A Compiler Infrastructure for Accelerator Generators Extended Abstract

Rachit Nigam* Samuel Thomas* Zhijing Li Adrian Sampson
Cornell University

1. Motivation

Accelerating interest in specialized and reconfigurable architectures has amplified the need for compilers that automatically generate hardware implementations from high-level descriptions. The traditional approach is high-level synthesis (HLS), which aims to automatically compile C and C++ programs to hardware. However, software languages are a poor fit for generating hardware. HLS compilers need to bridge the semantic chasm between sequential, von Neumann software languages and the resource-constrained, pervasively parallel hardware setting. The result is a complex, general-purpose programming model that does not excel at expressing any specific category of accelerator architecture.

A more promising approach is to raise the level of abstraction with domain-specific languages (DSLs) and generate a more specialized class of architecture [4, 8]. While successful, such compilers remain significant one-off engineering feats. An accelerator compiler writer needs not only to decide on an architectural style, such as streaming pipelines [4] or systolic arrays [2]; they then must face the problem of realizing the architecture as intertwined data and control paths using a register transfer level (RTL) target language. These DSL-to-accelerator compilers often invent several intermediate languages on the way to hardware, and reimplement general-purpose optimizations irrelevant to their domain.

We present Calyx, a new intermediate language (IL) and open-source infrastructure for building compilers that generate hardware accelerators. The key novel idea in Calyx is to augment a structural hardware representation with a novel *control* language that lets compiler frontends separate computation resources from their execution schedule. Calyx offers two main benefits over emitting RTL directly: (1) The program's control flow is not obfuscated by implementation in control circuitry, so the Calyx compiler can perform optimizations that are sensitive to the control flow. (2) Frontend compilers can leave the tedious job of implementing the control circuitry, such as finite state machines (FSMs), to the Calyx compiler.

This paper describes the design of the Calyx IL and a series of passes that translate high-level programs to SystemVerilog implementations. We use Calyx to implement two compilers as case studies: one that generates systolic arrays for linear algebra computations, and one that implements a recently proposed imperative language for accelerator design. Calyx makes it tractable to implement these compilers and obtain efficient hardware without directly manipulating RTL descriptions.

*Equally contributing authors.

2. Limitations of the State of the Art

- **RTL Intermediate Languages.** FIRRTL [7] and other intermediate languages for hardware design [3, 12, 14, 15] let tools generate, optimize, and transform RTL descriptions. They are appropriate for representing complete hardware implementations, but for DSL frontends, they inherit the same abstraction gap problem as any other RTL language. Calyx focuses on explicitly representing the control flow of the accelerator design and enables control-flow-sensitive optimizations. Such optimizations are infeasible in RTL-level languages since control flow is encoded using structural elements and recovering it, in general, is not possible.
- **FSM-Based Intermediate Languages.** Intermediate languages that use finite state machines with some representation of the data path have been used extensively in traditional HLS compilers [5, 11, 13]. While able to express any sequential circuit, such languages suffer from two problems: (1) In an effort to represent cycle-accurate schedules, the languages impose restrictions on the data path. (2) They typically impose a particular implementation strategy for execution schedules—for example, using one top-level FSM instead of many small ones. Calyx does not restrict the data path specification, and it does not tie the abstract execution schedule to any concrete implementation. Frontends can control the implementation strategy based on domain-specific information (cf Section 4.4 in the main paper).
- **Hardware DSL Compilation.** Research languages such as Spatial [8], HeteroCL [9], Dahlia [10], and Aetherling [4] aim to combat the expressivity problems of traditional HLS. They are not designed as general-purpose compiler ILs, however, and they typically implement specialized internal ILs that reflect the semantics of the language. Calyx aims to provide a common, unifying target IL for hardware-focused DSLs that is language independent but empowers frontends to exert fine-grained control over the generated architecture.
- **High-Level Synthesis.** Traditional HLS compilers extend C, C++, or OpenCL with ad hoc annotations to express hardware-level concerns [1, 6, 16, 17]. They are convenient for kernels that fit their parallel loop-based mold, but they make it difficult to express more specialized and domain-specific architectures. For example, the Aetherling compiler [4] opts to generate RTL directly because C-based HLS is a poor match for the kinds of high-throughput streaming pipelines it targets. Calyx aims to let similar compiler designs generate exactly the architecture they want without resorting to low-level RTL engineering.

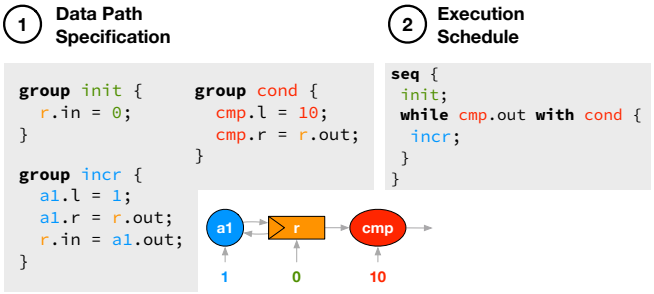

Figure 1: Specifying designs using Calyx's control language.

3. Key Insight

Calyx is an intermediate language (IL) that separates the specification of an accelerator's data path from its execution schedule. Hardware DSL compilers specify structural components and define their execution schedule using an imperative control language that includes loops, conditionals, sequencing, and parallel execution.

Figure 1 shows a Calyx program that implements a counter. It uses *groups* (1) to specify three structural graphs: `init`, which initializes the register with the value 0, `incr`, which increments the value in the register, and `cond`, which computes the exit condition for the counter. To determine when these groups run, the Calyx program defines an execution schedule (2) using the *control language*. The `seq` control statement first executes the graph defined by `init` and then uses the `while` statement to increment the value in the register until the value on `cmp.out` is 0. An equivalent RTL implementation of the same hardware would need to realize the control logic in concrete components and wires; Calyx represents it abstractly. Furthermore, groups can transparently read from and write to ports without multiplexing; the Calyx compiler automatically generates this additional logic based on the schedule.

Calyx's split between the data path and the control specification lets frontends precisely specify the structural components in a hardware design without intertwining it with the control logic. This novel split allows optimization, analysis, and compositional compilation of accelerator designs.

4. Main Artifacts

- A modular, pass-based compiler for Calyx that optimizes and lowers Calyx programs to synthesizable SystemVerilog.
- As a first case study, a systolic array generator targeting Calyx for linear algebra acceleration.
- As a second case study, a Calyx-based backend for Dahlia [10], an imperative language that was originally implemented using a commercial HLS tool as a backend.

5. Key Results and Contributions

Contributions

- We present the design of the Calyx IL and its novel approach to separating a structural description from an imperative control program that orchestrates its execution schedule.
- We describe how to efficiently compile Calyx programs with high-level control constructs to synthesizable RTL.
- We show how to enrich Calyx programs with domain specific information, such as the latency of custom operations generated by a compiler frontend, and demonstrate how the Calyx compiler's pass-based infrastructure can exploit this information to improve the efficiency of generated hardware.

Empirical results

- Our Calyx-based systolic array generator's implementations for matrix multiply are 5.3× faster on average than the designs generated by Vivado HLS and only 1.14× larger on average. At the input largest size, Calyx is 11× faster while using only 1.76× more area.
- Our implementation of a new Calyx backend for the Dahlia [10] programming language generates designs that are within a small factor of a heavily optimized commercial HLS toolchain.
- We extend our Calyx backend for Dahlia to generate domain-specific latency information and show that a custom pass implemented to use this information generates designs that are 50% faster than the baseline compiler.

Advantages over past work

- Unlike previous work on RTL-level ILs [3, 7], Calyx lifts the level of abstraction by providing a high-level control language to specify the execution schedule of a program. Specifying the schedule in this high-level language allows the Calyx compiler to perform control-flow-sensitive optimizations to hardware designs.
- Calyx differs from past work that proposes custom DSLs [4, 8] and uses internal ILs to optimize designs. Calyx is a generic IL and does not make specific architectural choices for the programs represented in it. Language frontends can integrate low-level RTL-like datapaths when needed with high-level imperative descriptions when convenient.

6. Why ASPLOS

Calyx is a programming language for generating custom architectures. It combines ideas from language design and compiler construction with techniques from computer architecture to generate high-performance domain-specific hardware.

7. Citation for Most Influential Paper Award

Calyx initiated a category of research on designing and implementing *accelerator design languages*, which are now seen as a distinct category from low-level hardware description languages (HDLs) and old-fashioned C-derived high-level synthesis (HLS) compilers. The paper highlighted the need for investment in generalized, reusable infrastructure to implement specialized, narrowly focused accelerator compilers. The open-source Calyx infrastructure has since become the foundation for hundreds of domain-specific accelerator compilers, and it served as a research enabler for work on optimizing, analyzing, and debugging hardware accelerators.

References

[1] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2011.

[2] J. Cong and J. Wang. PolySA: Polyhedral-based systolic array auto-compilation. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018.

[3] Ross Daly, Lenny Truong, and Pat Hanrahan. Invoking and linking generators from multiple hardware languages using CoreIR. In *Workshop on Open-Source EDA Technology (WOSET)*, 2018.

[4] David Durst, Matthew Feldman, Dillon Huff, David Akeley, Ross Daly, Gilbert Louis Bernstein, Marco Patrignani, Kayvon Fatahalian, and Pat Hanrahan. Type-directed scheduling of streaming accelerators. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2020.

[5] Nikil D Dutt, Tedd Hadley, and Daniel D Gajski. An intermediate representation for behavioral synthesis. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, 1991.

[6] Intel. Intel High Level Synthesis Compiler. URL `https://www.altera.com/products/design-software/high-level-design/intel-hls-compiler/overview.html`.

[7] Adam M. Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, and Jonathan Bachrach. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In *International Conference on Computer-Aided Design (ICCAD)*, 2017.

[8] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Spatial: a language and compiler for application accelerators. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2018.

[9] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. HeteroCL: A multi-paradigm programming infrastructure for software-defined reconfigurable computing. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2019.

[10] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. Predictable accelerator design with time-sensitive affine types. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2020.

[11] Sameer D Sahasrabuddhe, Hakim Raja, Kavi Arya, and Madhav P Desai. Ahir: A hardware intermediate representation for hardware generation from high-level programs. In *20th International Conference on VLSI Design held jointly with 6th International Conference on Embedded Systems (VLSID'07)*, 2007.

[12] Fabian Schuiki, Andreas Kurth, Tobias Grosser, and Luca Benini. LLHD: A multi-level intermediate representation for hardware description languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2020.

[13] Rohit Sinha and Hiren D Patel. synASM: a high-level synthesis framework with support for parallel and timed constructs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2012.

[14] Sheng-Hong Wang, Akash Sridhar, and Jose Renau. LNAST: a language neutral intermediate representation for hardware description languages. In *Second Workshop on Open-Source EDA Technology (WOSET)*, 2019.

[15] Clifford Wolf. Yosys manual. `http://www.clifford.at/yosys/files/yosys_manual.pdf`.

[16] Xilinx Inc. Vivado Design Suite User Guide: High-Level Synthesis. UG902 (v2017.2) June 7, 2017. `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_2/ug902-vivado-high-level-synthesis.pdf`.

[17] Zhiru Zhang, Yiping Fan, Wei Jiang, Guoling Han, Changqi Yang, and Jason Cong. AutoPilot: A platform-based ESL synthesis system. In *High-Level Synthesis*, pages 99–112. 2008.