A Toolkit for Designing Hardware DSLs

Griffin Berlstein
Cornell University
griffin@cs.cornell.edu

Rachit Nigam
Cornell University
rnigam@cs.cornell.edu

Chris Gyurgyik
Cornell University
cpg49@cornell.edu

Adrian Sampson
Cornell University
asampson@cs.cornell.edu

***Abstract*—Recent years have seen a renewed interest in open-source hardware design tools everywhere in the stack—from new register-transfer-level (RTL) languages to open-source flows for fabricating silicon [1]. While innovation in the traditional hardware stack promises better, faster, and more portable tools, key innovations are still needed to democratize hardware design. Specifically, domain specific languages (DSLs) allow experts to concisely express computations without delving into low-level hardware details are needed to enable widespread use of hardware accelerators. In order to simplify the gargantuan task of implementing, optimizing, and lowering such hardware domain specific languages (DSLs), we have been building Calyx, an intermediate language (IL) and a compiler infrastructure for accelerator generators. We demonstrate how Calyx IL's novel separation between the structural and control-flow aspects of an accelerator design enables it to: (1) simplify frontends by efficiently encoding their semantics, (2) enable novel optimization passes that cannot be performed in traditional hardware ILs, and (3) allow us to build software-like debugging infrastructure.**

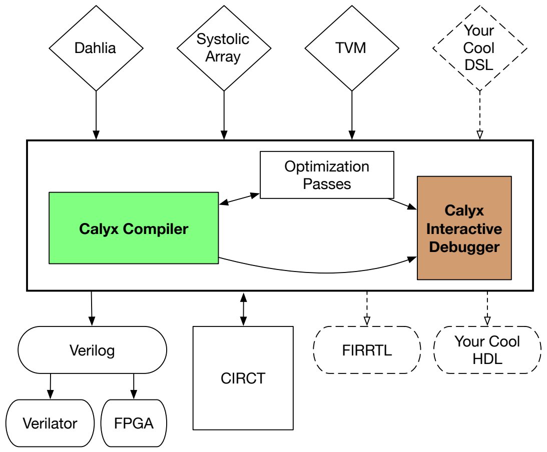


Fig. 1: The Calyx infrastructure for hardware DSLs.

I. Introduction

The rise of open-source hardware tools has begun to revolutionize the system stack for general silicon and FPGA design. Above the level of these general-purpose tools, however, there is a need for commensurate advances in infrastructure for building abstractions for high-level hardware accelerator generation using domain-specific languages (DSLs). We identify two problems in the space that need to be addressed to make designing new hardware DSLs as painless as software DSLs. First is the problem of a robust and optimized compiler infrastructure like LLVM [2] that allows developers to quickly take advantage of heavily optimized optimization and lowering passes. Such an infrastructure needs to perform all of the repetitive optimizations and lowering tasks well so that DSL designers can focus more on developing new abstractions. Second, there has been comparatively little work in the fundamental problem of debugging. While low-level hardware engineers have access to a wide range of verification tools, high-level domain experts do not have the same luxury: those wishing to debug their DSL-generated accelerator are left with only their wits and a waveform debugger. While traditional debugging workflows may be a good fit for hardware experts designing traditional hardware such as a CPU, the disconnect between waveforms and DSL semantics make them a non-starter for domain experts when implementing application-specific accelerators. While DSL users write programs with abstractions like loops and tensors, they have to debug them at the level of wires and registers.

This work is open source: https://capra.cs.cornell.edu/calyx/

The fundamental problem of waveform debugging is that RTL lacks control-flow information. In every cycle, any port *may* change and contribute to the bug; this means that such tools must necessarily capture the entire state of the design and present this overwhelming information to the user. When trying to isolate a bug, the user is initially faced with every single wire and register's state at every time step. While tractable in small cases, this approach is hard to scale: a simple 8×8 matrix multiplier can take thousands of cycles. Without some insight about the bug, identifying the problem, and localizing the root cause, is vastly more complicated than it should be. If small linear algebra kernels are this unapproachable, debugging a full neural network accelerator is nigh impossible. Contrast this experience to debugging a software program where a user can a debugger to step execution, create breakpoints, and view intermediate results while executing their program. These abstractions of software debuggers, however, are difficult to port to RTL languages since they are fundamentally attached to abstractions such as clock cycles instead of control structures such as loops.

Our solution to the problems of reusability and debugging is building a new toolkit and ecosystem based on the Calyx [3] infrastructure (Figure 1). Calyx, like LLVM, uses a small but expressive intermediate language (IL) that can describe, optimize, and lower hardware accelerators specified in a multitude of DSLs. The key innovation of Calyx is a separation between the control-flow flow and structural representation of an accelerator design. By explicitly encoding the control flow information, Calyx enables novel, reusable optimizations (Section III) as well as an interactive, software-like debugging tool (Section IV).

II. The Calyx Intermediate Language

Calyx's mixed representation of control and structure, along with a precise characterization of isolation and timing properties enables Calyx to support optimizations that can transform both low-level structural aspects of the design, such as the finite-state machine encoding, to higher-level properties such as the amount of parallelism and resource-sharing.

A Calyx program consists of components which encapsulate the structure and control flow of a sub-circuit. They correspond to hardware modules in an RTL design. Each component defines its input and output ports along with three sections: (1) **`cells`** which declare the subcomponents used by a component, (2) **`wires`** which describe the connection between ports of subcomponents, and (3) **`control`** which describes the control-flow of the component using a software-like imperative language.

The following Calyx component defines a component that doubles the value of its input:

```
component double(in: 32) -> (out: 32) {
  cells {
    adder = std_add(32);
  }
  wires {
    adder.left = in; adder.right = in;
    out = adder.out;
  }
  control { }
}
```

The program defines the subcomponent `adder` which is an instance of the `std_add` module. Next, it accepts inputs using the 32-bit port `in`, and generates a 32-bit output `out`. The input port of the component is connected to the inputs the adder while the output port is connected to the output of the adder `adder.out`. Since the **`control`** section is empty, this component is *purely-structural* and looks very similar to RTL languages.

A. Groups and Control

Calyx uses the **`group`** keyword to define a collection of assignments that collectively perform some action. For example, the following group definition increments the value in a register `r`:

```
group incr {
  adder.left = r.out;
  adder.right  = 32'd1;
  r.in = adder.out;
  r.write_en = 1'd1;
  incr[done] = r.done;
}
```

Assignments within a group act like any other set of assignments: they are continuously active and can operate over multiple clock cycles. However, by defining a *done condition*, the group can state when the assignments have performed a meaningful action such as increment a register. Once the action is performed a group can be stopped from executing the assignments.

The ability to control the execution of groups is provided by the **`control`** program. For example, the following control program will increment the value of the register `r` three times by executing the assignments in `incr` thrice:

```
control { seq { incr; incr; incr; } }
```

The **`seq`** operator executes the groups sequentially. In order to know when a group is done with its computation, the **`seq`** operator uses the *done condition* of the group. Calyx supports several control operators: **`seq`** for sequencing execution, **`par`** for parallel execution, **`if`** for conditional execution, **`while`** for encoding loops, and **`invoke`** for function call-like semantics.

The combination of groups and control operators allows hardware DSLs to easily encode their semantics into hardware. For example, our frontend for an imperative, loop-based language [4] can directly map its control flow into Calyx operators. While it only has five operators, we've used Calyx to develop a several DSLs that can generate a wide variety of architectures: an imperative, loop-based language, a systolic array generator, a frontend for TVM [5], and an implementation for a ternary content addressable memory (TCAM).

For a longer overview on designing a frontend with Calyx, please refer to the Calyx documentation.[1]

III. Reusable Optimizations

Along with an intermediate representation, Calyx implements a modular, pass-based compiler infrastructure similar to LLVM [2]. Unlike hardware ILs such as FIRRTL [6], LLHD [7], or LNAST [8], Calyx explicitly encodes the control-flow of the hardware design. Unlike software ILs such as LLVM, Calyx explicitly represents the structural aspects of the design, such as which adder performs which computation. In this way, Calyx can injest software ILs as frontends and emit hardware ILs as backends.

Calyx's mixed representation enables it to perform optimizations that utilize both the control-flow information as well as the structural information. We overview one such optimization called *register unsharing* which attempts to tradeoff logic for memory in the final design by instantiating additional registers when possible.

Unlike traditional software compilers, where minimizing the number of used registers is always optimal, in hardware designs, the target dictates the profitability of register-reuse. For example, on a traditional FPGA, each slice may contain a stateful element making registers readily available compared to complex multiplexers. On the other hand, on ASIC processes, logic becomes relatively cheap while registers become expensive.

[1]https://capra.cs.cornell.edu/docs/calyx/tutorial/frontend-tut.html

Because of this, we implemented a pass to "unshare" or separate the uses of registers when possible. In conjunction to Calyx's already available implementation of a register sharing pass, this enables fine-grained memory-logic trade-offs by DSL designers and application users. In pseudocode, the optimization can perform the following transformation:

```
let x = 10; x := x + 1; x := x + 10;
```

into:

```
let x = 10; let y = x + 1; let z = y + 10;
```

In the first program, there are three writes to the register `x` which means the generated hardware will instantiate a $4 : 1$ multiplexer for one register. The second program instead instantiates three registers and no multiplexers whatsoever.

The core implementation of the optimization first calculates the reaching definitions of registers and then uses this information to rewrite future register uses. In traditional software optimization parlance, a definition is *killed* when it is overwritten. For example, the statement `x := x + 1` *kills* the original write to `x` making this statement available for register unsharing.

The analysis uses groups to calculate when a register definition is killed:

```
group write_x {
  x.in = 32'd10; x.write_en = 1'd1;
  write_x[done] = x.done;
}
```

Since the port `x.in` is used in this group, the pass can infer that any previous definition of `x` has been killed by this pass. Next, it uses the control program to analysis when a particular register definition can be rewritten:

```
seq {
  write_x; read_x_and_y;
  write_y_to_x; read_x;
}
```

In this control program, if the group `read_x_and_y` makes use of `x`, its definition cannot be rewritten. However, if `write_y_to_x` overwrites the value of `x`, then uses of `x` in `read_x` can be rewritten. The Calyx implementation of this pass uses a parallel CFG [9] representation to make it work with general Calyx programs.

In this way, Calyx's explicit representation of control and structural information enables unique control-flow-oriented hardware optimizations. Calyx's open source implementation features 27 passes that validate, optimize, and lower Calyx programs for simulation, synthesis, or FPGA execution.

IV. Interactive Debugger

While the Calyx ecosystem already supports running Calyx programs through both simulation and FPGA execution, debugging Calyx programs remains challenging—programs must first be lowered to RTL, losing the rich control-flow information encoded in the Calyx level and rely on waveform or FPGA debugging techniques. Our key observation is that by leveraging the high-level control flow representation, we can enable Calyx program to both simulate faster and provide software-like debugging features. To this end, we've implemented the Calyx interactive debugger (CIDR) which can simulate Calyx programs and provide a software-like debugging experience with breakpoints, stepping execution, and printing out intermediate values. By directly building this system on top of the Calyx infrastructure, we provide added benefits to frontends using Calyx—they can immediately provide their users a debugging experience that does not require understanding the semantics of RTL programs.

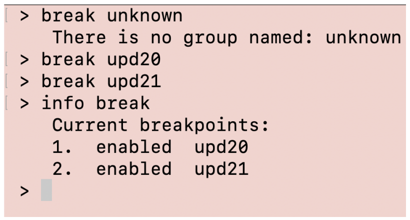
CIDR is able to sidestep some of the conventional challenges of hardware simulation and debugging by dint of Calyx's representation. Efficient hardware simulation [10] relies on the fact that while a hardware *may* have most of its assignments active every cycle, in practice, a majority of them do not change every cycle. By leveraging these low-activity factors in RTL designs, efficient simulators can avoid having to compute every single assignment every cycle. The challenge, however, is that the information of which assignments are inactive in a given cycle is not present in the source RTL program. This forces simulators to develop complex heuristics which identify inactive regions and avoid simulating them to save resources; however, these heuristics are necessarily conservative and so simulators must fall back to simulating these regions, lest the simulator fail to preserve the semantics of the accelerator. Thus RTL's structure, or lack thereof, imposes a clear cost in both runtime and code complexity.

In contrast, Calyx's control program encode *exactly* which assignments are active in a given clock. For example, if a program uses Calyx's **`seq`** operator, the Calyx program can infer that only the set of assignments in current group is going to be active. This means that CIDR knows, at all times, which parts of the accelerator are running and how they relate to each other as well as the rest of the execution schedule. This removes some of the complexity of simulation and suggests the possibility of a fast simulator for Calyx without the need for complex heuristics. Table I reports on preliminary experiments comparing the end-to-end simulation times for Verilator [11] and CIDR using the Polybench [12] benchmarks. Because CIDR is a "pure" interpreter, it sidesteps the compilation phase required for Verilator simulation: the running times in the table for Verilator include C++ generation, C++ compilation, and simulation execution. In these benchmarks, CIDR consistently outperforms Verilator. While larger benchmarks are needed to create a more complete picture, the initial results are promising.

Beyond the potential to increase simulation speeds, this additional control flow information means that CIDR programs are more closely related to source-level programs generated by frontends. This enables CIDR to provide *structured debugging* of source-level programs. We demonstrate CIDR's capabilities using three features: stepping execution, breakpoints, and inspecting intermediate values. A traditional RTL simulator can support execution stepping at the granularity of cycles since it does not have access to any additional control information. In contrast, CIDR can step execution at the level of both

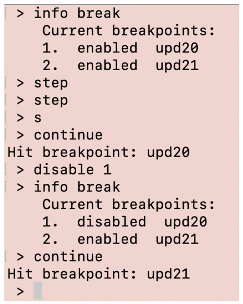
Benchmark	CIDR (seconds)	Verilator (seconds)
symm	1.8 ± 0.003	10.4 ± 0.064
gesummv	1.7 ± 0.013	15.2 ± 0.142
trisolv	1.5 ± 0.023	9.6 ± 0.174
3mm	2.3 ± 0.018	12.4 ± 0.119
bicg	1.5 ± 0.019	9.8 ± 0.083
trmm	1.5 ± 0.015	9.6 ± 0.050
2mm	2.0 ± 0.050	14.0 ± 0.384
durbin	1.5 ± 0.012	10.4 ± 0.013
gemver	1.9 ± 0.023	18.0 ± 0.502
syrk	1.8 ± 0.018	10.5 ± 0.077
mvt	1.7 ± 0.016	14.4 ± 0.137
syr2k	2.1 ± 0.014	11.5 ± 0.092
doitgen	2.9 ± 0.002	9.4 ± 0.036
ludcmp	1.6 ± 0.010	14.7 ± 0.089
gemm	1.8 ± 0.006	10.8 ± 0.069
atax	1.8 ± 0.016	12.2 ± 0.383
lu	1.5 ± 0.017	9.9 ± 0.059

TABLE I: End-to-end simulation time for Polybench benchmarks averaged over 10 runs.



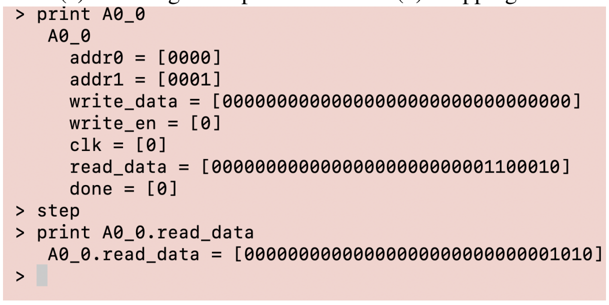
```
> break unknown
   There is no group named: unknown
> break upd20
> break upd21
> info break
   Current breakpoints:
   1.  enabled  upd20
   2.  enabled  upd21
>
```

(a) Enabling breakpoints.



```
 > info break
    Current breakpoints:
    1.  enabled  upd20
    2.  enabled  upd21
 > step
 > step
 > s
 > continue
Hit breakpoint: upd20
 > disable 1
 > info break
    Current breakpoints:
    1.  disabled  upd20
    2.  enabled  upd21
 > continue
Hit breakpoint: upd21
 >
```

(b) Stepping execution.



```
> print A0_0
   A0_0
      addr0 = [0000]
      addr1 = [0001]
      write_data = [00000000000000000000000000000000]
      write_en = [0]
      clk = [0]
      read_data = [00000000000000000000000001100010]
      done = [0]
> step
> print A0_0.read_data
   A0_0.read_data = [00000000000000000000000000001010]
>
```

(c) Printing intermediate values.

Fig. 2: The interface for Calyx Interactive Debugger (CIDR).

cycles and *groups*. This allows users to step over whole set of assignments that they know to be behaving correctly: for example if the group that increments a register works correctly, the user can immediately step over *all* the assignments and cycles needed to run the group. Figure 2b shows how a user can step through execution of groups. Next, CIDR support breakpoints at the level of groups and control programs (Figure 2a). Again, unlike RTL simulators which can only provide breakpoints when a particular value changes, CIDR accomplishes this using Calyx's mixed representation. Finally, CIDR enables inspection of intermediate values in each port of a subcomponent. Figure 2c shows how a user can output the value of a port during execution. While this is similar to capabilities provided by RTL and waveform debugging, since CIDR is capable of showing the user only the portions of the circuit which are currently relevant, it greatly reduces the cognitive burden of looking through dozens of port signals to figure out which ones might be contributing to the current bug.

Open-source infrastructure like CIDR is a boon for compiler developers and users of hardware DSLs. Domain-specific users can immediately run their code and receive feedback without needing to understand the semantics of low-level RTL programs and use a software-like debugging features when they run into problems. Compiler developers can focus on designing new abstractions and features for their DSLs instead of spending time on designing a debugging experience from scratch. Instead, they can extend the CIDR to support different debugging modes that present to users information in a domain-specific manner—for example using port information to reconstruct arrays or tensors present in the surface language.

Finally, CIDR makes it easier to develop and maintain the Calyx compiler infrastructure itself. Unlike many compilers, the Calyx compiler is modular—composed of many distinct passes which, in concert, transform Calyx programs. Simulating Calyx programs means that it is possible to perform differential testing of the program's behavior under different passes to validate that the passes preserve semantics in the appropriate way. While Verilator simulation supports this methodology, it is hard to know if individual passes are correct since the simulation path necessarily invokes the lowering passes, any of which could potentially obscure or introduced a bug. Source-level simulation gets rids of this requirements and enables testing of Calyx programs at all levels.

V. Future Work

Calyx's mixed representation presents several opportunities to advance the state of the art and build robust, open-source tools. We are currently focused on three thrusts of Calyx development: First, we are building a robust simulation and debugging infrastructure that allows DSL developers to connect source-level semantics to the generated hardware designs. We plan to add features such as rematerializing language-level structures so that users of the Calyx debugger can view data at the level of data structures instead of ports. Second, we are integrating Calyx with the CIRCT [13] ecosystem which represents a united effort to build an open-source hardware infrastructure. While we've already made substantial progress, by making CIRCT a first-class citizen in the Calyx ecosystem, we will enable the wider software compiler to build robust hardware accelerator generators. Finally, we plan to formalize the semantics of Calyx and implement automatic verification technologies on top of it. Such verification tools can benefit from the explicit control-flow representation of Calyx to scale better than purely hardware-focused verification tools.

VI. Conclusion

Calyx represents a step towards a future where designing hardware accelerator takes a weekend instead of months. We believe that by bridging the gap between domain-experts and hardware accelerators represents a step forward in the ambitious goal of accelerator ubiquity.

References

[1] Google, “SkyWater open source PDK,” 2021, https://github.com/google/skywater-pdk.

[2] C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation,” in *International Symposium on Code Generation and Optimization (CGO)*, 2004.

[3] R. Nigam, S. Thomas, Z. Li, and A. Sampson, “A compiler infrastructure for accelerator generators,” in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[4] R. Nigam, S. Atapattu, S. Thomas, Z. Li, T. Bauer, Y. Ye, A. Koti, A. Sampson, and Z. Zhang, “Predictable accelerator design with time-sensitive affine types,” in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2020.

[5] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, “{TVM}: An automated end-to-end optimizing compiler for deep learning,” in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[6] A. M. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach, “Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations,” in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2017.

[7] F. Schuiki, A. Kurth, T. Grosser, and L. Benini, “LLHD: A multi-level intermediate representation for hardware description languages,” in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2020.

[8] S.-H. Wang, A. Sridhar, and J. Renau, “LNAST: A language neutral intermediate representation for hardware description languages,” in *Second Workshop on Open-Source EDA Technology (WOSET)*, 2019.

[9] D. Grunwald and H. Srinivasan, “Data flow equations for explicitly parallel programs,” in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1993.

[10] S. Beamer and D. Donofrio, “Efficiently exploiting low activity factors to accelerate rtl simulation,” in *Design Automation Conference (DAC)*, 2020.

[11] Veripool, “Verilator,” 2021, https://www.veripool.org/wiki/verilator.

[12] Louis-Noel Pouchet. (2021) PolyBench/C: The Polyhedral Benchmark Suite. [Online]. Available: http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/

[13] The CIRCT authors, “CIRCT: Circuit IR compilers and tools,” 2021, https://github.com/llvm/circt.