HLS from PyTorch to System Verilog with MLIR and CIRCT

Mike Urbach
Alloy Computing, LLC
USA

Morten B. Petersen
EPFL
Switzerland

ABSTRACT

Recent work using High-Level Synthesis (HLS) for AI accelerator design has largely been based on innovations at the frontend level. These flows often communicate design intent to vendor HLS tools by generating C++, which can obfuscate high-level information vital for effective HLS. Using open-source compiler infrastructure, we have implemented an HLS compiler that explicitly models domain-specific concepts at every level of abstraction from PyTorch to RTL. We show the utility of this approach by demonstrating dynamically and statically scheduled HLS flows that effectively leverage domain-specific IRs at the appropriate levels of abstraction.

1 INTRODUCTION

We believe one of the core challenges to High-Level Synthesis (HLS) is its use of languages such as C++ that suffer from the von Neumann bottleneck [1]. Hardware is massively concurrent, but the von Neumann programming model makes reasoning about the flow of data between memory and computation difficult. An HLS compiler for this programming model faces challenges when analyzing memory accesses and control flow. These pitfalls can be avoided by building HLS around languages that capture computations in a higher level of abstraction than word-at-a-time programming.

Consider the programming models of popular machine learning (ML) frameworks, taking PyTorch [11] as an example. These capture dataflow and computation in a truly high-level way: a matrix multiply is a single function call in the source and a single construct in the IR. Compare this to C++, where a matrix multiply might be represented by nested loops, memory accesses, and computation. Representing domain-specific concepts like matrix multiply explicitly can convey high-level intent directly to an HLS compiler.

Recent work in high-level programming models for accelerator design often focuses on frontend transformations, but eventually generates C++ to communicate hardware design intent to proprietary HLS tools [2, 9, 14, 16]. We believe that much can be gained by rejecting C++ as a de facto IR for HLS in favor of domain-specific IRs at all stages of compilation. By constructing our HLS compiler with MLIR [8] and CIRCT [5], we can:

- Capture high-level source information that is vital to achieving good HLS results.
- Support optimizations at all levels of abstraction from the ML model to the hardware description.
- Decouple HLS from word-at-a-time programming with compiler infrastructure that models domain concepts explicitly.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
LATTE '22, March 1, 2022, Virtual, Earth
© 2022 Copyright held by the owner/author(s).

We see MLIR as a perfect fit for HLS tooling since it allows compilers to construct, analyze, and transform domain-specific IRs within a state-of-the-art compiler infrastructure. Now, we can connect ML frameworks like PyTorch to CIRCT via MLIR and focus on HLS innovations without reinventing the compiler frontend and backend. At each compiler stage, domain concepts are modeled explicitly using open-source representations, unlocking the potential to optimize, simulate, verify, and extend at each step.

We have developed a compiler stack that explores these ideas. Specifically, we demonstrate:

- An MLIR-based frontend for PyTorch and the high-level information it captures.
- A dynamically scheduled backend flow that uses latency-insensitive components with distributed control.
- A statically scheduled backend flow that computes an FSM with latency-insensitive control.
- A statically scheduled backend flow that computes an FSM with pipelined control.
- Example System Verilog generated by the flows.

Having both statically and dynamically scheduled paths allows us to generate high-performance circuits whether memory access patterns are affine or not. This work focuses on lowering PyTorch through well-known abstractions, which we see as a pragmatic first step with the intent to pave a path for other frontends and novel IRs. To our knowledge, this is the first fully open-source compiler that is able to compile ML programs into hardware description language using MLIR end-to-end, including the hardware description levels.

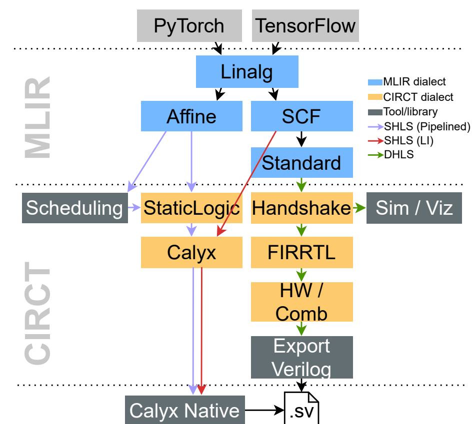

Figure 1: With MLIR and CIRCT, we convert PyTorch programs to RTL through either a dynamically scheduled (DHLS) or statically scheduled (SHLS) flow.

Figure 2: A PyTorch example (i). The input IRs for two backend flows as affine loops (ii) and a CDFG (iii). The mid-level IRs for each flow as a static pipeline (iv) and a dataflow graph (v). SystemVerilog modules generated by each flow (vi) and (vii).

2 PYTORCH FRONTEND

For example, we will take a program written in Python using PyTorch (see Figure 2, Listing i). The torch-mlir [13] compiler infrastructure can convert this into MLIR's Linalg dialect. With upstream MLIR tooling, we convert Linalg to dialects at the different levels of abstraction accepted by the backend flows, as shown in Figure 1 and Figure 2, Listings ii and iii. We focus on lowering, but Linalg also supports transformations like tiling, fusion, and sparsification.

3 DYNAMIC FLOW

In the dynamically scheduled flow, we convert the program to a CDFG, using MLIR's Standard dialect. This is then converted to a dataflow program in the style of Dynamatic [7]. Dataflow is expressed through the Handshake dialect (see Figure 2, Listing v), which models compositional elastic circuits with distributed control [3, 6]. Our stack has been shown to generate substantially smaller circuits than Dynamatic, by taking advantage of optimizations in CIRCT at both the dataflow and hardware level [12].

4 STATIC, LATENCY-INSENSITIVE FLOW

In the static, latency-insensitive flow, we convert the program to the SCF (Structured Control Flow) dialect in MLIR in order to represent looping and conditional constructs at a higher level of abstraction than in Section 3. From there, we construct Calyx IR [10], which captures these abstractions in an explicit schedule. On the backend, we use the native Calyx compiler to synthesize an FSM and datapath, using latency-insensitive connections like the dynamic flow.

5 STATIC, PIPELINED FLOW

To leverage more high-level information in the static flow, we convert the program to the Affine dialect in MLIR. The polyhedral model enables us to enforce structured iterations, capture detailed memory access dependences using MLIR, and compute an optimal schedule using CIRCT's scheduling infrastructure. This is captured in the StaticLogic dialect as a pipelined while loop IR (see Figure 2, Listing iv), and can be lowered to Calyx IR as in Section 4. Unlike the static, latency-insensitive flow, we synthesize a fully static schedule, which allows the Calyx compiler to optimize the FSM and avoid generating latency-insensitivity circuitry.

6 CONCLUSION AND FUTURE WORK

We now have three complementary, narrow paths leading from ML programs to hardware descriptions and can start to widen each path with more support.

For the dynamic path, we expect to generate higher performance circuits through new buffering techniques, canonicalizations, and memory analyses. On the static paths, we plan to support optimizations such as in ScaleHLS [16], AutoSA [15], and others. We hope to incorporate target specific information and apply design space exploration to each path. Besides improving each path, we intend to combine both static and dynamic scheduling, allowing each to handle parts of the program that they are best suited for, similar to DASS [4]. We also plan to continue integrating Calyx with CIRCT to take advantage of CIRCT's backend optimizations and code generation.

By building these tools as open-source software, we hope to create a center of mass where new HLS innovations can be researched within a common framework that exposes domain-specific IRs at every level of the stack.

ACKNOWLEDGEMENTS

We thank Stephen Neuendorffer, Julian Oppermann, Hanchen Ye, Chris Gyurgyik, Rachit Nigram, Adrian Sampson, and the CIRCT community for their insightful discussions and contributions.

REFERENCES

[1] John Backus. 1978. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *Commun. ACM* 21, 8 (Aug 1978), 613–641.

[2] Endri Bezati, Mahyar Emami, Jörn W. Janneck, and James R. Larus. 2021. StreamBlocks: A compiler for heterogeneous dataflow computing (technical report). *CoRR* abs/2107.09333 (2021). arXiv:2107.09333

[3] Josep Carmona, Jordi Cortadella, Mike Kishinevsky, and Alexander Taubin. 2009. Elastic circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28, 10 (2009), 1437–1455.

[4] Jianyi Cheng, Lana Josipović, George A Constantinides, Paolo Ienne, and John Wickerson. 2021. DASS: Combining Dynamic and Static Scheduling in High-level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2021).

[5] CIRCT. [n.d.]. Circuit IR Compilers and Tools. Online. https://circt.llvm.org

[6] Stephen A Edwards, Richard Townsend, Martha Barker, and Martha A Kim. 2019. Compositional dataflow circuits. *ACM Transactions on Embedded Computing Systems (TECS)* 18, 1 (2019), 1–27.

[7] Lana Josipović, Radhika Ghosal, and Paolo Ienne. 2018. Dynamically scheduled high-level synthesis. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 127–136.

[8] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A compiler infrastructure for the end of Moore's law. *arXiv preprint arXiv:2002.11054* (2020).

[9] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. 2020. Predictable accelerator design with time-sensitive affine types. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 393–407.

[10] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. 2021. A compiler infrastructure for accelerator generators. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 804–817.

[11] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*. 8024–8035.

[12] Morten Borup Petersen. 2022. *A Dynamically Scheduled HLS Flow in MLIR*. Master's thesis. École Polytechnique Fédérale de Lausanne.

[13] torch mlir. [n.d.]. Torch-MLIR Project. Online. https://github.com/llvm/torch-mlir

[14] Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. 2017. Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 65–74.

[15] Jie Wang, Licheng Guo, and Jason Cong. 2021. AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 93–104.

[16] Hanchen Ye, Cong Hao, Jianyi Cheng, Hyunmin Jeong, Jack Huang, Stephen Neuendorffer, and Deming Chen. 2021. ScaleHLS: Scalable High-Level Synthesis through MLIR. *CoRR* abs/2107.11673 (2021). arXiv:2107.11673