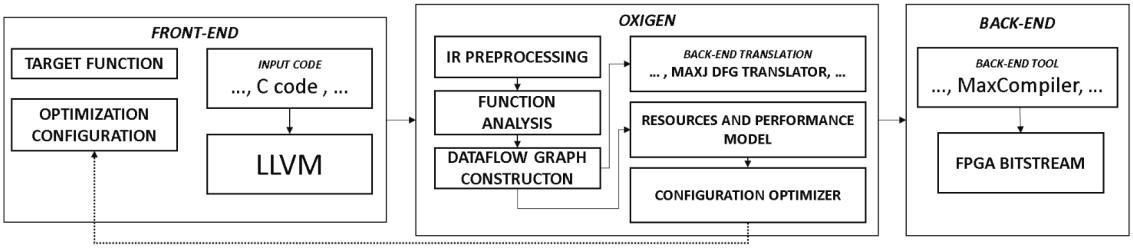

Figure 2.8: The infrastructure surrounding OXiGen [6].

HLS enhancements. Its architecture extracts essential kernels from high-level applications, facilitating the creation of specialised accelerators tailored for specific tasks. The Bambu HLS tool complements SODA-OPT by consuming the *LLVM-IR* and producing hardware descriptions for Application-Specific Integrated Circuit (ASIC) and FPGAs. Figure 2.9 provides an overview of this framework.

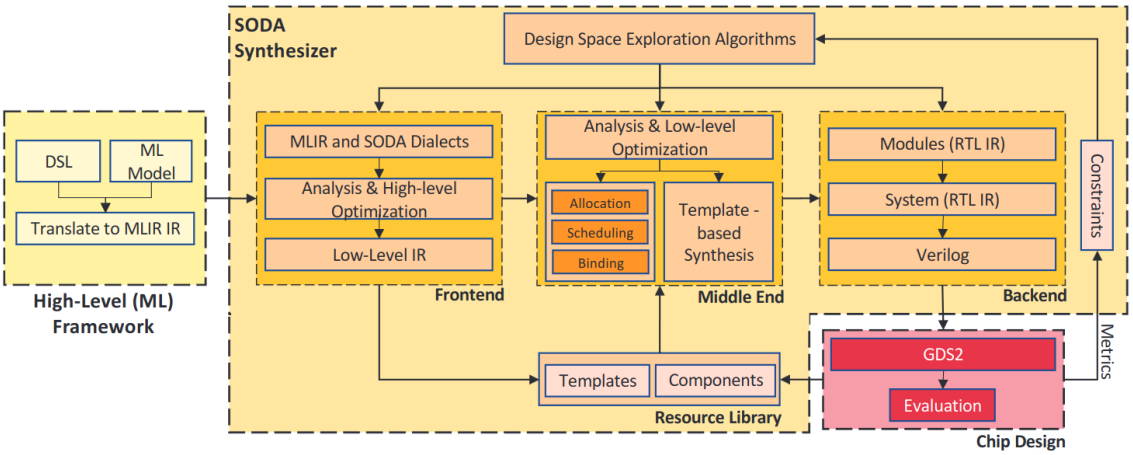

Figure 2.9: Overview of the SODA Synthesizer [7].

2.2.3 Relevant MLIR Work

The approaches most similar to the work presented in this dissertation are the CIRCT project, and the streaming dialect proposed in [9]. The Circuit IR Compilers and Tools (CIRCT) project is part of the Low Level Virtual Machine (LLVM) ecosystem and extends MLIR's capabilities into the realm of hardware design [8]. As a domain-specific compiler infrastructure, CIRCT focuses on optimising and transforming code from various high-level languages into Hardware Description Languages (HDLs) such as *SystemVerilog* [8]. It exploits the modular and extensible features of MLIR, enabling it to gain advantages in expressing hardware designs in a structured and composable fashion by resorting to dialects specifically designed for hardware description. These dialects encapsulate the semantics of hardware constructs and facilitate the transformation of high-level designs into code suitable for synthesis. Figure 2.10 briefly overviews its dialects and how they interact.

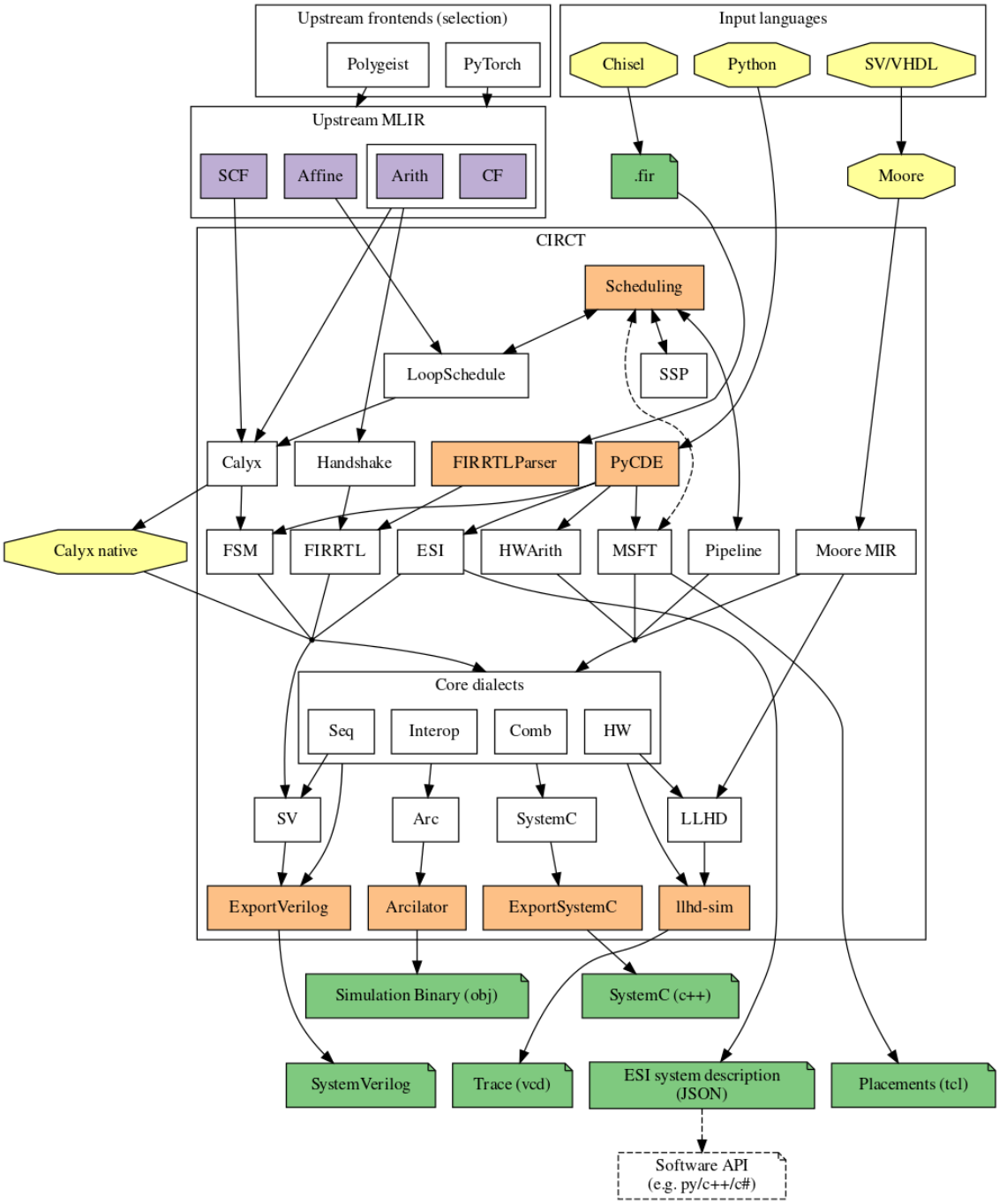

Figure 2.10: CIRCT dialects [8].

In the work presented in [9], the authors implemented a streaming abstraction as an MLIR dialect illustrated in Figure 2.11.

```
func.func @top(%in: !stream.stream<i64>)
    -> !stream.stream<i64> {
    %out = stream.map(%in) : (!stream.stream<i64>)
    -> !stream.stream<i64> {
        ^0(%val : i64):
        %0 = arith.constant 10 : i64
        %r = arith.addi %0, %val : i64
        stream.yield %r : i64
    }
    return %out : !stream.stream<i64>
}
```

Figure 2.11: "An example of a stream map operation expressed with a mixture of dialects. The computation consumes each input stream element and emits a new stream. Here, all values were increased by the constant 10" [9].

While the dialect expresses data-flow patterns (e.g., mapping of input streams into output streams), the computation performed over stream items is represented in existing dialects like *arith*. By defining a generic transformation of a data stream into CIRCT's *handshake* dialect, the flow re-utilises other existing core dialects to generate HDL. Specifically, each operation applied over a stream item is mapped to a *handshake* operation, which facilitates the incorporation of transformations to generate scheduled pipelines exploiting data parallelism and iteration overlapping. The approach was validated by generating circuits for a small set of synthetic cases for a setup consisting of a host CPU and Peripheral Component Interconnect Express (PCIe) based FPGA.

The *ArmSME* dialect [10], illustrated in Figure 2.12, is designed to target the Scalable Matrix Extension (SME).

```
%result = arm_sme.fmopa_2way %lhs, %rhs :
  vector<[8]xf16>, vector<[8]xf16> into vector<[4]x[4]xf32>
```

Figure 2.12: Operation involving the summation of two expanded outer products. It uses two one-dimensional scalable vectors as input and produces a two-dimensional scalable vector as output [10].

This dialect defines custom *LLVM-IR* intrinsic operations used specifically for Advanced RISC Machines (ARM) SME, facilitating the conversion of certain operations to SME-specific instructions. For example, the dialect allows the lowering of a matrix multiplication operation from a dialect *linalg* to ARM SME's Floating-Point Outer Product (FMOPA) operations. Other operations support tasks such as copying tiles, loading and storing tile slices, and performing various arithmetic and logical operations on matrices. Additionally, the dialect includes instructions for handling different data types and memory layouts, ensuring optimised performance for matrix operations. In terms of assembly code generation, the *ArmSME* dialect first generates *LLVM-IR*

dialect using LLVM intrinsics within the dialect. This dialect produces an *LLVM-IR* file that contains the regular computations and the intrinsics that represent the engine optimisations. This file is then processed by the LLVM compiler, which generates the final assembly code containing SME instructions and regular ARM assembly. This flow is illustrated in Figure 2.13

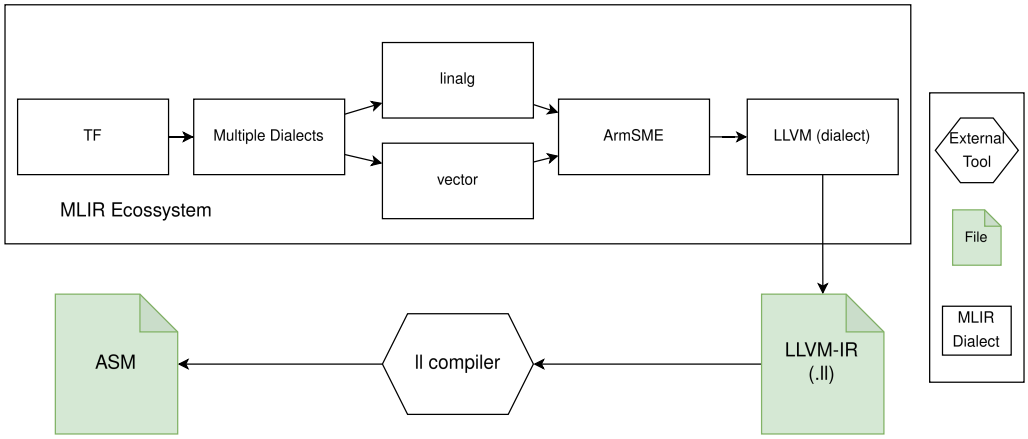

Figure 2.13: Generation of ARM SME assembly from TF dialect flow.

POM is an end-to-end optimising framework built on MLIR specifically designed for efficient FPGA-based accelerator generation [11]. POM introduces a multi-layered IR approach, which contrasts with traditional HLS tools that rely on a single-level IR. POM integrates the polyhedral model into MLIR, enabling advanced dependence analysis and a variety of loop transformations for optimising performance on FPGAs. The framework features a user-friendly DSL, illustrated in Figure 2.14, that separates algorithm specification from scheduling, allowing users to easily describe computations and explore different optimisation strategies without extensive code restructuring. Additionally, POM includes an automatic Domain Space Exploration (DSE) engine to identify high-performance optimisation schemes efficiently.

```
1 for (t= 0; t <_PB_TSTEPS; t++){
2     for (i= 1; i < _PB_N - 1; i++)
3         B[i] = 0.33333 * (A[i-1] + A[i] + A[i + 1]);
4     for (1 = 1; i < _PB_N - 1; i++)
5         Ali] = 0.33333 * (B[i-1] + B[i] + B[i + 11);
6 }
```

Figure 2.14: "Jacobi-1d described with POM DSL" [11].

Figure 2.15 illustrates the entire compilation workflow. Explicitly, POM distinctly segments the compilation process into three layers. Following a specific schedule, this approach systematically lowers the DSL through progressive transformations across different abstraction levels. At these various IR levels—namely the dependence graph IR, polyhedral IR, and MLIR *affine* dialect equipped with HLS pragma attributes—the process includes performing dependence analysis, code transformation, and hardware optimisation.

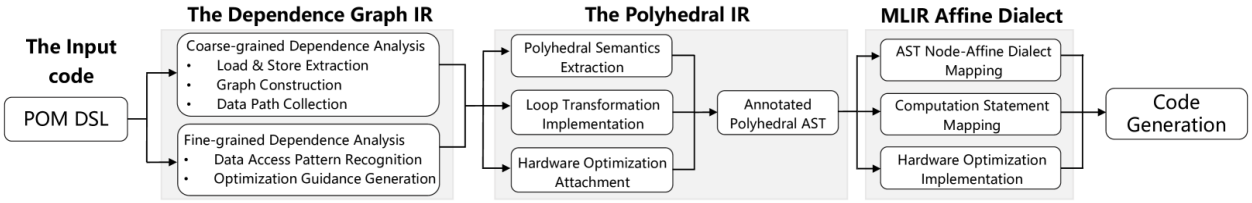

Figure 2.15: POM flow compilation [11].

CGRV-OPT is a compilation framework designed to optimise and deploy large-scale data-flow applications on heterogeneous architectures, specifically targeting CGRAs and RISC-V Central Processing Units (CPUs) [12]. CGRV-OPT, built on the MLIR framework, bridges the gap between high-level workload descriptions and low-level intermediate representations for various architectures. It simplifies the integration of CGRAs and CPUs, offering a user-friendly interface that enables developers, even with limited hardware expertise, to use CGRAs effectively. Key features include automated optimisations and transformations that enhance kernel performance and an automated hardware/software partitioning mechanism that intelligently maps kernels to either the CPU or CGRA based on performance models, maximising hardware utilisation. The structure of CGRV-OPT is illustrated in Figure 2.16. CGRV-OPT accepts input from AI models developed in *PyTorch* or *TensorFlow*, as well as programs written in *C*. This is achieved by integrating three open-source projects: Torch-MLIR, TF-MLIR, and toolPolygeist [22] [23] [24]. These projects translate applications into MLIR files, which CGRV-OPT can process. CGRV-OPT then performs a series of optimisations and transformations. First, it identifies and extracts all kernels. The optimisation process then focuses on improving the performance of these kernels, ultimately converting them into *LLVM-IR*. This *LLVM-IR* can then be passed to a toolchain to generate CGRA configuration data.

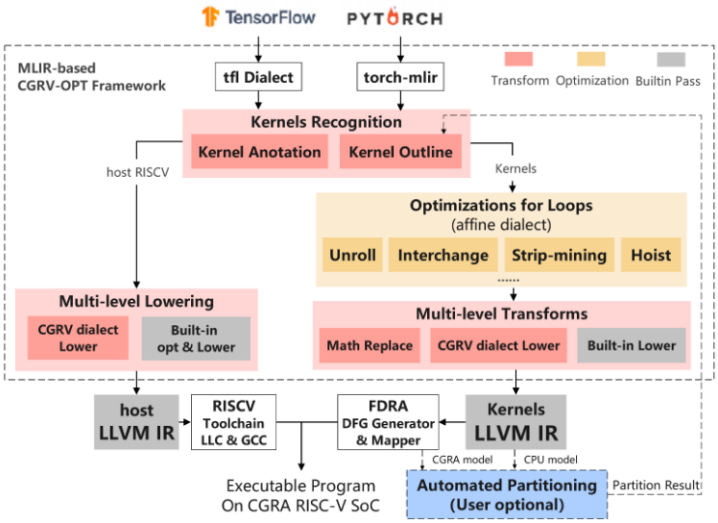

Figure 2.16: "Framework of CGRV-OPT. The blue section represents the optional automated software-hardware process" [12].

2.3 Summary

Table 2.1 summarises all the approaches, including some component or aspect of targeting various hardware platforms through high-level language support, DSL integration, and MLIR incorporation, demonstrating the diverse strategies of optimising code for different computing architectures.

Table 2.1: State-of-the-art summary.

Approach	High-level Language Support	DSL Support	MLIR Integration	Target Platforms	Publication Date
This work	✗	✓	✓	GPPs, FPGAs	2024
StreamIt [25]	✗	✓	✗	GPPs	2002
MaxCompiler [4]	✓	✓	✗	FPGAs	2011
OXiGen [6]	✓	✗	✗	ASICs, FPGAs	2018
CIRCT [8]	✓	✗	✓	ASICs, FPGAs	2020
SARA [5]	✓	✓	✗	CGRAs (Plasticine)	2021
SODA & Bambu [7]	✓	✗	✓	ASICs, FPGAs	2022
Work in [9]	✗	✗	✓	ASICs, FPGAs	2022
POM [11]	✗	✓	✓	ASICs, FPGAs	2024
CGRV-OPT [12]	✗	✗	✓	ASICs, CGRAs	2024

Chapter 3

Support for Streaming SIMD Paradigms Via MLIR

This chapter introduces a newly developed intermediate representation specifically designed for streaming and vector operations called Structured Representation Language (SRL). By providing multiple formats, including a *JavaScript Object Notation (JSON)* based format for tool interfacing, a human-readable textual representation, and a custom MLIR dialect, *SRL* bridges the gap between high-level abstractions and low-level hardware-specific implementations.

This chapter also delves into the *SRL* Compiler, which is responsible for transforming these high-level representations into either assembly code of a compatible target or other outputs like the envisioned Hardware Description Language (HDL) output support.

The following sections will explain the previously existing compilation support for one of our target backends and how it was adapted as part of our proposed approach. They will also explain the various facets of the multiple aspects of *SRL*, including its *JSON* format, Domain Specific Language (DSL), and Multi-Level Intermediate Representation (MLIR) dialect. Each section provides detailed insights into the structure, functionality, and use cases of each *SRL* format, highlighting the implemented compilation steps.

3.1 Previous *LLVM-IR* Analysis Based *UVE* Generation

In the previously existing approach for *Unlimited Vector Extension (UVE)* code generation, as developed by the *UVE* extension authors, the compilation of arbitrary *C/C++* code begins with Clang transforming it into *LLVM-IR* code (i.e., *ll* format files). Then, the subsequent step is to analyse this Intermediate Representation (IR), looking for vectorisation and streaming optimisation opportunities and extracting the memory access patterns. In the final compilation stage, *UVE* representing those optimisations are injected as inline assembly in the original *ll* file, which will be consumed by the ll compiler that will generate the binaries. In summary, *UVE* is generated by analysing streaming opportunities at the *LLVM-IR* level. This analysis is then used to generate *UVE* assembly, which is directly injected as inline assembly into the *LLVM-IR*. The Figure 3.1

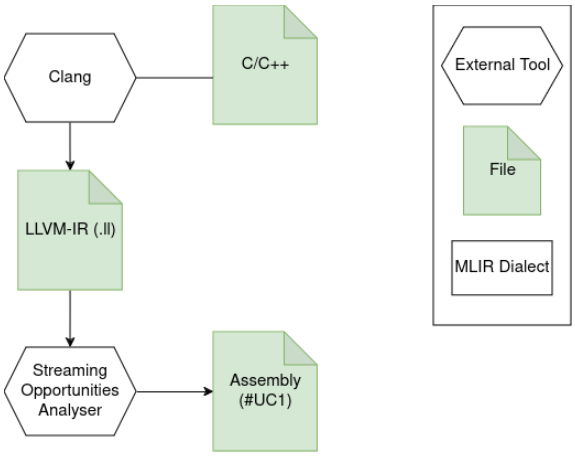

Figure 3.1: Current approach compilation flow.

illustrates the currently implemented solution. Our approach enhances this flow by exploiting higher-level information that is no longer present at *LLVM-IR* level but can be retained through MLIR dialects that are domain or context-specific, including our own proposed dialect.

3.2 Proposed MLIR Based Compilation for Stream & Vector Computing

This work will focus on the design of Structured Representation Language (SRL), both the DSL and MLIR dialect, and develop the *SRL* Compiler where these representations allow the automatic detection and representation of deterministic and non-deterministic computing structures and memory access schemes.

Specifically, instead of directly generating assembly from streaming and vectorisation opportunities identified through the analysis over *LLVM-IR*, these insights are re-utilised to generate a *JSON* that serves as the basis for *SRL* code generation.

Then, the compiler will generate an MLIR dialect equivalent to the DSL, and then translate the MLIR directly to *UVE* in its earlier (*UVE* 1) and its more recent specification (*UVE* 2) custom instructions [26] [27]. Finally, code generated by the *SRL* Compiler from a few *C/C++* benchmark kernels will be tested, specifically for *UVE* 1, since the simulation of *UVE* 2 code is currently not supported. The testing platform for *UVE* 1 was based on a modified version of the Spike simulator [28].

The Figure 3.2 describes the two approaches presented, the old and the new path, and can be used to compare them regarding the compilation of *C/C++*.

The main difference between the approach presented in this work (new path) and the previous one is the addition of the MLIR layer before the *UVE* assembly generation step. The principal advantage of this new approach lies in the flexibility and extensibility provided by MLIR. This addition allows for the possibility of targeting different architectures beyond *UVE* and extending other MLIR compilers that have dialects representable in a streaming abstraction.

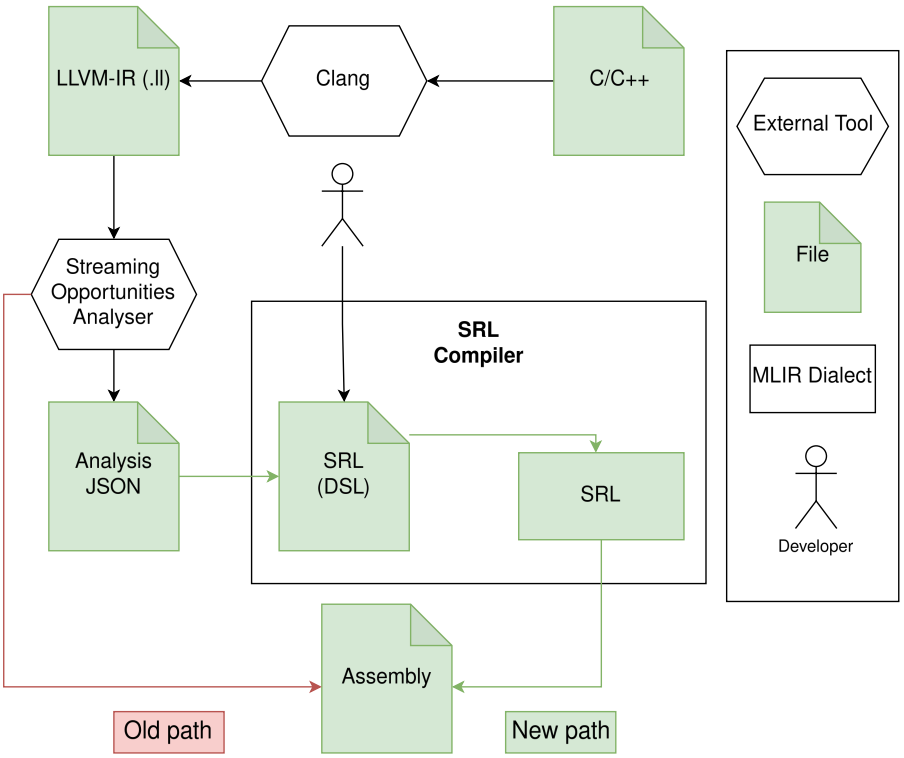

Figure 3.2: Comparison of the old path and the new path for *C/C++* compilation.

3.3 Structured Representation Language

Structured Representation Language (SRL) is a domain-specific representation for streaming and vector operations with three distinct formats: one in *JSON* for more straightforward interface with other tools, a DSL in a textual representation designed for use by human software developers, and an MLIR dialect used for the compilation backend, by integrating with other MLIR dialects to generate the final binaries. Briefly, *SRL* is organised into inputs, streams, and computation.

Firstly, inputs are defined by their identifiers and types (e.g., `int`, `float`) and are used similarly to function arguments or variable declarations.

Secondly, streams represent sequences of data elements processed in a specific order, enabling efficient data with complex access patterns. They facilitate the flow of data between different parts of the computation, allowing for the continuous processing of data elements as they become available.

Streams also have identifiers and types, can be specified as either input or output, have a base pointer, and include a list of dimensions (ranging from 1D to 8D). The dimensions are determined by an initial offset, a step, and a length (number of elements). Additionally, dimension modifiers can be declared to change the stride or length at runtime, allowing for more complex patterns, such as triangular access patterns. It is also possible to represent auxiliary streams, which is vital to guarantee code integrity.

Finally, for computation, *SRL* represents vector operations between streams, benefiting from

```
    "inputs": [
        {
            "datatype": "ptr",
            "name": "foo"
        },
        {
            "datatype": "i64",
            "name": "bar",
            "value": "1"
        }
    ],
...
```

Figure 3.3: Example of the **`inputs`** section of a *JSON* file.

the identification of streaming and vector operations performed over *LLVM-IR* highlighting computation segments as Controlflow Graphs (CFGs) and Dataflow Graphs (DFGs). The following subsections demonstrate the three different *SRL* formats.

3.3.1 JavaScript Object Notation Format

The JavaScript Object Notation (JSON) format was created to facilitate easier interfacing with other tools. It serves as a structured entry point. Specifically, the *JSON* format is used to formalise, in a parsable way, the current outputs of the Low Level Virtual Machine (LLVM) based analysis. This facilitates integration with *SRL* generation, which can then replicate the current information captured by the analysis by directly consuming this *JSON* output. This process gradually drove the development of the *SRL* syntax presented here. The *JSON* structure represents a function's data and control flows in a detailed and organised manner.

It includes the following fields:

- **`functionName`**
- **`program`**
- **`inputs`**
- **`streams`**
- **`vectors`**
- **`controlFlows`**
- **`dataFlows`**.

The **`functionName`** field specifies the name of the kernel function. Following this, the **`program`** field outlines the program's CFGs in sequence. The **`inputs`** field is an array of input specifications for the function, each described by its data type, name, and an optional initial value. If the input does not have an optional initial value, it represents a function argument instead of a variable declaration. Input examples are described in Figure 3.3 where `foo` represents a function argument and `bar` represents a variable declaration.

Then, the **`streams`** field defines the data streams used in the program. It specifies the stream's name, datatype, stream type, base variable, and one to eight dimensions, which include initialisation, stride, and the number of elements, These dimension can also have modifiers. For example,

```
...
{
    "name": "foo",
    "datatype": "f32",
    "type": "OutputStream",
    "base": "bar",
    "dims": [
        {
            "init": "baz",
            "stride": "qux",
            "nElements": "quux"
        }
        ...
    ]
    "mods": [
        {
            "type": "N_ELEMENTS",
            "behaviour": "inc",
            "amount": "baz",
            "dimAssociated": "baz",
            "dimTarget": "qux"
        }
    ],
    "scalar": false
}
...
```

Figure 3.4: Example of the `stream` element of a *JSON* file.

the stream `foo`, described in Figure 3.4, is defined as an output stream based on `bar`. This stream have a dimension initialised at `baz`, having a stride of `qux`, and containing `quux` elements. It also includes a modifier that adjusts the number of elements. This modifier is of type `N_ELEMENTS`, with a behaviour of `inc` (increment), specifying an amount of `baz` to increase the dimension length. It is associated with dimension `baz` and targets dimension `qux`. The **`name`** attribute is used as the stream identifier. The **`datatype`** attribute in *JSON* indicates the data type of the stream. The data types include 32-bit floating-point numbers (`f32`), void, 32-bit integers (`i32`), 64-bit integers (`i64`), and pointers (`ptr`). The `f32` type attribute identifies the type of the stream and can be either `InputStream` or `OutputStream`. Finally, the **`scalar`** is a special field indicates whether the stream is to be iterated element by element.

The **`vectors`** field lists vectors used in the function, detailing their data type, name, and assigned value. These vectors represent Single Instruction Multiple Data (SIMD) vectors with the same element in all the positions. These vectors can also be defined in the DFG. In Figure 3.5 **`vectors`** field lists the vector `"u4"` as a 32-bit float with the value of `"A"` described.

The **`controlFlows`** field details the control flow structures within the program. Within the *JSON*, each CFG contains five fields: a list of the initial DFGs, inner CFGs which correspond to nested loops that appear before initial DFGs, a name tag for identification, and the stopping condition. In the example shown in Figure 3.6, control flow begins at `"node::1"` with the condition `"whilePresent::u1::0"`, with no inner CFGs or end nodes specified. The DFG nodes are defined in the DFG *JSON* field that will be explained later. The condition can be interpreted as "while dimension 0 of the stream `u1` is present".

```
...
{
    "datatype": "f32",
    "name": "u4",
    "value": "A"
}
...
```

Figure 3.5: Example of the "vector" element of a *JSON* file.

```
...
{
    "begin": [
        "node::1"
    ],
    "condition": "whilePresent::u1::0",
    "end": [],
    "innerControlFlow": [],
    "tag": "loop::1"
}
...
```

Figure 3.6: Example of a "CFG" element of a *JSON* file.

Finally, the **dataFlows** field represents the computation segment as a DFG, with each node containing optional arguments, subsequent nodes in the flow, and the type of operation or action performed by the node. The **dataFlows** field includes nodes performing various operations. For instance, in Figure 3.7 `node::1` assigns `u1`, while `node::2` performs addition of the `node::3` and `node::4`.. This DFG is also illustrated in Figure 3.8 for easier visualisation.

```
...
{
    "node::1": {
        "args": [
            "u1"
        ],
        "children": [
            "node::2"
        ],
        "type": "="
    },
    "node::2": {
        "children": [
            "node::3",
            "node::4"
        ],
        "type": "+"
    }
}
...
```

Figure 3.7: Example of a "DFG" element of a *JSON* file.

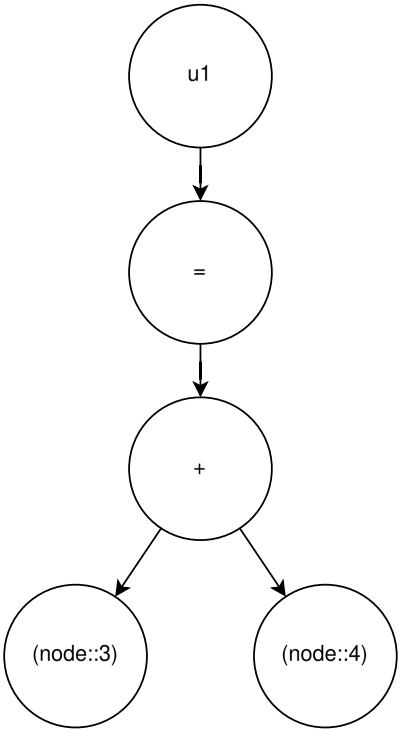

Figure 3.8: Visualisation of the DFG illustrated on Figure 3.7. `node::1` assigns a value to `u1`, while `node::2` performs an addition operation involving the outputs of `node::3` and `node::4`.

3.3.2 Domain Specific Language

The *SRL* DSL elements were initially defined based on the outputs of the *LLVM-IR* analysis in their *JSON* format. Besides being able to represent the analysis outputs (e.g., stream configurations and computations), the syntax, inspired by *C++* to be familiar to developers, was designed for readability, making it human writable while being tailored for the specific use case of stream operations. Although the primary purpose of this DSL is to be automatically generated, it can also be used by experts for manual optimisations or debugging a streaming platform using DSL linked as external *C* functions. In this last case, DSL function names must match the corresponding external *C* functions that will be accelerated to be then linked at the assembly level.

This DSL operates in a function-based manner and encompasses inputs, streams, and computations.

Regarding DSL, function syntax includes the function name, a list of arguments and the return type. Following this, the body of the program contains the stream declarations and the computations, all enclosed in curly braces.

Depending on their origins, inputs are represented by standard variables or function arguments. Figure 3.9 illustrates a simple function with two input declarations (`foo` and `bar`). The function `simple_function` takes one input argument, `foo`, which is of type **`i32`** (32-bit integer). Inside the function, a local variable `bar` is declared and initialised to 7, and it is of type **`i64`** (64-bit integer). The syntax for declaring a function argument follows the pattern of specifying the type

first, followed by the variable name, while local variables inside the function are declared and initialised.

```
void simple_function(i32 foo){
    i64 bar = 7;
}
```

Figure 3.9: Example of inputs declaration in the DSL.

Streams can be perceived as an Object Oriented Programming (OOP) class, where their configurations are methods. Variable declarations follow a type-identifier format, with types including standard data types like **i32**, **i64**, **f32**, **ptr**, and **void**. These declarations can also involve initialisation. To ensure flexibility in stream declarations, the DSL supports templated types, allowing for parameterised types akin to *C++* templates. For instance, **InputStream**<**f32**> and **OutputStream**<**f32**> indicate stream objects parameterised with a floating-point type, using angle brackets '< >'. These stream objects can invoke configuration methods using dot (.) notation, such as **appendDim** and **appendSizeMod**. Figure 3.10 illustrates the declaration and configuration generic stream objects.

```
void configureStreams() {
    InputStream<f32> foo;
    foo.appendDim(0, 100, 1);   //init, nElements, stride
    foo.appendDim(0, 100, 1);
    foo.appendSizeMod(inc, 1, 0, 1);    // increment mode, increment ammont,
                                        //dimAssociated, dimTarget

    OutputStream<f32> bar;
    bar.appendDim(0, 100, 1);
}
```

Figure 3.10: Example of stream declarations and configurations using the human-readable DSL version of SRL.

The function `configureStreams` declares an **InputStream** and an **OutputStream**, both parameterised with the **f32** type. These stream objects invoke the **appendDim** method using dot notation to configure their dimensions, specifying the *init*, *stride*, and *number of elements* of the stream. The first stream also invoke the **appendSizeMod** method.

Regarding control, the DSL includes the **whilePresent** loop, which checks the state of streams. The **whilePresent** syntax consists of the identifier of the stream and an integer representing the number of the dimension to use in the control flow of the computation, followed by a block with the stream computation. For computation, *SRL* supports arithmetic operations (`*`, `/`, `+`, `-`), assignments (`=`), function calls, and method invocations.

Figure 3.11 illustrates the use of the **whilePresent** loop in *SRL*. The `computeStream` function declares two **InputStream** and one **OutputStream**, both parameterised with the **f32** type. These stream objects invoke the **appendDim** method to configure the streams with similar dimensions. The **whilePresent** loop checks the state of `foo`, using the first dimension (indicated by

the integer 0) to control the loop. Inside the loop, an arithmetic operation is performed where each element of `bar` is multiplied by `baz` and assigned to `foo`. Conceptually, this code represents the exact computation as a segment of `C` code where the arithmetic operations are done using an iterator.

```
void computeStream(ptr foo_address, ptr bar_address, ptr baz_address){
    OutputStream<f32> foo(foo_address);
    InputStream<f32> bar(bar_address);
    InputStream<f32> baz(baz_address);
    foo.appendDim(0, 100, 1);
    bar.appendDim(0, 100, 1);
    baz.appendDim(0, 100, 1);

    whilePresent(foo, 0) {
        foo = bar * baz;
    }
}
```

Figure 3.11: Example of a `whilePresent` loop and arithmetic operations in *SRL*

For the list of this DSL refer to Appendix C.

3.3.3 Multi-Level Intermediate Representation Dialect

The *srl* dialect functions as a bridge between our DSL and the MLIR ecosystem. It translates high-level DSL constructs into a format compatible with MLIR, facilitating further optimisation and integration with various MLIR dialects. Moreover, the *srl* dialect plays a unique role as an intermediary interface, bridging high-level dialects, such as *linalg*, and lower-end targets like *UVE* assembly or specialised hardware. This evolution would enhance the identification and optimisation of streaming and vector operations. By integrating *SRL* directly into the MLIR ecosystem, it could potentially replace the current *LLVM-IR* analysis flow, illustrated in Figure 3.2, with a more direct compilation flow from *C/C++* code, thus streamlining the process of identifying and optimising streaming and vector operations.

Additionally, control flow elements, such as loops and conditionals, that are mapped to *SRL* constructs align with existing MLIR dialects. In addition to this, *SRL* handles arithmetic, logical operations, and loops through standard MLIR operations, such as those in the *arith* dialect. This straightforward integration positions *SRL* as a potential target for future integration with other MLIR dialects, including *affine* [29], *linalg* [30], *vector* [31].

Concerning its constructs, in the *srl* dialect, the newly created stream type is represented by `stream<>`. The configuration of these streams, including operations like appending dimensions, is managed through MLIR *SRL*-specific operations. These operations ensure that stream properties are correctly set up and maintained within the MLIR framework, allowing for efficient manipulation and utilisation of streaming data. Figure 3.12 express the DSL code in Figure 3.11 by illustrating the declaration and configuration of streams in MLIR using the *srl* dialect. It creates two `InputStream` of type `stream<f32>` using the `srl.new_input_stream` operation and configures it with the `srl.append_dim` operation to set its dimensions. Similarly, an `OutputStream`

is created and configured. Moreover, it also possible to use modifiers at MLIR level such as **srl.append_size_mod**.

```
func computeStream(%foo_address: ptr, %bar_address: ptr, %baz_address: ptr): void {
    %zero = arith.constant 0 : i64
    %size = arith.constant 100 : i64
    %stride = arith.constant 1 : i64

    %foo = srl.new_output_stream %foo_address: stream<f32>
    %bar = srl.new_output_stream %bar_address: stream<f32>
    %baz = srl.new_output_stream %baz_address: stream<f32>

    srl.append_dim %foo, %zero, %size, %stride: void
    srl.append_dim %bar, %zero, %size, %stride: void
    srl.append_dim %baz, %zero, %size, %stride: void

    while_present(%foo, 0) {
        srl.add %foo, %bar, %baz: void
    }
}
```

Figure 3.12: Example of stream declaration and configuration in MLIR using the *srl* dialect

One major challenge of mirroring DSL constructs is ensuring a valid MLIR representation for stream types and operations. Unlike traditional MLIR representations, which use Single Static Assignment (SSA) for variables, stream operations in *SRL* must preserve the identity of stream registers. This is essential to prevent the compatibility issues encountered in some Circuit IR Compilers and Tools (CIRCT) dialects, which are important to avoid. It is considered that in general, and for compatibility with our *UVE* target, a stream register at this level will be associated with a concrete register implemented by a specific streaming-capable hardware, so there is no flexibility for register re-assignments after operations. That is, register names and states must be accurately maintained.

To adjust the SSA nature of MLIR, the *srl* dialect uses the return/modified variable as the first operand, similarly to regular assembly. This approach is also applied to stream object methods. Figure 3.12 illustrates the DSL code adhering to SSA principles, particularly in the addition operation (`srl.add %foo, %bar, %baz: void`) where `%foo` is the SSA variable that will hold the result of the addition, while `%bar` and `%baz` are the operands. Unlike representing the computation as `%foo = srl.add %bar, %baz : stream<f32>`, which would violate SSA by redeclaring `%foo` (previously declared as `%foo = srl.new_output_stream %foo_address: stream<f32>`), this method ensures SSA compliance.

For detailed information on MLIR operations related to the *srl* dialect please refer Appendix C.

3.4 SRL Compiler

This section discusses the different translation steps of *SRL* Compiler to convert from the *JSON* format of *SRL* to *UVE* assembly. *SRL* Compiler is the tool responsible for optimising and converting the DSL into the dialect and converting the dialect into assembly. This compiler was developed

using AnyWeaver instead of the LLVM's MLIR libraries. This choice was made for two main reasons: LLVM's libraries are extensive and difficult to extend, partly due to code complexity and partly due to the underlying compilation of any LLVM based compiler itself. Furthermore, our in-house LARA based tools already have good code analysis and transformation support. This choice does not imply compatibility problems with other MLIR tools because it is possible to pass the IR between different tools using MLIR's textual representation, which can be emitted and parsed natively using the standard distributed MLIR tools.

Moreover, like the majority of MLIR compilers, this compiler is structured in two main parts: the translator (srl-translate), to convert between different formats, and the optimiser (srl-opt), to optimise at the MLIR layer. In this work, the focus will be on the translations (srl-translate). The translations are the following:

1. *JSON* to *SRL* native,
2. *SRL* native to *srl* dialect,
3. *srl* dialect to *UVE* assembly;

Using this compiler for arbitrary *C/C++* code begins with Clang parsing the source code and transforming it into *LLVM-IR*. Subsequent steps involve analysing the *LLVM-IR* to identify data streaming opportunities and extract access patterns. This data is then emitted as *JSON* format in the structure defined, containing essential streaming and vectorisation information. This *JSON* serves as the basis for generating the DSL, converting it into the dialect, and finally producing the *UVE* assembly. Outputs of these translations are presented as results in Chapter 4.

3.4.1 Transforming the JSON format into DSL

This subsection details the process of translating *JSON* input into DSL code. The transformation leverages prior analysis performed on the *LLVM-IR* generated by Clang, stored in the predefined *JSON* format. This facilitates converting high-level code into the *srl* dialect.

First, the *JSON* attributes for inputs are mapped directly to DSL declarations. The `datatype` attribute in *JSON* specifies the variable type, such as `ptr`, `f32`, and `i64`, which correspond to pointer, float, and integer types in DSL. The `name` attribute provides the variable's name used in the DSL. If a `value` attribute is present, it initialises the variable in the DSL.

For example:

```
"inputs": [
        {
            "datatype": "ptr",
            "name": "foo"
        }
        {
            "datatype": "i64",
            "name": "bar",
            "value": "1"
        }
```

```
    ],
```

is translated in to following DSL code:

```
void example_function(ptr foo)
{
    i64 bar = 1;
}
```

Once the variables are declared, streams are configured. Each stream declaration in *JSON* includes attributes like **name**, **datatype**, **type**, **base**, and **dims**. These attributes are translated to their DSL equivalents. The **name** attribute becomes the stream's name in DSL. The **type** and **datatype** attributes are combined into a single DSL declaration, such as **OutputStream**<**f32**>. The **base** attribute specifies the base address or identifier for the stream, which is passed as an argument in the stream's constructor. Dimensional attributes in *JSON* are translated to method calls in DSL to append dimensions to the stream.

As an illustration:

```
{
        "name": "foo",
        "datatype": "f32",
        "type": "OutputStream",
        "base": "bar",
        "dims": [
            {
                "init": "baz",
                "stride": "qux",
                "nElements": "quux"
            }
        ]
},
```

becomes in DSL:

```
    OutputStream<f32> foo(bar);
    foo.appendDim(baz, qux, quux);
```

Control flow elements in *JSON* are converted into corresponding control structures in DSL. *JSON* properties like **begin**, **condition**, and **end** are translated into loops or conditional blocks in DSL. The **begin** property specifies the starting nodes, while the **condition** defines the loop's condition. The DSL code inside the loop executes the operations defined by these nodes.

Lastly, data flow configurations in *JSON*, which define operations and assignments within a computational graph, are translated into corresponding DSL code. Each node in *JSON* represents an operation, with attributes specifying its type, arguments, and children nodes. These nodes are transformed into DSL expressions and statements that execute the specified operations.

The process begins with reading the *JSON* file and storing its information in specialised classes representing different elements of *SRL*, such as function names, streams, inputs, DFGs, and CFGs. Inputs without a specific known value are treated as function arguments to the generated DSL function call, while others are treated as local variables with known initial values. Stream objects

are translated into their corresponding statements in the *SRL* syntax. For computations, the *SRL* code is generated based on the **program** field in the *JSON*.

3.4.2 Transforming the DSL into Dialect

In this step, AnyWeaver consumes a Java ARchive (JAR) file for an ANTLR generated parser for our *SRL* DSL grammar. Then, AnyWeaver consumes the DSL code file and generates an Abstract Syntax Tree (AST). The AST nodes are then visited, utilising visitor patterns to traverse and process the nodes. The system operates through several key components and processes that collectively facilitate this transformation.

After parsing the AST, a declarations map is created for each function node, where the keys are variable names and the values are their corresponding nodes. This map is constructed by searching for function declaration arguments and general declarations within the given node, combining the results, and iterating over them to populate the map. If a variable is re-declared, an error is thrown, indicating the lines of both declarations.

At the system's core are the visitor functions, each designed to handle a specific node type within the syntax tree. These functions interpret the syntax and semantics of each node, converting them into the corresponding MLIR code. The traversal starts with a main visitor function configured to determine the appropriate visit function for each node type using a configuration object. When processing identifiers, the visitors use the declarations map to ensure that each variable or function is correctly recognised throughout the code. Figure 3.13 illustrates an example of a visitor function.

```
function visitObjectType(node: Joinpoint): string[] {

    checkVisitorType(node, "ObjectType");

    const classNameChild = node.children[0];
    const templatedTypeChild = node.children[1];

    let className = visitIdentifier(classNameChild)[1];
    let code: string = ""

    code = objectNameToMlirType[className]
    if (!code) {
        throw new Error("Undefined class name " + className);
    }

    if (templatedTypeChild) {
        const datatype = visitTemplatedType(templatedTypeChild)[1];
        code += '<' + datatype + '>';
    }

    return ['', code, className];
}
```

Figure 3.13: Visitor function for converting an object type from a DSL node to an MLIR type. This function processes the class name and any templated types, generating the corresponding MLIR type code.

Moreover, a counter is maintained to generate unique names for SSA (Static Single Assignment) variables, which is crucial for the correct formation of MLIR code. Specific visit functions handle different IR constructs, ensuring that each type of node is accurately translated into MLIR. For instance, there are dedicated functions for processing identifiers, integer operations, function declarations, types, variable declarations, loops, method calls, binary operations, and function calls. Each of these visit functions ensures that the corresponding IR constructs are translated into valid and efficient MLIR code.

3.4.3 Transforming the Dialect into UVE assembly

The conversion from the MLIR dialect to *UVE* assembly involves translating high-level constructs into lower-level instructions suitable for execution on the hardware ensuring that constants, variables, streams and control flow constructs are accurately represented and optimised for *UVE* architectures.

For constant initialisation, MLIR constants are defined using the `arith.constant` operation are translated into *UVE* assembly using the `addi` instruction. This operation loads the constant values into specific registers. For instance, a constant defined in MLIR would be translated to an equivalent *UVE* assembly representation that initialises these constants in temporary registers.

For example, this MLIR statement:

```
%foo = arith.constant 42 : i32
```

Is converted into this Reduced Instruction Set Computer Five (RISC-V) assembly:

```
addi t0, x0, 42
```

This is a temporary solution because repeatedly loading constants with `addi` instructions within loops or across multiple operations can cause inefficiencies, leading to redundant and costly performance overhead.

For stream declaration and configuration, we need a first pass in the generated AST from the parser to collect all the stream information necessary for generating valid *UVE* because the order of instructions is vital for correct stream configuration. The *srl* dialect facilitates this process, as the configuration order in *SRL* is not as strict as in assembly. Additionally, this pass creates a variable usage map, which is necessary in the register assignment phase and checks if all the variables used have been declared.

In the register assignment phase, identified streams are assigned to *UVE* registers in order from `u0` to `u32`, which are reserved for *UVE* streams. This assignment generates a variable register assignment map, ensuring each stream is correctly mapped to its corresponding register. Following this, the streams' declaration is generated. After this, visitors are used to generate the computation, ignoring the MLIR statements related to the streams' declaration and configuration since they have already been generated in the previous step.

After this step, the process of creating a *UVE* 1 stream involves several steps, which are outlined below:

1. **Determine the Instruction Type:**
    - Identify the appropriate instruction type (e.g. half-word, word, double-work) based on the stream's data type (e.g. float, double).
    - Map the data type to its corresponding instruction representation (e.g. `.w`, `.d`).
2. **Stream Type Configuration:**
    - **Auxiliary Streams:**
        - Configure with an empty configuration string.
    - **Input and Output Streams:**
        - Retrieve the specific *UVE* stream register from the register map using the stream's name.
        - Select the instruction template according to the stream type.
        - Remove any unnecessary suffixes for one-dimensional streams.
3. **Construct the Configuration String:**
    - Concatenate the instruction, data type, stream register, and address register.
    - Append dimension-specific declarations to manage the spatial properties.
4. **Dimension-Specific Declarations:**
    - **First Dimension:**
        - Obtain the number of elements and stride from the register map.
        - Append these details to the configuration string.
        - Apply modifiers using specific instruction formats if required.
    - **Intermediate Dimensions (if any):**
        - Process each dimension similarly by retrieving initialisation values, strides, and the number of elements.
        - Apply any modifiers to the stream register.
    - **Last Dimension (if present):**
        - Finalise the stream setup with an **`ss.end`** instruction.
        - Append the initialisation value, stride, and number of elements to the configuration string.
5. **Final Configuration String:**
    - Return the complete configuration string, encapsulating all necessary instructions and configurations.

In contrast, *UVE* 2's process simplifies the handling of dimensions by initially addressing all but the last dimension, consolidating initialisation values, strides, and number of elements into a single append instruction (`ss.app`). Modifiers are then applied in the same manner as *UVE* 1. The last dimension, if present, is appended with an end instruction similar to *UVE* 1.

The computation, such as loops and arithmetic operations in MLIR, are translated into corresponding constructs in *UVE* assembly using visitors. This approach remains consistent across both *UVE* 1 and *UVE* 2, with no differences in the visitors used for computation generation. Figure 3.14 exemplifies one of these computation instructions visits.

```
function visitWhilePresentOp(node: Joinpoint): CodePreVisitReturn {

    checkVisitorType(node, "WhilePresentOp");

    const ssaIdSreamNameChild = node.children[1];
    const integerChild = node.children[2];
    const regionChild = Query.searchFrom(node, "Region").getFirst() as Joinpoint;

    const dimNumber = visitInteger(integerChild).code;
    const streamRegister = visitSsaId(ssaIdSreamNameChild).code;

    const loopTag = ".Loop_" + String(loopTagNumber++);
    const loopControl =
        `${'0' === dimNumber ? 'so.b.nc' : 'so.b.ndc.${dimNumber}'}
        ${streamRegister}, ${loopTag}`;

    let code = loopTag + ':\n';
    code += visitRegion(regionChild).code;
    code += loopControl;

    return { code: code, pre: "" }
}
```

Figure 3.14: Example MLIR to *UVE* visitor used for computation generation. This visitor is used to transform a **`while_present`** loop in a conditional branch.

3.5 Future Solution for Fully Integrated MLIR Flow

In the future, we aim to remove the analysis of *LLVM-IR* because the loss of information at such a low level can lead to the misuse of the streaming and vector optimisations provided by SRL. For this, since upstream dialects can be generated from *C/C++* code, placing *srl* dialect at an intermediary stage between these upstream dialects and lower-end dialects or targets will allow for future compilation support starting from high-level code, replacing the current analysis over *LLVM-IR* and integrating this process of identifying streaming and vector opportunities natively, into a one-shot compilation flow. Shown in Figure 3.15, the *C/C++* input shown in is fed directly into the MLIR ecosystem, bypassing the use of the *SRL* DSL if desired, although the DSL remains a valuable tool for explicit manipulation of data streams and vector operations by a human code developer knowledgeable about the capabilities of the target hardware. In this solution, *SRL* can

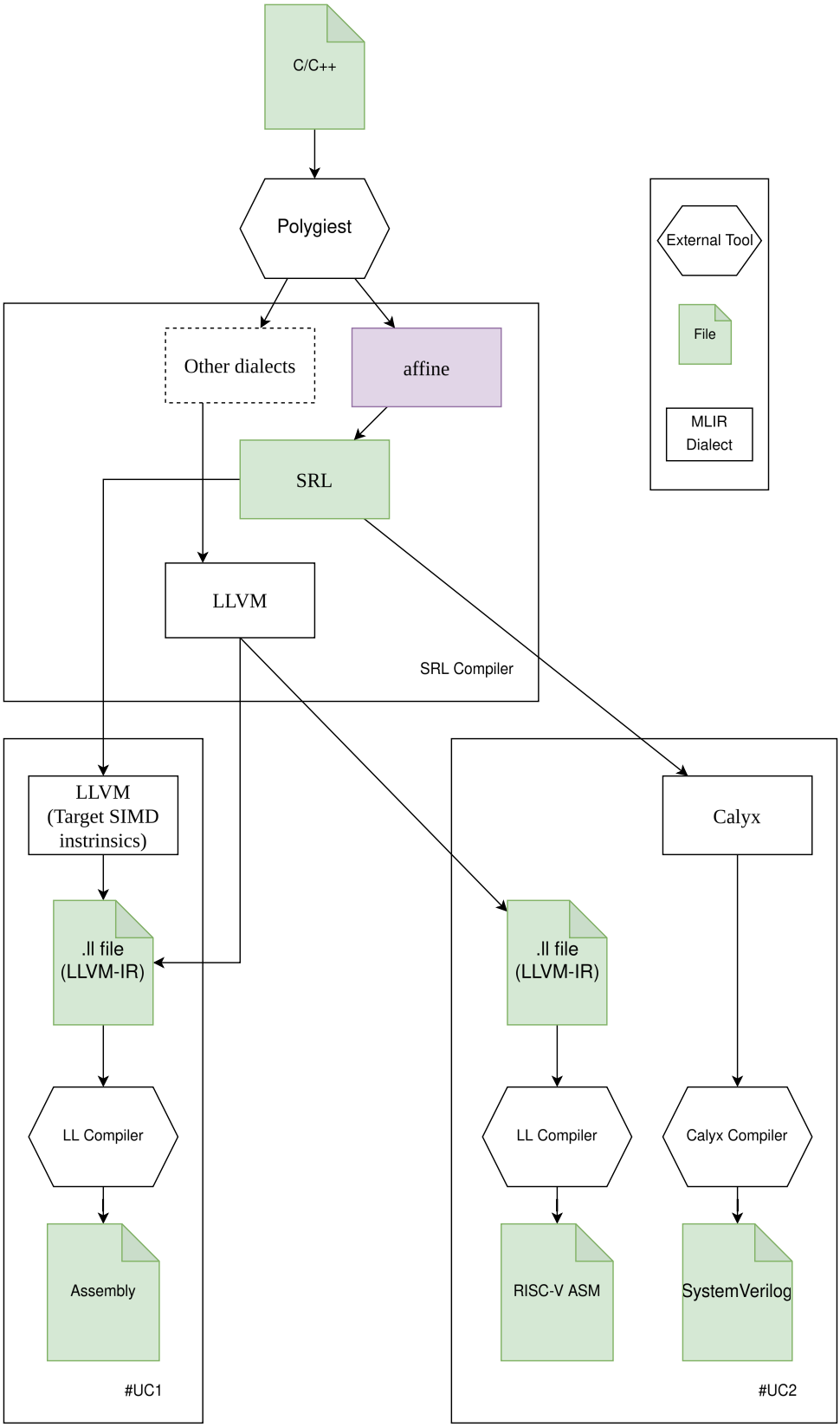

Figure 3.15: Future solution for streaming paradigms via MLIR.

be seen as a compiler plugin for those who wish to utilise streaming and vector SIMD engines or generate custom hardware for this purpose. For targeting SIMD extensions, we will generate *LLVM-IR* intrinsics to better integrate with existing MLIR work and resolve the linking issues present in the current approach. In this case, assembly code will be generated by the LLVM compiler. We also intend to explore possible optimisations within *srl* dialect.

In addition to this advantage regarding the use case of targeting GPPs with SIMD extensions, i.e., Use Case (UC) 1 in Figure 3.2, this flow also facilitates other targets, like HDL generation from *SRL* information, as explained in the next section.

3.5.1 Circuit IR Compilers and Tools Analysis

Concerning the UC 2, to find a path from *srl* dialect onto HDL, possible paths between hardware related dialects in Circuit IR Compilers and Tools (CIRCT) were tested, which is the subset shown in Figure 3.16. The test scopes were limited to *C/C++*, so transformation paths starting from the Polygiest compiler [24], the current *C/C++* entry point into MLIR, were tested. Paths shown in green represent successful transformations, dotted paths are reported as possible, while the path between *fsm* and *sv* (dialect for the *SystemVerilog* syntax) failed, even for the provided example cases. It was determined that the *calyx* dialect supports processing code containing the established dialects *affine*, *sfc*, *arith*, and *cf*. Therefore, placing the *srl* dialect above this level and transforming it into code relying on these dialects provides a viable path to generate HDL code implementing streaming and vector operations, fulfilling the requirement. Additionally, Calyx has its native DSL, providing a second lowering path into HDL using Calyx-specific tools [32]. In this regard, the *SRL* DSL serves a similar function by allowing a different entry point into MLIR that does not depend on Polygiest.

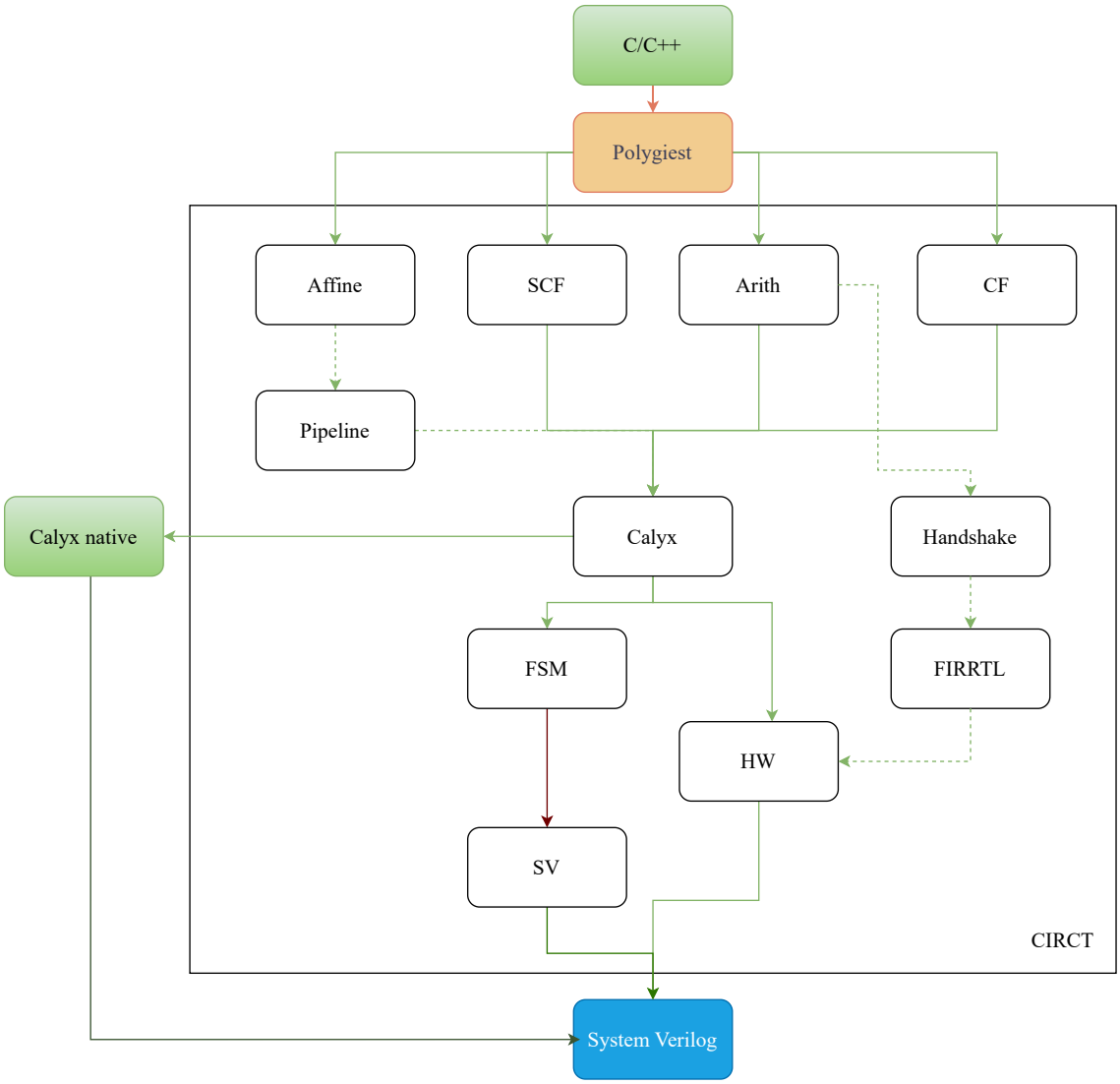

Figure 3.16: Subset of current CIRCT ecosystem, focusing on dialects and transformation paths which take *C/C++* as input (adapted from figure in [8]).

Chapter 4

Results & Discussion

The following chapter presents a comprehensive overview of the results, demonstrating the compiler's efficiency in mapping high-level operations to low-level vectorised instructions. This includes a detailed analysis of the compilation process for the Single Precision A X Plus Y (SAXPY) kernel, encompassing all intermediate stages of compilation. Furthermore, the results discuss the generation of RISC-V Vector Extension (RVV) proposal assembly and conclude with a performance evaluation of the generated code compared to conventional Reduced Instruction Set Computer Five (RISC-V). All compilation intermediate stages can be found in Appendix B.

4.1 Experimental Setup

To validate both *Structured Representation Language (SRL)* and its compiler, four different C kernels were used: SAXPY (Figure 4.1), Triangular Solve (TRISOLV) (Figure 4.3), 3 Matrix Multiplication (3MM) (Figure 4.5) and General Matrix Multiplication (GEMM) (Figure 4.7). Consequently, to validate the compilation and test the performance of these C kernels, *Unlimited Vector Extension (UVE)* 1 was used as the target platform since, at the time of this work, *UVE* 2 did not have a working testing platform. The testing platform for *UVE* 1 was a modified version of the Spike simulator [28].

Previously, work compared the original Instruction Set Architecture (ISA) with Arm Scalable Vector Extension (SVE) and Neon, using a modified version of the gem5 ISS [15]. This earlier evaluation was dependent on the micro-architecture used, specifically the Arm Cortex-A76, which had better metrics (e.g., RAM usage) but did not support the *UVE* version used in this work.

To evaluate the SRL Compiler, the four benchmarks were compiled firstly using using only RISC-V instructions, which serves as the performance comparison baseline, and then using *UVE* instructions. The RISC-V assembly was directly derived by compiling *C/C++* with GCC using full optimisation ('-O3' flag) and auto-vectorisation. In contrast, the *UVE* code was generated by the SRL Compiler leveraging the *LLVM-IR* analysis pass. For both RISC-V and *UVE*, the kernel functions were linked to the main program using an 'extern *C*' function declaration, and binary files were generated for execution on Spike. Due to the lack of full back-end compiler support for

the SRL Compiler, the kernel functions were initially compiled with the '-O0' optimisation level, resulting in empty body functions. The generated *UVE* assembly code was manually inserted into the compiled output.

Table 4.1 provides a clear comparison between the RISC-V and *UVE* evaluation processes, highlighting the key aspects and differences.

Table 4.1: Comparison of Evaluation Process: RISC-V vs *UVE*

Aspect	**RISC-V**	***UVE***
Assembly Generation	GCC with `-O3` and auto-vectorisation	GCC generated assembly function empty body with `-O0` containing UVE generated by *LLVM-IR* analysis pass and SRL Compiler
Function Linking	Linked using `extern C` function declaration	
Simulator	Modified Spike [28].	
Evaluation Aspect	Instruction count.	
Kernel Data Input Size	2500	

Results were obtained with Spike configured with a UVE 's vector registers with a maximum length of 512 and matrices and vectors with a size of 2500.

4.2 Translation Results Overview

In the SAXPY *C* code (Figure 4.1), a simple loop iterates over the elements of two arrays, performing arithmetic operations. In contrast, in the *UVE* assembly code (Figure 4.2), to enhance performance, streams are configured to represent the complex access patterns of the loops, enabling more efficient data handling. The streams are configured using the stride, which in this case is one, along with the number of elements in each array. A vectorised version of the value `A` is created to enable vectorised operations. Then, the computation inside the **`for`** loop is substituted by the loop delimited by **`so.b.nc`** (lines 10 to 13) which is equivalent to a **`while_present`** loop for the first dimension in *srl* dialect, containing equivalent arithmetic operations.

```
1
2 #define SIZE 32
3
4 void saxpy(float dest[SIZE],
5     float src[SIZE],
6     float A) {
7
8     for (int i = 0; i < SIZE; i++) {
9
10         dest[i] += src[i] * A;
11
12     }
13 }
```

Figure 4.1: SAXPY C Kernel.

```
1     addi t1, x0, 0 # %sn = 0
2     addi t2, x0, 1 # %snm1 = 1
3     ss.st.w u1, %[x], %[size], t3
4     ss.cfg.vec u1
5     ss.ld.w u2, %[y], %[size], t3
6     ss.cfg.vec u2
7     ss.ld.w u3, %[x], %[size], t3
8     ss.cfg.vec u3
9     so.v.dp.w u4, %[A], p0
10    .Loop_0:
11    so.a.mul.fp u5, u2, u4, p0
12    so.a.add.fp u1, u3, u5, p0
13    so.b.nc u1, .Loop_0
```

Figure 4.2: SAXPY *UVE* result.

In the TRISOLV *C* code (Figure 4.3), a nested loop iterates over the elements of the arrays to solve a triangular system of linear equations. The outer loop initialises each element of `x` with the corresponding value of `b`, and then the inner loop updates `x[i]` by subtracting the product of elements from `L` and `x`. Finally, each `x[i]` is divided by the diagonal element of L. In the *UVE* assembly code (Figure 4.4), the use of size modifiers plays a key role in managing the triangular access shape. The streams in the assembly code employ two dimensions with a size modifier to accommodate the triangular nature of the data access. Specifically, the **`ss.app.mod.siz.inc`** instructions, associated to the first dimension of each stream, adjust the size of the data streams dynamically, which allows the assembly code to handle varying access patterns efficiently. The size modifier adjustments are reflected in instructions like **`so.a.mul.fp`**, **`so.a.sub.fp`**, and **`so.a.div.fp`** within the conditional branch loop. These instructions replace the arithmetic operations (multiplication, subtraction, and division) in the *C* nested **`for`** loop, ensuring that the iterations are processed in a vectorised manner.

```
1  void trisolv(float *L,
2      float *b,
3      float *x) {
4
5      for (int i = 0; i < SIZE; ++i) {
6
7          x[i] = b[i];
8          for (int j = 0; j < i; ++j) {
9
10             x[i] -= L[i * SIZE + j]
11                 * x[j];
12
13         }
14
15         x[i] = x[i] / L[i * SIZE + i];
16
17     }
18
19 }
```

Figure 4.3: TRISOLV C Kernel.

```
1  addi t1, x0, 3 # %snm1 = 4
2  addi t2, x0, 3 # %snp1 = 6
3  addi t3, x0, 3 # %sn = 5
4  addi t4, x0, 1 # %one = 1
5  addi t5, x0, 0 # %zero = 0
6  ss.sta.ld.w u1, %[x],t3, t3
7  ss.app.mod.siz.inc u1, t1, t4
8  ss.end u1, t5, t5, t4
9  ss.cfg.vec u1
10 ss.sta.ld.w u2, %[z],t3, t5
11 ss.app.mod.siz.inc u2, t1, t4
12 ss.end u2, t5, t5, t4
13 ss.cfg.vec u2
14 ss.ld.w u3, %[y],t3, t4
15 ss.ld.w u4, %[x],t3, t2
16 ss.st.w u5, %[z],t3, t4
17 .Loop_0:
18 so.v.dp.w u6, t5, p0
19 .Loop_1:
20 so.a.mul.fp u6, u2, u1, p0
21 so.a.sub.fp u7, u7, u6, p0
22 so.b.ndc.1 u1, .Loop_1
23 so.a.adde.fp u6, u7, p0
24 so.a.add.fp u6, u6, u3, p0
25 so.a.div.fp u5, u6, u4, p0
```

Figure 4.4: TRISOLV *UVE* result.

In the 3MM *C* code (Figure 4.5), the nested loops iterate over the arrays `src1`, `src2`, and `src3` to compute the matrix multiplication. Each element of `src3` is computed as the dot product of the corresponding row of `src1` and column of `src2`. The array `src3` is initialised to zero before the multiplication process begins. In contrast, the 3MM *UVE* assembly code (Figure 4.6) optimises this process by configuring streams to handle the array data efficiently. The streams are set up with appropriate strides and lengths for `src1`, `src2`, and `src3`, enabling vectorised operations. The loops are replaced by specialised instructions like **`so.a.mul.fp`** and **`so.a.add.fp`** to perform parallel multiplication and accumulation operations, significantly enhancing performance.

The GEMM *C* implementation (Figure 4.7) initialises each element of matrix C by scaling

```c
void _3mm(float *src1,
    float *src2,
    float *src3,
    uint64_t si,
    uint64_t sj,
    uint64_t sk) {

    int i,j,k;
    for (i = 0; i < si; i++) {
        for (j = 0; j < sj; j++){
            src3[i*sj+j] = 0;
            for (k = 0; k < sk; k++){
              src3[i*sj+j] +=
                src1[i*sk+k]
                *src2[k*sj+j];

            }
        }
    }

}
```

Figure 4.5: 3MM C Kernel.

```
addi t1, x0, 0 # %zero = 0
addi t2, x0, 1 # %one = 1
ss.sta.ld.w u1, %[src1],%[sk], %[si]
ss.app u1, t1, t1, %[sj]
ss.end u1, t1, t2, %[sk]
ss.cfg.vec u1
ss.sta.ld.w u2, %[src2],t1, %[si]
ss.app u2, t1, t2, %[sj]
ss.end u2, t1, %[sj], %[sk]
ss.cfg.vec u2
ss.sta.st.w u3, %[src3],%[sj], %[si]
ss.end u3, t1, t2, %[sj]
ss.cfg.vec u3
.Loop_0:
so.v.dp.w u4, t1, p0
.Loop_1:
so.a.mul.fp u5, u1, u2, p0
so.a.add.fp u4, u4, u5, p0
so.b.ndc.1 u2, .Loop_1
so.a.adde.fp u3, u4, p0
so.b.nc u2, .Loop_0
```

Figure 4.6: 3MM *UVE* result.

it with a factor beta, then it iterates through the elements of matrices A and B, performing a series of multiply-add operations. On the other hand, the *UVE* assembly code (Figure 4.8) demonstrates a vectorised approach to implementing the GEMM operation. While the declaration of bi-dimensional streams are used to access data from the matrices A, B and C, the use of instructions like **`so.a.mul.fp`** and **`so.a.add.fp`** in the *UVE* code corresponds to the arithmetic operations in the *C* implementation.

Figure 4.9 shows a comparison between the instruction count between conventional RISC-V code, generated by GCC, and RISC-V with *UVE* generated by the SRL Compiler. This figure illustrates that the SAXPY kernel experiences a performance increase of over 41,8 times, the performance TRISOLV kernel increases by more than 18,9, the 3MM kernel performs more than 25 and GEMM kernel over 31,9 times. All of these results were related to kernel data input data sizes of 2500 elements. Note, these results are similar to the previous obtained with the LLVM-IR analysis, although this uses the new created abstraction and compiler in the compilation middle end.

In the code generated by GCC for RISC-V instructions, loops iterate over one or multiple-dimension arrays, performing arithmetic operations in each iteration that may not fully utilise the capabilities of hardware for performance optimisation. In contrast, *UVE* assembly code enhances performance. Streams are configured to represent the complex access patterns of the loops, enabling more efficient data handling ensuring that the computational loop focuses solely on performing the arithmetic operations, which are optimised for execution on the *UVE* architecture in a vectorised way. By configuring the streams in the preamble, the *UVE* code minimises the overhead associated with data access within the loop. This approach allows for better utilisation of the hardware's vector processing capabilities, reducing the number of instructions executed per iteration and improving overall execution speed.

```c
void gemm(float alpha,
    float beta,
    float *C,
    float *A,
    float *B,
    uint64_t si,
    uint64_t sj,
    uint64_t sk){

  int i, j, k;

  for (i = 0; i < si; i++) {
    for (j = 0; j < sj; j++)
      C[i * si + j] *= beta;
    for (k = 0; k < sk; k++) {
       for (j = 0; j < sj; j++)
        C[i * si + j] += alpha
            * A[i * si + k]
            * B[k*sk + j];
    }

  }

}
```

Figure 4.7: GEMM C Kernel.

```
addi t1, x0, 0 # %zero = 0
addi t2, x0, 1 # %one = 1
ss.sta.ld.w u1, %[C],%[sj], %[si]
ss.end u1, t1, t2, %[sj]
ss.cfg.vec u1
ss.sta.st.w u2, %[C],%[sj], %[si]
ss.end u2, t1, t2, %[sj]
ss.cfg.vec u2
ss.sta.ld.w u3, %[C],%[sj], %[si]
ss.app u3, t1, t1, %[sk]
ss.end u3, t1, t2, %[sj]
ss.cfg.vec u3
ss.sta.st.w u4, %[C],%[sj], %[si]
ss.app u4, t1, t1, %[sk]
ss.end u4, t1, t2, %[sj]
ss.cfg.vec u4
ss.sta.st.w u5, %[B],t1, %[si]
ss.app u5, t1, %[sj], %[sk]
ss.end u5, t1, t2, %[sj]
ss.cfg.vec u5
ss.sta.st.w u6, %[A],%[sk], %[si]
ss.app u6, t1, t2, %[sk]
ss.end u6, t1, t1, %[sj]
ss.cfg.vec u6
so.v.dp.w u7, %[beta], p0
so.v.dp.w u8, %[alpha], p0
.Loop_0:
.Loop_1:
so.a.mul.fp u1, u2, u7, p0
so.b.ndc.1 u1, .Loop_1
.Loop_2:
so.a.mul.fp u9, u6, u8, p0
so.a.mul.fp u10, u9, u5, p0
so.a.add.fp u3, u10, u4, p0
so.b.ndc.2 u3, .Loop_2
so.b.nc u1, .Loop_0
```

Figure 4.8: GEMM *UVE* result.

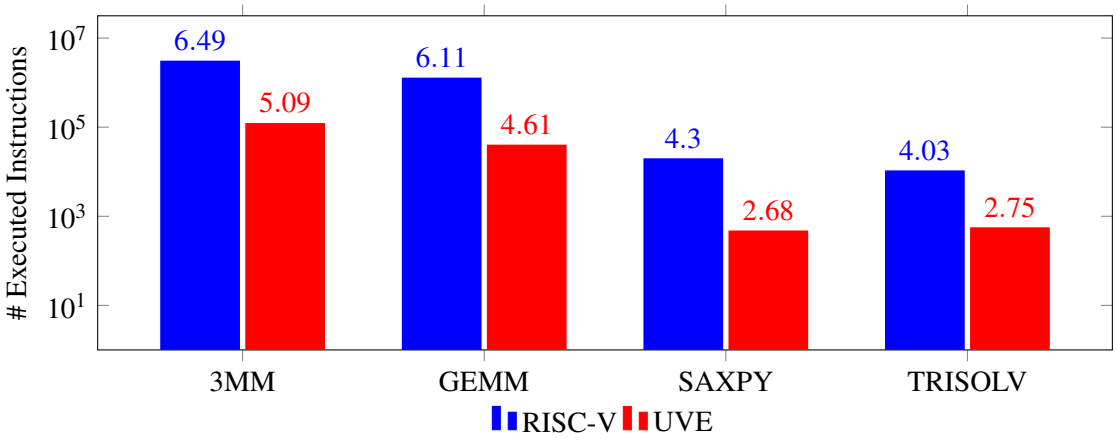

Figure 4.9: Comparison of Instruction Counts for Kernels using conventional RISC-V and UVE code generated by the SRL Compiler

The size of the Single Instruction Multiple Data (SIMD) bus, in this case 512 bits, is important to consider when assessing performance gains. While our analysis mostly concentrates on

instruction count, which might not always be a good indicator of potential speedup, it does show that using *srl* can be advantageous when aiming for effective SIMD platforms.

4.3 SAXPY Full Transformation Analysis

This section presents a detailed analysis of the complete transformation process applied to the *Single Precision A X Plus Y (SAXPY)* equation. Chosen for its simplicity and relevance, the SAXPY operation serves as an ideal example to demonstrate various translation techniques from source code to *LLVM-IR* and subsequent transformations.

While the complete example is available in Appendix B for reference, we will provide it here in full, accompanied by a step-by-step breakdown to enhance clarity and understanding of each phase in the transformation.

4.3.1 JSON to DSL Conversion

The code in Figure B.1 illustrates the original SAXPY kernel operation. This kernel is then passed to Clang to generate a *.ll* file. This file containing *LLVM-IR* instructions is passed through the analysis done in the previous work and generates the *JavaScript Object Notation (JSON)* illustrated in Figure B.2. This *JSON* is transformed to the Domain Specific Language (DSL) code illustrated in Figure B.3 passing through the various translation steps.

The given *JSON* format for the inputs of the SAXPY kernel is as follows:

```
    "inputs": [
        {
            "datatype": "ptr",
            "name": "x"
        },
        {
            "datatype": "ptr",
            "name": "y"
        },
        {
            "datatype": "f32",
            "name": "A"
        },
        {
            "datatype": "i64",
            "name": "size"
        },
        {
            "datatype": "i64",
            "name": "sn",
            "value": "0"
        },
        {
            "datatype": "i64",
            "name": "smn1",
            "value": "1"
        }
```

```
28     ],
```

Each attribute in the *JSON* format is directly translated into the corresponding declaration in the DSL.

Here is the corresponding DSL code derived from the *JSON* format:

```
1 void saxpy(ptr x, ptr y, f32 A, size)
2 {
3     i64 sn = 0;
4     i64 snm1 = 1;
5 }
```

In the DSL, the function `saxpy` takes three parameters: `x`, `y`, `A` and `size` corresponding to the inputs in the *JSON* format. Inside the function, the integer variables `sn` and `snm1` are initialised with their respective values as specified in the *JSON* input.

Then, the stream configurations are translated to DSL. This illustrates the first stream declaration in the SAXPY kernel in its *JSON* format:

```
1 {
2     "name": "u1",
3     "datatype": "f32",
4     "type": "OutputStream",
5     "base": "x",
6     "dims": [
7         {
8             "init": "sn",
9             "stride": "snm1",
10            "nElements": "size"
11        }
12    ]
13 },
14 {
15    "name": "u2",
16    "base": "y",
17    "datatype": "f32",
18    "type": "InputStream",
19    "dims": [
20        {
21            "init": "sn",
22            "stride": "snm1",
23            "nElements": "size"
24        }
25    ]
26 },
27 {
28    "name": "u3",
29    "base": "x",
30    "datatype": "f32",
31    "type": "InputStream",
32    "dims": [
33        {
34            "init": "zero",
35            "stride": "stride",
36            "nElements": "size"
```

```
        }
    ]
}
```

The **name** attribute is directly translated to the DSL declaration, where the stream is named u1.

The **type** and **datatype** attributes in *JSON*, which in this case are OutputStream and f32, are converted directly in the DSL to **OutputStream**<**f32**>.

The **base** attribute sets the base address or identifier for the stream to x. In the DSL, this is provided as the argument to the stream constructor: u1(x).

In the DSL, these dimensional attributes are translated into a method call that appends the dimension to the stream. The **appendDim** method adds the dimension to the stream **u1**. It takes the values for **init**, **stride**, and **nElements** as parameters that are already declared in the *JSON* inputs field.

This stream within *JSON* is translated into the *SRL* DSL code as follows:

```
OutputStream<f32> u1(x);
u1.appendDim(sn, snm1, size);

InputStream<f32> u2(y);
u2.appendDim(sn, snm1, size);

InputStream<f32> u3(x);
u3.appendDim(sn, snm1, size);
```

The other streams, u2 and u3, are configured similarly.

After this, concerning control flow in the *JSON*:

```
"controlFlows": [
    {
        "begin": [
            "node::0",
            "node::1"
        ],
        "condition": "whilePresent::u1::0",
        "end": [],
        "innerControlFlow": [],
        "tag": "loop::1"
    }
],
```

This control flow is transformed into a block, specifically a loop in DSL. The **begin** property specifies the nodes at which the control flow starts, in this case, `node::1` that will be replaced by the corresponding Dataflow Graph (DFG) code. The condition property, here given as `whilePresent::u1::0`, which indicates a whilePresent loop for `u1` iterating until the end of the dimension 0 is translated to **whilePresent**`(u1, 0)`.

This translation result on the following DSL code:

```
whilePresent(u1, 0)
{
    // node::1 code
```

```
    }
```

In the DSL, the `whilePresent(u1, 0)` statement initiates a loop that continues as long as elements in `u1` at index 0 are present. The code corresponding to `node::0` and `node::1` is executed inside this loop.

Finally, considering the following *JSON* configuration defining data flows:

```
"dataFlows": [
    {
        "node::0": {
            "args": ["u4"],
            "children": ["node::01"],
            "type": "="
        },
        "node::01": {
            "args": ["A"],
            "children": [],
            "type": "constantToStream"
        }
    },
    {
        "node::1": {
            "args": ["u1"],
            "children": ["node::2"],
            "type": "="
        },
        "node::2": {
            "children": ["node::3", "node::4"],
            "type": "+"
        },
        "node::3": {
            "args": ["u3"],
            "children": [],
            "type": "STREAM::load"
        },
        "node::4": {
            "children": ["node::5", "node::6"],
            "type": "*"
        },
        "node::5": {
            "args": ["u2"],
            "children": [],
            "type": "STREAM::load"
        },
        "node::6": {
            "args": ["u4"],
            "children": [],
            "type": "INPUT::value"
        }
    }
]
```

In this *JSON* structure, nodes and their relationships define data operations and assignments within a computational graph. This configuration is translated into *SRL* DSL as follows, where several operations have been concatenated into a single DSL statement:

```
// Node 0: Assignment u4 = constantToStream(A)
AuxStream<f32> u4 = constantToStream(A);

// Node 1: Assignment u1 = u3 + u2 * u4
u1 = u3 + u2 * u4;
```

4.3.2 DSL to MLIR Dialect Conversion

The transition from the DSL, illustrated in Figure B.3 to the MLIR ecosystem illustrated in Figure B.4. This process involved several 1:1 systematic substitutions and transformations to align the DSL with the MLIR syntax.

Initially, input constants from the DSL were replaced with Single Static Assignment (SSA) variables in MLIR, namely `%sn` and `%snm1`, which were defined using the *arith* dialect, which is the MLIR standard mechanism for declaring constants.

More precisely, regarding the example, this DSL statement:

```
i64 sn = 0;
i64 snm1 = 1;
```

Is translated to:

```
%sn = arith.constant 0: i64
%snm1 = arith.constant 1: i64
```

Concerning stream declarations translation into the MLIR ecosystem.

```
OutputStream<f32> u1(x);
```

Is translated to:

```
%u1 = srl.new_output_stream %x: stream<f32>
```

In this translation, the `OutputStream<f32>` type from the DSL is mapped to the MLIR type `stream` parameterised with the `f32` data type. The argument `x`, which represents the base address of the stream, is converted into its corresponding MLIR variable, the previous declaration in the code.

Method calls in the DSL, namely the ones for stream configuration, such as:

```
u1.appendDim(sn, smn1, size);
```

were also translated into MLIR instructions. For instance, this method call was converted to:

```
srl.append_dim %u1, %sn, %smn1, %size: void
```

In this transformation, the MLIR variable corresponding to the stream (`%u1`) became the first argument, followed by the method arguments (`sn`, `smn1`, `size`), which were replaced by their equivalent SSA variables.

Finally, computational statements in the DSL are substituted with the corresponding DSL computation instructions in MLIR. For instance, the DSL statement:

```
u1 = u3 + u2 * u4;
```

is translated to:

```
srl.mul %t0, %u2, %u4: void
srl.add %u1, %u3, %t0: void
```

This approach uses the left-hand operand of the instruction as a write target as the first operand to maintain the SSA nature of MLIR while accurately reflecting the original DSL operations.

4.3.3 MLIR to UVE Assembly Conversion

Respecting constant initialisation, the MLIR constants `%sn` and `%snm1` are directly translated to *UVE* assembly using the **`addi`** instruction to load these constants into registers **`t1`**, and **`t2`** respectively. This is a temporary solution and is valid for both *UVE* 1 and *UVE* 2 generation.

Summarily, this:

```
%sn = arith.constant 0: i64
%smn1 = arith.constant 1: i64
```

is translated to:

```
addi t1, x0, 0 # %sn = 0
addi t2, x0, 1 # %smn1 = 1
```

Concerning stream declaration and configuration in MLIR:

```
%u1 = srl.new_output_stream %x: stream<f32>
srl.append_dim %u1, %sn, %smn1, %size: void
%u2 = srl.new_input_stream %y: stream<f32>
srl.append_dim %u2, %sn, %smn1, %size: void
%u3 = srl.new_input_stream %x: stream<f32>
srl.append_dim %u3, %sn, %smn1, %size: void
```

In *UVE* 1 assembly, the **`ss.st.w`** and **`ss.ld.w`** instructions declare streams, and **`ss.cfg.vec`** configures them, resulting in:

```
ss.st.w u1, %[x], t2, t3
ss.cfg.vec u1

ss.ld.w u2, %[y], t2, t3
ss.cfg.vec u2

ss.ld.w u3, %[x], t2, t3
ss.cfg.vec u3
```

On the other hand, in *UVE* 2 assembly, the process uses different instructions for stream configuration:

```
    ss.sta.st.w.v u1, %[x]
    ss.end u1, t1, t2, t3

    ss.sta.ld.w.v u2, %[y]
    ss.end u2, t1, t2, t3

    ss.sta.ld.w.v u3, %[x]
    ss.end u3, t1, t2, t3
```

In UVE2, the stream operations use the **ss.sta.st.w.v** and **ss.sta.ld.w.v** instructions for starting the configuration, followed by the **ss.end** instruction to complete the configuration.

The constant to stream MLIR operation is converted to a stream using the **so.v.dp.w** instruction:

```
    so.v.dp.w u4, %[A], p0
```

The **while_present** loop in MLIR is translated to a loop in *UVE* assembly. The loop operations include multiplying streams and adding the results to the output stream. The **so.a.mul.fp** and **so.a.add.fp** instructions perform these operations.

```
    .Loop_0:
    so.a.mul.fp u5, u2, u4, p0
    so.a.add.fp u1, u3, u5, p0
    so.b.nc u1, .Loop_0
```

This final translation to *UVE* assembly demonstrates the compiler's ability to generate code valid code for different versions of *UVE*. The flexibility shown in handling platform-specific constraints for both *UVE* 1 and *UVE* 2 highlights the compiler's adaptability. Furthermore, overall, the successful translation of the SAXPY kernel and the generation of efficient *UVE* assembly code validate the *SRL* framework and its compiler.

4.4 Generating RVV Assembly Proposal

To illustrate the advantage of implementing our SRL approach through MLIR, now is presented an exploratory exercise showing how the *srl* dialect can be lowered to another stream capable target.

Specifically, this example will show how SRL DSL statements can represent concepts existing in other instruction sets, namely the RVV extension. Thus, we illustrate the usefulness of the DSL and dialect at this intermediate lowering stage, by showing a viable path to support multiple backends.

WE will use SAXPY as an example lowering for RVV. In this case, stream configuration is the following:

```
%sn = arith.constant 0: i64
%snm1 = arith.constant 1: i64
%u1 = srl.new_output_stream %x: stream<f32>
srl.append_dim %u1, %sn, %snm1, %size: void
%u2 = srl.new_input_stream %y: stream<f32>
```

```
srl.append_dim %u2, %sn, %snm1, %size: void
%u3 = srl.new_input_stream %x: stream<f32>
srl.append_dim %u3, %sn, %snm1, %size: void
```

This configuration sets up streams, which will be utilised to produce the configuration for the vectors. The RVV assembly code employs specific instructions to manage vector length and data loading for vector operations. The configuration and functions are shown in the RVV code that follows:

```
vsetvli t0, %[size], e32, m8, ta, ma # Set vector length and other parameters
vle32.v v0, (%[x]) # Load vector elements from address %[x] into v0
```

The vector processing is first configured in the RVV assembly code using the instruction `vsetvli t0, %[size], e32, m8, ta, ma`. This determines the parameters for the next vector operations as well as the vector length. Based on the total size given in `%[size]`, the number of elements that may be processed in a single vector instruction is stored in the register `t0`. While `m8` sets the register to operate with a group of eight elements at a time, the `e32` parameter indicates that each element in the vector is a 32-bit floating-point number. The way incomplete vector operations are handled when the data size is not an exact multiple of the vector length is controlled by the flags `ta` (tail agnostic) and `ma` (mask agnostic).

The vector elements are loaded from memory into the vector register `v0` by the instruction `vle32.v v0, (%[x])` once the vector configuration is set. The base address `%[x]` points to the location of the data array, and the instruction loads 32-bit floating-point elements from this address into the vector register for further processing. In RVV, only one register setup is required to configure vector length because the vector architecture operates on a block of contiguous memory as a unified vector. On the other hand, the streaming abstraction necessitates setting up three distinct streams (%u1, %u2, and %u3), each of which will handle input or output independently, for various data flows. Each stream is configured with its own dimensions, including base, stride, and size, which allows for more complex memory access patterns, such as non-contiguous or interleaved data, as opposed to the contiguous memory model of RVV.

Thus, while RVV needs only one setup for handling contiguous data vectors, the streaming model offers more flexibility and precision, which is particularly beneficial when dealing with complex or irregular memory access patterns. This additional versatility comes at the cost of needing multiple streams for configuration.

The construct `while_present(%u1, 0)` in the *srl* dialect denotes a high-level abstraction for managing the stream-based processing loop. The second input, `0`, indicates that it processes until the stream is exhausted. This loop iterates across the stream data, performing calculations as long as elements are present in the stream `%u1`.

This high-level construct is replaced by a conditional branch loop when it is lowered to RVV assembly. For this, the assembly equivalent is as follows:

```
loop_0:
    # computation here
    sub a0, a0, t0 # Decrement the loop counter a0 by vector length t0
```

```
    bnez a0, loop_0 # Branch to loop_0 label if a0 is not zero
```

Register `t0` stores the vector length, or the amount of elements processed every iteration, in RVV assembly, while `a0` serves as the loop counter, tracking the remaining elements.

The loop reduces the number of unprocessed items by decreasing `a0` by `t0` at the end of each iteration. The loop finishes when `a0` hits zero, indicating that all elements have been handled.

The beginning of the loop is indicated by the label `loop_0`. `bnez a0, loop_0` determines whether `a0` is non-zero, and `sub a0, a0, t0` decreases `a0` by the vector length. If it is, the loop continues; otherwise, it terminates.

Regarding the computation inside the while_present loop:

```
    srl.mul %t0, %u2, %u4: void
    srl.add %u1, %u3, %t0: void
```

It is simply translated into:

```
    vfmacc.vf v8, %[A], v0 # Multiply elements in v0 by constant A and accumulate to v8
    vse32.v v8, [%y] # Sum the values in y to v8
```

In the `while_present` loop, the *srl* operations perform a multiplication and an addition between streams, which translates in RVV assembly to a vector multiply-accumulate instruction (`vfmacc.vf`) and a store operation (`vse32.v`). While the store operation puts the result to memory, the multiply-accumulate instruction multiplies vector elements by a constant and adds the result to a target register.

Figure 4.10 shows an illustration of the final lowering suggestion.

```
vsetvli t0, %[size], e32, m8, ta, ma # Set vector length and other parameters
vle32.v v0, (%[x]) # Load vector elements from address %[x] into v0
vse32.v v8, [%y] # Sum the values in y to v8
loop_0:
    vfmacc.vf v8, %[A], v0 # Multiply elements in v0 by constant A and accumulate to v8
    vse32.v v8, [%y] # Sum the values in y to v8
    sub a0, a0, t0 # Decrement the loop counter a0 by vector length t0
    bnez a0, loop_0 # Branch to loop_0 label if a0 is not zero
```

Figure 4.10: Lowering proposal of SAXPY kernel from *srl* dialect to RVV extension instructions.

This proposal highlights the promise of the *srl* dialect as a prospective target for future advancements, by demonstrating its ability to be integrated with other platforms that aim to optimise code execution through vectorisation-like operations and streaming.

It is technically possible to lower the *srl* dialect to RVV, although there are a few obstacles to overcome. The fundamental functions of *srl*, like multiply and addition operations transfer quite well to RVV instructions. While RVV works best with contiguous blocks of memory, *srl* allows for complex stream setups and non-contiguous memory access patterns. Because of this disparity, in order to target RVV, intricate memory access patterns in *srl* might need further transformations to simplify complex access expressions into instructions compatible with RVV.

The lowering process is further complicated by *srl*'s complex control over multiple streams and different data access patterns. While vectorised operations may be handled effectively by RVV, attention is needed when transferring *srl*'s adaptable stream-based methodology to RVV's more homogeneous memory model. Therefore, even if the transition is feasible, making sure the final RVV code is accurate and performant will take an extensive amount of effort.

In addition, translating the *srl* dialect to hardware dialects like Calyx demonstrates the feasibility of generating Hardware Description Language (HDL) capable of implementing the abstractions expressed through *SRL*.

Chapter 5

Conclusions and Future Work

5.1 Conclusion

In the landscape of contemporary computing, characterised by the shifts in Moore's Law and Dennard scaling, this work proposes addressing post-Moore's Law challenges by investing in novel compilation approaches for custom computing systems.

It exploits the Multi-Level Intermediate Representation (MLIR) ecosystem as a crucial element for compilation onto our currently planned targets, i.e. custom hardware generation and streaming-focused extensions.

To achieve our objectives, we developed the syntax, parsers, and required compilation steps for a Domain Specific Language (DSL) type representation of our Structured Representation Language (SRL), as well as a dialect representation, compatible with MLIR framework. Our work included an extensive analysis of current Unlimited Vector Extension (UVE) generation techniques, serving as the basis to implement our own MLIR based approach for code generation, which achieved functionally correct code, versus a legacy Low Level Virtual Machine (LLVM) based compilation flow. Furthermore, we propose future compilation steps for supporting other state-of-the-art targets capable of streaming and vector computations, such as RISC-V Vector Extension (RVV) extension.

In summary, this work tackles contemporary compilation challenges, aiming to provide contributions to the state-of-the-art regarding support for heterogeneous architectures.

5.2 Future Work

In the future, we plan to employ the *srl* dialect as an interface between high-level languages like *C/C++* and *Python*, as well as Machine Learning frameworks and streaming dedicated platforms using MLIR. To do this, we plan to convert other dialects like *vector*, *affine* or *linalg* and even infer streaming opportunities associated with logical and arithmetic operations inside MLIR. This aims to eliminate users' need to be familiar with the native SRL and broaden its applicability.

To sum up, the desired future solution is illustrated in Figure 3.15 for a *C/C++* compiler using the SRL MLIR plugin. In this solution, SRL can be seen as a compiler plugin for those wishing to utilise streaming and vector Single Instruction Multiple Data (SIMD) engines or generate custom hardware for this purpose.

Appendix A

Publications

This section includes revelant publications in the following conferences:

- LATTE'24 - Workshop on Languages, Tools, and Techniques for accelerator Design (LATTE 2024)is a workshop embedded in the Architectural Support for Programming Languages and Operating Systems conference (ASPLOS), a CORE A* conference. April 27, 2024, San Diego, CA, United States of America (https://capra.cs.cornell.edu/latte24/).
- ARC24 - The 20th International Symposium on Applied Reconfigurable Computing (ARC 2024) March 20 - 22, 2024, Aveiro, Portugal (https://arc2024.av.it.pt/).

Multi-Target DSL and MLIR Dialect for Streaming

Manuel Cerqueira da Silva
University of Porto
INESC TEC
Portugal
manuel.m.carvalho@inesctec.pt

Luís Crespo
INESC-ID, Instituto Superior Técnico,
Universidade de Lisboa
Portugal
luis.miguel.crespo@tecnico.ulisboa.pt

Nuno Neves
INESC-ID, Instituto Superior Técnico,
Universidade de Lisboa
Portugal
nuno.neves@inesc-id.pt

Nuno Paulino
INESC TEC / University of Porto
Portugal
nuno.m.paulino@inesctec.pt

João Bispo
University of Porto / INESC TEC
Portugal
jbispo@fe.up.pt

ABSTRACT

With the shift towards custom architectures, there is a growing need for new compilation approaches. Our focus lies in crafting an Multi-Level Intermediate Representation (MLIR) dialect capable of jointly representing streaming and vector processing operations. We introduce our own Domain Specific Language (DSL) intended to bridge the gap between the MLIR layer and code generation suited for streaming-oriented hardware accelerators. The Structural Representation Language (SRL) abstracts streaming and vector concepts, serving as an intermediary step towards generating code containing our proposed dialect from diverse input sources, a task we aim to explore in future work. We present the syntaxes of the SRL DSL and the dialect, demonstrating how they can be leveraged to target general-purpose processors with SIMD coprocessors, as well options like FPGAs and CGRAs.

1 INTRODUCTION

In this work, we propose a data streaming DSL and an equivalent MLIR dialect as an intermediary layer, contributing to code generation suited for hardware accelerators. Data streaming, in this context, refers to a continuous data flow between the memory and the processor, or co-processor. Our focus is on creating a dialect to represent this type of access, together with operations over vector data types (i.e., SIMD-like).

To validate the created abstractions, we will consider: 1) targeting an existing instruction set extension for RISC-V called Unlimited Vector Extension (UVE)[2, 4], which relies on a RISC-V core modified with streaming and vector hardware, and 2) generating HDL for co-processors that implement the expressed stream and vector operations. By decoupling the memory access from computational tasks, streaming enables parallel execution and pre-fetching. This decoupling can lead to improved vectorisation, especially in scenarios where memory access patterns are complex and irregular.

LATTE '24, March 26, 2024, San Diego, CA, United States of America

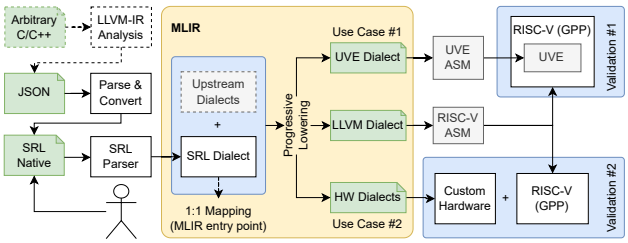

Figure 1: Compilation flow from SRL Native (DSL) to two hardware targets, where the DSL can be provided either by user input, or as a JSON generated from LLVM-IR analysis.

2 Structural Representation Language (SRL)

Our DSL, named SRL, has three distinct representations: one in JSON for easier interface with other tools, another in a textual representation designed for use by human software developers, and an MLIR dialect used for the compilation back-end, for integration with other MLIR dialects in order to generate the final binaries.

The development of the SRL DSL and respective dialect are motivated by the objective of supporting our current primary use case, the UVE assembly, a RISC-V instruction set extension supported by a RISC-V core with a streaming engine capable of SIMD-like operations [2]. To be precise, the SRL is meant to replace the current compiler-based approach for generating UVE code, which relies on analysing LLVM-IR to identify streaming opportunities [4], and generating an executable through a modified linking process.

However, through this approach, the information that is lost at the LLVM-IR level complicates the analysis, leading to under-utilisation of UVE's support for complex access patterns. That is, this flow is a raising step to recover computational streaming abstractions, that should reside at a higher level or mid-level, and then be lowered directly into the streaming-capable hardware.

To address this, we aim to leverage SRL (both DSL and dialect) to preserve this contextual and structural information, including loop structures, at a higher level, supporting more sophisticated compilation onto the SIMD-like hardware which implement UVE, but also enabling the possibility of targeting other back-ends, e.g., HDL generation.

To support this development we are representing the output of the current LLVM-IR analysis through the aforementioned JSON

```
// Dimensions
d1 = Dim(init: x_offset, step: step, length: len);
d2 = Dim(init: y_offset, step: step, length: len);
d3 = Dim(init: output_offset, step: step, length: len);

// Streams
xStream = InputStream<f32>(base: x_address, dims: [d1]);
yStream = InputStream<f32>(base: y_address, dims: [d2]);
outStream = OutputStream<f32>(base: output_address, dims: [d3]);

//Computation iterates until the end of the stream.
whilePresent(xStream.dim[0]) {
    outStream = xStream+yStream*mul_factor;
}
```

Figure 2: Proposed SRL DSL syntax for saxpy function

```
// Stream declaration
%d1 = srl.dim %d1_off, %step, %len : dim_type
%d2 = srl.dim %d2_off, %step, %len : dim_type
%d3 = srl.dim %d3_off, %step, %len : dim_type

%xStream = srl.create_input_stream %x_addr, %d1: stream_type
%yStream = srl.create_input_stream %y_addr, %d2: stream_type
%outStream = srl.create_output_stream %y_addr, %d3: stream_type

// Computation
%control_dim = srl.get_control_dim %xStream : dim

srl.while_present (%control_dim) : (dim_type) -> i32 {
    srl.mult %u0, %yStream, %mult_factor : f32
    srl.add %outStream, %xStream, %u0 : f32
}
```

Figure 3: MLIR Level implementation of the SAXPY function including SRL dialect statements

entry point, benefiting from the already implemented identification of streaming and vector operations performed over state-of-the-art benchmark kernels.

2.1 SRL DSL

We initially defined the SRL DSL syntax based on the outputs of the LLVM-IR analysis in their JSON format, as mentioned. Besides being able to represent the analysis outputs (e.g., stream configurations, computation), the syntax was defined for readability, to be human writable.

The SRL is organised into inputs (live-ins), outputs (live-outs), streams and computation. Inputs and outputs are defined by their identifiers and types (e.g. int, float). Streams also have identifiers and types, and are specified to be either input or output, have a base pointer, and a list of dimensions (ranging from 1D to 8D). The dimensions are determined by an initial offset, a step, and a length. It is also possible to declare dimension modifiers that change the stride or length in runtime, which allows for more complex patterns, such as triangular access patterns. In respect to computation, in addition to common scalar arithmetic and logical operations, SRL also represents vector operations between streams.

Within the JSON representation, the computation segment is represented as a Dataflow Graph, separate from input/output specification, stream initialisation, or control. The control flow operations are implemented through attributes of the streams themselves, e.g, an empty status flag.

To exemplify the use of the DSL, consider the *Single precision A X plus Y (SAXPY)* equation:

$$\text{dest}[i] = \text{src2}[i] + \text{src1}[i] \times \text{value}, \quad \text{for } i = 0 \text{ to size} - 1$$

Figure 2 shows an implementation of this equation using the textual form of the DSL. This example defines two input streams, *xStream* and *yStream*, and an output stream, *outStream*. It then performs the SAXPY operation until the end of the stream, utilising the stream dimension to control the flow. The streaming abstraction removes all explicit memory accesses during computation. The process of continuously fetching data in the configured pattern is now a separate problem and can be tackled concurrently, i.e., by a specialized hardware engine. Every time an input stream variable is read, a datum is consumed and the following read will fetch the next datum. The same logic applies for output streams. The writes or reads to the stream variables are thus agnostic to access latency or stalls, as they block until the operation is possible.

2.2 SRL Dialect

The SRL dialect mirrors our DSL, serving as an entry-point into the MLIR ecosystem. Figure 3 exemplifies our proposal for the SRL dialect, given the features we must represent, and the abstraction layer we have chosen. Regarding the stream declaration, the statement:

```
d1 = Dim(init: x_offset, step: step, length: len);
u1 = InputStream<f32>(base: x_address, dim: [d1]);
```

easily converts into the MLIR representation:

```
%d1 = srl.dim %d1_off, %step, %len : dim_type
%xStream=srl.create_input_stream %x_addr, %d1:stream_type
```

Development and use of the dialect faces certain challenges, particularly in ensuring a valid MLIR representation of both stream types and operations. For instance, stream register assignment and use must be handled differently, since MLIR relies on SSA representation, while stream register names must be preserved despite being read/written, as each stream is bound to a specific hardware registers after being configured.

However, most SRL operations, i.e. arithmetic, logical and loops are straightforwardly integrated into existing MLIR representations (i.e., *arith*). This characteristic positions our dialect as a potential future target for integration with other MLIR dialects, such as *arith*, *scf*, *tensor*, or *vector* [3]). The significant distinction between our dialect and these upstream dialects is its role as an intermediary interface. The novelty lies in attempting to create a MLIR dialect to bridge higher-level dialects down to targets (e.g., the UVE assembly) or specialised hardware (e.g., through use of hardware oriented dialects) capable of streaming and vectorisation.

Furthermore, since upstream dialects can be generated from C/C++ code, placing our SRL dialect at an intermediary stage between these upstream dialects and lower-end dialects or targets will allow for future compilation support starting for high-level code, replacing the current analysis over LLVM-IR and integrating this process of identifying streaming and vector opportunities natively,

```
    ss.ld.w u1, %[src1], %[size], %[stride] # Configures stream data
    ss.cfg.vec u1 # Configure vector registers as a vector dimension
    ss.ld.w u10, %[src2], %[size], %[stride] # Same as u1
    ss.cfg.vec u10
    ss.st.w u3, %[dest], %[size], %[stride]
    ss.cfg.vec u3 # Same as u1
    so.v.dp.w u4, %[value], p0 # Stores 'value' in u4 register
  .L1%=: # Perform a vector addition (u3 = u10 + u1 * u4)
    so.a.mul.fp u5, u1, u4, p0
    so.a.add.fp u3, u10, u5, p0
    so.b.nc u1, .L1%= # Branch to L1 if stream not complete
```

Figure 4: UVE assembly implementation of SAXPY function

into a one-shot compilation flow. That is, the C/C++ input shown in Figure 1 is fed directly into the MLIR ecosystem, bypassing the use of the SRL DSL if desired, although the DSL remains a useful tool for explicit manipulation of data streams and vector operations.

The following sections explain the remainder of the envisioned lowering processes into to our two mentioned targets, UVE assembly (Section 2.3), and Hardware Description Language (HDL) (Section 2.4).

2.3 Lowering to UVE Assembly

The UVE is composed of approximately 80 instructions, operating on a set of 32 configurable vector/streams registers (`u0` to `u31`) and 16 predicate registers (`p0` to `p15`). Computation instructions are predicated via the predicate registers, and so computations only happen on each datum of a vector based on it's respective predicate.

Figure 4 shows the result of lowering the MLIR code in Figure 3 to this assembly. The streams are configured to read (i.e., `u1` and `u10`) or write (`u3`) data from specified addresses, through arguments such as `src1`. The '*create_input_stream*' is replaced with stream loading (`ss.ld.d`) and configuration (`ss.cfg.vec`) instructions. A vector register `u4` holds the same constant floating-point value in all its lanes. The `WhilePresent` loop from the original code is substituted with a conditional branch which checks the stream state, and the loop body contains all computation instructions. While most of the operations were easily translated to UVE instructions, there were some challenges encountered, particularly in dealing with register assignment, due to SSA representation as previously mentioned.

To generate a final executable, this code can be handled as an assembly file and linked against a host `C/C++` application via a function call.

2.4 Lowering to HDL

To generate HDL capable of implementing the abstractions expressed through SRL, the streaming tasks must be translated to modules capable of complex memory access patterns, and compute operations are substituted with their hardware counterparts.

We tested possible paths using such hardware dialects that may enable this translation [1]. For instance, using Calyx [6] and its dialect we can successfully generate HDL (namely, SystemVerilog). This dialect can be generated from upstream dialects, as we intend to do with he SRL dialect. Thus, despite challenges to accurately generating valid fully custom streaming hardware structure descriptions, there is a viable path to translating SRL to Calyx or other hardware dialects. Furthermore, since we aim to support generation of SRL dialect from `C/C++`, this would be akin to implementing flow similar to traditional High-Level-Synthesis (HLS) within MLIR.

Alternatively, some concrete non von Neumann platforms can be considered as possible to targets, such as CGRAs that consume data streams in a manner analogous to UVE [5].

3 CONCLUSIONS AND FUTURE WORK

This work addresses challenges in contemporary computing post-Moore's Law by proposing novel compilation approaches for custom computing systems. It emphasises the importance of the MLIR project for compiling onto planned targets such as custom hardware generation and the UVE for RISC-V. The approach involves defining a syntax and parser for a DSL to represent streaming and vector computations, mapping it onto its respective dialect in the MLIR framework, and lowering it to one of the targets currently considered, chief of which is a RISC-V core with a streaming engine.

The integration of SRL into the middle-end of the compilation process will entail relocating the current analysis conducted on LLVM-IR to this stage by generating a JSON that will produce SRL. This enables the compilation of arbitrary C++ code for the UVE.

Future plans include using the JSON interface for high-level languages and machine learning frameworks, optimising for streaming within the SRL dialect, and expanding its applicability to other SIMD architectures beyond UVE for instance, the RISC-V "V" vector extension (in this case, the dialect *vector* could be a target of SRL lowering). Overall, this work contributes to supporting heterogeneous architectures' compilation challenges.

ACKNOWLEDGMENTS

This work was funded by Fundação para a Ciência e a Tecnologia (FCT), under project 2022.06780.PTDC (DOI: 10.54499/2022.06780.PTDC). The authors also acknowledge the contributions from projects 2022.11626.BD, and UIDB/50021/2020 (DOI: 10.54499/UIDB/50021/2020).

REFERENCES

[1] Manuel Cerqueira da Silva, Luís Sousa, Nuno Paulino, and João Bispo. 2024. A DSL and MLIR Dialect for Streaming and Vectorisation. In *Applied Reconfigurable Computing. Architectures, Tools, and Applications*, Iouliia Skliarova, Piedad Brox Jiménez, Mário Véstias, and Pedro C. Diniz (Eds.). Springer Nature Switzerland, Cham, 181–190.

[2] Joao Mario Domingos, Nuno Neves, Nuno Roma, and Pedro Tomás. 2021. Unlimited Vector Extension with Data Streaming Support. In *Proc. of the 48th Annual ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 209–222. https://doi.org/10.1109/ISCA52012.2021.00025

[3] LLVM MLIR Contributors. Accessed 2024. *MLIR Documentation: Vector Dialect*. https://mlir.llvm.org/docs/Dialects/Vector/ Accessed on: March 8, 2024.

[4] Nuno Neves, Joao Mario Domingos, Nuno Roma, Pedro Tomás, and Gabriel Falcao. 2022. Compiling for Vector Extensions With Stream-Based Specialization. *IEEE Micro* 42, 5 (2022), 49–58. https://doi.org/10.1109/MM.2022.3173405

[5] Nuno Neves, Pedro Tomás, and Nuno Roma. 2020. Reconfigurable Stream-based Tensor Unit with Variable-Precision Posit Arithmetic. In *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 149–156. https://doi.org/10.1109/ASAP49362.2020.00033

[6] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. 2021. A Compiler Infrastructure for Accelerator Generators. In *Proc. of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 804–817. https://doi.org/10.1145/3445814.3446712

A DSL and MLIR Dialect for Streaming and Vectorisation*

Manuel Cerqueira da Silva[1], Luís Sousa[2,1][0000−0002−1925−8939], Nuno Paulino[2,1][0000−0001−5547−0323], and João Bispo[1,2][0000−0002−3017−9449]

[1] Faculty of Engineering, University of Porto, Portugal
[2] INESC TEC, Portugal
up201806391@edu.fe.up.pt, lm.sousa@fe.up.pt, nuno.m.paulino@inesctec.pt, jbispo@fe.up.pt

Abstract. This work addresses the contemporary challenges in computing, caused by the stagnation of Moore's Law and Dennard scaling. The shift towards heterogeneous architectures necessitates innovative compilation strategies, prompting initiatives like the Multi-Level Intermediate Representation (MLIR) project, where progressive code lowering can be achieved through the use of *dialects*. Our work focuses on developing an MLIR dialect capable of representing streaming data accesses to memory, and Single Instruction Multiple Data (SIMD) vector operations. We also propose our own Structured Representation Language (SRL), a Design Specific Language (DSL) to serve as a precursor into the MLIR layer and subsequent inter-operation between new and existing dialects. The SRL exposes the streaming and vector computational concepts to a higher-level, and serves as intermediate step to supporting code generation containing our proposed dialect from arbitrary input code, which we leave as future work. This paper presents the syntaxes of the SRL DSL and of the dialect, and illustrates how we aim to employ them to target both General-Purpose Processors (GPPs) with SIMD co-processors and custom hardware options such as Field-Programmable Gate Arrayss (FPGAs) and Coarse-Grained Re-configurable Arrays (CGRAs).

Keywords: DSL, MLIR, Compilation, Streaming, Vectorisation, SIMD, Heterogeneous Systems

1 Introduction

In the past decade, there has been a noteworthy transformation in the field of computing, marked by the stagnation of Moore's Law and Dennard scaling [13]. This shift has introduced a period where the conventional methods of achieving performance improvements confront unprecedented challenges. The spotlight has turned towards heterogeneous architectures, which encompass diverse processing

* This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT), under project 2022.06780.PTDC (DOI:10.54499/2022.06780.PTDC). We also acknowledge the contributions from FCT grant SFRH/BD/10002/2022.

units and have the potential to leverage hardware specialisation [13]. However, the compilation of applications for such systems, or for application-specific computing engines, poses difficulties when relying on traditional high-level source code. In response, initiatives such as the MLIR project have emerged [6]. Nested within the LLVM ecosystem, MLIR introduces a unified Intermediate Representation (IR), utilising the concept of dialects. These dialects can represent different functionalities or paradigms of data manipulation. The intention is to facilitate generation of code for targets (i.e., hardware platforms or processors) containing heterogeneous components.

This work contributes to this by introducing a streaming DSL and an equivalent MLIR dialect as an intermediary layer for code generation tailored to heterogeneous architectures. In this context, streaming refers to a sequence of data flowing between the memory and the processor. Specifically, we will focus on designing a dialect capable of representing data stream accesses and operations on vector data types (i.e., SIMD-like). To validate the new dialects' abstractions, the compilation targets will include an existing instruction set extension named the Unlimited Vector Extension (UVE) for RISC-V dedicated for streaming and vector operations. Additionally, our validation process will encompass the generation of hardware for supporting these operations [2].

The paper is organised as follows: Section 2 summarises the related work, including existing efforts in introducing into the MLIR ecosystem dialects for hardware generation, and other DSLs for streaming computation; Section 3 presents our proposed DSL and respective MLIR dialect for streaming and vectorisation, and Section 4 concludes the paper.

2 Related Work

Existing approaches for streaming, and other computation paradigms such as vectorisation or fully custom compute, exist at different abstraction levels. Some designs chose to present the unconventional hardware capabilities to the programmer explicitly via DSLs, while in contrast, compiler level approaches delegate the task of identifying streaming opportunities during code generation. We briefly summarise both cases in the following sections.

2.1 DSLs for Streaming Computation

Existing DSLs for streaming (also referred to as dataflow), propose adding specific syntax to high-level languages to expose the computing paradigm. This serves as a wrapper to functionalities implemented by a middle-layer, which may still require the use of dedicated compilers to fully realise.

The *StreamIt* methodology [3] involves the utilisation of various techniques aimed at representing and manipulating abstractions that serve as representations of data streams and operations. This design choice provides the programmer with a notable degree of control over the configuration of the dataflow, enabling the exploration of diverse design variations. The implementation of

StreamIt is carried out in Java, capitalising on the language's mature features. However, it is worth noting, as acknowledged by the authors, that there exists an opportunity for improvement in terms of enhancing the clarity of the syntax employed in the language.

In the MaxCompiler [7] flow for generating FPGA bitstreams, the developer is required to provide a host application in `C/C++`, and kernels and a manager encapsulating those kernels in MaxJ, a Java-based high-level DSL with operator overloading. Kernels delineate the computations chosen for hardware acceleration, while the manager configures their interfaces for interaction with the host application. The *FAST* language [4] serves as an imperative language for dataflow models. It utilizes the MaxComplier as a back-end, but provides support for `C/C++` by introducing a transformation step which generates MaxJ code. The code is transformed based on user supplied source annotations in the form of pragmas. This approach does not require modifying the target `C/C++` code with custom DSL syntax, but is only suitable for targeting FPGAs.

The SARA [16] approach generates code for the Plasticine [12] CGRA by using Spatial [5], a DSL for hardware accelerator specification, as a front-end, whereas SARA proper is the proposed back-end compiler. The use of Spatial as a means of representing compute kernels mandates that developers articulate the kernel code within the confines of the DSL, potentially impacting code portability. Therefore, the approach offers a structured flow for optimisation exploration, albeit with the trade-off of language-specific algorithm expression, and only specifically for CGRAs.

2.2 Compiler Support for Streaming Computation

While previously presented approaches made use of some compiler support, these were custom compilers tailored for the respective DSLs or hardware targets. It was also up to the developer to exploit the features exposed by the DSLs. In contrast, approaches presented in this section identify streaming opportunities during code generation by relying on existing compilation front-ends which produce LLVM-IR code (or MLIR dialects).

The *OXiGen* [11] flow is based on extracting dataflow graphs from LLVM-IR code generated from any front-end language compilable to this representation. The targeted LLVM-IR code is restricted to a user-specified function name. OXiGen then infers streams, albeit with access pattern limitations, from memory accesses performed through any iteration variable in the outermost loop dimension. Vectorization is inserted by modifying input and output data types of the inferred streams. Graphs are finally translated into MaxJ code for use with the MaxCompiler [7], i.e., only FPGAs are targeted.

SODA-OPT and Bambu [1] emerge as a possible option in the domain of high-level compilation and hardware synthesis. These tools harness MLIR to contribute to the generation of hardware accelerators. SODA-OPT utilises MLIR to generate finely-tuned LLVM-IR code optimised for High-Level-Synthesis (HLS) enhancements. Its architecture extracts essential kernels from high-level applications, facilitating the creation of specialised accelerators tailored for specific

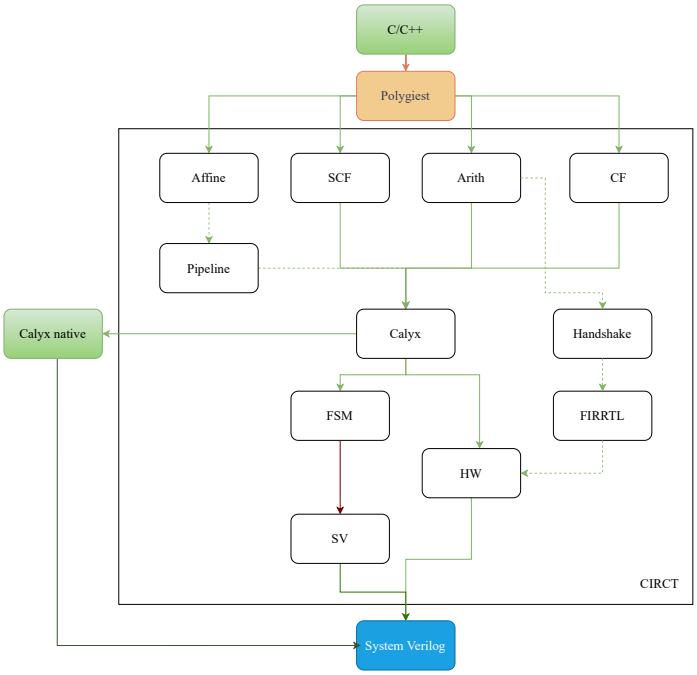

Fig. 1. Subset of current CIRCT ecosystem, focusing on dialects and transformation paths which take `C/C++` as input (complete figure in [14])

tasks. The Bambu HLS tool complements SODA-OPT by consuming the LLVM-IR and producing hardware descriptions for ASICs and FPGAs.

Most similar to our approach is work the Circuit IR Compilers and Tools (CIRCT) project, and the streaming dialect proposed in [15]. The CIRCT project is part of the LLVM ecosystem and extends MLIR's capabilities into the realm of hardware design [14]. Figure 1 shows a `C/C++` focused subset view of CIRCT's work-in-progress for the integration of several representations for hardware generation in the MLIR ecosystem. As a domain-specific compiler infrastructure, CIRCT focuses on optimising and transforming code from various high-level languages into Hardware Description Languages (HDLs) such as SystemVerilog [14]. It exploits the modular and extensible features of MLIR, enabling it to gain advantages in expressing hardware designs in a structured and composable fashion, by resorting dialects specifically designed for hardware description. These dialects encapsulate the semantics of hardware constructs and facilitate the transformation of high-level designs into code suitable for synthesis.

In the recent work presented in [15], the authors implemented a streaming abstraction as an MLIR dialect. While the dialect expresses dataflow patterns (e.g., mapping of input streams into output streams), the computation performed over stream items is expressed in existing dialects like *arith*. By defining a generic

transformation of a data stream into CIRCT's *handshake* dialect, the flow reutilises other existing core dialects to generate HDL. Specifically, each operation applied over a stream item is mapped to a *handshake* operation, which facilitates the incorporation of transformations to generate scheduled pipelines exploiting data parallelism and iteration overlapping. The approach was validated by generating circuits for a small set of synthetic cases, for a setup consisting of a host CPU and PCIe-based FPGA.

3 DSL and MLIR Dialect for Streaming and Vectorization

Our work aims to develop components in the MLIR ecosystem to achieve support towards architectures capable of exploiting fast memory access, via streaming, and data parallelism, via vectorization. To achieve this, we propose defining a syntax and parser for a text-based Intermediate Representation (IR), capable of representing structural data-flow information, and to use this DSL as an easily mappable entry point into a new MLIR dialect. We believe that this will be a viable path for emission of code onto different streaming-capable architectures. Namely, onto GPPs with streaming extensions, and onto reconfigurable hardware, such as FPGAs or CGRAs, as shown in the compilation flow in Figure 2.

Relative to the state-of-the-art, existing CIRCT dialects focus on custom HDL generation from high-level languages (e.g., generating function accelerators), and no layer exists to represent this kind of computing model prior to committing to a particular architecture. Instead, our approach is most similar to [15], where a stream dialect is proposed. However, we aim to also include vectorisation over both primitive and structured data types, as well as demonstrating support for multiple targets beyond FPGAs. The novel components are the SRL DSL (*SRL Native*), the mapping of this input into the SRL dialect, and the lowering steps onto compatible targets.

We propose the SRL DSL in order to easily expose the features of the SRL dialect to a higher-level, suitable to be written by a human developer, similar to

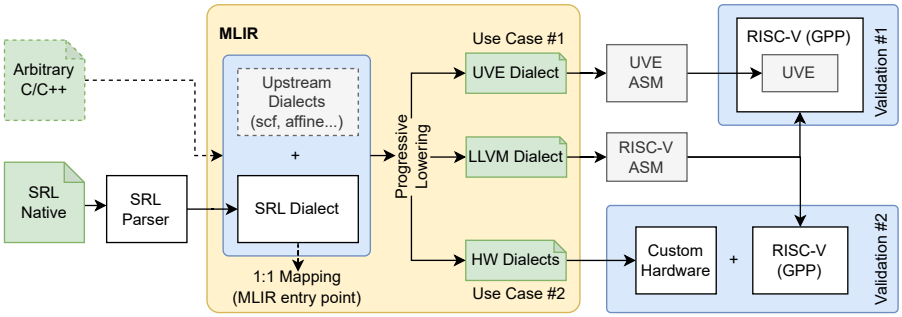

Fig. 2. Compilation flow from SRL Native (i.e., the SRL DSL) to two hardware targets. Support for SRL Dialect generation from standard source code is planned future work.

existing approaches. Simultaneously, the DSL is a first entry point into the MLIR eco-system to allow easy generation of the SRL at this stage. The entry point from `C/C++` will be done via Polygeist, or from LLVM-IR produced by Clang. Further transformation work will then be required to derive the same information the native SRL DSL expressiveness allows. Alternatively, some minor *pragma* based annotations in the `C/C++` could be used as hints.

The SRL Dialect is then lowered to a compatible target. In our work, we will first consider targeting the UVE RISC-V extension [2,9]. This will also require defining a UVE Dialect. In this case, the streaming and vector features are integrated into a single processor, but we will also consider targeting architectures where the specialised computing is exploited by a co-processor managed by an baseline RISC-V. Depending on the input SRL, this co-processor will be realised by either exploiting the viable paths identified in the CIRCT ecosystem to generate HDL (e.g., generating SystemVerilog via the *HW* dialect, or the Calyx IR [10]. See Figure 1 below). This co-processor HDL, coupled to a soft-core RISC-V, can then be deployed to FPGA targets. Alternatively, CGRA overlays coupled to the same RISC-V can be targeted by modifying the lowering process as required.

3.1 SRL DSL

The SRL DSL allows for expression of computational loops where the operands are data streams, and where the data types can be vector types. To illustrate the application of the proposed DSL, consider an example of the *Single precision A X plus Y (SAXPY)* kernel, represented by the equation:

$$\text{dest}[i] = \text{src2}[i] + \text{src1}[i] \times \text{value}, \quad \text{for } i = 0 \text{ to } \text{size} - 1$$

The provided code snippet, representing this equation, delineates the structure of the SRL, highlighting how it captures streaming-specific optimisations:

```
func saxpy(src1, src2, dest, size, stride, value) {
    u1 = Stream<f16>(src1, stride, size);  // Stream declarations
    u10 = Stream<f16>(src2, stride, size); // (with f16 vector data type)
    u3 = Stream<f16>(dest, stride, size);
    float u4 = value;
    // Computation
    for (int i = 0, i < size; i++) {
        f16 u5 = u1.load(i) * u4;
        u3.store(i, u10.load(i) + u5);
    }
}
```

Listing 1.1. Proposed SRL DSL syntax for saxpy function

This SRL example starts with the declaration of three 16 bits floating-point streams, namely *u1*, *u3*, and *u10*. Additionally, a scalar floating-point variable *u4* is initialised with the value of *value*. The subsequent computation involves a loop iterating over the range of *size*. Within each iteration, *u5* is calculated as the product of the element at index i in *u1* and the scalar *u4*. This result is then utilised to update the element at index i in *u3* by adding it to the corresponding element fetched from *u10*.

3.2 SRL Dialect

We intend to lower the SRL dialect into two types of targets, so we needed to determine the required computational features, at which abstraction level the dialect should be placed, and determine what existing lowering paths that can be re-utilised given this placement.

Since UVE assembly is the most concrete architectural target, the currently proposed dialect semantics and constructs are designed to facilitate this lowering. To support this target, RISC-V assembly with UVE instructions has to be generated, so the dialect must be placed at an abstraction level near, or above, the standard upstream dialects (as shown in Figure 2), in order to still allow for generation of LLVM-IR dialect using conventional compilation.

To find a path from this layer onto HDL for the second validation path, we tested possible paths between hardware related dialects in CIRCT, which is the subset previously shown in Figure 1. We limit our scope to `C/C++`, so we tested transformation paths starting from the Polygeist compiler [8], the current `C/C++` entry point into MLIR. Paths shown in green represent successful transformations, dotted paths are reported as possible, but no examples were available for testing, while the path between *FSM* and *SV* (dialect for the SystemVerilog syntax) failed, even for the provided example cases. We determined that the Calyx dialect supports processing of code containing the established dialects *affine*, *sfc*, *arith*, and *cf*. Thus if we place SRL dialect above this level, and transform it into code relying on these dialects, there is a viable path to generate HDL code implementing the streaming and vector operations, fulfilling our requirement.

The following code snippet exemplifies our proposal for the SRL dialect, given the features we must represent, and the abstraction layer we have chosen.

```
func @saxpy(%src1: memref<? x f16>, %src2: memref<? x f16>,
    %dest: memref<? x f16>, %size: index, %stride: index, %value: f16) {
    ; Stream declaration
    %u1 = srl.create_float_stream %src1, %stride, %size:
        memref<? x f16>, index, index -> stream<? x f16>
    %u3 = srl.create_float_stream %dest, %stride, %size:
        memref<? x f16>, index, index -> float16_stream
    %u10 = srl.create_float_stream %src2, %stride, %size:
        memref<? x f16>, index, index -> float16_stream
    %u4 = srl.constant %value : f16
    ; Computation
    %zero = srl.constant 0 : index
    %one = srl.constant 1 : index
    ; Define the loop
    srl.for %i = %zero to %size step %one {
        %u5 = srl.load %u1 index: %i
        %u5 = srl.mul %u5, %u4 : f16
        %u10_i = srl.load %u10 index: %i
        %u5 = srl.add %u5, %u10_i : f16
        srl.store %u3, %u5 index: %i
    }
    return
}
```

Listing 1.2. MLIR Level implementation of the SAXPY function including SRL dialect statements

This dialect functions as a near 1:1 representation of our DSL, serving as a means to integrate our DSL into the MLIR environment. This creates a pathway for diverse lowering pathways for our streaming and vectorisation DSL by leveraging MLIR.

Regarding the stream declaration, the statement "`u1 = stream<f16>(src1, stride, size);`" transitions smoothly into the MLIR representation: `%u1 = srl.create_float_stream %src1, %stride, %size: memref<?xf16>, index, index -> stream<?xf16>`. Operations like load or multiplication are straightforwardly integrated into the MLIR representation. This code exemplifies the intrinsic streaming computation pattern within the SRL dialect, mirroring the characteristics of SRL itself. This characteristic positions it as a potential future target for integration with other sources focused on optimising code execution through streaming and vectorisation operations.

3.3 Lowering Example into UVE Assembly

The particular streaming engine we will target in this work implements the RISC-V UVE instruction set extension as its interface [2]. This framework emerges as an innovative solution tailored to address challenges inherent in SIMD extensions, specifically those optimised for fixed-size registers, such as ARM Scalable Vector Extension (SVE) and RISC-V Vector (RVV). Within the streaming paradigm employed by UVE, a direct streaming mechanism facilitates the seamless flow of input directly into the register file, eliminating explicit memory accesses and their overhead.

While work exists on a compiler based approach to generate UVE code [9], it relies on analysing LLVM-IR to determine streaming opportunities (e.g., identifying iterators and bounds), and then regenerating a modified executable with UVE assembly. We posit that at LLVM-IR level the information loss complicates the analysis, thus under-utilising UVE's support for complex access patterns. We aim to use SRL (both DSL and dialect) to retain this contextual and structural (e.g., loop structure) information at a higher-level. Consider the example of the following translation of the code in Listing 1.2 into UVE assembly:

```
1   ss.ld.d u1, %[src1], %[size], %[stride] #Loads stream data to register
2   ss.cfg.vec u1 #Configure vector registers as a vector dimension
3   ss.ld.d u10, %[src2], %[size], %[stride] #Same as u1
4   ss.cfg.vec u10
5   ss.st.d u3, %[dest], %[size], %[stride]
6   ss.cfg.vec u3 #Same as u1
7   so.v.dp.d u4, %[value], p0 #loads to u4 the 'value'
8 .L1%=: #Perform a vector addition between u10 and u5, store the result in u3
9   so.a.mul.fp u5, u1, u4, p0
10  so.a.add.fp u3, u10, u5, p0
11  so.b.nc u1, .L1%= #Branch to L1 if stream not complete
```

Listing 1.3. UVE assembly implementarion of SAXPY function

In the SRL dialect, as in the UVE assembly, the code follows the same sequence: it loads data from the source (`u1` and `u10`), configures vectors, stores data to the destination (`u3`), and executes the dot product operations, multiplication, and addition in a loop. The 'create_float_stream' operation was

replaced with the stream load (ss.ld.d) and configuration (ss.cfg.vec) operations. The computation operations were substituted with equivalent UVE instructions, eliminating the need for implicit loads and stores present in the MLIR dialect. It is important to note that these passes or conversions from SRL dialect to UVE are proposals and are yet to be implemented.

4 Conclusions and Future Work

In the landscape of contemporary computing, characterised by the shifts in Moore's Law and Dennard scaling, this work proposes addressing post-Moore's Law challenges, by investing in novel compilation approaches for custom computing systems. It focuses on the MLIR project as a crucial element for compilation onto our currently planned targets, i.e., custom hardware generation, and the Unlimited Vector Extension (UVE) for the RISC-V.

To achieve our objectives, we presented an approach based on defining a syntax and parser for a Structured Representation Language (SRL), mapping it to the MLIR framework, and exploring pathways for efficient integration into existing dialects. In the future, we plan to employ the SRL dialect as an interface for high-level languages like `C/C++` and Python, as well as Machine Learning frameworks. This aims to eliminate the necessity for users to be familiar with the native SRL. Lastly, we intended to implement optimisations specifically for streaming within the SRL dialect, and broaden the applicability of the SRL dialect to target other SIMD architectures beyond UVE.

In summary, this work tackles contemporary compilation challenges, aiming to provide contributions to the state-of-the-art regarding support for heterogeneous architectures.

Disclosure of Interests The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Agostini, N.B., Curzel, S., Amatya, V., Tan, C., Minutoli, M., Castellana, V.G., Manzano, J., Kaeli, D., Tumeo, A.: An MLIR-based Compiler Flow for System-Level Design and Hardware Acceleration. In: Proc. of the 41st IEEE/ACM International Conference on Computer-Aided Design. pp. 1–9 (2022)
2. Domingos, J.M., Neves, N., Roma, N., Tomás, P.: Unlimited Vector Extension with Data Streaming Support. In: Proc. of the 48th Annual ACM/IEEE International Symposium on Computer Architecture (ISCA). pp. 209–222 (2021). `https://doi.org/10.1109/ISCA52012.2021.00025`
3. Gordon, M.I., Thies, W., Karczmarek, M., Lin, J., Meli, A.S., Lamb, A.A., Leger, C., Wong, J., Hoffmann, H., Maze, D., Amarasinghe, S.: A Stream Compiler for Communication-Exposed Architectures. In: Proc. of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems. p. 291–303. Association for Computing Machinery (2002). `https://doi.org/10.1145/605397.605428`

4. Grigoras, P., Niu, X., Coutinho, J.G.F., Luk, W., Bower, J., Pell, O.: Aspect driven compilation for dataflow designs. In: Proc. of the 24th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP). pp. 18–25. IEEE (jun 2013). https://doi.org/10.1109/ASAP.2013.6567545, http://ieeexplore.ieee.org/document/6567545/
5. Koeplinger, D., Feldman, M., Prabhakar, R., Zhang, Y., Hadjis, S., Fiszel, R., Zhao, T., Nardi, L., Pedram, A., Kozyrakis, C., et al.: Spatial: A language and compiler for application accelerators. In: Proc. of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 296–311 (2018)
6. Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., Zinenko, O.: MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In: Proc. of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO). pp. 2–14 (2021). https://doi.org/10.1109/CGO51591.2021.9370308
7. Maxeler Technologies: MaxCompiler White Paper (2011), https://www.maxeler.com/media/documents/MaxelerWhitePaperMaxCompiler.pdf, accessed July 4, 2024
8. Moses, W.S., Chelini, L., Zhao, R., Zinenko, O.: Polygeist: Raising C to Polyhedral MLIR. In: 30th International Conference on Parallel Architectures and Compilation Techniques (PACT). pp. 45–59 (2021). https://doi.org/10.1109/PACT52795.2021.00011
9. Neves, N., Domingos, J.M., Roma, N., Tomás, P., Falcao, G.: Compiling for Vector Extensions With Stream-Based Specialization. IEEE Micro **42**(5), 49–58 (2022). https://doi.org/10.1109/MM.2022.3173405
10. Nigam, R., Thomas, S., Li, Z., Sampson, A.: A Compiler Infrastructure for Accelerator Generators. In: Proc. of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. p. 804–817 (2021). https://doi.org/10.1145/3445814.3446712
11. Peverelli, F., Rabozzi, M., Del Sozzo, E., Santambrogio, M.D.: OXiGen: A tool for automatic acceleration of c functions into dataflow FPGA-based kernels. In: Proc. of the 32nd IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). pp. 91–98. Institute of Electrical and Electronics Engineers Inc. (aug 2018). https://doi.org/10.1109/IPDPSW.2018.00023
12. Prabhakar, R., Zhang, Y., Koeplinger, D., Feldman, M., Zhao, T., Hadjis, S., Pedram, A., Kozyrakis, C., Olukotun, K.: Plasticine: A reconfigurable architecture for parallel patterns. In: Proc. of the 44th Annual ACM/IEEE International Symposium on Computer Architecture (ISCA). pp. 389–402 (2017). https://doi.org/10.1145/3079856.3080256
13. Shalf, J.: The future of computing beyond Moore's Law. Philosophical Transactions of the Royal Society A **378**(2166), 20190061 (2020)
14. The LLVM Project: CIRCT - Circuit IR Compilers and Tools (2020), https://circt.llvm.org/, accessed July 4, 2024
15. Ulmann, C.: Multi-Level Rewriting for Stream Processing to RTL compilation. Master thesis, ETH Zurich, Zurich (2022). https://doi.org/10.3929/ethz-b-000578713
16. Zhang, Y., Zhang, N., Zhao, T., Vilim, M., Shahbaz, M., Olukotun, K.: SARA: Scaling a Reconfigurable Dataflow Accelerator. In: Proc. of the 48th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA). pp. 1041–1054 (2021). https://doi.org/10.1109/ISCA52012.2021.00085

Appendix B

Code Translations

This appendix explores the process of translating code through various representations, from high-level procedural languages to intermediate and target-specific formats. The pipeline is demonstrated using examples of the Single Precision A X Plus Y (SAXPY), Triangular Solve (TRISOLV), 3 Matrix Multiplication (3MM), and General Matrix Multiplication (GEMM) kernels.

The process begins with C code, which is converted into JavaScript Object Notation (JSON) format by an analysis tool. This JSON format is then transformed into a Domain Specific Language (DSL) for higher-level abstraction. The DSL is further translated into the Multi-Level Intermediate Representation (MLIR), which supports complex optimisations. Finally, MLIR is converted into target formats, Unlimited Vector Extension (UVE) 1 and UVE 2. UVE 1 has been validated for correctness, while UVE 2 is awaiting further validation, due to the lack of an Instruction Set Simulator (ISS) for it.

Each section details the translation of different kernels through these stages, providing insights into the process and highlighting areas for potential improvement.

B.1 Single Precision A X Plus Y

Figure B.1 illustrates the original C kernel of SAXPY, showing the straightforward implementation of the operation in a procedural programming language.

Figure B.2 presents the result of the analysis tool, which converts the C code into a JSON format. This representation captures the structure and operations of the SAXPY kernel in a machine-readable format, facilitating subsequent transformations.

Figure B.3 illustrates the conversion of the JSON representation into a DSL. The DSL format provides a higher-level abstraction of the SAXPY operation, which can be further manipulated and optimised for various execution environments.

Following the DSL representation, Figure B.4 demonstrates the transformation of the SAXPY operation into the MLIR. MLIR is a versatile intermediate representation designed to support complex optimisations and transformations in compiler pipelines.

```c
void saxpy(float dest[], float src[], float A, int size) {

    for (int i = 0; i < n; i++) {
        dest[i] += src[i] * A;
    }
}
```

Figure B.1: SAXPY kernel C source code.

Finally, Figure B.5 and Figure B.6 show the conversions of the MLIR representation into the UVE 1 and UVE 2 formats, respectively.

B.2 Triangular Solve

Appendix B.2 illustrates the original C kernel for the TRISOLV algorithm, demonstrating the straightforward implementation of the operation in a procedural programming language.

Figure B.7 presents the result of the analysis tool, which converts the C code into a JSON format. This JSON representation captures the structure and operations of the TRISOLV kernel in a machine-readable format, facilitating subsequent transformations.

Figure B.8 shows the conversion of the JSON representation into a DSL. The DSL format provides a higher-level abstraction of the TRISOLV operation, enabling further manipulation and optimization for various execution environments.

Following the DSL representation, Figure B.9 demonstrates the transformation of the TRISOLV operation into the MLIR. MLIR is a versatile intermediate representation designed to support complex optimizations and transformations in compiler pipelines.

Finally, Figure B.10 and Figure B.11 illustrate the conversions of the MLIR representation into the UVE 1 and UVE 2 formats, respectively.

B.3 3 Matrix Multiplication

Figure B.12 provides the original C kernel for the 3MM (3D Matrix Multiply) algorithm, showcasing its implementation in a procedural programming language.

Figure B.13 presents the analysis tool output, which translates the C code for 3MM into a JSON format. This format captures the structure and operations of the 3MM kernel, making it suitable for further transformations.

Figure B.14 illustrates the conversion from the JSON format into a DSL. The DSL representation abstracts the 3MM operation, facilitating further optimisation and manipulation for various execution contexts.

Following the DSL representation, Figure B.15 demonstrates the transformation of the 3MM operation into MLIR.

Finally, Figure B.16 and Figure B.17 show the conversions of the MLIR representation into UVE 1 and UVE 2 formats.

```
{
  "functionName": "saxpy",
  "program": ["loop::1"],
  "inputs": [
    {"datatype": "ptr", "name": "x"},
    {"datatype": "ptr", "name": "y"},
    {"datatype": "f32", "name": "A"},
    {"datatype": "i64", "name": "size"},
    {"datatype": "i64", "name": "sn", "value": "0"},
    {"datatype": "i64", "name": "smn1", "value": "1"}
  ],
  "streams": [
    {
      "name": "u1", "datatype": "f32", "type": "OutputStream", "base": "x",
      "dims": [{"init": "sn", "stride": "smn1", "nElements": "size"}]
    },
    {
      "name": "u2", "base": "y", "datatype": "f32", "type": "InputStream",
      "dims": [{"init": "sn", "stride": "smn1", "nElements": "size"}]
    },
    {
      "name": "u3", "base": "x", "datatype": "f32", "type": "InputStream",
      "dims": [{"init": "zero", "stride": "stride", "nElements": "size"}]
    }
  ],
  "controlFlows": [
    {
      "begin": ["node::0", "node::1"],
      "condition": "whilePresent::u1::0",
      "end": [], "inner_cfg": [], "tag": "loop::1"
    }
  ],
  "dataFlows": [
    {
      "node::0": {"args": ["u4"], "children": ["node::01"], "type": "="},
      "node::01": {"args": ["A"], "children": [], "type": "constantToStream"}
    },
    {
      "node::1": {"args": ["u1"], "children": ["node::2"], "type": "="},
      "node::2": {"children": ["node::3", "node::4"], "type": "+"},
      "node::3": {"children": [], "args": ["u3"], "type": "STREAM::load"},
      "node::4": {"children": ["node::5", "node::6"], "type": "*"},
      "node::5": {"children": [], "args": ["u2"], "type": "STREAM::load"},
      "node::6": {"children": [], "args": ["u4"], "type": "INPUT::value"}
    }
  ]
}
```

Figure B.2: JSON generated by the Low Level Virtual Machine (LLVM)-Intermediate Representation (IR) analysis (SAXPY kernel).

```
1
2 void saxpy(ptr x, ptr y, f32 A, i64 size)
3 {
4     i64 sn = 0;
5     i64 snm1 = 1;
6
7     OutputStream<f32> u1(x);
8     u1.appendDim(sn, snm1, size);
9
10    InputStream<f32> u2(y);
11    u2.appendDim(sn, snm1, size);
12
13    InputStream<f32> u3(x);
14    u3.appendDim(sn, snm1, size);
15
16    AuxStream<f32> u4 = constantToStream(A);
17    whilePresent(u1, 0)
18    {
19        u1 = u3 + u2 * u4;
20    }
21 }
```

Figure B.3: Translation result from JSON to SRL (SAXPY kernel).

```
1     func.func @saxpy(%x: ptr, %y: ptr, %A: f32, size: i64) -> (void) {
2     %sn = arith.constant 0: i64
3     %snm1 = arith.constant 1: i64
4     %u1 = srl.new_output_stream %x: stream<f32>
5     srl.append_dim %u1, %sn, %snm1, %size: void
6     %u2 = srl.new_input_stream %y: stream<f32>
7     srl.append_dim %u2, %sn, %snm1, %size: void
8     %u3 = srl.new_input_stream %x: stream<f32>
9     srl.append_dim %u3, %sn, %snm1, %size: void
10    %t0 = srl.new_aux_stream: stream<f32>
11    %u4 = srl.constant_to_stream %A: stream<f32>
12    srl.while_present(%u1, 0) {
13        srl.mul %t0, %u2, %u4: void
14        srl.add %u1, %u3, %t0: void
15    }
16 }
```

Figure B.4: Translation result from DSL to MLIR dialect (SAXPY kernel).

```
1     addi t1, x0, 0 # %sn = 0
2     addi t3, x0, 1 # %snm1 = 1
3
4     ss.st.w u1, %[x], %[size], t3
5     ss.cfg.vec u1
6     ss.ld.w u2, %[y], %[size], t3
7     ss.cfg.vec u2
8     ss.ld.w u3, %[x], %[size], t3
9     ss.cfg.vec u3
10
11    so.v.dp.w u4, %[A], p0
12
13    .Loop_0:
14    so.a.mul.fp u5, u2, u4, p0
15    so.a.add.fp u1, u3, u5, p0
16    so.b.nc u1, .Loop_0
```

Figure B.5: Translation result from MLIR dialect to UVE 1 (SAXPY kernel).

```
addi t1, x0, 3 # %snm1 = 3
addi t2, x0, 3 # %snp1 = 3
addi t3, x0, 3 # %sn = 3
addi t4, x0, 1 # %one = 1
addi t5, x0, 0 # %zero = 0
ss.sta.ld.w.v.1 u1, %[x]
ss.app u1, t5, t3, t3
ss.app.mod.siz.inc.1 u1, t4
ss.end u1, t5, t5, t4
ss.cfg.vec u1

ss.sta.ld.w.v.1 u2, %[z]
ss.app u2, t5, t3, t5
ss.app.mod.siz.inc.1 u2, t4
ss.end u2, t5, t5, t4
ss.cfg.vec u2

ss.ld.w u3, %[y]

ss.ld.w u4, %[x]

ss.st.w u5, %[z]

.Loop_0:
so.v.dp.w u6, t5, p0

.Loop_1:
so.a.mul.fp u7, u1, u2, p0
so.a.sub.fp u6, u6, u7, p0

so.b.ndc.1 u1, .Loop_1
so.a.adde.fp u7, u6, p0
so.a.add.fp u7, u3, u7, p0
so.a.div.fp u5, u7, u4, p0

so.b.nc u1, .Loop_0
```

Figure B.6: Translation result from MLIR dialect to UVE 2 (SAXPY kernel).

```c
#define SIZE 5

void trisolv(float *L, float *b, float *x) {

    for (int i = 0; i < SIZE; ++i) {
        x[i] = b[i];
        for (int j = 0; j < i; ++j) {
            x[i] -= L[i * SIZE + j] * x[j];
        }
        x[i] = x[i] / L[i * SIZE + i];
    }
}
```

B.4 General Matrix Multiplication

Figure B.18 shows the original C kernel for the GEMM operation, presenting a direct implementation in a procedural programming language.

Figure B.19 depicts the result of the analysis tool, which converts the GEMM C code into a JSON format. This JSON representation encapsulates the structure and operations of the GEMM kernel in a machine-readable format, aiding in subsequent transformations.

Figure B.20 illustrates the conversion of the JSON representation into a DSL. This DSL format abstracts the GEMM operation, allowing for further manipulation and optimisation tailored to different execution environments.

Subsequently, Figure B.21 shows the transformation of the GEMM operation into the MLIR. MLIR supports advanced optimisations and transformations in compiler pipelines.

Finally, Figure B.22 and Figure B.23 display the conversions of the MLIR representation into the UVE 1 and UVE 2 formats, respectively.

```
{
  "functionName":"trisolv",
  "program":["loop::1"],
  "inputs":[
    {"datatype":"ptr","name":"x"},
    {"datatype":"ptr","name":"y"},
    {"datatype":"ptr","name":"z"},
    {"datatype":"i64","name":"snm1","value":"3"},
    {"datatype":"i64","name":"snp1","value":"3"},
    {"datatype":"i64","name":"sn","value":"3"},
    {"datatype":"i64","name":"one","value":"1"},
    {"datatype":"i64","name":"zero","value":"0"}
  ],
  "streams":[
    {"base":"x","datatype":"f32","dims":[
      {"init":"zero","nElements":"sn","stride":"sn"},
      {"init":"zero","stride":"one","nElements":"zero"}
    ],"name":"u1","type":"InputStream","mods":[
      {"type":"N_ELEMENTS","behaviour":"inc","ammount":"one","dimAssociated":"zero",
      "dimTarget":"snm1"}
    ],"scalar":false},
    {"name":"u2","base":"z","datatype":"f32","dims":[
      {"init":"zero","stride":"zero","nElements":"sn"},
      {"init":"zero","stride":"one","nElements":"zero"}
    ],"mods":[
      {"type":"N_ELEMENTS","behaviour":"inc","ammount":"one","dimAssociated":"zero",
      "dimTarget":"snm1"}
    ],"type":"InputStream"},
    {"name":"u3","base":"y","datatype":"f32","dims":[
      {"init":"zero","stride":"one","nElements":"sn"}
    ],"type":"InputStream","scalar":true},
    {"base":"x","datatype":"f32","dims":[
      {"init":"zero","stride":"snp1","nElements":"sn"}
    ],"name":"u4","type":"InputStream","scalar":true},
    {"name":"u5","base":"z","datatype":"f32","dims":[
      {"init":"zero","stride":"one","nElements":"sn"}
    ],"type":"OutputStream","scalar":true}
  ],
  "controlFlows":[
    {"condition":"whilePresent::u1::0","begin":["node::0"],
      "innerControlFlow":["loop::2"],"end":["node::4","node::5","node::6"],"tag":"loop::1"},
    {"begin":["node::1","node::3"],"condition":"whilePresent::u1::1",
      "end":[],"innerControlFlow":[],"tag":"loop::2"}
  ],
  "dataFlows":[
    {"node::0":{"args":["u6"],"children":["node::01"],"type":"="},
      "node::01":{"args":["zero"],"children":[],"type":"constantToStream"}},
    {"node::1":{"args":["u7"],"children":["node::2"],"type":"="},
      "node::2":{"args":[],"children":["node::21","node::22"],"type":"*"},
      "node::21":{"children":[],"args":["u1"],"type":"STREAM::load"},
      "node::22":{"children":[],"args":["u2"],"type":"STREAM::load"}},
    {"node::3":{"children":["node::30"],"args":["u6"],"type":"="},
      "node::30":{"children":["node::31","node::32"],"args":[],"type":"-"},
      "node::31":{"children":[],"args":["u6"],"type":"STREAM::load"},
      "node::32":{"children":[],"args":["u7"],"type":"STREAM::load"}},
    {"node::4":{"children":["node::31","node::32"],"args":[],"type":"reduce"},
      "node::31":{"children":[],"args":["u7"],"type":"STREAM::load"},
      "node::32":{"children":[],"args":["u6"],"type":"STREAM::load"}},
    {"node::5":{"children":["node::30"],"args":["u7"],"type":"="},
      "node::30":{"children":["node::31","node::32"],"args":[],"type":"+"},
      "node::31":{"children":[],"args":["u3"],"type":"STREAM::load"},
      "node::32":{"children":[],"args":["u7"],"type":"STREAM::load"}},
    {"node::6":{"children":["node::30"],"args":["u5"],"type":"="},
      "node::30":{"children":["node::31","node::32"],"args":[],"type":"/"},
      "node::31":{"children":[],"args":["u7"],"type":"STREAM::load"},
      "node::32":{"children":[],"args":["u4"],"type":"STREAM::load"}}
  ]
}
```


Figure B.7: JSON generated by the LLVM-IR analysis (TRISOLV kernel).

```
    void trisolv(ptr x, ptr y, ptr z)
{
    i64 snm1 = 3;
    i64 snp1 = 3;
    i64 sn = 3;
    i64 one = 1;
    i64 zero = 0;

    InputStream<f32> u1(x);
    u1.appendDim(zero, sn, sn);
    u1.appendDim(zero, one, zero);
    u1.appendSizeMod(inc, one, zero, snm1);

    InputStream<f32> u2(z);
    u2.appendDim(zero, zero, sn);
    u2.appendDim(zero, one, zero);
    u2.appendSizeMod(inc, one, zero, snm1);

    InputStream<f32> u3(y);
    u3.appendDim(zero, one, sn);
    u3.setScalar();

    InputStream<f32> u4(x);
    u4.appendDim(zero, snp1, sn);
    u4.setScalar();

    OutputStream<f32> u5(z);
    u5.appendDim(zero, one, sn);
    u5.setScalar();

    whilePresent(u1, 0)
    {
        AuxStream<f32> u6 = constantToStream(zero);
        whilePresent(u1, 1)
        {
            AuxStream<f32> u7 = u1 * u2;
            u6 = u6 - u7;
        }
        reduce(u7, u6);
        u7 = u3 + u7;
        u5 = u7 / u4;
    }
}
```

Figure B.8: Translation result from JSON to SRL (TRISOLV kernel).

```
func.func @trisolv(%x: ptr, %y: ptr, %z: ptr) -> (void) {
    %snm1 = arith.constant 3: i64
    %snp1 = arith.constant 3: i64
    %sn = arith.constant 3: i64
    %one = arith.constant 1: i64
    %zero = arith.constant 0: i64
    %u1 = srl.new_input_stream %x: stream<f32>
    srl.append_dim %u1, %zero, %sn, %sn: void
    srl.append_dim %u1, %zero, %one, %zero: void
    srl.append_size_mod inc, %u1, %one, %zero, %snm1: void
    %u2 = srl.new_input_stream %z: stream<f32>
    srl.append_dim %u2, %zero, %zero, %sn: void
    srl.append_dim %u2, %zero, %one, %zero: void
    srl.append_size_mod inc, %u2, %one, %zero, %snm1: void
    %u3 = srl.new_input_stream %y: stream<f32>
    srl.append_dim %u3, %zero, %one, %sn: void
    srl.set_scalar %u3: void
    %u4 = srl.new_input_stream %x: stream<f32>
    srl.append_dim %u4, %zero, %snp1, %sn: void
    srl.set_scalar %u4: void
    %u5 = srl.new_output_stream %z: stream<f32>
    srl.append_dim %u5, %zero, %one, %sn: void
    srl.set_scalar %u5: void
    srl.while_present(%u1, 0) {
        %u6 = srl.constant_to_stream %zero: stream<f32>
        %u7 = srl.new_aux_stream: stream<f32>
        srl.while_present(%u1, 1) {
            srl.mul %u7, %u1, %u2: stream<f32>
            srl.sub %u6, %u6, %u7: stream<f32>
        }
        srl.reduce %u7, %u6: void
        srl.add %u7, %u3, %u7: void
        srl.div %u5, %u7, %u4: void
    }
}
```

Figure B.9: Translation result from DSL to MLIR dialect (TRISOLV kernel).

```
addi t1, x0, 3 # %snm1 = 4
addi t2, x0, 3 # %snp1 = 6
addi t3, x0, 3 # %sn = 5
addi t4, x0, 1 # %one = 1
addi t5, x0, 0 # %zero = 0
ss.sta.ld.w u1, %[x],t3, t3
ss.app.mod.siz.inc u1, t1, t4
ss.end u1, t5, t5, t4
ss.cfg.vec u1

ss.sta.ld.w u2, %[z],t3, t5
ss.app.mod.siz.inc u2, t1, t4
ss.end u2, t5, t5, t4
ss.cfg.vec u2

ss.ld.w u3, %[y],t3, t4

ss.ld.w u4, %[x],t3, t2

ss.st.w u5, %[z],t3, t4

.Loop_0:
so.v.dp.w u6, t5, p0

.Loop_1:
so.a.mul.fp u6, u2, u1, p0
so.a.sub.fp u7, u7, u6, p0

so.b.ndc.1 u1, .Loop_1
so.a.adde.fp u6, u7, p0
so.a.add.fp u6, u6, u3, p0
so.a.div.fp u5, u6, u4, p0
```

Figure B.10: Translation result from MLIR dialect to UVE 1 (TRISOLV kernel).

```
addi t1, x0, 3 # %snm1 = 3
addi t2, x0, 3 # %snp1 = 3
addi t3, x0, 3 # %sn = 3
addi t4, x0, 1 # %one = 1
addi t5, x0, 0 # %zero = 0
ss.sta.ld.w.v.1 u1, %[x]
ss.app u1, t5, t3, t3
ss.app.mod.siz.inc.1 u1, t4
ss.end u1, t5, t5, t4
ss.cfg.vec u1

ss.sta.ld.w.v.1 u2, %[z]
ss.app u2, t5, t3, t5
ss.app.mod.siz.inc.1 u2, t4
ss.end u2, t5, t5, t4
ss.cfg.vec u2

ss.ld.w u3, %[y]

ss.ld.w u4, %[x]

ss.st.w u5, %[z]

.Loop_0:
so.v.dp.w u6, t5, p0

.Loop_1:
so.a.mul.fp u7, u1, u2, p0
so.a.sub.fp u6, u6, u7, p0

so.b.ndc.1 u1, .Loop_1
so.a.adde.fp u7, u6, p0
so.a.add.fp u7, u3, u7, p0
so.a.div.fp u5, u7, u4, p0

so.b.nc u1, .Loop_0
```

Figure B.11: Translation result from MLIR dialect to UVE 2 (TRISOLV kernel).

```
void _3mm(float *src1, float *src2, float *src3, uint64_t sizeI, uint64_t sizeJ, uint64_t sizeK) {

  int i,j,k;

    for (i = 0; i < sizeI; i++) {
        for (j = 0; j < sizeJ; j++){
          src3[i*sizeJ+j] = 0;
          for (k = 0; k < sizeK; k++)
              src3[i*sizeJ+j] += src1[i*sizeK+k] * src2[k*sizeJ+j];
      }
  }

}
```

Figure B.12: 3MM kernel C source code.

```
{
  "functionName": "_3mm",
  "program": ["loop::1"],
  "controlFlows": [
    {
      "condition": "whilePresent::u1::0", "begin": ["node::0"],
      "innerControlFlow": ["loop::2"], "end": ["node::4", "node::5", "node::6"],
      "tag": "loop::1"
    },{
      "condition": "whilePresent::u1::1","begin": ["node::1", "node::3"],
      "end": [],"innerControlFlow": [],
      "tag": "loop::2"
    }],
  "dataFlows": [{
      "node::0": { "args": ["u6"], "children": ["node::01"], "type": "=" },
      "node::01": { "args": ["zero"], "children": [], "type": "constantToStream" }
    },{
      "node::1": { "args": ["u7"], "children": ["node::2"], "type": "=" },
      "node::2": {
        "args": [], "children": ["node::21", "node::22"], "type": "*",
        "node::21": { "children": [], "args": ["u1"], "type": "STREAM::load" },
        "node::22": { "children": [], "args": ["u2"], "type": "STREAM::load" }
      }},{
      "node::3": {
        "children": ["node::30"], "args": ["u6"], "type": "=",
        "node::30": {
          "children": ["node::31", "node::32"], "args": [], "type": "-",
          "node::31": { "children": [], "args": ["u6"], "type": "STREAM::load" },
          "node::32": { "children": [], "args": ["u7"], "type": "STREAM::load" }
        }},{
      "node::4": {
        "children": ["node::31", "node::32"], "args": [], "type": "reduce",
        "node::31": { "children": [], "args": ["u7"], "type": "STREAM::load" },
        "node::32": { "children": [], "args": ["u6"], "type": "STREAM::load" }
      }},{
      "node::5": {
        "children": ["node::30"], "args": ["u7"], "type": "=",
        "node::30": {
          "children": ["node::31", "node::32"], "args": [], "type": "+",
          "node::31": { "children": [], "args": ["u3"], "type": "STREAM::load" },
          "node::32": { "children": [], "args": ["u7"], "type": "STREAM::load" }
        }},{
      "node::6": {
        "children": ["node::30"], "args": ["u5"], "type": "=",
        "node::30": {
          "children": ["node::31", "node::32"], "args": [], "type": "/",
          "node::31": { "children": [], "args": ["u7"], "type": "STREAM::load" },
          "node::32": { "children": [], "args": ["u4"], "type": "STREAM::load" }
        } }],"inputs": [
    { "datatype": "ptr", "name": "src1" },{ "datatype": "ptr", "name": "src2" },
    { "datatype": "ptr", "name": "src3" },{ "datatype": "i32", "name": "si" },
    { "datatype": "i32", "name": "sj" },{ "datatype": "i32", "name": "sk" },
    { "datatype": "i32", "name": "zero", "value": "0" },
    { "datatype": "i32", "name": "one", "value": "1" }
  ],
  "streams": [{
      "name": "Stream1", "datatype": "f32", "type": "InputStream", "base": "src1",
      "dims": [
        { "init": "zero", "nElements": "sk", "stride": "si" },
        { "init": "zero", "nElements": "sj", "stride": "0" },
        { "init": "zero", "nElements": "sk", "stride": "one" }
      ]},{
      "name": "Stream2", "datatype": "f32", "type": "InputStream", "base": "src2",
      "dims": [
        { "init": "zero", "nElements": "si", "stride": "0" },
        { "init": "zero", "nElements": "sj", "stride": "one" },
        { "init": "zero", "nElements": "sk", "stride": "sj" }
      ]},{
      "name": "Stream3", "datatype": "f32", "type": "OutputStream", "base": "src3",
      "dims": [
        { "init": "zero", "nElements": "si", "stride": "sj" },
        { "init": "zero", "nElements": "sj", "stride": "one" }
      ]}]
}
```

Figure B.13: JSON generated by the LLVM-IR analysis (3MM kernel).

```
1     void trisolv(ptr x, ptr y, ptr z)
2 {
3     i64 snm1 = 3;
4     i64 snp1 = 3;
5     i64 sn = 3;
6     i64 one = 1;
7     i64 zero = 0;
8
9     InputStream<f32> u1(x);
10    u1.appendDim(zero, sn, sn);
11    u1.appendDim(zero, one, zero);
12    u1.appendSizeMod(inc, one, zero, snm1);
13
14    InputStream<f32> u2(z);
15    u2.appendDim(zero, zero, sn);
16    u2.appendDim(zero, one, zero);
17    u2.appendSizeMod(inc, one, zero, snm1);
18
19    InputStream<f32> u3(y);
20    u3.appendDim(zero, one, sn);
21    u3.setScalar();
22
23    InputStream<f32> u4(x);
24    u4.appendDim(zero, snp1, sn);
25    u4.setScalar();
26
27    OutputStream<f32> u5(z);
28    u5.appendDim(zero, one, sn);
29    u5.setScalar();
30
31    whilePresent(u1, 0)
32    {
33        AuxStream<f32> u6 = constantToStream(zero);
34        whilePresent(u1, 1)
35        {
36            AuxStream<f32> u7 = u1 * u2;
37            u6 = u6 - u7;
38        }
39        reduce(u7, u6);
40        u7 = u3 + u7;
41        u5 = u7 / u4;
42    }
43 }
```

Figure B.14: Translation result from JSON to SRL (GEMM kernel).

```
func.func @trisolv(%x: ptr, %y: ptr, %z: ptr) -> (void) {
    %snm1 = arith.constant 3: i64
    %snp1 = arith.constant 3: i64
    %sn = arith.constant 3: i64
    %one = arith.constant 1: i64
    %zero = arith.constant 0: i64
    %u1 = srl.new_input_stream %x: stream<f32>
    srl.append_dim %u1, %zero, %sn, %sn: void
    srl.append_dim %u1, %zero, %one, %zero: void
    srl.append_size_mod inc, %u1, %one, %zero, %snm1: void
    %u2 = srl.new_input_stream %z: stream<f32>
    srl.append_dim %u2, %zero, %zero, %sn: void
    srl.append_dim %u2, %zero, %one, %zero: void
    srl.append_size_mod inc, %u2, %one, %zero, %snm1: void
    %u3 = srl.new_input_stream %y: stream<f32>
    srl.append_dim %u3, %zero, %one, %sn: void
    srl.set_scalar %u3: void
    %u4 = srl.new_input_stream %x: stream<f32>
    srl.append_dim %u4, %zero, %snp1, %sn: void
    srl.set_scalar %u4: void
    %u5 = srl.new_output_stream %z: stream<f32>
    srl.append_dim %u5, %zero, %one, %sn: void
    srl.set_scalar %u5: void
    srl.while_present(%u1, 0) {
        %u6 = srl.constant_to_stream %zero: stream<f32>
        %u7 = srl.new_aux_stream: stream<f32>
        srl.while_present(%u1, 1) {
            srl.mul %u7, %u1, %u2: stream<f32>
            srl.sub %u6, %u6, %u7: stream<f32>
        }
        srl.reduce %u7, %u6: void
        srl.add %u7, %u3, %u7: void
        srl.div %u5, %u7, %u4: void
    }
}
```

Figure B.15: Translation result from DSL to MLIR dialect (GEMM kernel).

```
addi t1, x0, 0 # %zero = 0
addi t2, x0, 1 # %one = 1
ss.sta.ld.w u1, %[src1],%[sk], %[si]
ss.app u1, t1, t1, %[sj]
ss.end u1, t1, t2, %[sk]
ss.cfg.vec u1

ss.sta.ld.w u2, %[src2],t1, %[si]
ss.app u2, t1, t2, %[sj]
ss.end u2, t1, %[sj], %[sk]
ss.cfg.vec u2

ss.sta.st.w u3, %[src3],%[sj], %[si]
ss.end u3, t1, t2, %[sj]
ss.cfg.vec u3

.Loop_0:
so.v.dp.w u4, t1, p0

.Loop_1:
so.a.mul.fp u5, u1, u2, p0
so.a.add.fp u4, u4, u5, p0

so.b.ndc.1 u2, .Loop_1
so.a.adde.fp u3, u4, p0

so.b.nc u2, .Loop_0
```

Figure B.16: Translation result from MLIR dialect to UVE 1 (GEMM kernel).

```
addi t1, x0, 3 # %snm1 = 3
addi t2, x0, 3 # %snp1 = 3
addi t3, x0, 3 # %sn = 3
addi t4, x0, 1 # %one = 1
addi t5, x0, 0 # %zero = 0
ss.sta.ld.w.v.1 u1, %[x]
ss.app u1, t5, t3, t3
ss.app.mod.siz.inc.1 u1, t4
ss.end u1, t5, t5, t4
ss.cfg.vec u1

ss.sta.ld.w.v.1 u2, %[z]
ss.app u2, t5, t3, t5
ss.app.mod.siz.inc.1 u2, t4
ss.end u2, t5, t5, t4
ss.cfg.vec u2

ss.ld.w u3, %[y]

ss.ld.w u4, %[x]

ss.st.w u5, %[z]

.Loop_0:
so.v.dp.w u6, t5, p0

.Loop_1:
so.a.mul.fp u7, u1, u2, p0
so.a.sub.fp u6, u6, u7, p0

so.b.ndc.1 u1, .Loop_1
so.a.adde.fp u7, u6, p0
so.a.add.fp u7, u3, u7, p0
so.a.div.fp u5, u7, u4, p0

so.b.nc u1, .Loop_0
```

Figure B.17: Translation result from MLIR dialect to UVE 2 (GEMM kernel).

```
void gemm(float alpha, float beta, float *C, float *A, float *B, uint64_t sizeI,
    uint64_t sizeJ, uint64_t sizeK){

  // C (sizeI x sizeJ) * beta  += alpha * A (sizeI x sizeK) * B (sizeK x sizeJ)

  int i, j, k;

  for (i = 0; i < sizeI; i++) {
    for (j = 0; j < sizeJ; j++)
      C[i * sizeI + j] *= beta;
    for (k = 0; k < sizeK; k++) {
       for (j = 0; j < sizeJ; j++)
        C[i * sizeI + j] += alpha * A[i * sizeI + k] * B[k*sizeK + j];
    }
  }

}
```

Figure B.18: GEMM kernel C source code.

```
{
  "controlFlows": [{
    "condition": "whilePresent::Stream1::0",
    "inner_cfg": ["loop::1_0", "loop::1_1"],
    "tag": "loop::1"
  },{ "begin": ["node::5"], "condition": "whilePresent::Stream3::1", "tag": "loop::1_1" },
  { "begin": ["node::1"], "condition": "whilePresent::Stream1::1", "tag": "loop::1_0" }
]
  "dataFlows": [{
    "node::1": { "args": ["Stream1"], "children": ["node::2"], "type": "=" },
    "node::2": { "children": ["node::3", "node::4"], "type": "*" },
    "node::3": { "args": ["Stream2"], "type": "STREAM::load" },
    "node::4": { "args": ["constantbeta"], "type": "INPUT::value" },
    "node::5": { "args": ["Stream3"], "children": ["node::6"], "type": "=" },
    "node::6": { "children": ["node::7", "node::8"], "type": "+" },
    "node::7": { "args": ["Stream6"], "type": "STREAM::load" },
    "node::8": { "children": ["node::9", "node::12"], "type": "*" },
    "node::9": { "children": ["node::10", "node::11"], "type": "*" },
    "node::10": { "args": ["Stream5"], "type": "STREAM::load" },
    "node::11": { "args": ["constantalpha"], "type": "INPUT::value" },
    "node::12": { "args": ["Stream4"], "type": "STREAM::load" }
  }],
  "inputs": [
    { "datatype": "f32", "name": "input1", "value": "%alpha" },
    { "datatype": "f32", "name": "input0", "value": "%beta" },
    { "datatype": "ptr", "name": "input2", "value": "%C" },
    { "datatype": "ptr", "name": "input10", "value": "%A" },
    { "datatype": "ptr", "name": "input12", "value": "%B" },
    { "datatype": "i64", "name": "input3", "value": "0" },
    { "datatype": "i64", "name": "input5", "value": "%0" },
    { "datatype": "i64", "name": "input11", "value": "%1" },
    { "datatype": "i64", "name": "input4", "value": "%wide.trip.count80" },
    { "datatype": "i64", "name": "input6", "value": "%wide.trip.count" },
    { "datatype": "i64", "name": "input8", "value": "%wide.trip.count73" },
    { "datatype": "i64", "name": "input9", "value": "%wide.trip.count67" },
    { "datatype": "i64", "name": "input7", "value": "1" }
  ],
  "streams": [{
      "base": "input2", "datatype": "f32", "dims": [
        { "init": "input3", "nElements": "input4", "stride": "input5" },
        { "init": "input3", "nElements": "input6", "stride": "input7" }
      ], "name": "Stream1", "type": "OutputStream"},{
      "base": "input2", "datatype": "f32", "dims": [
        { "init": "input3", "nElements": "input4", "stride": "input5" },
        { "init": "input3", "nElements": "input6", "stride": "input7" }
      ], "name": "Stream2", "type": "InputStream"},{
      "base": "input2", "datatype": "f32", "dims": [
        { "init": "input3", "nElements": "input4", "stride": "input5" },
        { "init": "input3", "nElements": "input8", "stride": "input3" },
        { "init": "input3", "nElements": "input9", "stride": "input7" }
      ], "name": "Stream3", "type": "OutputStream"},{
      "base": "input12", "datatype": "f32", "dims": [
        { "init": "input3", "nElements": "input4", "stride": "input3" },
        { "init": "input3", "nElements": "input8", "stride": "input5" },
        { "init": "input3", "nElements": "input9", "stride": "input7" }
      ], "name": "Stream4", "type": "InputStream"},{
      "base": "input10", "datatype": "f32", "dims": [
        { "init": "input3", "nElements": "input4", "stride": "input11" },
        { "init": "input3", "nElements": "input8", "stride": "input7" },
        { "init": "input3", "nElements": "input9", "stride": "input3" }
      ], "name": "Stream5", "type": "InputStream"},{
      "base": "input2", "datatype": "f32", "dims": [
        { "init": "input3", "nElements": "input4", "stride": "input5" },
        { "init": "input3", "nElements": "input8", "stride": "input3" },
        { "init": "input3", "nElements": "input9", "stride": "input7" }
      ], "name": "Stream6", "type": "InputStream"}],
  "vectors": [{ "datatype": "f32", "name": "constantbeta", "value": "input0" },
    { "datatype": "f32", "name": "constantalpha", "value": "input1" }]
}
```


Figure B.19: JSON generated by the LLVM-IR analysis (GEMM kernel).

```
    void trisolv(ptr x, ptr y, ptr z)
{
    i64 snm1 = 3;
    i64 snp1 = 3;
    i64 sn = 3;
    i64 one = 1;
    i64 zero = 0;

    InputStream<f32> u1(x);
    u1.appendDim(zero, sn, sn);
    u1.appendDim(zero, one, zero);
    u1.appendSizeMod(inc, one, zero, snm1);

    InputStream<f32> u2(z);
    u2.appendDim(zero, zero, sn);
    u2.appendDim(zero, one, zero);
    u2.appendSizeMod(inc, one, zero, snm1);

    InputStream<f32> u3(y);
    u3.appendDim(zero, one, sn);
    u3.setScalar();

    InputStream<f32> u4(x);
    u4.appendDim(zero, snp1, sn);
    u4.setScalar();

    OutputStream<f32> u5(z);
    u5.appendDim(zero, one, sn);
    u5.setScalar();

    whilePresent(u1, 0)
    {
        AuxStream<f32> u6 = constantToStream(zero);
        whilePresent(u1, 1)
        {
            AuxStream<f32> u7 = u1 * u2;
            u6 = u6 - u7;
        }
        reduce(u7, u6);
        u7 = u3 + u7;
        u5 = u7 / u4;
    }
}
```

Figure B.20: Translation result from JSON to SRL (GEMM kernel).

```
func.func @trisolv(%x: ptr, %y: ptr, %z: ptr) -> (void) {
    %snm1 = arith.constant 3: i64
    %snp1 = arith.constant 3: i64
    %sn = arith.constant 3: i64
    %one = arith.constant 1: i64
    %zero = arith.constant 0: i64
    %u1 = srl.new_input_stream %x: stream<f32>
    srl.append_dim %u1, %zero, %sn, %sn: void
    srl.append_dim %u1, %zero, %one, %zero: void
    srl.append_size_mod inc, %u1, %one, %zero, %snm1: void
    %u2 = srl.new_input_stream %z: stream<f32>
    srl.append_dim %u2, %zero, %zero, %sn: void
    srl.append_dim %u2, %zero, %one, %zero: void
    srl.append_size_mod inc, %u2, %one, %zero, %snm1: void
    %u3 = srl.new_input_stream %y: stream<f32>
    srl.append_dim %u3, %zero, %one, %sn: void
    srl.set_scalar %u3: void
    %u4 = srl.new_input_stream %x: stream<f32>
    srl.append_dim %u4, %zero, %snp1, %sn: void
    srl.set_scalar %u4: void
    %u5 = srl.new_output_stream %z: stream<f32>
    srl.append_dim %u5, %zero, %one, %sn: void
    srl.set_scalar %u5: void
    srl.while_present(%u1, 0) {
        %u6 = srl.constant_to_stream %zero: stream<f32>
        %u7 = srl.new_aux_stream: stream<f32>
        srl.while_present(%u1, 1) {
            srl.mul %u7, %u1, %u2: stream<f32>
            srl.sub %u6, %u6, %u7: stream<f32>
        }
        srl.reduce %u7, %u6: void
        srl.add %u7, %u3, %u7: void
        srl.div %u5, %u7, %u4: void
    }
}
```

Figure B.21: Translation result from DSL to MLIR dialect (GEMM kernel).

```
addi t1, x0, 3 # %snm1 = 4
addi t2, x0, 3 # %snp1 = 6
addi t3, x0, 3 # %sn = 5
addi t4, x0, 1 # %one = 1
addi t5, x0, 0 # %zero = 0
ss.sta.ld.w u1, %[x],t3, t3
ss.app.mod.siz.inc u1, t1, t4
ss.end u1, t5, t5, t4
ss.cfg.vec u1

ss.sta.ld.w u2, %[z],t3, t5
ss.app.mod.siz.inc u2, t1, t4
ss.end u2, t5, t5, t4
ss.cfg.vec u2

ss.ld.w u3, %[y],t3, t4

ss.ld.w u4, %[x],t3, t2

ss.st.w u5, %[z],t3, t4

.Loop_0:
so.v.dp.w u6, t5, p0

.Loop_1:
so.a.mul.fp u6, u2, u1, p0
so.a.sub.fp u7, u7, u6, p0

so.b.ndc.1 u1, .Loop_1
so.a.adde.fp u6, u7, p0
so.a.add.fp u6, u6, u3, p0
so.a.div.fp u5, u6, u4, p0
```

Figure B.22: Translation result from MLIR dialect to UVE 1 (GEMM kernel).

```
addi t1, x0, 3 # %snm1 = 3
addi t2, x0, 3 # %snp1 = 3
addi t3, x0, 3 # %sn = 3
addi t4, x0, 1 # %one = 1
addi t5, x0, 0 # %zero = 0
ss.sta.ld.w.v.1 u1, %[x]
ss.app u1, t5, t3, t3
ss.app.mod.siz.inc.1 u1, t4
ss.end u1, t5, t5, t4
ss.cfg.vec u1

ss.sta.ld.w.v.1 u2, %[z]
ss.app u2, t5, t3, t5
ss.app.mod.siz.inc.1 u2, t4
ss.end u2, t5, t5, t4
ss.cfg.vec u2

ss.ld.w u3, %[y]

ss.ld.w u4, %[x]

ss.st.w u5, %[z]

.Loop_0:
so.v.dp.w u6, t5, p0

.Loop_1:
so.a.mul.fp u7, u1, u2, p0
so.a.sub.fp u6, u6, u7, p0

so.b.ndc.1 u1, .Loop_1
so.a.adde.fp u7, u6, p0
so.a.add.fp u7, u3, u7, p0
so.a.div.fp u5, u7, u4, p0

so.b.nc u1, .Loop_0
```

Figure B.23: Translation result from MLIR dialect to UVE 2 (GEMM kernel).

Appendix C

SRL Specification

Introduction

The following section documents the Structured Representation Language (SRL) dialect and Domain Specific Language (DSL) methods.

C.1 SRL MLIR Operations

C.1.1 srl.new_input_stream

Arguments: `%x: ptr`

Return Type: `stream<f32>`

Description: Creates a new input stream from the given pointer. This operation initialises a stream that can be used to read data.

C.1.2 srl.new_output_stream

Arguments: `%z: ptr`

Return Type: `stream<f32>`

Description: Creates a new output stream from the given pointer. This operation initialises a stream that can be used to write data.

C.1.3 srl.append_dim

Arguments: `%stream, %init, %stride, %n_elements`

Return Type: `void`

Description: Appends a new dimension to the specified stream. This operation updates the stream configuration to include the additional dimension, which is defined by the following parameters:

- `%stream`: The target stream to which the dimension is being appended.
- `%init`: The initial position for the first element of the new dimension.
- `%stride`: The stride value, which determines the spacing between consecutive elements in the new dimension.
- `%n_elements`: The number of elements in the new dimension.

This allows for more complex data layouts by increasing the dimensionality of the stream.

C.1.4 srl.append_size_mod

Arguments: `inc, %amount, %associatedDim, %targetDim`

Return Type: `void`

Description: Appends a size modification to the specified stream. This modifies the size of the stream based on the provided parameters.

For instance, `srl.append_size_mod(inc, %amount, %associatedDim, %targetDim)` uses the increment mode to adjust the size of the stream, with a modification amount of %amount, associating it with dimension %associatedDim, and targeting dimension %targetDim.

C.1.5 srl.set_scalar

Arguments: `%stream`

Return Type: `void`

Description: Sets the stream to be iterated value by value.

C.1.6 srl.constant_to_stream

Arguments: `%constant`

Return Type: `stream<f32>`

Description: Converts a constant value into a stream. This operation initialises a stream with the provided constant value.

C.1.7 srl.new_aux_stream

Arguments: None

Return Type: `stream<f32>`

Description: Creates a new auxiliary stream. This stream can be used for intermediate computations within operations.

C.1.8 srl.while_present

Arguments: `(%stream: stream, %condition: int)`

Return Type: `void`

Description: Executes a loop while the specified stream is present. This operation iterates over the stream as long as it contains data.

C.1.9 srl.mul

Arguments: `%result, %stream1, %stream2`

Return Type: `void`

Description: Multiplies elements of the given streams and stores the result in the result stream. This operation performs element-wise multiplication.

C.1.10 srl.sub

Arguments: `%result, %minuend, %subtrahend`

Return Type: `void`

Description: Subtracts elements of the subtrahend stream from the minuend stream and stores the result in the result stream. This operation performs element-wise subtraction.

C.1.11 srl.reduce

Arguments: `%result, %stream`

Return Type: `void`

Description: Reduces the elements of the stream to a single value. This operation performs a reduction operation, such as summing or finding the minimum.

C.1.12 srl.add

Arguments: `%result, %stream1, %stream2`

Return Type: `void`

Description: Adds elements of the given streams and stores the result in the result stream. This operation performs element-wise addition.

C.1.13 srl.div

Arguments: `%result, %numerator, %denominator`

Return Type: `void`

Description: Divides elements of the numerator stream by the denominator stream and stores the result in the result stream. This operation performs element-wise division.

C.2 SRL DSL Methods

C.2.1 InputStream

Arguments: `ptr`

Return Type: `InputStream<f32>`

Description: Creates a new input stream from the given pointer. This stream is used to read data.

C.2.2 OutputStream

Arguments: `ptr`

Return Type: `OutputStream<f32>`

Description: Creates a new output stream from the given pointer. This stream is used to write data.

C.2.3 foo.appendDim

Arguments: `init, nElements, stride`

Return Type: `void`

Description: Appends a new dimension to the `foo` stream. This operation initialises the dimension with the specified starting value, sets the number of elements, and defines the stride between elements. For example, `foo.appendDim(0, 100, 1)` initialises the dimension at 0, specifies 100 elements, and sets the stride to 1.

C.2.4 foo.appendSizeMod

Arguments: `inc, amount, associatedDim, targetDim`

Return Type: `void`

Description: Appends size modification to foo stream. This modifies the size of the stream according to the provided parameters. For example, `foo.appendSizeMod(inc, 1, 0, 1)` uses the increment mode to adjust the size of the stream, with a modification amount of 1, associating it with dimension 0 and targeting dimension 1.

C.2.5 foo.setScalar

Arguments: `stream`

Return Type: `void`

Description: Sets a scalar value for the specified stream. This configures the stream with a scalar value to be used in subsequent operations.

C.2.6 constantToStream

Arguments: `constant: f32`

Return Type: `AuxStream<f32>`

Description: Converts a constant value into an auxiliary stream. This initializes a stream with the provided constant value.

C.2.7 whilePresent

Arguments: `stream, condition`

Return Type: `void`

Description: Executes a loop while the specified stream is present. This iterates over the stream as long as it contains data.

C.2.8 mul

Arguments: `result, stream1, stream2`

Return Type: `AuxStream<f32>`

Description: Multiplies elements of the given streams and stores the result in an auxiliary stream. This performs element-wise multiplication.

C.2.9 sub

Arguments: `result, minuend, subtrahend`

Return Type: `AuxStream<f32>`

Description: Subtracts elements of the subtrahend stream from the minuend stream and stores the result in an auxiliary stream. This performs element-wise subtraction.

C.2.10 reduce

Arguments: `result, stream`

Return Type: `AuxStream<f32>`

Description: Reduces the elements of the stream to a single value. This performs a reduction operation, such as summing or finding the minimum.

C.2.11 add

Arguments: `result, stream1, stream2`

Return Type: `AuxStream<f32>`

Description: Adds elements of the given streams and stores the result in an auxiliary stream. This performs element-wise addition.

C.2.12 div

Arguments: `result, numerator, denominator`

Return Type: `AuxStream<f32>`

Description: Divides elements of the numerator stream by the denominator stream and stores the result in an auxiliary stream. This performs element-wise division.

References

[1] United Nations. THE 17 GOALS, Accessed 2024. Accessed on: March 8, 2024. URL: https://sdgs.un.org/goals.

[2] Konrad Komisarczyk, Lorenzo Chelini, Kanishkan Vadivel, Roel Jordans, and Henk Corporaal. PET-to-MLIR: A polyhedral front-end for MLIR. In *Proc. of the 23rd Euromicro Conference on Digital System Design (DSD)*, pages 551–556. IEEE, 2020.

[3] Paul Grigoras, Xinyu Niu, Jose G. F. Coutinho, Wayne Luk, Jacob Bower, and Oliver Pell. Aspect driven compilation for dataflow designs. In *Proc. of the 24th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, pages 18–25. IEEE, jun 2013. URL: http://ieeexplore.ieee.org/document/6567545/, doi:10.1109/ASAP.2013.6567545.

[4] Maxeler Technologies. MaxCompiler White Paper, 2011. Accessed on October 23, 2024. URL: https://www.maxeler.com/media/documents/MaxelerWhitePaperMaxCompiler.pdf.

[5] Yaqi Zhang, Nathan Zhang, Tian Zhao, Matt Vilim, Muhammad Shahbaz, and Kunle Olukotun. SARA: Scaling a Reconfigurable Dataflow Accelerator. In *Proc. of the 48th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, pages 1041–1054, 2021. doi:10.1109/ISCA52012.2021.00085.

[6] Francesco Peverelli, Marco Rabozzi, Emanuele Del Sozzo, and Marco D. Santambrogio. OXiGen: A tool for automatic acceleration of c functions into dataflow FPGA-based kernels. In *Proc. of the 32nd IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 91–98. Institute of Electrical and Electronics Engineers Inc., aug 2018. doi:10.1109/IPDPSW.2018.00023.

[7] Nicolas Bohm Agostini, Serena Curzel, Vinay Amatya, Cheng Tan, Marco Minutoli, Vito Giovanni Castellana, Joseph Manzano, David Kaeli, and Antonino Tumeo. An MLIR-based Compiler Flow for System-Level Design and Hardware Acceleration. In *Proc. of the 41st IEEE/ACM International Conference on Computer-Aided Design*, pages 1–9, 2022.

[8] The LLVM Project. CIRCT - Circuit IR Compilers and Tools, 2020. Accessed on October 23, 2024. URL: https://circt.llvm.org/.

[9] Christian Ulmann. Multi-Level Rewriting for Stream Processing to RTL compilation. Master thesis, ETH Zurich, Zurich, 2022. doi:10.3929/ethz-b-000578713.

[10] Finn Wilkinson and Simon McIntosh-Smith. An initial evaluation of arm's scalable matrix extension. In *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 135–140. IEEE, 2022.

[11] Weichuang Zhang, Jieru Zhao, Guan Shen, Quan Chen, Chen Chen, and Minyi Guo. An optimizing framework on mlir for efficient fpga-based accelerator generation. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 75–90, 2024. doi:10.1109/HPCA57654.2024.00017.

[12] Jiahang Lou, Xuchen Gao, Yiqing Mao, Yunhui Qiu, Yihan Hu, Wenbo Yin, and Lingli Wang. An agile deploying approach for large-scale workloads on cgra-cpu architecture. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6, 2024.

[13] John Shalf. The future of computing beyond Moore's Law. *Philosophical Transactions of the Royal Society A*, 378(2166):20190061, 2020.

[14] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *Proc. of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, 2021. doi:10.1109/CGO51591.2021.9370308.

[15] Joao Mario Domingos, Nuno Neves, Nuno Roma, and Pedro Tomás. Unlimited Vector Extension with Data Streaming Support. In *Proc. of the 48th Annual ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 209–222, 2021. doi:10.1109/ISCA52012.2021.00025.

[16] RISCV Foundation. Spike RISC-V ISA Simulator, Accessed 2024. Accessed on: March 8, 2024. URL: https://github.com/riscv-software-src/riscv-isa-sim.

[17] SPECS-FEUP. Lara framework, Accessed 2024. Accessed on: March 8, 2024. URL: https://github.com/specs-feup/lara-framework.

[18] SPECS-FEUP. Lara framework, Accessed 2024. Accessed on: March 8, 2024. URL: https://github.com/specs-feup/anyweaver.

[19] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A Stream Compiler for Communication-Exposed Architectures. *SIGOPS Oper. Syst. Rev.*, 36(5):291–303, oct 2002. doi:10.1145/635508.605428.

[20] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Plasticine: A reconfigurable architecture for parallel patterns. In *Proc. of the 44th Annual ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 389–402, 2017. doi:10.1145/3079856.3080256.

[21] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, et al. Spatial: A language and compiler for application accelerators. In *Proc. of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 296–311, 2018.

[22] LLVM MLIR Contributors. The torch-mlir project, Accessed 2024. Accessed on: March 8, 2024. URL: https://github.com/llvm/torch-mlir.

[23] LLVM MLIR Contributors. The tensorflow-mlir project, Accessed 2024. Accessed on: March 8, 2024. URL: https://github.com/llvm/llvm-project/tree/main/mlir/.

[24] William S. Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. Polygeist: Raising C to Polyhedral MLIR. In *30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 45–59, 2021. doi:10.1109/PACT52795.2021.00011.

[25] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A Stream Compiler for Communication-Exposed Architectures. In *Proc. of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, page 291–303. Association for Computing Machinery, 2002. doi:10.1145/605397.605428.

[26] High-Performance Computing Architectures and Systems. UVE 2.0 - Unlimited Vector Extension with Data Streaming Support, Accessed 2024. Accessed on: March 8, 2024. URL: https://github.com/hpc-ulisboa/UVE.

[27] High-Performance Computing Architectures and Systems. UVE 2.0 - Unlimited Vector Extension with Data Streaming Support, Accessed 2024. Accessed on: March 8, 2024. URL: https://github.com/hpc-ulisboa/UVE2.

[28] Ana Fernandes, Luís Crespo, Nuno Neves, Pedro Tomás, Nuno Roma, and Gabriel Falcao. Functional validation of the risc-v unlimited vector extension. *IEEE Embedded Systems Letters*, 2024.

[29] LLVM MLIR Contributors. MLIR Documentation: Affine Dialect, Accessed 2024. Accessed on: March 8, 2024. URL: https://mlir.llvm.org/docs/Dialects/Affine/.

[30] LLVM MLIR Contributors. MLIR Documentation: Linalg Dialect, Accessed 2024. Accessed on: March 8, 2024. URL: https://mlir.llvm.org/docs/Dialects/Linalg/.

[31] LLVM MLIR Contributors. MLIR Documentation: Vector Dialect, Accessed 2024. Accessed on: March 8, 2024. URL: https://mlir.llvm.org/docs/Dialects/Vector/.

[32] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. A Compiler Infrastructure for Accelerator Generators. In *Proc. of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, page 804–817, 2021. doi:10.1145/3445814.3446712.