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

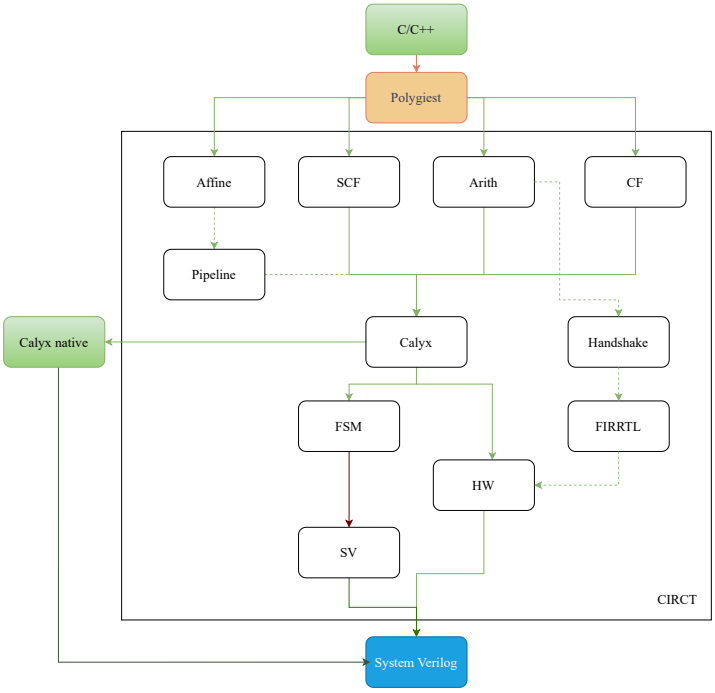

Fig. 1. Subset of current CIRCT ecosystem, focusing on dialects and transformation paths which take `C/C++` as input (complete figure in [14])

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 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 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

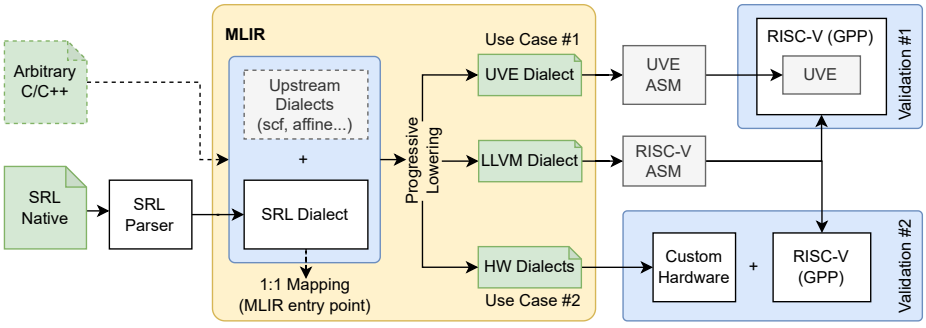

Fig. 2. Compilation flow from SRL Native (i.e., the SRL DSL) to two hardware targets. Support for SRL Dialect generation from standard source code is planned future work.

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:

```
  ss.ld.d u1, %[src1], %[size], %[stride] #Loads stream data to register
  ss.cfg.vec u1 #Configure vector registers as a vector dimension
  ss.ld.d u10, %[src2], %[size], %[stride] #Same as u1
  ss.cfg.vec u10
  ss.st.d u3, %[dest], %[size], %[stride]
  ss.cfg.vec u3 #Same as u1
  so.v.dp.d u4, %[value], p0 #loads to u4 the 'value'
.L1%=: #Perform a vector addition between u10 and u5, store the result in u3
  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
```

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 3, 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 3, 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