Automated Codesign of Domain-Specific Hardware Accelerators and Compilers

Priyanka Raina, Fredrik Kjolstad, Mark Horowitz, Pat Hanrahan, Clark Barrett, Kayvon Fatahalian
Stanford University, praina@stanford.edu

Topics: Architectures, applications, programming systems and codesign methodologies.

I. Challenge

With the slowdown in technology scaling, domain-specific hardware accelerators will play a major role in improving the performance and energy-efficiency of computing systems. In their Turing Lecture, John Hennessy and David Patterson make a strong case for this trend, and predict that it will lead to a new golden age of computer architecture [6]. However, because the applications that run on these systems, such as image classification, speech recognition, language modeling, recommendation systems and scientific computing, are evolving rapidly with advances in machine learning, the accelerators must be programmable, to avoid quickly becoming obsolete. Such accelerators require a complete compiler system in order to be useful, and this compiler must get updated as the accelerator hardware evolves. The methodology for evolving accelerators, and more importantly their compilers, is more or less a completely manual process today, where large engineering teams study the accelerator architecture in detail and make the necessary modifications to the compiler and the low-level libraries to leverage the accelerator. Because of the large overhead of maintaining the entire software stack, real-world usage of an accelerator lags far behind its design. A key challenge, therefore, is to automate the co-design of programmable accelerators and the compilers that map applications to them, for fast-changing application domains.

II. Opportunity

We propose to tackle this challenge with CGRA accelerators and compilers that adapt as the hardware evolves.

A. CGRA as an Accelerator Template

Our approach to solving this problem has been by using coarse-grained reconfigurable arrays (CGRAs). A CGRA is similar to an FPGA but with larger compute and memory units, and word-level interconnect as shown in Fig. 1. By tuning the amount of configurability in these units and the interconnect, we can create more specialized (closer to ASICs) or more general-purpose accelerators (closer to FPGAs). Thus, a CGRA provides a standard accelerator template for a compiler to target. For example, a CGRA specialized for neural networks would look similar to a hand-designed neural network accelerator like TPU [7] with compute units implementing multiply-accumulate operations, and the interconnect supporting systolic connections between them. To map applications to CGRAs, we have created a compiler shown in Fig. 2, which takes applications written using a high-level library such as Halide [10], lowers it to a dataflow graph based intermediate representation (IR) called CoreIR [3], and then maps, places and routes the graph onto the CGRA. Using this compiler, we can accelerate a wide range of dense linear algebra applications, such as those in image processing and machine learning, on our CGRA and achieve 7 to 25× lower energy than an FPGA.

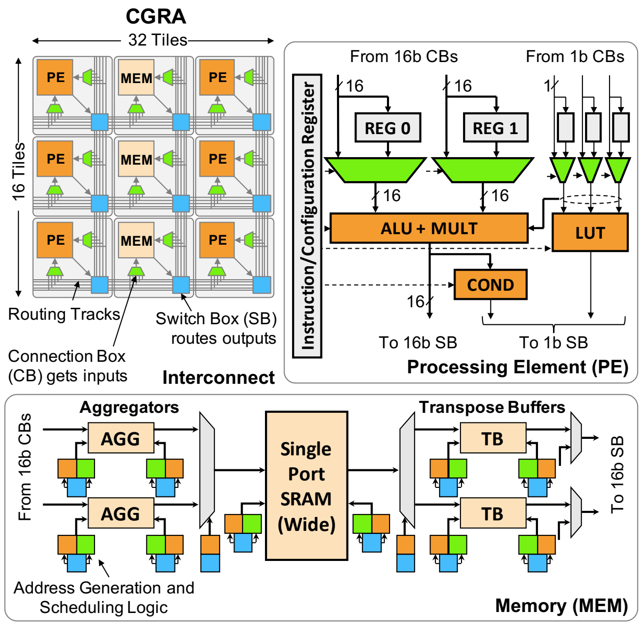

Fig. 1: A baseline CGRA architecture with processing element (PE) tiles, memory (MEM) tiles and a statically-configured interconnect.

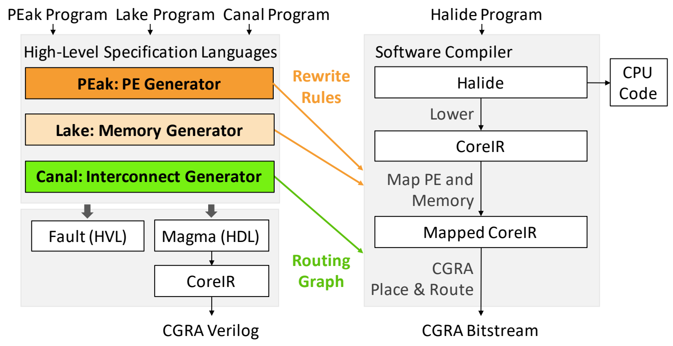

Fig. 2: End-to-end hardware generation and software compilation flow, starting with programs written in PEak, Lake, Canal, and Halide.

B. Accelerator-Compiler Codesign

The key insight to our approach is that, unlike previous work, our compiler *automatically updates* as the CGRA hardware evolves. We achieved this by creating mini specification languages—PEak for processing elements, Lake for memories, and Canal for interconnects—for formally specifying the hardware units, and then from those specifications automatically deriving both the hardware implementation and the collateral needed by the compiler as shown in Fig. 2 [1].

For example, the compiler for PEak, which is our specification language for processing elements (PEs), generates RTL Verilog, a functional model, and the rewrite rules the application compiler needs to map applications, all from a single PE specification. One of the primary enablers of PEak is its ability to leverage advanced SMT (Satisfiability Modulo Theories) solvers [2]. The PEak compiler synthesizes a collection of rewrite rules from a compiler IR (like CoreIR) to a PE ISA by finding an instruction in the ISA that is formally equivalent to one or more instructions in the IR. Similarly, from Lake specifications we generate memory hardware with programmable addressing logic, and the collateral needed by the compiler to map access patterns from applications to these memories. We perform this mapping from access patterns to configuration of address generators also using SMT solvers. Finally, Canal takes a set of (potentially heterogeneous) PE and memory cores and a specification of the interconnect. It generates the hardware, the routing graph that place-and-route tools need to map the dataflow graph onto the generated hardware, and the configuration bitstream that implements the routing result on the hardware, from the specification. As a result, a change in the design of any component automatically propagates through the flow to affect dependent components without manual intervention, and the compiler continually updates with the hardware.

C. Large-Scale Automated Design Space Exploration

Architects often explore many alternatives when designing an accelerator to achieve the best performance, power and area trade-offs. They analyze application kernels to find common sequences of operations that they can make faster or more energy-efficient. This is often done incrementally by proposing a design change, implementing it, then reevaluating the efficiency. A major impediment to design space exploration is implementing the software changes needed to compile the application to the new accelerator. The techniques we describe make it easy to modify an accelerator using PEak, Lake and Canal, and automatically derive a code generator so that the application can be compiled. This enables quick iterative design. Using such hardware-compiler codesign approaches, there is an opportunity to automate large-scale design space exploration (DSE) of accelerator architectures.

As a step in this direction, we are creating a PE DSE framework, that analyzes application graphs using subgraph mining [4] and maximal independent set analysis [5] and generates an ordered list of frequent subgraphs. It then uses subgraph merging [9] to merge several frequent subgraphs to generate a candidate PE graph, which it automatically converts into a PEak specification. From this, the PEak compiler generates both PE hardware and the rewrite rules required by the application mapper as described before. Finally, it synthesizes the CGRA with these specialized PEs and evaluates it using the mapping produced by the compiler. Using this method, we show that optimizing the PE for image processing reduces area by 29.6% to 32.5% and energy by 44.5% to 65.25%. Building on this initial work, given a set of applications in a domain, we hope to automatically produce an accelerator specialized for that domain. Finally, reinforcement learning techniques like [8], in conjunction with such a system, are very promising for performing fast DSE.

III. Timeliness

With the slowdown of Moore's law, hardware specialization is the most promising technique for continued improvement of scientific computing systems. The lack of a structured approach for evolving the software stack, as the underlying hardware becomes more specialized, has been one of the biggest impediments to its adoption. Our approach provides a systematic way of thinking about accelerators as specialized CGRAs and employs a combination of new programming languages and formal methods to automatically generate the accelerator hardware and its compiler from a single source of truth. Furthermore, it enables the creation of DSE frameworks that automatically generate accelerator architectures that approach the efficiencies of hand-design ones, with a significantly lower design effort. This has the potential to massively improve productivity of hardware-software engineering teams and enable quicker customization and deployment of complex accelerator-rich computing systems.

References

[1] R. Bahr, C. Barrett, N. Bhagdikar, A. Carsello, R. Daly, C. Donovick, D. Durst, K. Fatahalian, K. Feng, P. Hanrahan, T. Hofstee, M. Horowitz, D. Huff, F. Kjolstad, T. Kong, Q. Liu, M. Mann, J. Melchert, A. Nayak, A. Niemetz, G. Nyengele, P. Raina, S. Richardson, R. Setaluri, J. Setter, K. Sreedhar, M. Strange, J. Thomas, C. Torng, L. Truong, N. Tsiskaridze, and K. Zhang. Creating an agile hardware design flow. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.

[2] C. Barrett, P. Fontaine, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[3] R. Daly and L. Truong. Invoking and linking generators from multiple hardware languages using coreir. In *WOSET*, 2018.

[4] M. Elseidy et al. Grami: Frequent subgraph and pattern mining in a single large graph. *Proc. VLDB Endow.*, March 2014.

[5] H. Cheng et al. *Mining Graph Patterns*. Springer US, 2010.

[6] J. Hennessy and D. Patterson. A new golden age for computer architecture. *Commun. ACM*, 62(2):48–60, January 2019.

[7] N. P. Jouppi et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 1–12, New York, NY, USA, 2017. Association for Computing Machinery.

[8] A. Mirhoseini et al. Chip placement with deep reinforcement learning, 2020.

[9] N. Moreano et al. Efficient datapath merging for partially reconfigurable architectures. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2005.

[10] 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.*, 48(6):519–530, June 2013.