Resource Sharing on Hardware Accelerators through Control-Flow Based Optimizations

Caleb Kim
Cornell University
USA

1 PROBLEM AND MOTIVATION

Domain-Specific Languages (DSLs) are useful for accelerator design, as they provide high-level abstractions in exchange for a more limited application [13] or architectural [4] domain. Calyx [11] is a shared intermediate language for DSL-to-hardware compilers. DSLs target Calyx, before Calyx optimizes the design and lowers it to synthesizable RTL. One optimization is resource sharing, which collapses multiple copies of the same hardware into a single module. We demonstrate resource sharing using techniques traditional to software compilers, such as live-range analysis, dominator analysis, and inlining. This is possible due to Calyx's software-like control flow and hardware-like structure (i.e., explicit representation of hardware modules).

2 RELATED WORK

2.1 The Calyx Language

Figure 1 shows function `my_fun` and its Calyx equivalent. Calyx designs consist of *components*, which have input and output ports that define their interface (Figure 1b, line 1).

The `cells` section contains the submodules for the component. `my_fun` instantiates six 32-bit primitive cells (lines 2–6).

The `wires` section contains unordered, continuous assignments that wire together cell ports. Assignments can be organized into groups. Group `assign_y` (lines 8–11) corresponds to `let y = x1 + 4` (Figure 1a, line 2).

The `control` section handles execution: `seq` for sequential execution, `par` for parallel execution, `invoke` for "calling" cells (e.g., line 23), `if` statements, and `while` loops.

```
1   fn my_fun(x1: int32, x2: int32) {
2       let y = x1 + 4;
3       let z = (y * 2) + (x2 * 3);
4       return z;
5   }
```

(a) Software Langauge

```
1   component my_fun(x1: 32, x2:32) -> (return:32) {
2     cells{
3       y = reg(32); z = reg(32);
4       add1 = add(32); add2 = add(32);
5       mult1 = mult(32); mult2 = mult(32);
6     }
7     wires{
8       group assign_y {
9         add1.left = x1; add1.right = 4; y.in = add1.out;
10        y.go = 1; assign_y[done] = y.done;
11      }
12      group assign_z {
13        add2.left = mult1.out; add2.right = mult2.out;
14        z.in = add2.out; z.go = 1;
15        assign_z[done] = z.done;
16      }
17      return = z.out;
18    }
19    control{
20      seq {
21        assign_y;
22        par{
23          invoke mult1(left = y.out,right = 2);
24          invoke mult2(left = x2; right = 3);
25        }
26        assign_z;
27      }
28    }
29  }
```

(b) Calyx

Figure 1: Software function and Calyx equivalent

2.2 Related Approaches

Many resource sharing approaches, such as SDC [3], first establish resource (and other) constraints before checking whether a design can be generated given those constraints. This approach is monolithic; *all* optimizations (resource sharing, timing, etc.) are required for compilation, meaning we cannot isolate optimizations for verification, debugging, etc. Calyx uses a pass-based, LLVM-like [8] compiler. Passes, which take in and then emit valid Calyx code, can be skipped, added, and rearranged.

Vericert-Fun [12] shares functions rather than individual operators. However, functions that (1) call functions or (2) load/store memory are inlined rather than shared. Inlining increases the number of instructions in a given function, which increases the complexity of the FSM coordinating its execution, potentially hurting performance [2, 10].

Prior to our work, Calyx had a register sharing pass to minimize register usage [11]. We improve this pass by (1) generalizing sharing to arbitrary Calyx components and (2) providing the ability to share across inlined components without blowing up FSM complexity.

2.3 Register Sharing in Calyx

Calyx's original register sharing pass worked by constructing live ranges for each register and using a greedy coloring algorithm to share registers with non-overlapping ranges [11]. In Figure 1b, `y` is dead by line 25, while `z` is not live until line 26. Therefore, `y` and `z` can be shared.

To compute liveness, Calyx mostly uses a standard data-flow formulation. However, `par` blocks are unique since each thread *must* execute (unlike `if` branches, which *may* execute). Therefore, Calyx introduces p-nodes, which correspond to `par` blocks and recursively contain CFGs corresponding to the `par` block's threads [11].

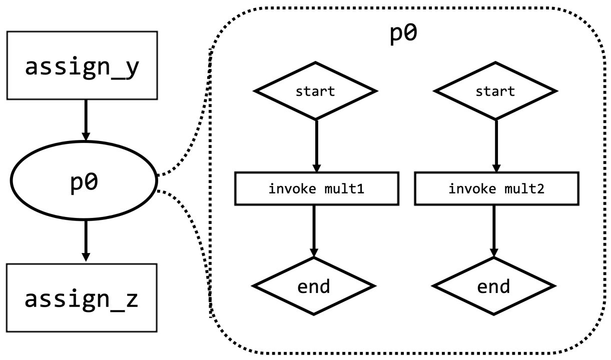


Figure 2: CFG for Figure 1b. Since `p_0` is a p-node, `assign_z` is dominated by *both* `invoke` statements.

3 APPROACH AND UNIQUENESS

3.1 Component Sharing

Sharing arbitrary Calyx components (multipliers, non-primitive components like `my_fun`, etc.) lets us share hardware modules other than registers. However, this presents an additional challenge: while registers completely overwrite their state at each use, other components may be more complicated. Consider a hypothetical component `counter` that counts the number of times it has been invoked. Since its state is not *completely* overwritten at each use, (e.g., its current state being 1 still affects its next state—2), liveness is not a useful concept for determining when we should share `counters`. Therefore, we should not share such components.

We implement a separate pass to conservatively detect if a component is *shareable* (i.e., state is *completely* overwritten at each use), using the following criteria: (1) it only instantiates cells that are themselves shareable (2) in its control flow, any *possible* read from a stateful (i.e., non-combinational) cell *must* be preceded by a write to it.

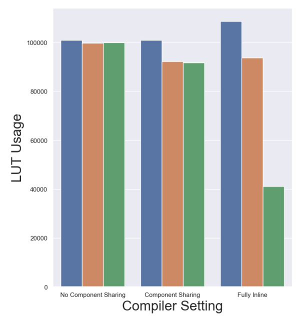
Property (1) is easily checked. To check (2), we implement a mostly traditional dominator analysis [1]. However, since each `par` thread *must* execute, we take the union—rather than intersection—of dominators when predecessors are ends of p-node threads (see Figure 2).

3.2 Bounded Sharing

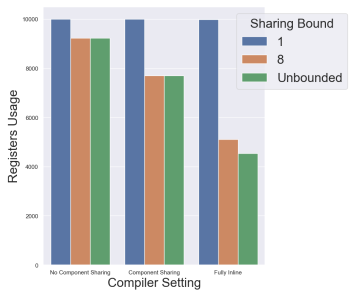
Sharing comes with tradeoffs. While sharing components decreases usage of that component, it also requires new MUXes to determine when and where to wire that component's ports [5]. Therefore, we implement sharing bounds, letting us specify the maximum number of times a cell can be shared. We do this by bounding the total number of times any color can be used in the greedy coloring algorithm from Section 2.3.

3.3 FSM-Aware Inlining

In Calyx, we implement inlining by replacing the `invoke` of a component with its control flow, which lets us share cells instantiated within different "parent" components. To address the increased FSM complexity that comes with inlining [2, 10], we introduce an annotation to Calyx control statements that instructs the compiler to generate separate FSMs. This makes the inlined control flow visible to the sharing pass, while generating circuitry as if the components were still separate. Calyx's semantics are ideal for this technique, which requires both software-like control-flow (for inlining) and hardware-like structure (for FSMs).

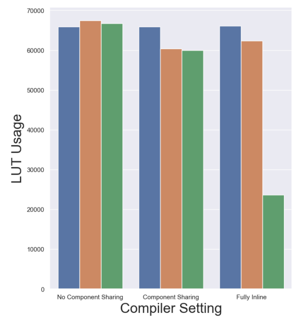


(a) LUT Usage

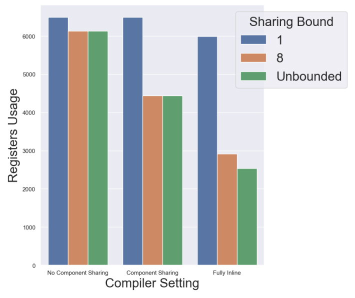


(b) Register Usage

Figure 3: GoogleNet Resource Usage



(a) LUT Usage



(b) Register Usage

Figure 4: MobileNet Resource Usage

4 RESULTS

We use Calyx to simulate inference for six neural networks: AlexNet [7], GoogleNet [16], LeNet [9], MobileNet [14], SqueezeNet [6], and VGG [15].

For each design, each operation (e.g., conv2D) is compiled into a Calyx component, and is properly invoked using Calyx's control flow semantics. Calyx compiles each design in less than four seconds. We measured resource estimates across two variables:

(1) compiler setting:
   (a) without component sharing
   (b) with component sharing
   (c) FSM-aware inlining then sharing
(2) sharing bound:
   (a) 1
   (b) 8
   (c) unbounded

Figure 3 and Figure 4 show resource usage (as reported by Vivado 2020.2) for GoogleNet and MobileNet, though the trends hold across all designs. Increasing sharing bound has a more drastic effect on LUT/register usage after the introduction of component sharing. Resource usage is smallest overall when designs are inlined and sharing is unbounded. Interestingly, unbounded sharing does not increase LUT usage. Analyzing the synthesized Verilog, we hypothesize that unbounded sharing simplifies logic for FSMs because they read from fewer hardware modules.

REFERENCES

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison-Wesley Longman Publishing Co., Inc., USA.

[2] Benjamin Carrion Schafer. 2015. Process selection for maximum resource sharing in High-Level Synthesis. In *Electronic System Level Synthesis Conference (ESLsyn).*

[3] J. Cong and Zhiru Zhang. 2006. An efficient and versatile scheduling algorithm based on SDC formulation. In *Design Automation Conference (DAC).*

[4] David Durst, Matthew Feldman, Dillon Huff, David Akeley, Ross Daly, Gilbert Louis Bernstein, Marco Patrignani, Kayvon Fatahalian, and Pat Hanrahan. 2020. Type-Directed Scheduling of Streaming Accelerators. In *Conference on Programming Language Design and Implementation (PLDI).*

[5] Stefan Hadjis, Andrew Canis, Jason H. Anderson, Jongsok Choi, Kevin Nam, Stephen Brown, and Tomasz Czajkowski. 2012. Impact of FPGA Architecture on Resource Sharing in High-Level Synthesis. In *International Symposium on Field Programmable Gate Arrays.*

[6] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. *arXiv preprint arXiv:1602.07360* (2016).

[7] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2017. ImageNet Classification with Deep Convolutional Neural Networks. *Commun. ACM* (2017).

[8] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization (CGO).*

[9] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* (1998).

[10] Marco Minutoli, Vito Giovanni Castellana, Antonino Tumeo, and Fabrizio Ferrandi. 2015. Inter-procedural resource sharing in High Level Synthesis through function proxies. In *International Conference on Field Programmable Logic and Applications (FPL).*

[11] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. 2021. A compiler infrastructure for accelerator generators. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[12] Michalis Pardalos, Yann Herklotz, and John Wickerson. 2022. Resource Sharing for Verified High-Level Synthesis. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM).*

[13] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. 2017. Programming Heterogeneous Systems from an Image Processing DSL. *ACM Trans. Archit. Code Optim.* (2017).

[14] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *Conference on Computer Vision and Pattern Recognition (CVPR).*

[15] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[16] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going Deeper With Convolutions. In *Conference on Computer Vision and Pattern Recognition (CVPR).*