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

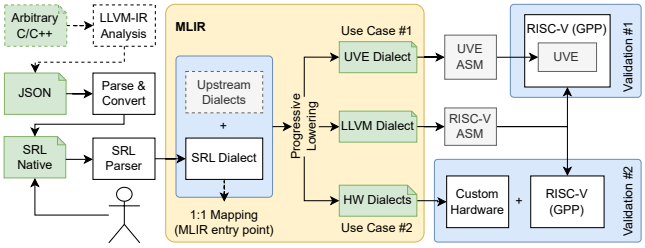

Figure 1: Compilation flow from SRL Native (DSL) to two hardware targets, where the DSL can be provided either by user input, or as a JSON generated from LLVM-IR analysis.

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 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 envisoned 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