Creating an Agile Hardware Design Flow

Rick Bahr, Clark Barrett, Nikhil Bhagdikar, Alex Carsello, Ross Daly, Caleb Donovick, David Durst, Kayvon Fatahalian, Kathleen Feng, Pat Hanrahan, Teguh Hofstee, Mark Horowitz, Dillon Huff, Fredrik Kjolstad, Taeyoung Kong, Qiaoyi Liu, Makai Mann, Jackson Melchert, Ankita Nayak, Aina Niemetz, Gedeon Nyengele, Priyanka Raina, Stephen Richardson, Raj Setaluri, Jeff Setter, Kavya Sreedhar, Maxwell Strange, James Thomas, Christopher Torng, Leonard Truong, Nestan Tsiskaridze, Keyi Zhang
Stanford University
Email: praina@stanford.edu

***Abstract*—Although an agile approach is standard for software design, how to properly adapt this method to hardware is still an open question. This work addresses this question while building a system on chip (SoC) with specialized accelerators. Rather than using a traditional waterfall design flow, which starts by studying the application to be accelerated, we begin by constructing a complete flow from an application expressed in a high-level domain-specific language (DSL), in our case Halide, to a generic coarse-grained reconfigurable array (CGRA). As our understanding of the application grows, the CGRA design evolves, and we have developed a suite of tools that tune application code, the compiler, and the CGRA to increase the efficiency of the resulting implementation. To meet our continued need to update parts of the system while maintaining the end-to-end flow, we have created DSL-based hardware generators that not only provide the Verilog needed for the implementation of the CGRA, but also create the collateral that the compiler/mapper/place and route system needs to configure its operation. This work provides a systematic approach for desiging and evolving high-performance and energy-efficient hardware-software systems for any application domain.**

***Index Terms*—accelerator architectures, DSLs, compilers**

I. Introduction

Digital design tools and methodology have improved dramatically, letting us create billion-plus-transistor SoCs with accelerators we use every day. Unfortunately, completing these designs (with software) takes many years, and costs hundreds of millions of dollars [1]. Interestingly, a *waterfall*-like approach, which starts by studying an application and creating a hardware specification, and then continues by going through a number of refinements, is still used for most accelerator designs. The waterfall approach suffers from twin issues of changing application requirements and incomplete knowledge/understanding of the problem, making the resulting system less useful than desired. To avoid these issues, we explore an agile end-to-end hardware/software design flow where one incrementally updates hardware and software to generate an accelerator. The resulting flow is shown in Figure 1, and it can generate a customizable coarse-grained reconfigurable array (CGRA), along with the software infrastructure for mapping Halide [2] applications to the CGRA for execution.

This work is funded by DARPA's Domain-Specific SoC (DSSoC) program and Stanford's Agile Hardware Center and SystemX Alliance.

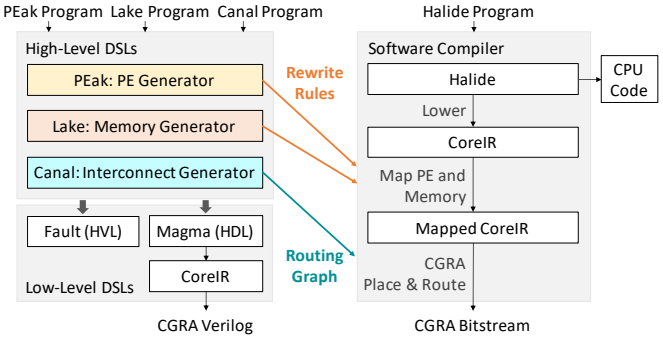


Fig. 1. End-to-end hardware generation and software compilation flow, starting with programs written in PEak, Lake, Canal, and Halide.

Our approach leverages recent work on creating and using hardware generators [3]–[6] to improve design productivity, and builds upon prior work on building/using CGRAs [7]–[10]. Like TVM [11] and HPVM [12], we are trying to construct a system that can map applications to hardware. Our flow has two main distinguishing features: (i) we utilize programming languages' semantics to address the problem of maintaining consistency between all layers of the end-to-end flow; and (ii) we create a modular system by using a number of small languages that each target one domain of the overall flow.

Any end-to-end flow is an integration of many layers of software and hardware. By having templates/generators create the layers in the flow, the parameters between the different layers quickly become dependent on each other. For example, if changing a parameter creates a new instruction in the CGRA's processing element (PE), the configuration for the layer mapping applications to the CGRA also needs to change.

Our main contribution is recognizing that the integration problem is fundamentally about managing the composition of the end-to-end flow's layers so that the cross-layer constraints are always satisfied, enabling developers to continuously compile and measure the applications on the hardware. Unlike configuration files, languages' semantics are sufficiently expressive to communicate both configuration values and how changes to those values impact other layers in the system. Thus, we have created three DSLs—PEak for PEs, Lake for memories, and Canal for interconnects—for

978-1-7281-1085-1/20/$31.00 ©2020 IEEE

Authorized licensed use limited to: Stanford University. Downloaded on November 18,2026 at 04:28:42 UTC from IEEE Xplore. Restrictions apply.

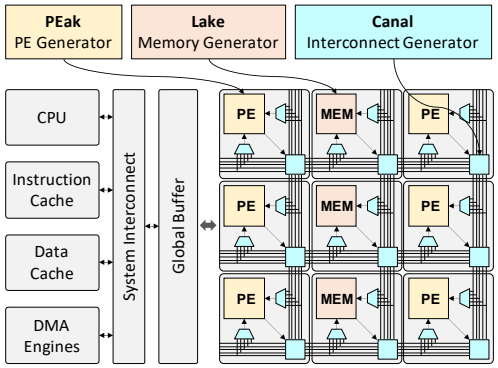


Fig. 2. Our island-style CGRA contains processing element (PE) and memory (MEM) tiles communicating through an interconnect, generated from PEak, Lake, and Canal DSLs.

specifying different parts of the CGRA as shown in Figure 1.

By writing our design configurations in these DSLs, we obtain a **single source of truth** for each layer. These languages have different "backends" that ensure different tools in the flow have a consistent view of the design. For example, the compiler of PEak, our DSL for processing elements, generates RTL Verilog, a functional model, and the rewrite rules the application compiler needs to map applications to it. Tying these disparate operations together requires an understanding of what programs mean, which our DSL approach provides.

II. CGRA Hardware and Compiler

Figure 2 shows the CGRA hardware that is generated by our DSLs and targeted by the software compiler. The software compiler, shown on the right in Figure 1, is divided into three main steps; compiling a Halide application to a CoreIR graph, mapping it to a graph of PE and MEM tiles, and performing place and route (P&R) on the mapped graph.

CoreIR [13] is an LLVM-inspired hardware IR and compiler framework and is leveraged by the RTL generation flow for the CGRA, and independently by the Halide compiler as its output target. CoreIR defines a standardized serializable graph-format, semantically-precise bitvector and stateful operations based on SMT-Lib [14], and a set of useful optimizations.

To create a flexible compiler framework for an ever-changing CGRA specification, multiple parts of the compiler need to be parameterized by the specification. PEak and Lake provide the mapper with a set of rewrite rules. Canal provides the P&R tool with tile and routing information.

A. Halide Compilation

Applications are written in Halide [2], a C++ embedded DSL for image processing and machine learning applications, that decouples scheduling from algorithms. As shown in Figure 3, our compilation flow consists of two stages. First, we extend the Halide scheduling primitives to specify what part of the application will be accelerated as well as to define the memory hierarchy and parallelism. Adding hardware scheduling primitives enables us to explore data tiling and traversal choices and to generate a configuration of the CGRA that maximizes the overall energy-efficiency and performance.

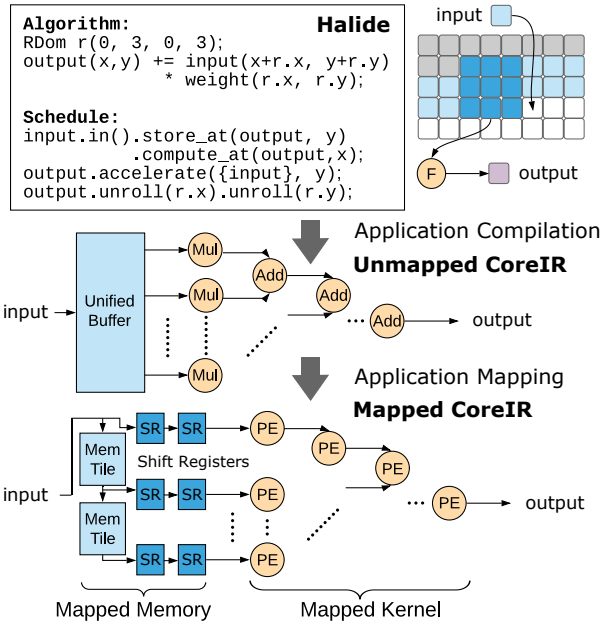


Fig. 3. Application compilation and mapping for a 3×3 convolution.

This Halide language is then lowered to Halide's internal intermediate representation (Halide IR). In this representation, computational kernels are represented by statements enclosed in for-loops, and memory operations are represented by reads and writes to unbounded, multi-dimensional arrays.

Next, the compiler lowers the application to the target intermediate representation, CoreIR. It does this by translating each compute statement into CoreIR's bitvector primitives and by performing a memory extraction pass to transform loop nests into streaming memories called unified buffers. This data-flow graph of unified buffer memories and computation kernels is then passed to the mapper.

B. Application Mapping

Application mapping transforms the Halide-generated, unmapped CoreIR graph into a semantically equivalent mapped CoreIR graph containing PE and MEM tiles. These PE and MEM tiles are defined by the particular CGRA specification. The transformations for computational kernels and unified buffers into PE and MEM tiles are informed by the PEak and Lake specifications respectively.

1) Memory Mapping: The unified buffer abstraction manages the dataflow between application kernels. We transform the loop control flow and data flow into an access pattern by mapping an n-dimensional loop to an n-dimensional address space. Memory mapping uses polyhedral analysis-based rewrite rules to take the unified buffers in the application and recursively break them into simpler unified buffers that can be mapped to the CGRA MEM tiles. Section III-B2 provides more details about memory rewrite rules.

2) Kernel Mapping: Kernel mapping produces a graph of PEs that minimizes a cost metric, typically total area or energy. Mapping is done in two phases: CGRA-independent optimizations and CGRA-dependent instruction selection. The

Authorized licensed use limited to: Stanford University. Downloaded on November 18,2026 at 04:28:42 UTC from IEEE Xplore. Restrictions apply.

first phase performs common optimizations including constant folding, common sub-expression elimination, and dead code elimination. The second phase performs instruction selection using rewrite rules that specify how to map CoreIR patterns to configured PE tiles. The PEak compiler generates these rewrite rules automatically from the PE specification (see Section III-A). Given these rules and a cost metric, the instruction selector finds a complete cover of the CoreIR graph with the rewrite rules that minimize the total cost.

C. Application Placement and Routing

Finally, we place and route the mapped CoreIR graph onto the CGRA. We first partition the input graph into multiple computation kernels where each kernel represents a densely connected graph component. *Global placement* places these kernels on the CGRA using an analytic solver. *Detailed placement* inside each kernel optimizes the placement result. Routing is done through an iterative algorithm which resolves resource overuse while optimizing for metrics such as delay. The routing result is used to generate the configuration bitstream for the CGRA. These steps require the routing graph corresponding to the CGRA, as well as information on how to set configuration registers to implement the routing. Canal provides this information, as described in Section III-C3.

III. Domain-Specific Languages for CGRA Hardware Generation

We use three DSLs to specify our CGRA. A specification written in these DSLs is the single source of truth for different systems that interpret it to generate the hardware, rewrite rules for mapping to the hardware, and other collateral. Using these DSLs, a change in the design of any component automatically propagates through the flow to affect dependent components without manual intervention.

A. PEak: Processing Element Generator

PEak is an embedded Python DSL for specifying PEs inspired by Bell and Newell's ISP notation for describing computer structures [15]. A PEak specification defines an instruction set (ISA), declares state, and describes the semantics of each instruction as a function from inputs and current state to outputs and next state. Figure 4 shows the multiple interpretations of a single PEak specification. The PEak compiler uses `magma` [4] to generate hardware and SMT [14] to generate mapper rewrite rules from the specification. It is executable in Python, so it also serves as a functional model of the PE hardware. The interface of the specification is tested to ensure consistency between the functional model and the hardware.

1) PEak Specification: PEak applies multiple interpretations [16] to the PE specification through the use of an abstract type system. Each PEak sub-component (functional model, hardware generator, and rewrite rule generator) provides a separate concrete implementation of the language's primitive abstract types. For example, PEak defines an abstract `BitVector` type that supports the `&` operator. Evaluating the expression `a & b` with the implementation of `BitVector`

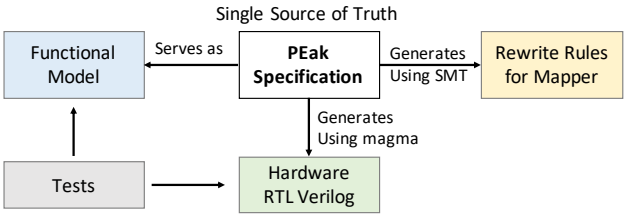


Fig. 4. From a specification of a PE, PEak automatically generates its functional model, hardware description, and rewrite rules for the mapper.

as an executable Python type performs a functional simulation. Using `magma`'s `Bits` type constructs a circuit. Using the `SMTBitVector` type, constructs an SMT formula.

PEak provides the primitive abstract types `Bit` and `BitVector` (signed and unsigned). To aid formal analysis, the semantics of `Bit` and `BitVector` are consistent with SMT-lib [14]. PEak also provides enums and algebraic data types (sum/tagged union and product/struct types) to aid the specification of ISAs.

The example code in Figure 5 and Figure 7 defines the ISA and functional specification[1] of a simple PE. Separating the encoding of the ISA from the functional specification lets designers easily modify the instruction decode logic without modifying the functional specification, and forces type-safe interaction with instructions. Since `Opcode` is not a `BitVector`, a direct comparison of `inst.op` to a `BitVector` will cause an error. Instead, the user must refer to a member of the `Opcode` Enum.

In the functional specification, `__init__` defines subcomponents and state like registers and memories (including pipeline registers). The example PE has two sub-components, a `Data` and a `Bit` register. The `__call__` method defines the semantics of each PE instruction by determining the desired behavior of each `inst`. Both the ISA and the functional specification can be tested using Python execution.

2) Generating PE Hardware: PEak relies on `magma` [4], a Python-embedded hardware construction language, to compile specifications to RTL Verilog. PEak's syntax extends `magma`'s `sequential` circuit syntax with rich types that describe ISAs using `magma`'s type protocol. `magma`'s type protocol lets new types be defined by implementing an interface that allows magma to interpret the new type as if it were one of magma's built-in primitive types. For example, PEak's sum type provides a syntax that forces type-safe interaction with variants. The implementation of the type protocol allows `magma` to interpret sum type values as `magma Bits`. This allows sum types to provide syntax-level constraints while reusing the semantics of `BitVector` for the hardware implementation.

Lowering a PEak specification to `magma` is a straightforward process that captures the functional intent of the designer. The `__call__` method simply defines the state machine transition function that is executed on every positive edge of the clock. The PEak language encourages high-level

[1]We distinguish the functional specification (the function of a PE [15]) from the functional model (a software executable model of a hardware component).

Authorized licensed use limited to: Stanford University. Downloaded on November 18,2026 at 04:28:42 UTC from IEEE Xplore. Restrictions apply.

```
class Opcode(Enum):
  Add = 0
  And = 1
class Instruction(
    Product):
  op = Opcode
  invert_A = Bit
  scale_B = Bit
  reg_out = Bit
# Data is BitVector
Data = Unsigned[16]
```

Fig. 5. PE ISA specification.

```
pe = PE()
inst = Instruction(
  Opcode.Add,
  Bit(0), # invert_A
  Bit(1), # scale_B
  Bit(0)) # reg_out
out, flag = pe(
  inst,
  Data(2), # A
  Data(3), # B
  Data(5), # C
  Bit(0))  # c_in
assert out==Data(17)
assert flag==Bit(0)
```

Fig. 6. PE python execution.

```
class PE(Peak):
  def __init__(self):
    self.o_reg = Register(Data)
    self.f_reg = Register(Bit)

  def __call__(self,
               inst: Instruction,
               A: Data,
               B: Data,
               C: Data,
               c_in: Bit
               ) -> (Data, Bit):
    if inst.invert_A:
      A = ~A
    if inst.scale_B:
      B = B*C
    if inst.op == Opcode.Add:
      # adc = add with carry
      res, flag = A.adc(B, c_in
    )
    else: # inst.op == Opcode.
    And
      res = A & B
      flag = (res == 0)
    if inst.reg_out:
      res = self.o_reg(res)
      flag = self.f_reg(flag)
    return res, flag
```

Fig. 7. PE functional specification.

specifications that eschew low-level details such as resource sharing, clock gating, and data gating. Instead of requiring that these details be captured at the PEak level, these concerns are addressed by optimization passes in the compiler tool-chain. `magma`'s compiler intermediate representation (CoreIR [13]) is based on SMT [14] which enables formal equivalence checking of the input/output pairs of each pass.

The `fault` [17] Python package is used to test `magma` circuits with the function call syntax in Figure 6. By wrapping the generated `magma` circuit in a `fault` tester object, designers directly reuse functional model tests for the hardware description, and `fault` generates a test bench that verifies the `magma` circuit using a hardware simulator such as Verilator.

3) Generating Rewrite Rules for Kernel Mapping: Mapping CoreIR graphs requires rewrite rules that specify how particular CoreIR patterns map to PEs. To generate rewrite rules, the `__call__` method is transformed into a normal form where each name is assigned to once, there is a single return at the end of the function, sub-components are called once, and all `if` blocks are transformed into ternary expressions. Once in this form, applying `__call__` to abstract SMT variables (in the same way they are applied to concrete python variables in Figure 6) produces a symbolic execution of the circuit. This symbolic execution can be used to generate rewrite rules from a CoreIR IR node using a quantified SMT query:

$$\exists inst\ \forall inputs : IRNode(inputs) == PE(inst, inputs)$$

If the SMT solver finds an $inst$, we have a rewrite rule between $IRNode$ and $inst$. If the SMT solver does not find a rewrite, we know that none exists.

Further, a similar technique can be used to ensure optimizations do not change the behavior of a design. For example,

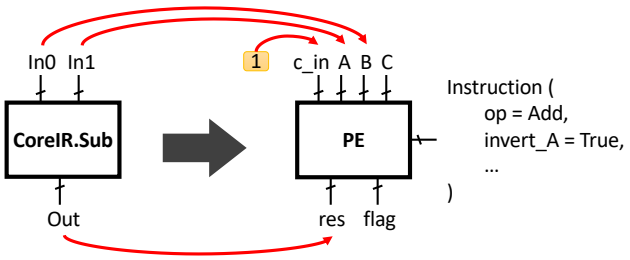


Fig. 8. An example rewrite rule for kernel mapping that maps CoreIR subtract to the simple PE from Figure 7.

suppose a rewrite rule has been discovered between $IRNode$ and $inst$ for PE. An optimized PE OPE can be verified with:

$$\forall inputs : IRNode(inputs) == OPE(inst, inputs)$$

B. Lake: Memory Generator

While PEak starts with a high-level specification of a PE, Lake starts with a low-level hardware-centric specification of a memory module to make it easy for hardware designers to perform design space exploration. From this specification, Lake creates the technology-dependent RTL Verilog, a high-level specification of this hardware that can be used in a polyhedral rewrite system, and a mechanism to set the configuration registers from the rewrite system output.

1) Lake Hardware Specification: Lake memory modules contain one or more memory units, blocks that select or combine inputs to create an output, and a graph interconnecting these units to each other and the ports. Figure 9 shows an example memory module with three memory units. For each memory unit, the rewrite system needs to know the memory capacity, number of ports, port width, and the read/write delay, all of which are easily extracted from the hardware specification. It also needs to understand the capability of the address generators. Our system currently supports nested affine loops (with the user specifying the number of loops and constraints on the loop values), with the innermost loop as a normal loop or a vector of addresses (user specifies max length). The latter allows our system to support some non-affine address patterns.

This system makes it easy both to express an efficient memory (like using a wide fetch memory to emulate a multi-ported memory) and to extract its specification for the rewrite system. Figure 9 shows the hardware needed to utilize a wide memory: a small memory to aggregate data before writing to the wide memory and another small memory to re-order and serialize the output data after reading from the wide memory. By passing each units's parameters to the rewrite system, Lake does not need to explicitly compute the access patterns supported by the overall memory module. The rewrite system leverages polyhedral analysis to analyze the address pattern using the parameters for each memory unit separately.

The designer also specifies how units and ports are interconnected as well as functions that combine multiple inputs into a single output. These mux-like functions can be used to bypass unused internal units or bundle extra ports from multiple Lake

Authorized licensed use limited to: Stanford University. Downloaded on November 18,2026 at 04:28:42 UTC from IEEE Xplore. Restrictions apply.

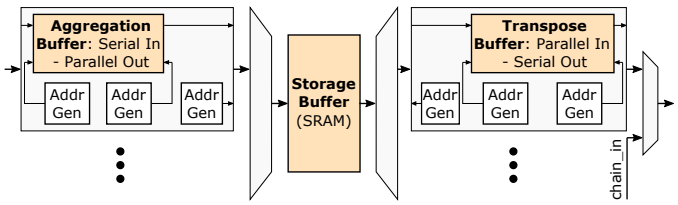


Fig. 9. Example Lake memory module with three memory units.

memory modules to form a memory with larger capacity or bandwidth without using additional hardware. For example, our current design has an extra data input port, an output port, and a mux in the hardware for the rewrite system to chain Lake modules together to double the memory capacity.

2) Generating Rewrite Rules for Memory Mapping: The memory rewrite rules map each unified buffer required by the application to hardware memory modules generated from Lake. There are two types of rewrite rules: hardware-independent and hardware-dependent. Hardware-independent rewrite rules use polyhedral analysis on the access patterns of the unified buffer to determine data reuse. This reduces the bandwidth/capacity of the buffer. For example, memory bandwidth can be reduced by inserting registers if the data is fetched multiple times, while capacity can be reduced if we overwrite the data after it becomes obsolete. Figure 10(a) shows this rewrite for a fully unrolled 1D convolution with window size of 3. While the application-level unified buffer specifies a memory the size of the image with three output ports (since the downstream kernel needs to read three pixels in parallel), it can be rewritten into two shift registers by analyzing the reuse pattern in the memory accesses.

The hardware-dependent rewrite rules transform abstract memories into concrete hardware memory modules using the parameters extracted from the Lake specification (Section III-B1). If the application-level unified buffer needs more bandwidth or capacity than what is available in a memory tile, the compiler uses memory banking or chaining, respectively, as shown Figure 10(b) and (c). Since hardware memories may have a wider fetch width, we also include a vectorization rewrite rule to map to them as shown in Figure 10(d).

Since the rewrite system works on extracted memory specifications, it has the specification for each address generator. However, it does not know how to configure the hardware to implement that specification as it has no knowledge of the actual hardware. To determine this configuration state, we first extract a formal model of the address generation logic from the Verilog RTL. Using this model and the knowledge of which bits are the configuration state, we then use an SMT solver to find a setting of configuration bits that generates the required address pattern for that generator, similar to PEak.[2]

C. Canal: Interconnect Generator

Canal takes a set of (potentially heterogeneous) PE and memory cores and a specification of the interconnection network. It then generates the hardware (with the cores snapped

[2]We have a working prototype of this system and are manually mapping some configurations. We expect the full system to be operational in 6 months.

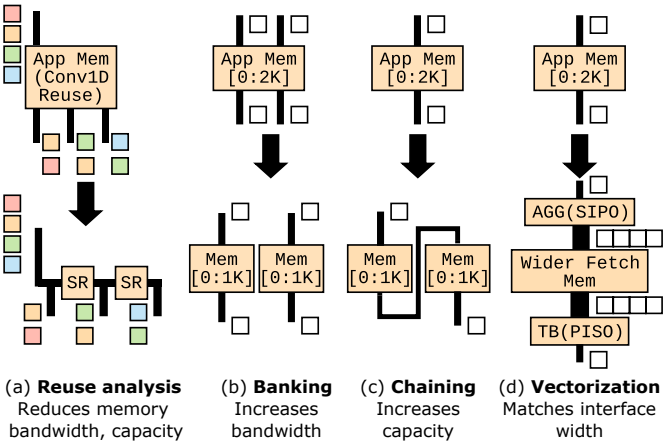


Fig. 10. Memory mapping rewrite rule examples: (a) is hardware-independent, while (b), (c) and (d) are hardware-dependent rewrite rules.

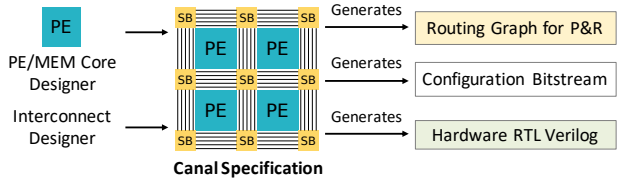


Fig. 11. Canal is a single source of truth for generating hardware, place-and-route collateral, the configuration bitstream, and a functional model (not shown).

into the network at designer-specified locations), the routing graph that place-and-route tools need to map the dataflow graph onto the generated hardware, the configuration bitstream that implements the routing result on the hardware, and a functional model (Figure 11). It allows designers to easily explore interconnect parameters like network topology, placement of pipeline registers, and switchbox design.

1) Canal Specification: A Canal program is a directed graph that abstractly represents the structure of the interconnect. Vertices are terminals, and directed edges are wired connections. Vertices can have multiple incoming edges, which abstracts away low-level multiplexers. Each vertex can be annotated with attributes. A coordinate attribute enables reinterpreting the graph on a grid-based layout, and a type attribute marks a vertex as a tile port or a pipeline register.

Using an abstract graph-based DSL has several advantages over a simple hardware generator with parameters. A graph allows staged generation (e.g. use passes to insert pipeline registers). Different standard interconnect topologies can easily be imported and modified.

2) Generating Interconnect Hardware: We generate the RTL description automatically by following several rules: 1) Every edge is a directed wire connection; 2) Vertices with more than one incoming edge generate multiplexers; 3) Multiplexer select bits follow the incoming edge ordering; 4) Vertices with attributes for special hardware types (e.g. a pipeline register) generate that hardware. Canal also verifies structural correctness by comparing the connectivity of the generated hardware (extracted from the RTL) with the original abstract graph using standard graph isomorphism algorithms.

Authorized licensed use limited to: Stanford University. Downloaded on November 18,2026 at 04:28:42 UTC from IEEE Xplore. Restrictions apply.

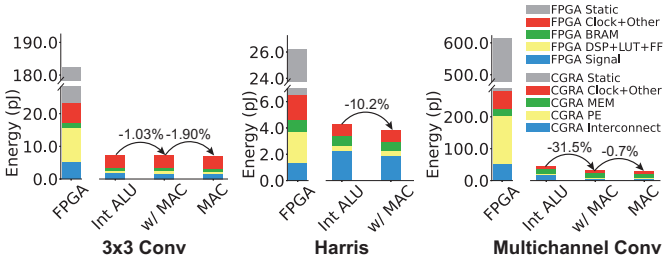


Fig. 12. Energy/op of three different CGRAs (PE with an integer ALU, integer ALU + MAC, and MAC only) and an FPGA in the same technology.

3) Generating Routing Graph for Place-and-Route: Canal mechanically transforms the abstract graph into a routing graph required by the P&R tools to map the application dataflow graph onto precisely this instance of generated hardware. It also verifies the structural connectivity of the transformation against the original abstract graph, and includes timing-related information (e.g. wire delays) in the routing graph for timing-driven P&R.

4) Generating Configuration Bitstream: The output of the place-and-route tool is a routing result that describes which connections must be made (in the reconfigurable interconnect) in order to implement the application dataflow graph. Canal takes the routing result and generates a configuration bitstream that creates these connections on the generated hardware.

IV. Results

Using our DSLs, we created Garnet, the latest iteration of our CGRA SoC, with a 32×16 array of PE and memory tiles, a second level memory called global buffer and an ARM Cortex M3 processor (Figure 2). Each PE tile has a 16-bit, two-input, fixed point ALU, and some registers. Each memory tile contains 2 KB of SRAM and flexible address generators. An interconnect with five 16-bit tracks and five 1-bit tracks connects the tiles.

To show the flexibility of our design flow, we generate three versions of the 32×16 CGRA and use our software compiler to map 3×3 convolution, Harris corner detector, and a neural network layer (multichannel convolution) onto the different CGRAs. In the first version, the PE on the CGRA has a 16-bit, two-input, integer ALU. In the second version, the ALU has an additional, specialized multiply-accumulate (MAC) instruction. The third version is most specialized and only has a MAC unit. The CGRAs are synthesized, placed, and routed in TSMC 16nm technology and run at 200 MHz.

Figure 12 shows the energy/op consumed by each version of the CGRA. We compare the CGRAs with the FPGA on the Xilinx ZCU102 development board programmed with Vivado 2017.2 toolchain, which is in the same TSMC 16nm technology. Adding a specific MAC instruction to the PE reduces energy because fewer PEs are needed to execute convolutions, resulting in less inter-tile communication. Specializing the PE to have only a MAC instruction further reduces energy at the cost of configurability (this version can no longer run Harris). The CGRA consumes 6.92× to 25.3× less energy than the FPGA.

V. Conclusion

To facilitate agile hardware design, we need tools to maintain the end-to-end flow. This requires hardware generators, clean interfaces, and methods to communicate changing design features without a designer's manual intervention. Our framework and associated DSLs address these concerns by allowing the designer to separately deal with different concerns, and by seamlessly communicating changing design capability to all the layers in our flow. The result is an approach to agile hardware design that enables rapid integration of changing components and shorter design cycles.

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, pp. 519–530, June 2013.

[3] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a scala embedded language," in *DAC Design Automation Conference 2012*, pp. 1212–1221, 2012.

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

[5] K. Jaic and M. C. Smith, "Enhancing hardware design flows with myhdl," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, (New York, NY, USA), p. 28–31, Association for Computing Machinery, 2015.

[6] S. Jiang, B. Ilbeyi, and C. Batten, "Mamba: Closing the performance gap in productive hardware development frameworks," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2018.

[7] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix," in *International Conference on Field Programmable Logic and Applications*, pp. 61–70, Springer, 2003.

[8] V. Govindaraju, C. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim, "Dyser: Unifying functionality and parallelism specialization for energy-efficient computing," *IEEE Micro*, vol. 32, no. 5, pp. 38–51, 2012.

[9] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel patterns," in *2017 ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pp. 389–402, 2017.

[10] 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)*, pp. 1–13, IEEE, 2016.

[11] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An automated end-to-end optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation*, (Carlsbad, CA), pp. 578–594, USENIX Association, Oct. 2018.

[12] M. Kotsifakou, P. Srivastava, M. D. Sinclair, R. Komuravelli, V. Adve, and S. Adve, "Hpvm: Heterogeneous parallel virtual machine," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, (New York, NY, USA), p. 68–80, Association for Computing Machinery, 2018.

[13] R. Daly and L. Truong, "Invoking and linking generators from multiple hardware languages using coreir," 2018.

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

[15] C. G. Bell and A. Newell, "The pms and isp descriptive systems for computer structures," in *Proceedings of the May 5-7, 1970, spring joint computer conference*, pp. 351–374, 1970.

[16] J. O'Donnell, "Hydra: hardware description in a functional language using recursion equations and high order combining forms," *The Fusion of Hardware Design and Verification*, pp. 309–328, 1988.

[17] L. Truong, "fault." https://github.com/leonardt/fault, 2020.

Authorized licensed use limited to: Stanford University. Downloaded on November 18,2026 at 04:28:42 UTC from IEEE Xplore. Restrictions apply.