UDIR: Towards a Unified Compiler Framework for Reconfigurable Dataflow Architectures

Nikhil Agarwal*, Mitchell Fream*, Souradip Ghosh*, Brian C. Schwedock*†, Nathan Beckmann*
{nikhilag, mfream, souradip}@cmu.edu b.schwedock@samsung.com beckmann@cs.cmu.edu
*Carnegie Mellon University †Samsung

***Abstract*—Specialized hardware accelerators have gained traction as a means to improve energy efficiency over inefficient von Neumann cores. However, as specialized hardware is limited to a few applications, there is increasing interest in programmable, non-von Neumann architectures to improve efficiency on a wider range of programs. Reconfigurable dataflow architectures are a promising design, but the design space is fragmented and, in particular, existing compiler and software stacks are ad hoc and hard to use. Without a robust, mature software ecosystem, RDAs lose much of their advantage over specialized hardware.**

This paper proposes a unifying dataflow intermediate representation (UDIR) for reconfigurable dataflow compilers. Popular von Neumann compiler representations are inadequate for dataflow architectures because they do not represent the dataflow control paradigm, which is the target of many common compiler analyses and optimizations. UDIR introduces *contexts* to break regions of instruction reuse in programs. Contexts generalize prior dataflow control paradigms, representing where in the program tokens must be synchronized. We evaluate UDIR on four prior dataflow architectures, providing simple rewrite rules to lower UDIR to their respective machine-specific representations, and demonstrate a case study of using UDIR to optimize memory ordering.

I. Introduction

Von Neumann processors are fundamentally energy-inefficient, wasting 90% or more of their energy on instruction fetch, pipeline control, and data movement [7,9,10]. Over the past decade, architects have turned to *hardware specialization* to improve energy efficiency. Application-specific integrated circuits (ASICs) regularly improve performance and energy by 100× vs. vN processors [3,10,11]. But ASICs also increase design cost by orders of magnitude, and they are only applicable to a narrow subset of computations. [18] Consequently, energy efficiency remains out of reach for most programmers.

Dataflow to the rescue? Recent reconfigurable dataflow architectures (RDAs) and coarse-grained reconfigurable arrays (CGRAs) promise the best of both worlds: programmability *and* efficiency. At a high-level, an RDA is a grid of simple processing elements (PEs) connected by a network on-chip (NoC), which avoids the inefficiency of vN by spatially distributing compute and communication (vs. streaming instructions through a shared execution pipeline).

Poor compiler support hamstrings dataflow architectures. The design space of recent RDAs and CGRAs is vast; PEs can range from simple ALUs to full-fledged cores. Thus far, their compiler and software stacks have been ad hoc and limited in scope. Because of widely varying designs, compile-time representations are tightly coupled with each RDA's particular microarchitecture. Integrating and maintaining each software stack is expensive and impractical. Immature compiler and software support means that *RDAs lose much of their programmability advantage vs. specialized hardware.* For this reason, dataflow architectures need a unified compiler infrastructure.

What does a unified dataflow IR look like? vN and dataflow architectures are fundamentally different and require different IRs. Dataflow execution is inherently parallel and lacks a program counter. The dataflow control paradigm is to simply issue operations whenever their inputs are available, with additional control operators to steer tokens to the right destination [4]. In the presence of potential races between tokens, dataflow architectures require some *token synchronization* scheme to avoid issuing operations with mismatched inputs. Token synchronization is expensive — typically, more than half of all operations in a dataflow program. Hence, although vN IRs [13] are sufficient for many compiler analyses (e.g., constant propagation), a dataflow IR is needed to reason about dataflow execution.

Our solution: UDIR. We propose *UDIR*, a unifying dataflow intermediate representation for reconfigurable dataflow architectures. UDIR represents programs as a dataflow graph (DFG), which naturally expresses coarse-grained and instruction-level parallelism. UDIR simplifies common compilation steps, including dependence analysis, operation ordering, and instruction scheduling.

UDIR further introduces *contexts* to generalize token synchronization. Token synchronization is a critical dimension of dataflow architectures that distinguishes prior architectures: some designs enforce *ordered dataflow* [4, 8, 12, 17] by disallowing re-ordering of tokens, whereas *tagged dataflow* [14, 15, 20] architectures allow re-ordering to improve performance and give matching tokens a unique tag. We identify that *instruction reuse* defines where token synchronization is required, and UDIR breaks a program into regions called *contexts* that group sets of instructions that will be used together (e.g., a loop or function body). Contexts represent token synchronization because tokens must synchronize only when they cross a context boundary.

Summary of results. UDIR is implemented as an embedded IR in LLVM [13]. We evaluate UDIR on applications written in C, mapping imperative code to dataflow. We prove UDIR's generality by lowering it to a diverse set of four prior dataflow architectures: two ordered dataflow and two tagged dataflow architectures, each with a different token synchronization scheme. Moreover, we show that lowering UDIR to machine-specific dataflow representations requires only simple rewrite rules.

Contributions: This paper contributes the following:

- We develop UDIR, an embedded LLVM dataflow IR and compiler framework. UDIR abstracts dataflow semantics, enabling target-agnostic dataflow compiler optimizations for RDAs and CGRAs.
- We identify instruction reuse and token synchronization as a critical element of dataflow architectures, and propose a new *context* abstraction to represent the elements of token synchronization.
- We provide simple rewrite rules from UDIR to four prior dataflow architectures, showcasing UDIR's generality and ease of use.
- We present a case study where UDIR enables target-agnostic dataflow analysis by reducing redundant control flow.

II. Background

Dataflow vs. vN HW/SW interface. In a von Neumann core, instructions semantically execute in-order, sequenced by a program counter.

Table I: UDIR enables target-agnostic compilation for RDAs, encouraging the development of a mature compiler ecosystem for programmable dataflow architectures.

	vN	RDAs	FPGAs	**UDIR**
Frontend	✓	Ad hoc	HLS	✓
Generic optimization	✓	✗	Vendor-locked	✓
Target-specific optimization	✓	Limited	Vendor-locked	✓
Backend	✓	Platform-specific	Vendor-locked	✓
Performance and efficiency	✗	✓	✓	✓

In contrast, RDAs directly expose parallelism to the programmer, only ordering operations according to program dependencies. Dataflow semantics enable RDAs to distribute parallel operations spatially across an array of simple processing elements, dramatically improving energy efficiency vs. vN designs that attempt to re-discover parallelism at runtime via hardware [8].

Problems with traditional compiler IRs for dataflow. Existing compilers, such as LLVM and GCC, are targeted at von Neumann cores. In particular, they assume sequential execution in their intermediate representations. By assuming sequential execution, existing IRs do not express fine-grain dependencies (e.g., precise memory-ordering dependencies), and instead resort to coarse-grained ordering, (e.g., global memory fences). Nor do they support dataflow-specific program transformations, like spatial loop unrolling. This makes RDAs an unsuitable backend target for existing compilers, and calls for a high-level dataflow IR. As a dataflow IR, UDIR has support for the relaxed program-order semantics of dataflow machines, encoding more precise dependencies and control-flow information than a vN IR.

Prior dataflow IRs and compilers. In lieu of vN IRs, several prior dataflow IRs have been proposed. Pegasus IR [1] notably encodes dataflow information in a high-level IR by combining predicated- and gated-SSA forms. Some dataflow IRs [6, 22, 23] are designed to streamline generic optimizations (e.g. dead-code elimination) using a graphical IR structure. However, none of these IRs aims to be a high-level target for a wide variety of RDA designs. In particular, they do not abstract the various token synchronization schemes seen in prior dataflow architectures, a primary contribution of UDIR.

Other compilers for specialized architectures. Some specialized compilers target RDA-adjacent architectures, and they have built frameworks that feature common optimizations and target-specific backends. Recently, HPVM [5], ÂţIR [19], and TVM [2] have implemented high-level IRs and lowering passes for heterogeneous machines, custom accelerator designs, and accelerators for ML, respectively. However, these IRs do not fulfill the same purpose as UDIR. HPVM retains the program counter from vN IRs, ÂţIR targets RTL only, and TVM optimizes high level tensor operations.

III. Unified Dataflow IR

The unifying dataflow IR (UDIR) is an intermediate representation and compiler infrastructure for reconfigurable dataflow architectures. The UDIR compiler framework supports programs written in sequential languages, like C, leveraging the LLVM infrastructure for generic optimizations (e.g. constant folding). UDIR abstracts dataflow semantics needed across prior RDAs, including control flow, token synchronization, and memory ordering, and it can be easily lowered to prior dataflow ISAs via simple rewrite rules.

Table II: UDIR's instruction set.

Category	**Operator(s)**
Arithmetic	$+$, $-$, $\times$, $\div$, $<<$, $\neq$, etc.
Memory	`load`, `store`
Control flow	`select`, `steer`, `merge`, `order`
Stream	Affine iterators; e.g., $1, 2 \ldots N$
Token synchronization	`enter`, `exit`

A. UDIR's common infrastructure

UDIR enables common, target-agnostic compiler optimizations for RDAs. Prior RDA compilers tightly couple their ISA and the hardware, leaving little room for common and reusable optimizations. Instead of optimizing low-level microarchitectural features straight away, UDIR represents high-level dataflow semantics in a dataflow graph (DFG), as shown in Fig. 1.

UDIR's DFG is constructed from the generated LLVM IR of a program. Doing so enables UDIR to leverage LLVM's existing infrastructure for dependence analysis and optimize DFG operations. It uses LLVM's memory and arithmetic instructions, along with additional dataflow-specific instructions for control and token synchronization. UDIR represents control flow via `steer` operators, which forward or drop an input data token based on a boolean input token, and `merge`/`order` operators shown in Table II. Merge instructions forward one of two input tokens, based on a boolean decider. Order instructions simply forward tokens from one input once a token is received on both inputs, e.g., to order a store after two earlier loads. Currently, UDIR supports standard `if-else` and loop based control flow. Arbitrary indirection, such as `longjmp`, is the subject of ongoing research.

UDIR streamlines compilation of new RDAs. RDA designers can leverage UDIR's infrastructure to produce optimized DFGs without having to reason about dataflow semantics from scratch or from a vN representation. As described in Sec. III-C, compiler developers can lower to specific dataflow ISAs by rewriting UDIR's high-level primitives into machine-specific mechanisms or operations.

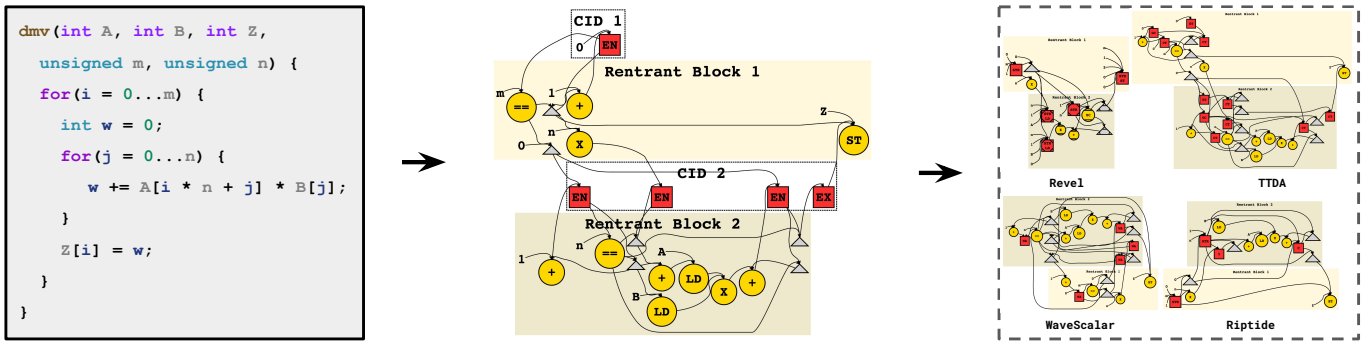
B. *Abstracting token synchronization via* contexts

A distinguishing feature of dataflow ISAs is the token-synchronization scheme in the presence of instruction reuse. Any dataflow ISA needs to understand where to synchronize, what to synchronize, and how to synchronize.

Key insight: *Where* synchronization is needed remains common across all dataflow ISAs. We identify that token synchronization is needed wherever there is *instruction reuse*, i.e, at the boundaries of reentrant code blocks. These points are where mis-matched, racing tokens may arrive and potentially cause incorrect execution. In the example code shown in Fig. 1, the outer loop ("Reentrant Block 1") and inner loop ("Reentrant Block 2") are the two reentrant blocks.

UDIR identifies and marks boundaries around reentrant code blocks and assigns them unique, static identifiers, called *context ids*. We define a context as the unique invocation of a reentrant code block — note that while contexts are dynamic, every context has a unique, static context id. In particular, each unique *callpoint* of a reentrant code block has a unique context id; this is essential to distinguish between invocations of a reentrant block from different call points within the same parent context.

At the context boundary, UDIR inserts instructions called `enter` and `exit`, which pass data through unmodified. The significance of `enter` and `exit` is that these operators demark where token synchronization is required. They are replaced with machine-specific synchronization mechanisms during lowering, as described below.



Sequential Source Code | UDIR | Machine Dataflow Representation

Figure 1: UDIR Compilation Flow

Synchronizing tokens. UDIR first identifies the values that cross the context boundary. For a function call, these values are readily identified by standard compilers, like LLVM, as the function arguments and return value(s). For loops, the values are loop live-ins, loop-carried dependencies, and loop live-outs.

Fig. 1 depicts contexts as yellow background, with red boxes showing the `enter` and `exit` points. `i`, `j`, and `w` are the live-ins that enter through inner loop context boundary (`CID 2`) and `i` alone enters through the outer loop context boundary (`CID 1`). `i` is also a live-out and exits through the outer loop context boundary (`CID 2`).

UDIR's contexts now define the minimal regions of token synchronization, demarking the instructions and tokens that require synchronization. How token synchronization is accomplished can vary vastly across architectures; e.g., TTDA uses a tag allocator to dynamically manage tags, WaveScalar increments tags monolithically, and RipTide has no tags as it enforces strict ordering among tokens.

C. Lowering UDIR to prior RDAs

UDIR can be lowered to prior RDAs with vastly different architectures and token synchronization schemes via simple rewrite rules.

Riptide [8] turns `enter` to `carry` or `invariant`, depending on whether the corresponding enter has a self edge. `carry` and `invariant` block the creation of new contexts until an inner context completes. Riptide simply deletes `exits`, since there is only one live context.

TTDA [14] turns `enter`(s) at a context boundary into one `get-context` and `change-tag`(s) operators. `get-context` calls the tag manager, which allocates a new tag from a global free list. The new tag is fanned out to all `change-tags` at the context boundary. `exit` is replaced with a pair of `extract-tag` and `change-tag` operators, which record and then restore the tag of the parent context.

Revel [21] combines an ordered-dataflow and systolic dataflow architecture, with support for streaming loads and stores. The lowering pass first analyzes UDIR's DFG to identify affine loads and stores and turns them into `streams`. It then turns outer-loop `enters` to `carry` and `invariant` operators, like Riptide. Since the systolic array is statically scheduled, the innermost loop's `enters` are fused to a `transfer` operator, which is a barrier that forwards tokens only when all have arrived. Finally, Revel deletes `exits` for outer loops, like RipTide, and inserts `transfers` for `exits` from the innermost loop.

WaveScalar [20] turns `enter` to `wave-advance`, which increments the tag of the input token. WaveScalar requires tags to increase monotonically, and must sequentialize execution to avoid tag conflicts. As a consequence, WaveScalar must carry *all* live tokens through `wave-advance` across every context, boundary adding a large number of `wave-advance` operators. Finally, WaveScalar deletes `exits`.

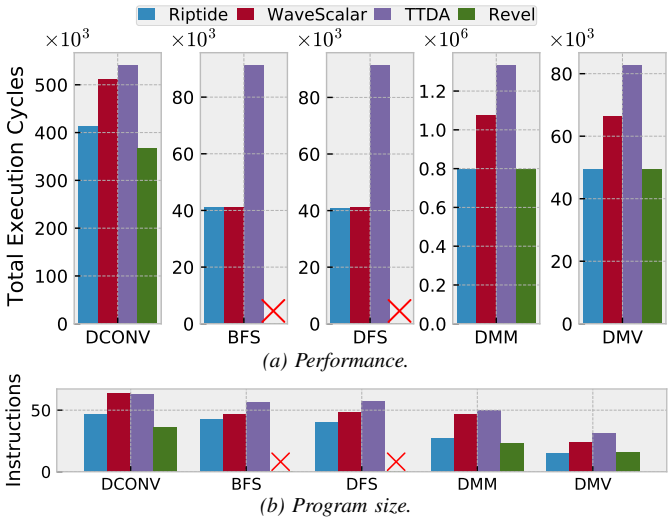
IV. Methodology

We evaluate UDIR on a suite of five commonly used kernels: BFS, DFS, DCONV, DMM and DMV. We use a high-level functional dataflow simulator to run the compiled dataflow graphs for each of four dataflow ISAs: Riptide, Revel, TTDA, WaveScalar. We model each target architecture at a high level, performing tag matching as described in each design, except for Revel the outer loops run on a vanilla ordered-dataflow architecture instead of Triggered Instructions [16]. We assume infinite buffering for input tokens. Each instruction is allowed to fire at most once per cycle. Our memory has a single port and 8 banks. All instructions finish in one cycle except loads and stores (bank conflicts), and TTDA's `get-context` instruction is modeled as four cycles to account for tag allocation overhead.

V. Evaluation

A. UDIR can target a wide range of dataflow architectures

UDIR allows us to compile our test suite to most of our target architectures. `BFS` and `DFS` both fail to compile for our Revel-esque target because they have conditional stores in their innermost loop, which are incompatible with Revel's systolic array. Without UDIR, there is no unified compilation support for these architectures. Some, like WaveScalar, did provide compilers which could ingest C, but others, like TTDA, didn't. TTDA targeted the functional language Id.



(a) Performance.

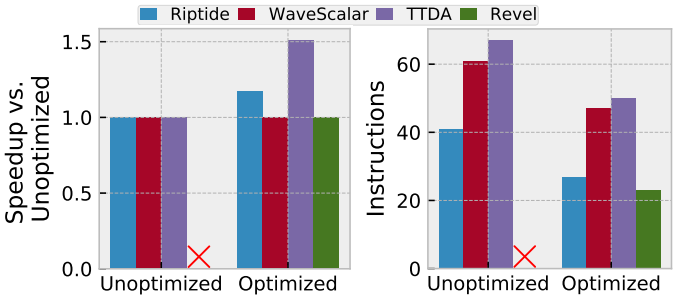
(b) Program size.

Figure 2: Performance and program size for five benchmarks on four different dataflow architectures. UDIR compiles these programs from C, leveraging LLVM and then lowers to each design via simple rewrite rules.

Fig. 2 shows the execution time and program size for compiling each benchmark from C to the different dataflow architectures. The significance of these results is that UDIR can correctly target very different dataflow architectures from a common, high-level dataflow representation. However, the results also enable comparison of different architectures. The cost of dynamic tag allocation is evident from TTDA's poor performance vs. the other architectures. Similarly, WaveScalar must sequentialize execution to guarantee monolithically increasing tags. Both TTDA and WaveScalar can achieve higher parallelism than RipTide or Revel; this potential advantage is not revealed in our simple simulations because all operations execute in one or a few cycles. However, by injecting variable latency, we see that TTDA and WaveScalar can perform significantly better than the ordered dataflow architectures (RipTide and Revel) because it can execute ready operations out-of-order.

B. UDIR enables target-agnostic dataflow optimization

In a vN processor, loads and stores are always ordered (as there is a total program ordering), meaning that work must be done at runtime to discover non-aliased memory accesses. Dataflow architectures do not enforce a total program ordering, so the programmer must ensure the existence of ordering edges between memory accesses according to the source language's memory model. This enables RDAs to encode memory ordering more precisely than vN machines, which can improve performance without complex hardware to track load-store aliasing. As a baseline, UDIR enforces ordering between all memory accesses in the original program order. We show that by relaxing this ordering, we can improve performance on multiple RDA targets.



(a) Performance. *(b) Program size.*

Figure 3: Case study of memory-ordering optimization on DMM. UDIR enables target-agnostic dataflow optimizations on the dataflow graph, which can then be lowered to any machine-specific representation.

We target the simplest case of memory ordering: when two memory accesses do not alias, e.g., when the `restrict` keyword is used in C. To construct the memory-ordering graph, we call LLVM's memory-alias interface. If the result is a strict no-alias, then we remove the ordering edge between memory accesses. This often simplifies the DFG and improves performance. Simplifying the DFG can also enable mapping of more complex programs to some architectures. With full memory-ordering, DMM cannot be lowered to our Revel-esque target since the inner-loop memory accesses cannot be promoted to streams. However, after optimization, the simpler DFG is lowered successfully.

For Riptide, WaveScalar, and TTDA, we see 1.17x, 1x and 1.51x speedups respectively. This optimization does not improve WaveScalar's performance because of the nature of WaveScalar tags. WaveScalar serializes the UDIR DFG because it needs to maintain monotonically increasing tags. Thus, after the optimization, DMM outer loops are serialized due to tag handling rather than memory ordering. In Riptide, outer loop iterations need only wait for preceding outer loop iterations to *begin* in order to maintain ordering, meaning multiple outer loop contexts can exist simultaneously. Because of this, removing the memory ordering constraint allows for more parallelism to be exploited. TTDA imposes the fewest constraints in that not only different outer loop contexts can exist simultaneously but they can also execute out of program order, giving it the largest performance uplift from this optimization. We also see a program size reduction of ≈15 instructions in each (0.66x, 0.77x, 0.75x) due to the roughly constant number of control flow instructions pruned.

This case study shows how UDIR enables target-agnostic dataflow optimizations. UDIR leverages existing LLVM memory-alias analysis to enable dataflow-specific relaxation of memory ordering.

VI. Conclusion

Modular compilation is critical for development of new and maintenance of old compiler infrastructures. To this end, UDIR significantly streamlines compilation flows for reconfigurable dataflow architectures. UDIR's context abstraction captures the differentiating feature of dataflow architectures, enabling target-agnostic compilation and simple lowering passes to new dataflow machines.

References

[1] M. Budiu and S. C. Goldstein, "Pegasus: An efficient intermediate representation," in *Technical Report CMU-CS-02-107*, May 2002.

[2] Chen *et al.*, "Tvm: End-to-end optimization stack for deep learning," Feb 2018. [Online]. Available: https://arxiv.org/abs/1802.04799

[3] W. J. Dally *et al.*, "Domain-specific hardware accelerators," *Commun. ACM*, vol. 63, no. 7, Jun. 2020.

[4] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic data-flow processor," in *ISCA*, 1975.

[5] K. et al, "Hpvm: Heterogeneous parallel virtual machine," in *PPoPP*, Feb. 2018.

[6] R. et al, "Rvsdg: An intermediate representation for optimizing compilers," in *TECS*, 2020.

[7] G. Gobieski *et al.*, "Snafu: an ultra-low-power, energy-minimal cgra-generation framework and architecture," in *ISCA*, 2021.

[8] G. Gobieski *et al.*, "Riptide: A programmable, energy-minimal dataflow compiler and architecture," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 546–564.

[9] G. Gobieski *et al.*, "Manic: A vector-dataflow architecture for ultra-low-power embedded systems," in *MICRO*, 2019.

[10] M. Horowitz, "Computing's energy problem (and what we can do about it)," in *ISSCC*, 2014.

[11] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," *arXiv preprint arXiv:1704.04760*, 2017.

[12] M. Karunaratne *et al.*, "Hycube: A cgra with reconfigurable single-cycle multi-hop interconnect," in *DAC*, 2017.

[13] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO*, Mar. 2004.

[14] R. S. Nikhil *et al.*, "Executing a program on the mit tagged-token dataflow architecture," *IEEE Transactions on computers*, 1990.

[15] G. M. Papadopoulos and D. E. Culler, "Monsoon: An explicit token-store architecture," *SIGARCH Comput. Archit. News*, vol. 18, no. 2SI, p. 82âĂŞ91, may 1990. [Online]. Available: https://doi.org/10.1145/325096.325117

[16] A. Parashar *et al.*, "Triggered instructions: a control paradigm for spatially-programmed architectures," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, 2013.

[17] R. Prabhakar *et al.*, "Plasticine: A reconfigurable architecture for parallel patterns," in *ISCA 44*, 2017.

[18] M. Satyanarayanan *et al.*, "The role of edge offload for hardware-accelerated mobile devices," in *HotMobile*, 2021.

[19] A. Sharifian *et al.*, "ÂţIR: An intermediate representation for transforming and optimizing the microarchitecture of application accelerators," in *MICRO*, Oct. 2019.

[20] S. Swanson *et al.*, "Wavescalar," in *MICRO 36*, 2003.

[21] J. Weng *et al.*, "A hybrid systolic-dataflow architecture for inductive matrix algorithms," in *HPCA*, 2020.

[22] A. M. Zaidi, "Accelerating control-flow intensive code in spatial hardware," in *Thesis*, Feb. 2015.

[23] A. M. Zaidi and D. Greaves, "A new dataflow compiler ir for accelerating control-intensive code in spatial hardware," in *IPDPS*, Dec. 2014.