Creating An Agile Hardware Flow

Rick Bahr, Clark Barrett, Nikhil Bhagdikar, Alex Carsello, Nate Chizgi, Ross G Daly, Caleb Donovick, David Durst, Kayvon Fatahalian, Pat Hanrahan, Teguh Hofstee, Mark Horowitz, Dillon Huff, Taeyoung Kong, Qiaoyi Liu, Makai Mann, Ankita Nayak, Aina Niemetz, Gedeon Nyengele, Priyanka Raina, Steve Richardson, Raj Setaluri, Jeff Setter, Daniel Stanley, Maxwell Strange, James Thomas, Leonard Truong, Xuan Yang, Keyi Zhang

***Abstract*—Although an agile approach is standard for software design, how to properly adapt this method to hardware is still an open question. Stanford's AHA (Agile HArdware) Project is working toward this goal while building SoCs with specialized accelerators. Rather than using a waterfall design flow, which typically starts by studying the application to be accelerated, we began by constructing a complete flow from application DSL to a generic coarse-grained reconfigurable array (CGRA), the first version of which we fabricated last year. Once we had a working system, we started working on ways to tune the application code, compiler, and CGRA to increase the efficiency of the resulting implementation. Our need to continually update parts of the system while maintaining the end-to-end flow required the use of hardware generators which not only provide the Verilog needed for the implementation, but also create the collateral that the compiler/mapper/P&R system needs.**

I. Introduction

Since the start of the ASIC revolution in the mid 1980s, the design tools, IP building blocks, and methodology used to create complex digital chips have improved dramatically, enabling us to create the modern billion plus transistor SoCs we use everyday. While the improved design methodology does allow us to build these complex chips, the time from design start to working system (with complete software) takes many years, and costs hundreds of millions of dollars [1].

Interestingly, the general approach toward designing an accelerator during this time has not changed: architecture students are still taught to study the application, then use that information to create an architecture to accelerate the application, then build the hardware, and then map the applications to this hardware. This strategy is very similar to the *waterfall* design approach in software, where one starts with a spec, and then goes through a number of levels of refinement.

The waterfall model suffers from twin issues of changing application requirements and incomplete knowledge/understanding of the problem, making the resulting system less useful than desired. So software designers have moved towards a more agile approach, aiming to create a simple end-to-end system with good interfaces, and then to work on improving the most pressing problems in that resulting system.

Stanford's AHA Center is exploring how best to incorporate agile design methodology in a hardware flow. The goal is to incrementally update hardware and software in an end-to-end flow to create an accelerator, rather than using the waterfall model described above. To this end, we spent most of last year creating a flow which can construct CGRA hardware along with the software infrastructure for mapping arbitrary Halide [2] applications to the CGRA for execution.

Clear interfaces (e.g. APIs in software), at the right level of abstraction, are essential for agile designs. For the CGRA, we wanted something like an ISA of a processor, to allow both the hardware and software to improve concurrently. Since most accelerators, including our CGRA, are spatial machines that exploit data parallelism and locality, we chose a dataflow representation of the algorithm as our *spatial ISA*. Thus the compiler assumes the hardware can execute any static dataflow graph, and the CGRA needs to provide tools that can map any static flow graph onto its fabric. We use CoreIR [3] to represent these graphs and have built a CoreIR targeting compiler, a set of tools for optimizing the generated CoreIR, and a mapper to map it to our CGRA. Many other groups have also chosen dataflow IRs in their hardware systems, including FIRRTL at UCB [4], and HPVM at UoI [5].

Building *Jade*, our first system, exposed a number of issues we did not initially consider. We are addressing these issues in the next generation of our design flow, *Garnet*, which we plan to tape out in June 2019. The main improvements in *Garnet* are:

- Maintaining end-to-end flow by insisting on a single source of truth for each stage in the flow. Thus each stage must inform both upstream and downstream tools of its requirements/capabilities. This facilitates better modularity as stages are updated throughout the design process.
- Enabling a separation of concerns by allowing passes on a generator's output to change the structure of the logical design to facilitate placement, power efficiency, and testing.

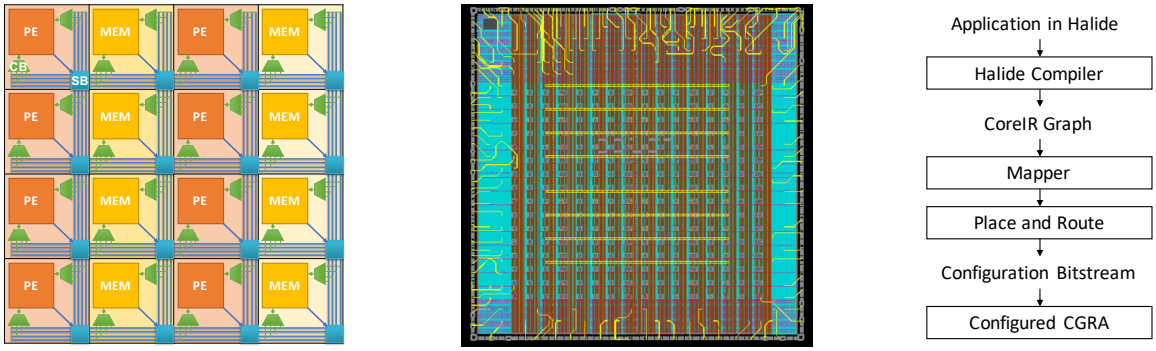
II. Jade

Our goal for **Jade** was to create the simplest system that could map an application to hardware. Fortunately, prior research created many tools we could build upon for this effort. The compiler infrastructure leveraged our previous work on mapping Halide imaging applications to hardware [6]; the hardware generator used Genesis2 [7] and Magma [8]; and our CGRA design was built off of knowledge gained in studying FPGAs [9] [10]. We chose a CGRA as our base architecture, since it is well matched to the type of spatial specialization we needed, and because our prior work indicated it was one of the most energy and area efficient programmable architectures possible [9].

We started with a very simple island-style design containing only two types of tiles: memory (MEM) and processor (PE). For interconnect, we used switch boxes (SB) to connect 16- and 1-bit wiring tracks to each other, and to connect outputs to a wiring track. Connection boxes (CB) selected which wiring track connects to which memory or processor input port. All these units were configured through a JTAG interface on the chip. Figure 1 shows the basic chip architecture. Our *Jade* tool flow compiles Halide applications to CoreIR, optimizes this representation, maps it to the CGRA, places and routes the mapped units on the CGRA, and finally generates the configuration bitstream to program the hardware with this mapping and routing. This flow is tested continuously.

Jade itself is written in Genesis2 [7], a hardware generation framework that uses Perl to meta-program hardware modules written in SystemVerilog. It was taped-out in Summer 2018, and we received packaged parts in January. These parts were fully functional, and our tool chain has successfully mapped and run Halide applications on this design.

While this tool chain works, as development progressed, its limitations became clear. While the blocks we created were flexible,



(a) CGRA Architecture (b) CGRA Die (c) Application Toolflow

Fig. 1. CGRA architecture and final chip. The architecture uses an island style design. (a) Each tile has a switch box (blue square) to connect outputs to tracks, and a connection box (green trapezoid) to connect its inputs to tracks. (b) CGRA die. (c) Series of steps to map a Halide application to our CGRA.

other blocks in the system needed to "know" those parameters, so it often took manual effort to update and share these parameters between different modules. Even when we made our generators automatically create the needed "configuration files," this meta file generation was separate from the hardware generation, and was not guaranteed to track the actual hardware design. Additionally, when we first tried to tape-out our chip, we realized that we needed to revise the structure of our logical design to make it more amenable to physical implementation. Going forward, we want to be able to keep physical concerns separate from our logical description. The refactoring for *Garnet*, our next version of the system, focuses on addressing these issues.

III. Garnet

In order to solve the problems with our first design, we are creating our next CGRA, **Garnet,** from a set of executable specifications that serve as a *single source of truth* for both generating the hardware itself and for generating the collateral that our P&R tools need to map applications to the CGRA. Generating arbitrary hardware from arbitrary higher-level specifications is an extremely difficult problem, but when we restrict the problem to generating only a specific type of hardware, such as an ALU, the problem becomes tractable. Thus, in order to generate our CGRA, we have created a set of domain-specific languages to describe each of the main components present in our CGRA, and also in many other hardware accelerators:

- **PEak** processing element specification language. From a PEak program, we generate an executable functional model as well as the RTL. We also utilize Satisfiability Modulo Theories (SMT) solvers to synthesize the mapping from our CoreIR hardware primitives to the generated processing element. This step automatically creates a backend code generator that maps CoreIR to the capabilities of our hardware block.
- **Karst** memory specification language. Designers can define the required storage element behaviors (queue, line buffer, double buffer, etc.). Using these definitions plus information about the physical memory macros, it creates the controller around that memory macro. It also synthesizes the tool to map larger memory structures onto a number of these blocks.
- **Canal** interconnect specification language. A Canal program takes a set of PE and memory cores generated by PEak and Karst, respectively, and a high-level specification of an interconnection network topology. It then realizes the network and snaps the cores into designer-specified locations within the network. This framework allows designers to quickly and easily explore various interconnect topologies, and it automatically produces correct collateral for placing and routing applications.

Feeding our Karst and PEak-generated cores to Canal gives us a grid of tiles in a reconfigurable interconnect. To finish the CGRA, we need to compose this interconnect with other components, such as a configuration controller and I/O pads.

In the new design flow, a change in the design of a component will automatically propagate through the flow to affect dependent components without manual intervention. For instance, if we add floating point instructions to the PE in PEak, it automatically generates the mapper and RTL that support floating point. Thus, we can easily map applications that use floating point numbers directly to the new hardware in the end-to-end flow. In addition, if we adjust the interconnect topology in Canal, co-designed P&R tools can automatically digest the new topology and produce updated configuration bitstreams without touching the PE or memory design.

In order to construct this final generator, we created **Gemstone**, a general infrastructure for creating hardware generators: PEak, Karst, Canal are Gemstone generators. In addition to this, Gemstone enables designers to write passes which take in an existing design (output from an evaluation of an embedded generator) and output a transformed version. For example, we can write passes on a design to embed some global signals into local blocks, move or duplicate logic to minimize wiring, or add power domains while maintaining a clean logical description at the generator level. Passes enable enhanced separation of concerns within a design specification.

IV. Conclusion

To facilitate agile hardware, one needs tools that help to maintain the end-to-end flow. This requires not only hardware generators and clean interfaces, but also methods to communicate changing design features without a designer's manual intervention. Our Gemstone framework and associated DSLs address these concerns by allowing the designer to separately deal with different concerns, and seamlessly communicate changing design capability to all the tools in our flow. The result is an approach to agile hardware design that allows rapid integration of changing components, and shorter design cycles. To encourage more discussion on agile hardware design, we have completely open-sourced both generations of our hardware design, software tool chain, as well as the Halide applications.

REFERENCES

[1] E. Sperling, "How Much Will That Chip Cost?" semiengineering.com/how-much-will-that-chip-cost/, March 2014, [Online].

[2] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *SIGPLAN Not.*, vol. 48, no. 6, pp. 519–530, Jun. 2013.

[3] R. Daly, L. Truong, and P. Hanrahan, "Invoking and linking generators from multiple hardware languages using CoreIR," in *Workshop on Open-Source EDA Technology (WOSET)*, no. 11, 2018.

[4] A. 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 *Proc. 36th Int'l Conf. on Computer-Aided Design*, ser. ICCAD '17. IEEE, 2017, pp. 209–216.

[5] M. Kotsifakou, P. Srivastava, M. Sinclair, R. Komuravelli, V. Adve, and S. Adve, "HPVM: Heterogeneous parallel virtual machine," in *PPoPP 2018 - Proceedings of the 23rd Principles and Practice of Parallel Programming*. ACM, 2 2018, pp. 68–80.

[6] J. Pu, S. Bell, X. Yang, J. Setter, S. Richardson, J. Ragan-Kelley, and M. Horowitz, "Programming heterogeneous systems from an image processing DSL," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 3, pp. 26:1–26:25, Aug. 2017.

[7] O. Shacham, O. Azizi, M. Wachs, W. Qadeer, Z. Asgar, K. Kelley, J. P. Stevenson, S. Richardson, M. Horowitz, B. Lee, A. Solomatnikov, and A. Firoozshahian, "Rethinking digital design: Why design must change," *IEEE Micro*, vol. 30, no. 6, pp. 9–24, Nov 2010.

[8] P. Hanrahan, "Magma github," https://github.com/phanrahan/magma/, [Online].

[9] A. Vasilyev, N. Bhagdikar, A. Pedram, S. Richardson, S. Kvatinsky, and M. Horowitz, "Evaluating programmable architectures for imaging and vision applications," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–13.

[10] J. Luu, I. Kuon, P. Jamieson, T. Campbell, A. Ye, W. M. Fang, K. Kent, and J. Rose, "VPR 5.0: FPGA CAD and architecture exploration tools with single-driver routing, heterogeneity and process scaling," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 4, no. 4, pp. 32:1–32:23, Dec. 2011. [Online]. Available: http://doi.acm.org/10.1145/2068716.2068718