AHA: An Agile Approach to the Design of Coarse-Grained Reconfigurable Accelerators and Compilers

KALHAN KOUL, JACKSON MELCHERT, KAVYA SREEDHAR, LEONARD TRUONG, GEDEON NYENGELE, KEYI ZHANG, QIAOYI LIU, JEFF SETTER, PO-HAN CHEN, YUCHEN MEI, MAXWELL STRANGE, ROSS DALY, CALEB DONOVICK, ALEX CARSELLO, TAEYOUNG KONG, KATHLEEN FENG, DILLON HUFF, ANKITA NAYAK, RAJSEKHAR SETALURI, JAMES THOMAS, NIKHIL BHAGDIKAR, DAVID DURST, ZACHARY MYERS, NESTAN TSISKARIDZE, STEPHEN RICHARDSON, RICK BAHR, KAYVON FATAHALIAN, PAT HANRAHAN, CLARK BARRETT, MARK HOROWITZ, CHRISTOPHER TORNG, FREDRIK KJOLSTAD, and PRIYANKA RAINA, Stanford University, USA

With the slowing of Moore's law, computer architects have turned to domain-specific hardware specialization to continue improving the performance and efficiency of computing systems. However, specialization typically entails significant modifications to the software stack to properly leverage the updated hardware. The lack of a structured approach for updating the compiler and the accelerator in tandem has impeded many attempts to systematize this procedure. We propose a new approach to enable flexible and evolvable domain-specific hardware specialization based on coarse-grained reconfigurable arrays (CGRAs). Our agile methodology 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. This enables the creation of design-space exploration frameworks that automatically generate accelerator architectures that approach the efficiencies of hand-designed accelerators, with a significantly lower design effort for both hardware and compiler generation. Our current system accelerates dense linear algebra applications but is modular and can be extended to support other domains. Our methodology has the potential to significantly improve the productivity of hardware-software engineering teams and enable quicker customization and deployment of complex accelerator-rich computing systems.

CCS Concepts: • **Software and its engineering → Compilers**; • **Computer systems organization → Parallel architectures**; • **Hardware → Hardware description languages and compilation**; **Application specific integrated circuits;**

This work was supported by the DSSoC DARPA grant, the Stanford AHA Agile Hardware Center and Affiliates Program, Intel's Science and Technology Center (ISTC), and the Stanford SystemX Alliance.

Authors' address: K. Koul, J. Melchert, K. Sreedhar, L. Truong, G. Nyengele, K. Zhang, Q. Liu, J. Setter, P.-H. Chen, Y. Mei, M. Strange, R. Daly, C. Donovick, A. Carsello, T. Kong, K. Feng, D. Huff, A. Nayak, R. Setaluri, J. Thomas, N. Bhagdikar, D. Durst, Z. Myers, N. Tsiskaridze, S. Richardson, R. Bahr, K. Fatahalian, P. Hanrahan, C. Barrett, M. Horowitz, C. Torng, F. Kjolstad, and P. Raina, Stanford University, 450 Serra Mall, Stanford, California 94305, United States; emails: {kkoul, melchert, skavya, lenny, nyengele, keyi, joeyliu, setter, pohan, yuchenm, mstrange, ross.daly}@stanford.edu, donovick@cs.stanford.edu, {ajcars, kongty, kzf}@stanford.edu, dillonhuff@gmail.com, {ankitan, setaluri, jjthomas, nikhil3, durst, zamyers, nestan, steveri, bahr}@stanford.edu, {kayvonf, hanrahan}@cs.stanford.edu, barrettc@stanford.edu, horowitz@ee.stanford.edu, ctorng@stanford.edu, kjolstad@cs.stanford.edu, praina@stanford.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Association for Computing Machinery.

1539-9087/2023/01-ART35 $15.00
https://doi.org/10.1145/3534933

Additional Key Words and Phrases: Hardware accelerators, coarse-grained reconfigurable arrays, domain-specific languages, image processing

ACM Reference format:
Kalhan Koul, Jackson Melchert, Kavya Sreedhar, Leonard Truong, Gedeon Nyengele, Keyi Zhang, Qiaoyi Liu, Jeff Setter, Po-Han Chen, Yuchen Mei, Maxwell Strange, Ross Daly, Caleb Donovick, Alex Carsello, Taeyoung Kong, Kathleen Feng, Dillon Huff, Ankita Nayak, Rajsekhar Setaluri, James Thomas, Nikhil Bhagdikar, David Durst, Zachary Myers, Nestan Tsiskaridze, Stephen Richardson, Rick Bahr, Kayvon Fatahalian, Pat Hanrahan, Clark Barrett, Mark Horowitz, Christopher Torng, Fredrik Kjolstad, and Priyanka Raina. 2023. AHA: An Agile Approach to the Design of Coarse-Grained Reconfigurable Accelerators and Compilers. *ACM Trans. Embedd. Comput. Syst.* 22, 2, Article 35 (January 2023), 34 pages.
https://doi.org/10.1145/3534933

1 INTRODUCTION

As technology scaling slows, domain-specific hardware accelerators will play an increasingly important role in improving the performance and energy efficiency of computing systems. With advances in machine learning, applications such as image classification, speech recognition, language modeling, recommendation systems, and scientific computing are changing at a rapid pace. Maintaining high end-to-end performance and efficiency requires that hardware accelerators, compilers, and applications evolve together in lockstep. Unfortunately, existing methodologies to achieve this involve significant manual effort. Large engineering teams study the accelerator architecture in detail and modify the compiler in an ad hoc manner, leveraging low-level libraries to target specific hardware features. Because of the large overhead in maintaining the software stack, it remains challenging to accelerate new domains or to accelerate existing domains as they evolve.

In this work, we propose automating the co-design of accelerators and compilers. We showcase this accelerator-compiler co-design by targeting a coarse-grained reconfigurable array (CGRA) [2]. CGRAs are a spatial-style architecture [15, 35, 42, 53] analogous to field-programmable gate arrays (FPGAs) but with coarser-grained processing and memory units along with a word-level interconnect, as shown in Figure 1. By tuning the amount of specialization in these units and the interconnect, we can span the space between application-specific integrated circuits (ASICs; less flexible, but more efficient) and FPGAs (more flexible, but much less efficient). For example, a CGRA specialized for neural networks would look similar to a hand-designed neural network accelerator such as the tensor processing unit (TPU) [22] with compute units implementing multiply-accumulate operations and the interconnect supporting only systolic connections between them. To map applications to CGRAs, we have created a compiler, shown in Figure 2, which takes applications written in the high-level domain-specific language (DSL) called Halide [44], lowers them to a dataflow graph intermediate representation (IR) called CoreIR [12], and then schedules, maps, places, and routes the graph to produce a CGRA bitstream. Using this compiler, we can accelerate a wide range of dense linear algebra applications, such as those in image processing and machine learning. While the current version of our accelerator-compiler system targets dense linear algebra applications, it must be noted that only portions of the system—the frontend language (Halide), static scheduling, and the design of the memory units—are tailored for such applications. Our system is modular and can be extended to support other application domains.

The key feature of our approach is that, unlike previous work, our compiler *automatically updates* as the CGRA hardware evolves. We achieve this by creating three mini domain-specific hardware specification languages—PEak for processing elements, Lake for memories, and Canal for interconnects. Programs written in each language represent the formal specification, a single source of truth, for each component. From this specification, our tools generate both the register-transfer

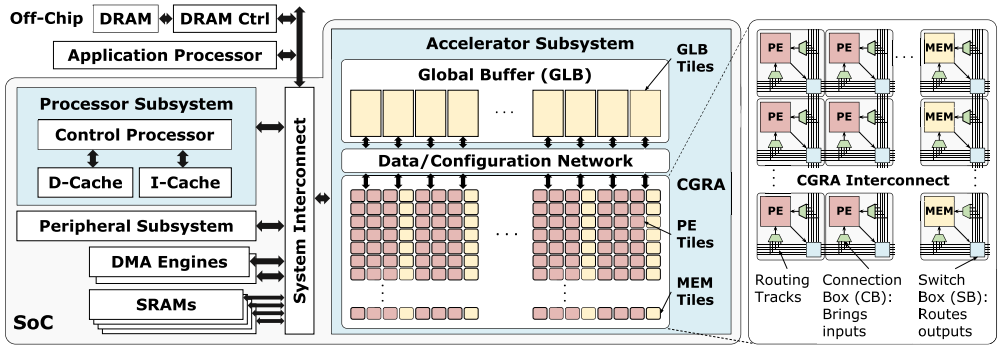


Fig. 1. Our system-on-chip (SoC) with a coarse-grained reconfigurable array (CGRA) accelerator. The CGRA has processing element (PE) and memory (MEM) tiles and a statically configured word-level interconnect.

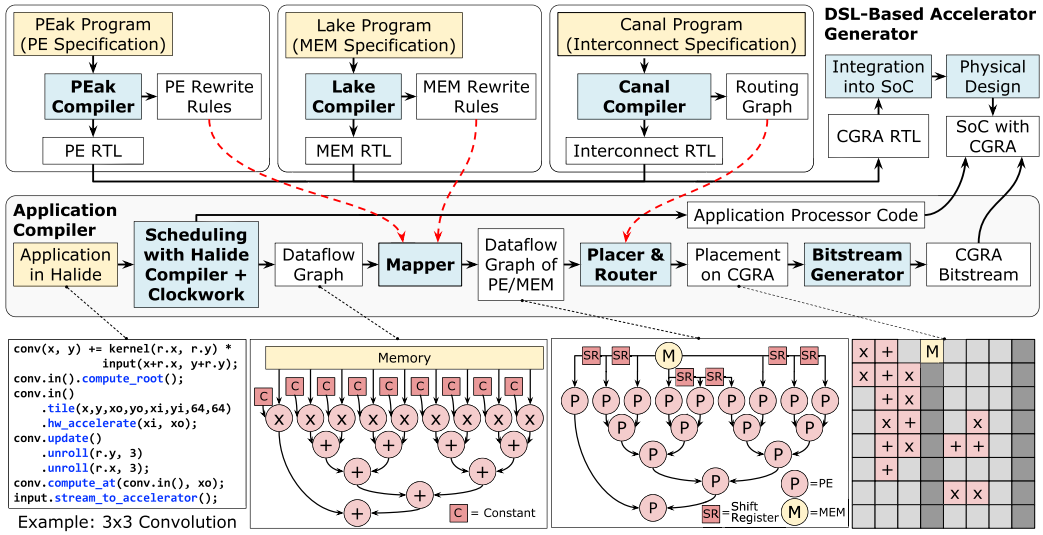


Fig. 2. Agile flow for co-design of CGRA accelerator and compiler. The compiler maps applications written in Halide to a CGRA bitstream. It receives updated collateral (rewrite rules, routing graph) from the DSL-based hardware generators, allowing it to compile to evolving CGRA hardware with no manual updates.

level (RTL) hardware description *and* the collateral needed by the application compiler as shown in Figure 2. Changes in the architecture are done simply by modifying the PEak, Lake, and Canal programs. As a result, any changes automatically propagate through the hardware generation and the compilation flow without manual intervention. These languages could be used to design other hardware accelerators (e.g., ASICs or FPGAs) and are not limited to CGRAs. We have chosen CGRAs as our target hardware for two key reasons: they allow exploration of a large design space and have the ability to accelerate a *domain* of applications. On the other hand, ASICs typically target a very limited number of applications and FPGAs are fine-grained and designed to handle applications in several different domains. Our flow enables quick iterative design and presents an opportunity to automate the large-scale design-space exploration of accelerator architectures that achieve energy efficiencies that beat general purpose architectures and approach ASICs.

2 RELATED WORK

With many researchers focused on hardware specialization, domain-specific accelerator generators and compilers that target accelerators have both become popular areas of study.

2.1 Domain-Specific Accelerator Generators

Many existing domain-specific accelerator generators start with an application or a class of applications and generate *specialized fixed-function hardware* to be instantiated as an ASIC or on an FPGA. In contrast, our work generates *programmable hardware* and a compiler that maps a variety of applications to it. Prior domain-specific accelerator generators that create fixed-function image processing pipelines include Aetherling [14], Darkroom [16], Halide-HLS [43], Hetero-Halide [30], HIPACC-FPGA [45], PolyMage-FPGA [10], Rigel [17], and SODA [8]. Similarly, DNNBuilder [60], DNNWeaver [46, 59], and VTA [37, 38] take deep neural network (DNN) models and generate hardware to accelerate them on FPGAs. DNNBuilder utilizes a line-buffer-based scheme and a fine-grained layer-based pipeline architecture, which reduces the pipeline latency and interlayer memory consumption, thereby reducing on-chip memory usage. DNNWeaver uses Caffe to generate synthesizable designs with several layer types and [59] uses Caffe [21] to optimize DNN blocking using double buffer structures. VTA develops a hardware-software flow for neural network layers using a modified Halide IR. It maps DNN layers onto operators that are then offloaded onto an FPGA. However, all of these systems generate *fixed-function* hardware on either FPGAs (which have high overhead) or ASICs (which are inflexible), as opposed to our work, which focuses on generating programmable CGRAs along with complete compilers to map to them.

There is also existing work on generating programmable accelerators as opposed to fixed-function ones. Work focused on CGRAs includes Plasticine [42], ultra-elastic CGRAs [48], CGRA-ME [9], DySER [15], and ADRES [35]. Other works propose configurable neural network accelerator generators such as MAGNet [54], MAERI [28], and AutoDNNchip [58]. However, all of these works must manually update their compilers for each subsequent generation of the hardware. In contrast, our work generates not only the programmable hardware but also the collateral needed by the application compiler as the hardware evolves.

2.2 Compilers that Target Accelerators

Existing compilers struggle to support flexible mapping onto complex, specialized hardware units. High-level synthesis (HLS) tools—including Catapult [36], LegUp [4], Vivado HLS [57], and others [20, 33]—compile C/C++ programs employing commonly used compiler front ends, schedule instructions to an intermediate representation (IR), map instructions to functional units, and emit code. This method is used for both FPGAs and ASICs in which the primitives – that is, registers, look-up tables (LUTs), and gates—are more fine grained than the compiler IR instructions. Unfortunately, when the primitives are *coarser grained* than the compiler IR instructions, this approach is no longer sufficient. For example, it is not straightforward for HLS compilers to determine how to fuse simple instructions to utilize coarse-grained PEs.

Moreover, while Catapult and Vivado HLS efficiently map arithmetic and exploit parallelism within individual loop bodies [34], they do not support optimizations for specialized memories [43] or across loops [61]. Efficiently exploiting parallelism across different loops in a loop nest requires manual effort to generate high-quality code for deeper pipelines [29]. Our compiler, described in Section 5, supports both automatic loop fusion and memory optimization.

HeteroCL [29] and Spatial [27] are both academic languages that support a more abstract programming model for accelerators. However, specifying the memory microarchitecture and its compiler interface is still left to the user. Similar to Halide, HeteroCL's frontend splits up the

algorithmic specification from the schedule and data types, but its backend implementation depends on manually written templates. Additionally, Spatial is able to map to different FPGA targets and to Plasticine, but it does not generate the compiler collateral needed to automatically track changes in the hardware. In contrast, we automatically generate the backend of our compiler, which maps the compiler IR to specific coarse-grained hardware primitives as the primitives evolve.

3 DSL-BASED ACCELERATOR GENERATOR

The key innovation in our flow is the use of high-level DSLs as inputs to our hardware generators as opposed to using less expressive parameters [1, 54]. Each component of the CGRA is written as a program in its own DSL: the PEs in PEak, the memories in Lake, and the interconnect in Canal. Each DSL's compiler generates *both* the hardware RTL and the collateral needed by the compiler to map applications onto the hardware, thereby always maintaining end-to-end system functionality. The component DSLs are written on top of the same host language—magma [49], which is a lower-level hardware generation DSL embedded in Python.

The goal of our system is to cover the general patterns required to construct a CGRA by using a composition of components written in different DSLs. Each DSL is tailored for the description of a specific sub-component by providing a specific abstraction: PEak uses sum types to describe instructions, Lake uses a streaming abstraction for describing memory access patterns, and Canal uses a graph abstraction for describing the interconnect. While it may be possible to describe other types of components than those intended by a DSL, the lack of other abstractions useful for other domains makes this an inconvenient burden unlikely to be taken on by a designer. The goal of the system is to have the designer understand which DSL best fits the description of the designer's component. If none do, then the designer is able to fall back on the magma host language that provides general purpose abstractions.

To minimize the burden on developers and to make the system easily accessible to hardware designers familiar with Python, the DSLs employ a shallow embedding architecture where each language is presented as a native Python library. These languages restrict the user input to the domain of interest by forcing the user to use specific application programming interfaces (APIs) or by performing analysis on the abstract syntax tree, which raises errors when encountering unsupported code. By restricting user input to a specific domain, the DSL compilers employ domain-specific optimizations to provide efficient hardware generation for their abstractions. By maintaining source mappings through the various compiler stages, the user can effectively iterate on the quality of the generated hardware by tuning the high-level code based on tool feedback.

Using the same host language for each DSL is a key design decision that facilitates the composition of domain-specific components during the construction of a complete system. By nature, DSLs are not designed for generic composition; they focus on specific abstractions useful for a limited domain. In contrast, a complex system incorporates different components written in different DSLs. Having each DSL compiler target the same general-purpose host hardware description language allows system designers to cleanly compose components. This avoids complexity in system integration and reduces the cognitive load of working with different language syntaxes for different parts of the system while also allowing the DSLs to share common features of the host language, such as advanced metaprogramming capabilities.

The following three sections are tightly connected. Section 3 describes our hardware DSLs, Section 4 gives an example of a generated CGRA architecture, and Section 5 describes our compiler that maps onto the generated hardware. For example, in Section 3, we describe PEak; in Section 4, we describe a processing element generated with PEak; and, in Section 5, we describe compute mapping, which utilizes collateral generated by PEak to map an application onto the PE

architecture described in Section 4. Figure 14(a) illustrates the intuition behind the PE hardware/-compiler automatic update. A specialized PE may implement multiple logical operations. The compiler is made aware of the possible mappings through a set of rewrite rules generated by the PEak DSL description.

3.1 PEak

A PEak specification for a PE defines its instruction set architecture (ISA), declares state, and describes the semantics of each instruction as a function from inputs and current state to outputs and next state. A PEak program can be compiled to a functional model, an RTL hardware description, and a formal model encoded using satisfiability modulo theories (SMT) [3], which is used to automatically synthesize a set of compiler rewrite rules that target the specified PE.

3.1.1 PEak Specification. The example code in Figures 3 and 5 defines the ISA and functional specification of a simple PE in PEak that supports two operations, can invert or scale the inputs, and can register the output. 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 interactions with instructions. In the functional specification, `__init__` defines sub-components and state (i.e., registers and memories, including pipeline registers). 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.

PEak applies multiple interpretations, a concept introduced by the Lava and Hydra functional HDLs [41], to the PE specification using 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 `a & b` with the implementation of `BitVector` as a Python type performs a functional simulation, with `magma`'s `Bits` type constructs a circuit, and with the `SMTBitVector` type constructs an SMT formula. To support multiple interpretations of control flow, the PEak compiler transforms input programs into single static assignment (SSA) form with *phi* nodes dispatching to a type-defined conditional operator. This technique allows the user to describe control flow using native Python `if` statements while leveraging dynamic dispatch to define the interpretation of control flow based on the input types.

3.1.2 Generating PE Hardware. PEak relies on `magma` 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, which defines new types by allowing `magma` to interpret the new type as 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` when generating hardware.

Lowering a PEak specification to `magma` is a straightforward process that captures the functional intent of the designer. The `__call__` method defines the state transition function that is executed on every positive edge of the clock. The PEak language encourages high-level specifications that eschew low-level details such as resource sharing, clock gating, and data gating. Rather than capturing these details at the PEak level, these concerns are addressed by optimization passes in the compiler tool chain. The `magma` compiler intermediate representation, CoreIR [12], is based on SMT [3], which enables formal equivalence checking of the input and output pairs of each pass. The `magma` circuits are tested with the `fault` [50] Python package using the function call syntax shown in Figure 4. Designers directly reuse functional tests for the hardware description as well, and `fault` generates a test bench for the design using a hardware simulator (e.g., Verilator [55]).

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

Fig. 3. PE ISA in PEak DSL.

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

Fig. 4. 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:
      res, flag = A.adc(B, c_in) # adc = add with carry
    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. 5. PE functional specification in PEak DSL.

To make writing PEak easier for developers, Peak's compiler performs a code transformation (SSA) that places certain restrictions on the user code. For example, the logic to transform the "return" statement from Python control flow semantics to hardware multiplexers requires that the user program "always returns" (i.e., there is a return statement in all possible paths of execution in the control flow graph). If the compiler finds that the user has violated this assumption, it will report an error (even though the user code might be valid Python).

By utilizing PEak to design our CGRAs, we can specify the operations in a PE, intraconnect of the PE, and number of inputs to and outputs from the PE. From the PE specification, the PEak backend automatically updates the application compiler by generating new rewrite rules for the PE, as described in the next subsection.

3.1.3 Generating Rewrite Rules for Compute Mapping. The mapping step in the application compiler requires rewrite rules (see Figure 2) that specify how subsets of CoreIR dataflow graphs map to hardware PEs. The PEak compiler transforms the `__call__` method into a normal form where each name is assigned to at most once, there is a single return at the end of the function, subcomponents are called once, and all `if` blocks are removed. We do this by first performing a fairly typical SSA pass. Additionally, we replace return statements with assignments to fresh names. Next, we inline the body of the if blocks and insert ternary expressions to select a final value of each assigned name. Finally, the return value must be inserted at the end of the function using a nested ternary expression over the possible return values. We then replace Boolean operators with their corresponding bitwise operators. Finally,ternary expressions are replaced with calls to the "ite" method on their condition, for example, `x if c else y` becomes `c.ite(x, y)`. Once in this form, applying `__call__` to abstract SMT variables (in the same way `__call__` is applied to concrete Python variables in Figure 4) 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:

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

The SMT definition of the *IRNode* and the *PE* are both constructed from the SMT interpretation of their respective PEak programs. For example, if we want to generate a rewrite rule for an add

operation, first we specify the operation in PEak. This can be done simply by writing a PEak program with two 16-bit inputs, a and b, that returns $a + b$. The *IRNode* SMT equation can be automatically generated using the SMT interpretation of this PEak program. It would include two 16-bit `BitVector` SMT variables and return the SMT equivalent of an add operation. Similarly, the *PE* SMT equation is automatically generated from the SMT interpretation of the PE specification.

If the SMT solver finds an *inst*, we have a rewrite rule between *IRNode* and *inst*. If it does not, we know that none exists. Further, a similar technique can be used to ensure that optimizations do not change the behavior of a design. For example, suppose that a rewrite rule has been discovered between *IRNode* and *inst* for *PE*. An optimized PE (*OPE*) can be verified with:

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

3.2 Lake

A Lake specification of a memory follows from the streaming memory abstraction [31] used in our application compiler (Section 5.2.2). Our abstraction bundles storage, address generation, and control logic in one structure for efficiency. It specifies the properties of the physical storage unit, including its type (registers or SRAM), depth, width, and ports. Each port is further specified by:

(1) The **iteration domain** of the statement instances in the application code that use the port (such as a read or write operation). The domain is defined by the bounds of loops in the loop nest surrounding the statement.
(2) The **access map** of the operations that maps points of the iteration domain to the value they read or write on the port.
(3) The **schedule** of all operations in the iteration domain that specifies the number of unstalled cycles between reset and each operation occurring.

Currently, our system supports cases in which both the access map and the schedule at each port are affine functions of the iteration domain (ID) variables. This allows the application compiler to use polyhedral analysis for loop fusion and memory optimization. The ID is implemented using a set of nested counters, whereas the access map and the schedule are generated by the address generator and schedule generator modules, respectively. The address generator and the schedule generator each implement an affine function of the ID counters, as shown in Figure 13.

The Lake compiler uses the Lake specification as the single source of truth from which it generates synthesizable Verilog, collateral for the application compiler describing the capabilities of the specified memory, and collateral for an SMT solver to generate the configuration bitstream that determines the data access pattern, as shown in Figure 6.

3.2.1 Lake Specification. A Lake specification allows the user to describe a memory as a graph, where vertices are memory ports and edges describe how these ports are connected, to allow for a composition of physical storage units. Ports are further grouped together in larger vertices corresponding to these storage units. To describe constraints on how the iteration domain and the address and schedule generators can interact with the physical storage unit, each port is characterized by the following properties:

- Port width: As a multiple of data word width.
- Operation performed: Either read, write, or read/write.
- Latency: The number of cycles between initiating the operation and actual data transfer.
- Initiation interval: The number of cycles between successive operations to the port.

Such a representation enables a user to describe simple memories such as a register file with one read port and one write port or a more complex memory such as a memory tile with two

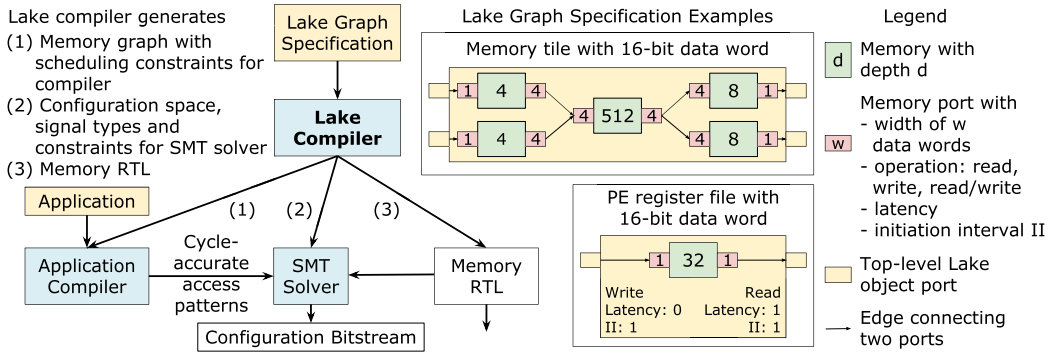


Fig. 6. Left: A program in the Lake DSL (Lake graph specification) serves as the single source of truth from which the Lake compiler generates (1) a description of capabilities and constraints of the memory hardware for the application compiler, (2) a list of configuration registers and interface signals as well as optional constraints on the configuration space for an SMT solver and (3) memory hardware. Right: Lake graph specifications for the memory tile in our CGRA (generated hardware shown in Figure 9) and the register file in our PEs (generated hardware shown in Figure 8).

read and two write ports. For example, we implement the memory tile in our CGRA with a wide single-ported memory to emulate a narrow multi-ported memory while achieving higher energy efficiency. To use such a wide memory, we require additional hardware: Figure 9 shows how SIPOs (serial-in parallel-out shift registers) aggregate data before writing to the wide memory and how PISOs (parallel-in serial-out shift registers) reorder and serialize data read from the wide memory. Figure 6 shows the graph specification for the register file in our PE (generated hardware in Figure 8) and our memory tile (generated hardware in Figure 9) in Lake.

Since Lake implements our streaming memory abstraction, a Lake specification allows the user to specify the functionality of the iteration domain, address generator, and schedule generator modules. Since our abstraction is implementation agnostic, the user can describe the implementation of address and schedule generators in a simple counter-based language, which is part of Lake and is based on scan from functional programming [18]. This description is not only given to the application compiler to communicate the capabilities of these modules but it also serves as a transformable IR for the Lake compiler to automatically apply rewrite rules to generate efficient RTLs (Section 3.2.2).

3.2.2 Generating Memory Hardware. The Lake compiler generates hardware from the Lake specification with the following rules:

- All memories are instantiated using SRAM macros if specified.
- Every edge is a directed wire connection between memory ports.
- Every edge has an associated schedule generator to generate read enables.
- Every edge has a set of shift registers that delay the write enable from the read enable based on the latency of the connected ports.
- Every edge has an associated iteration domain module to define the loop bounds for the affine access pattern.
- Every memory port has an associated address generator.
- Ports with more than one incoming edge generate multiplexers.
- Ports with more than one outgoing edge broadcast output data to all consumers.

The Lake compiler then automatically applies the following rewrite rules to generate more efficient hardware compared with the above baseline.

- Strength reduction: Multiplications in address and schedule generators are replaced with a sequence of additions [11]. Instead of computing affine access patterns with multiplications between iterators and strides (how much to increment loop variables by), we keep track of the previous access value and continually add the strides, dx and dy as shown in Figure 13, to the access value.
- Resource sharing: Nested counters in the iteration domain module are merged to remove redundant hardware.

For example, from the graph specification shown in Figure 6 for the memory tile, the Lake compiler generates the hardware shown in Figure 9, inferring multiplexers, address and schedule generators, and iteration domain modules.

Lake has the ability to describe a wide range of memories. Lake specifies three different qualities for each sub-memory in a tile: memory description, scheduling complexity, and addressing complexity. The memory description defines storage size (bitwidth, elements, banks), number of input and output ports, and whether the designer wants the ability to chain several memory tiles for increased capacity. Scheduling complexity determines the rate at which data can be written and read, how many reads and writes can be done in parallel per clock cycle, and whether memory allows for reads and writes in the same cycle. Finally, addressing complexity determines whether address generators are internal (in the MEM) or external (in the PE) and input and output address generator access pattern parameters if using internal generators. Similar to the PE, after changing the Lake description, the compiler will automatically update the memory mapping as described in the next subsection.

3.2.3 Generating Application Compiler Mapping Collateral. The Lake compiler generates the following collateral for the application compiler (see Section 5.2.2):

- The capacity and number of read, write, and read/write ports for each memory.
- The number of operations that can be done with a memory in one cycle (e.g., whether a read and write can be performed in one cycle).
- The port width for each memory port.
- The initiation interval for each memory port.
- A list of all the port connections to describe how the memory ports are connected.

The Lake compiler automatically generates a modified version of the Lake graph specification that connects at most one producer to at most one consumer along every edge to eliminate any multiplexers that would result in hardware generation. This representation is easier for the application compiler's polyhedral rewrite system (see Section 5.2.2) to consume.

Using this collateral, the application compiler generates application schedules that adhere to the constraints and capabilities of the generated hardware. These schedules directly specify the configuration register values for the generated memory hardware.

3.2.4 Generating SMT Solver Mapping Collateral. Given cycle-accurate schedules and access patterns generated by the application compiler (Section 5.2.2), the Lake compiler can use an SMT solver [51] to generate the configuration bitstream for memory hardware. The Lake compiler generates the following collateral for the SMT solver:

- RTL description of the memory hardware.
- A list of all interface signals and an annotation for each signal indicating whether it (1) is a configuration register that needs to be solved, (2) can be set to a constant value,

```
input logic                clk                                              CLK
input logic                rst_n                                            RSTN
input logic                tile_en                                          SET1
input logic [1:0] [15:0]   addr_in                                          SET0
input logic                strg_ub_agg_only_agg_write_sched_gen_0_enable     SOLVE
input logic [0:0] [15:0]   data_in                                          SEQUENCE
output logic               empty                                            X
```

Fig. 7. Excerpt from the collateral generated for the SIPO in Figure 9 for SMT solver indicating whether interface signals are clock (CLK) or reset (RSTN), can be assumed to be 1/0 (SET1 / SET0), are a configuration register value to be solved for (SOLVE), an input/output data sequence (SEQUENCE), or a value that does not matter (X).

(3) corresponds to a data input or output, (4) is a special signal (clock or reset), or (5) is a don't-care value (see example in Figure 7).

- An optional list of constraints on the configuration space based on the hardware (e.g., if the hardware has some number of bits for a configuration register where the entire range is not used, such as a three-bit signal with an upper bound of six, this constraint can be specified).
- An optional list of dependencies among configuration register values (e.g., if some configuration registers are not used depending on the value of another configuration, these dependencies can be specified).

To improve the scalability of the SMT solver for larger problems with Lake programs composed of multiple physical storage units, we can split the formal problem into several sub-problems that can be solved in parallel, one for each unit. For our memory tile in Figure 9, we solve sub-problems for the SIPOs, SRAM, and PISOs separately.

3.3 Canal

Canal represents a specification of the interconnect in the form of a directed graph. Notably, this specification is flexible enough to handle an arbitrary set of potentially heterogeneous PE and memory cores with different numbers of inputs and outputs. Like PEak and Lake, Canal generates both the hardware and the compiler collateral—in this case, the routing graph that the place-and-route (P&R) tool needs to map the dataflow graph onto the generated hardware, and the configuration bitstream that implements the routing result on the hardware. Canal allows designers to easily explore interconnect parameters, including network topology, placement of pipeline registers, and switchbox design.

3.3.1 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. For example, 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), and different standard interconnect topologies [13] can easily be imported and modified.

3.3.2 Generating Interconnect Hardware. We generate interconnect hardware by following these rules:

- Every edge is a directed wire connection.
- Vertices with more than one incoming edge generate multiplexers.

- Multiplexer select bits follow the incoming edge ordering.
- Vertices with special hardware attributes (e.g., a pipeline register) generate that hardware.

Canal also verifies structural correctness by comparing the connectivity of the generated hardware (from the RTL) with the original abstract graph using standard graph isomorphism algorithms [56]. Canal does not perform hardware optimizations; we rely on the Verilog synthesis tool (Design Compiler) to optimize the interconnect logic during synthesis.

Utilizing Canal to design our interconnect, we can specify the switchbox and connection topology, number of routing tracks, and pipeline register placement for a given interconnect. These choices affect the quality of routing and pipelining and, therefore, the maximum clock frequency at which an application runs on the CGRA. Canal then generates the relevant collateral needed for P&R and bitstream creation as described in the next two subsections.

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

3.3.4 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 configures these connections on the generated interconnect hardware.

4 ACCELERATOR ARCHITECTURE

Our CGRA [5] is composed of processing element (PE) and memory (MEM) tiles. Each tile has a core (PE or MEM) and interconnect components (five horizontal and five vertical bi-directional routing tracks, one switch box that routes all core outputs, and one connection box for each core input) which connect it to surrounding tiles. The PE, MEM and interconnect are specified in the PEak, Lake, and Canal DSLs, respectively, and composed together using `magma`. The CGRA is part of a system-on-chip (SoC) with a processor that drives applications on the CGRA. The SoC has four levels of memory hierarchy. Each PE has a small register file, the lowest level (L0) of the hierarchy. The MEM tile is at level one (L1). The CGRA is fed by the global buffer (L2), which is used for both bitstream configuration and streaming data in and out. Finally, the SoC is connected to an off-chip DRAM (L4). Each level of the memory hierarchy on the SoC is not a cache but rather implements the streaming memory abstraction described in Section 3.2 with software-programmable address and schedule generators, which can be targeted by our compiler. In the subsections that follow, we describe a baseline design for each block in our system and build upon that design with our architectural contributions. Three key points of novelty are (1) unique implementation of complex BFloat16 operations using several processing elements and memories, (2) memories optimized for the affine nature of access patterns in dense linear algebra applications, and (3) a specialized network for dynamic partial reconfiguration in the global buffer that allows fast application switching (3.5 μs) and multiple applications to run simultaneously.

4.1 Processing Element (PE) Core

Our baseline PE core consists of a 16-bit datapath that supports both the 16-bit integer (Int16) and 16-bit brain float (BFloat16) operations listed in Figure 8. A 1-bit datapath is driven by comparison operations or LUTs to control operations in other PEs. To implement complex BFloat16 operations (e.g., sine), we use multiple PEs with LUTs programmed in MEM tiles. For example, for non-local

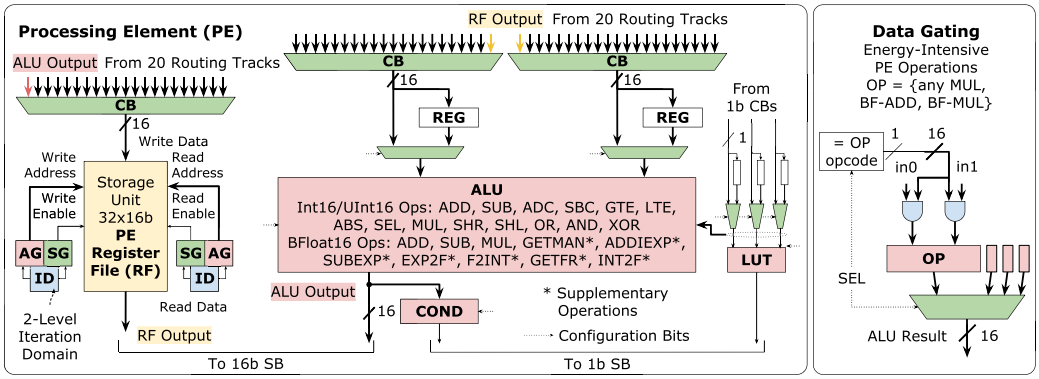


Fig. 8. PE architecture with an ALU, a LUT, and a small register file, and data-gating of complex operations.

means (NLM) application, this optimization avoids the latency overhead of having the CPU perform these computations (29.7× speedup) and avoids the area overhead of including the complex BFloat16 operations in each PE. On top of this baseline architecture, we data gate energy-intensive operations when they are not configured for use, as shown in Figure 8. Additionally, each PE contains a 32 × 16-bit register file, as the lowest level of the memory hierarchy for applications with heavy data reuse, such as neural networks.

4.2 Memory (MEM) Core

Our MEM architecture is specialized for dense linear algebra applications, which demonstrate affine access patterns. A baseline memory core consists of a narrow-width SRAM (width equal to the word-width of the datapath) with a simple affine function implementation (e.g., $s_x * x + s_y * y$ + offset) for generating the read-and-write address and enable signals. To improve this baseline architecture, instead of a narrow-width SRAM, a wide-width SRAM is used to increase the number of effective ports and to reduce access energy [53]. For example, using a 512 × 64-bit SRAM in the MEM core decreases the energy per access per byte by 2.0× from 1.65 pJ to 0.81 pJ when compared with a 16-bit-wide SRAM with the same capacity. Since we are using a wide-width SRAM, two serial-in, parallel-out (SIPO) and two parallel-in, serial-out (PISO) modules interface between the SRAM and the smaller-width datapath, supporting two input and two output ports. Further area optimizations are shown in Figure 9. First, sharing the configuration registers across address and schedule generators for the SIPOs, PISOs, and SRAM leads to 15% reduction in area. Second, replacing multipliers by adopting recurrence relations (as shown in Figure 13) to implement the affine functions in the address and schedule generators reduces area by another 39%. Finally, we add hardware to support memory tile chaining, which allows the compiler to map buffers in the application that exceed the capacity of a single MEM tile.

4.3 Interconnect

Both PE and MEM cores are connected over two island-style data interconnects, one for routing 16-bit signals and another for 1-bit signals. Figure 10 shows how switch boxes (SBs) and connection boxes (CBs) connect a tile's core to the data interconnect. SBs route data to and from tiles over 5 incoming and 5 outgoing 16-/1-bit routing tracks in each direction. Internally, SBs use the Imran topology [32], which "rotates" track numbers (i.e., IW2 links to ON3 mux instead of ON2 and so on) to increase routability. SBs also optionally pipeline data. CB muxes select input data for the core from the 20 routing tracks. One CB is instantiated for each input.

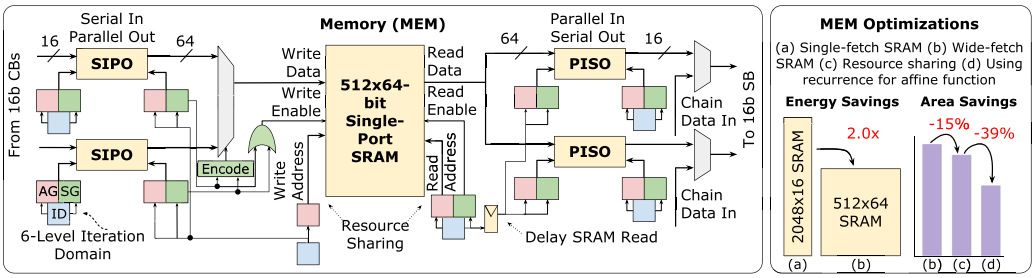


Fig. 9. Architecture of memory (MEM) core [5] and impact of energy and area optimizations.

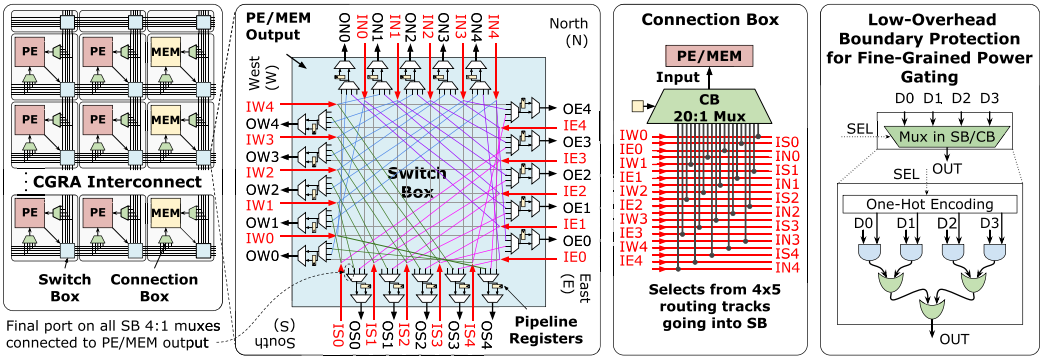


Fig. 10. CGRA has 16-bit and 1-bit (not shown) statically configured interconnects with low-overhead boundary protection for tile-level power gating.

Our CGRA also has tile-level power gating. This allows application P&R to turn off tiles that are not in use for a given application, eliminating leakage power consumption. To avoid the large overhead of adding isolation cells, we re-purpose the multiplexers already present in the SBs and CBs to provide isolation at power domain boundaries as shown in Figure 10 (right). Since all data going into a tile goes through an SB or a CB, instead of isolating all signals leaving an *off* domain, we isolate all signals entering an *on* domain [40]. This optimization reduces the area overhead of power domain boundary protection from 9% to 1%.

4.4 Global Buffer

The global buffer (GLB) is situated in between the DRAM and CGRA in the memory hierarchy. It serves the purpose of configuring the CGRA and streaming in and out data. It is composed of global buffer tiles, each corresponding to two columns of the CGRA, as shown in Figure 11. Each global buffer tile is composed of two SRAM banks, a load direct memory access (DMA), a store DMA, AXI, JTAG, and configuration switches to move data and configuration to and from the SoC, a ring switch to chain tiles for increased capacity, and an interconnect, which streams 16-bit words into the array. A baseline design would consist of one global buffer for data only, a separate configuration controller, and a 192 KB configuration buffer (two times the size of the CGRA configuration for double buffering).

Optimizations added to the global buffer include supporting fast dynamic partial reconfiguration (DPR), allowing flexible mapping of application kernels onto partial regions in the CGRA. Figure 11 shows how the GLB tiles not only stream data but also program partial bitstreams into the CGRA through a specialized configuration network. This allows the global buffer to double

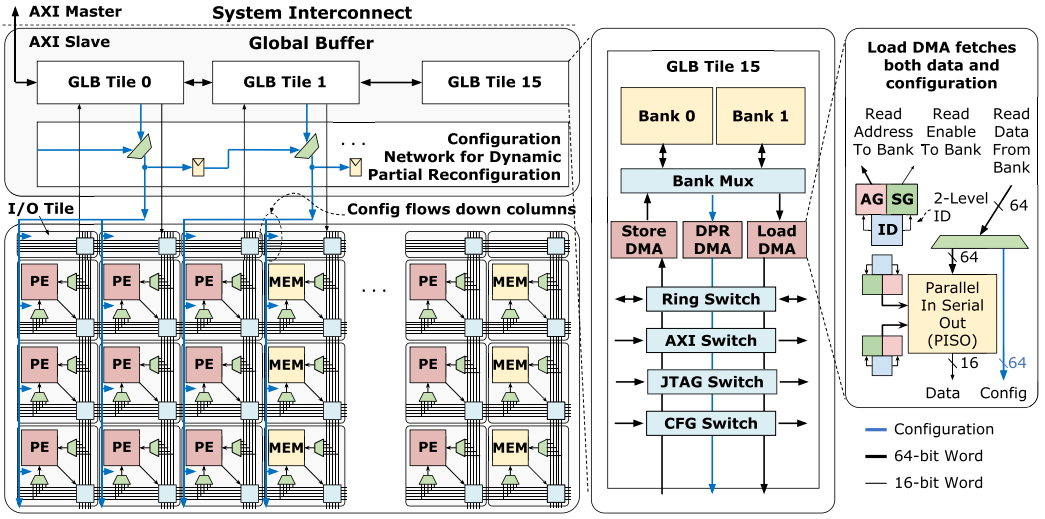


Fig. 11. Global buffer with dynamic partial reconfiguration.

	AXI-Lite	DPR	DPR Improvement
Peak throughput (MConfigs/s)	92.5	2345	25.3× ↑
Full CGRA configuration time (19824 registers) (μs)	214.3	8.5	25.3× ↓
Full CGRA configuration energy (mJ)	105.6	4.6	23.1× ↓
Application configuration time (Harris) (μs)	31.8	1.53	20.7× ↓

Fig. 12. Measured results for dynamic partial reconfiguration (DPR).

as a configuration memory and controller, reducing the 3.91 mm^2 baseline design to a 3.77 mm^2 design with a single global buffer and a specialized configuration network. Additionally, a user can set the configuration network to allow one GLB tile to broadcast the bitstream to all CGRA columns, one GLB tile to multicast to one or more columns, or multiple GLB tiles to unicast to their corresponding columns. Within the CGRA, the configuration interconnect distributes (32-bit address, 32-bit data) bundles vertically down each column and has a set of pipeline registers for higher configuration clock rates. This enables fast and flexible partial reconfiguration of the CGRA. Additionally, the GLB has the option to chain several tiles for applications that are data intensive. This allows an application that does not require a high input bandwidth to still access the capacity of the entire GLB. Figure 12 shows the impact of leveraging DPR to configure all CGRA partial regions in parallel, resulting in over 20× higher performance compared with configuring individual tiles through the system interconnect using the AXI-Lite protocol.

4.5 System on Chip

Figure 1 shows our SoC architecture. It contains the CGRA, GLB, and a processor (ARM Cortex M3 CPU), with an additional 128 KB of SRAM. The blocks on the SoC communicate over an ARM CoreLink NIC-400 system interconnect. Two DMA engines efficiently transfer data on and off chip over an ARM Thin Link (TLX) interface (48-bit) to and from a DRAM. Finally, there is also a peripheral subsystem to support interrupts, clocks, and UART. The CGRA accelerates the application, using the global buffer to store the input, output, and intermediate data. The CPU directs applications running on the CGRA, configuring the global buffer into different modes, sending the bitstream

and input data to the GLB and transferring the data out of the global buffer when the application is complete.

5 APPLICATION COMPILER

The role of the application compiler is to lower an application onto the CGRA hardware. This involves scheduling (e.g., tiling, unrolling), mapping logical operations onto physical hardware primitives (PEs and MEMs) that implement those operations, placement and routing on the CGRA interconnection network, pipelining for higher performance, CGRA bitstream generation given the final routing result, and, finally, code generation for the application processor.

5.1 Scheduling

Halide [44] is a multi-dimensional array processing language that serves as the frontend of our toolchain. Halide splits programs into two parts: the algorithm and the schedule. The algorithm specifies how to combine algebraic functions to compute the desired output values. The schedule specifies how to optimize performance or energy on the target hardware. Our compiler allows the user to quickly evaluate performance versus resource consumption trade-offs. The scheduling primitives we use are `split`, `reorder`, `tile`, `in`, and `unroll`. `Split` converts a single-loop variable into two nested variables. `Reorder` changes the ordering between innermost and outermost loops. `Tile` is syntactic sugar for splitting and reordering, which is commonly used for 2D image processing. `In` is used to specify a copy function to create memory hierarchies. This scheduling function explicitly copies data across the memory hierarchy, such as from a smaller memory tile to a larger global buffer memory. `Unroll` is used to remove loops by duplicating compute, which effectively decreases computation cycles given sufficient spatial resources.

The scheduling phase includes memory optimization passes that reduce the number of memory operations that must be considered for mapping. The simplest optimization is to inline constants. Naïvely, Halide functions with constant values (e.g., a convolution kernel with predetermined weights) would result in memories in which all updates are constant values. Inlining constants eliminates these memories. Second, iterative memory updates can be fused into a single update. For example, Halide has reduction domains to specify a series of read-modify-write operations (e.g., a convolution takes the sum of an $N \times N$ window). In the convolution example, Halide combines this computation, allowing us to use PEs connected in a multiply-add chain or tree, and only the final accumulated value is written to a memory. Then, the memory mapper is required to schedule only a single memory update. In Figure 13, we see that the Halide algorithm and schedule are transformed after scheduling, to a set of nested loops. The `blur` value is created using a single update consisting of four adds. Finally, we express LUTs as compute units in the compute graph rather than as memories, since they cannot be statically analyzed as they have data-dependent addresses for reading and writing. The compiler removes these LUT memories from the list of memories to be statically scheduled. In the compute mapping stage, these are mapped such that compute tiles dynamically calculate the address to read from or write into memory tiles.

The output of the scheduling phase is a compute graph emitted as a list of modules represented in the CoreIR JSON format (see example CoreIR graph visualized in Figure 2), as well as loop nests containing the memories' behavior that are fed into Clockwork [19], a tool that further optimizes memories using polyhedral analysis for higher performance.

5.2 Mapping

The mapping stage accepts a dataflow graph of logical operations (in CoreIR) that represents each application kernel. These logical operations must be mapped onto physical hardware units that can

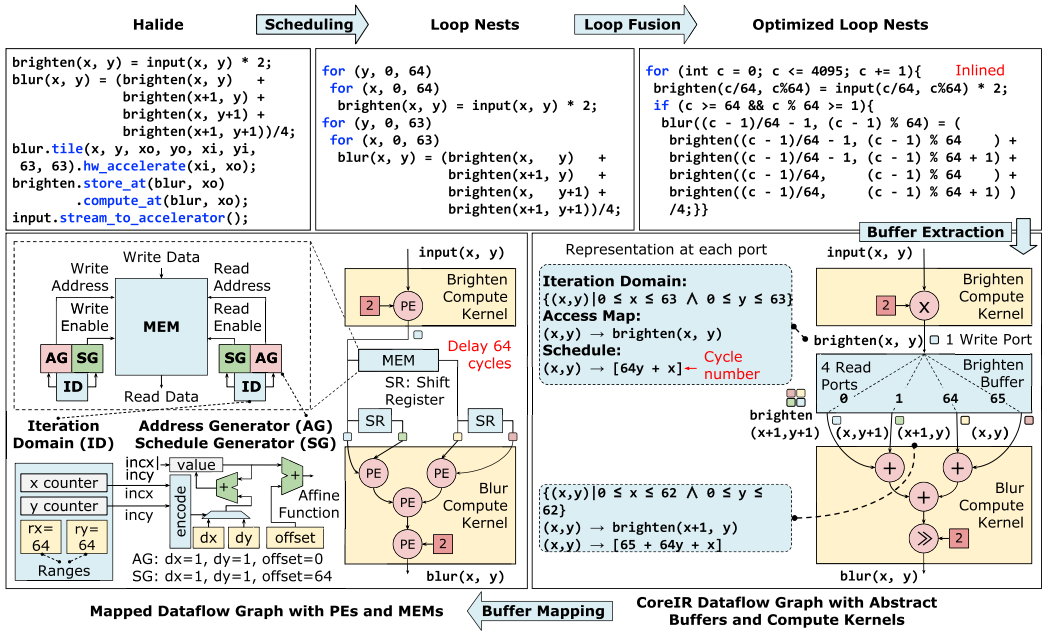


Fig. 13. Flow of a simple example Halide application (`brighten` then `blur`) through our compiler. The Halide application goes through scheduling which produces an intermediate representation with statements enclosed by loop nests. These loops are then fused to bring consumer statements as close as possible to the producer statements to minimize the amount of intermediate buffering. Buffer extraction then generates a dataflow graph of compute kernels and buffers, which is finally mapped to hardware PEs and MEMs on the CGRA.

then be configured to implement the operations. The mapping phase includes *compute mapping* onto PE tiles and *memory mapping* onto MEM tiles.

5.2.1 Compute Mapping. In the compute mapping stage, the logical operations to be transformed are primitive operations (e.g., adds, shifts, and multiplies). Using the method described in Section 3.1.3, we automatically generate a table of rewrite rules from the latest hardware specification of the PE. Each logical operation has a corresponding PE configuration that implements that operation. We also include rewrite rules for complex operations supported by the PE that can be described as a combination of multiple primitive operations, such as a fused multiply-add operation. Figure 14(a) shows how the compute mapping algorithm iteratively considers each rewrite rule and applies it to the nodes in the application graph until the graph of logical compute operations is transformed into a graph of PEs. A rewrite rule describes how to change a node in the application from a logical operation into a PE; thus, the application of a rewrite rule is simply replacing a logical operation node with a PE node. For example, in our rewrite rule table, we would have an entry for an add operation and the corresponding PE configuration implementing an add operation. In the application, we would apply this rewrite rule by replacing each add operation with the PE configured to implement the add operation. Compute mapping is complete when no logical operations remain to be mapped. In Figure 13, the CoreIR dataflow graphs of the `brighten` and the `blur` compute kernels get transformed into their mapped versions through the application of rewrite rules corresponding to multiplication, addition, and shift operations. Note that the order in which rewrite rules are considered dictates which will be applied first; thus, we ensure

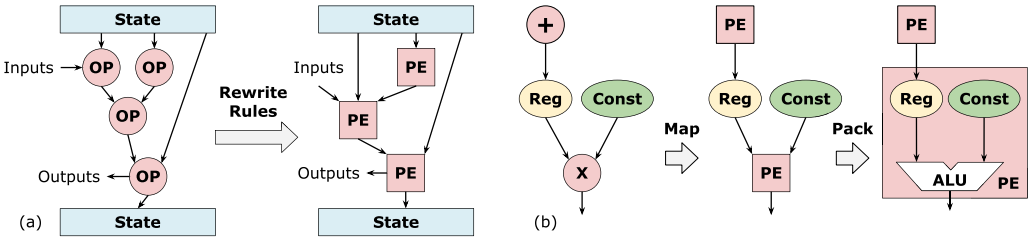


Fig. 14. Compute mapping and packing stages in the compilation flow. (a) Compute mapping transforms a graph of logical operations into a graph of physical PEs using rewrite rules automatically generated from the PEak specification. (b) The P&R packing stage combines registers, constants, and operations into PEs.

that rewrite rules that result in higher PE utilization (e.g., multiply by a constant) are considered before other rules (e.g., multiply).

5.2.2 Memory Mapping. In the memory mapping stage, the buffer abstraction described in Section 3.2 is extracted from the loop nest generated from Halide and is further mapped down to a concrete hardware implementation using the memories described in Sections 4.2 and 4.4. Memory mapping contains two separate steps, *buffer extraction* and *buffer mapping*. The first step uses polyhedral analysis to turn the multi-dimensional iteration space of a Halide-generated loop into one-dimensional cycle times at every buffer port. The second step takes the physical memory's resource constraints into consideration and recursively breaks the abstract buffers into simpler ones that can be directly mapped down to our CGRA.

As seen in Figure 13, the Halide-generated loop nest describes computation as operations on arrays over the iteration domain defined by index variables. Each read from and write to an array is bundled with a unique port on the corresponding buffer. For each port, the buffer extraction step computes an iteration domain, an access map that must be implemented by an address generator and, most importantly, a cycle-accurate schedule that maps operations to the cycle time when they will be executed in hardware. By using polyhedral analysis [19] to compute the data dependencies between operations, the memory extraction step fuses the loops and thus brings the consumer close to the producer yielding pipeline parallelism. For instance, in the `brighten` then `blur` application in Figure 13, the `brighten` operation produces an intermediate result which is further consumed by the 2×2 `blur` kernel. Polyhedral analysis fuses the loops and starts the `blur` operation as soon as we get the second pixel on the second row (cycle 65).

After buffer extraction, the memory mapper calculates storage duration from the access information of each port, from which it can infer the memory bandwidth and capacity requirements. The buffer mapping step derives a feasible and efficient memory implementation for each abstract buffer. The buffer mapping step may run into situations in which each physical memory may not have enough bandwidth or capacity or may be using an SRAM macro with a width that is higher than the width of the data. We address these challenges in the following ways:

- **Bandwidth:** Each physical memory on the accelerator may not have enough bandwidth to support computation without stalling. To address the need for high bandwidth, the buffer mapping stage has two strategies: shift register optimization and banking. Shift register optimization is possible when the delay between two ports is constant and the set of values that appear on the source subsumes the values that appear on the destination. For example, in Figure 13, the `brighten` buffer has constant delays of 0, 1, 64 and 65 between the write port and the four read portsl respectively. Remaining ports that cannot be served by shift registers are served from separate banks of memory with different address generation.

- **Capacity:** To map buffers with higher capacity than one memory tile on the CGRA, we chain several memory tiles into a single logical buffer. Our compiler statically analyzes the schedule and access map of each memory tile in the chain and sets the configuration accordingly.
- **Wide physical memories:** To make efficient use of a physical memory with a wide SRAM, the access pattern of the buffer is broken into sub-sequences with the same length as the SRAM width. The compiler strip-mines the innermost loops of the original program and adds wide loads and stores that are further mapped to the SIPO-to-SRAM and SRAM-to-PISO transfers shown in Figure 9.

Finally, this mapping produces configuration bits for each physical memory in the design.

5.3 Placement and Routing

Our compiler performs efficient power-domain-aware packing as well as placement and routing (P&R) of application kernels onto the CGRA. The P&R flow is similar to that of an FPGA with enhancements for power domain management. Utilizing fine-grained power domains, as described in Section 4.3, gives the P&R tool the freedom to choose any P&R topology. To select which tiles should be activated, the tool iteratively places and routes the application kernel onto the CGRA while trying to minimize the number of *on* tiles. Next, we describe the three phases: packing, placement, and routing.

5.3.1 Packing. Figure 14(b) shows how the packing stage groups operations into PE tiles, memory tiles, and the pipeline registers that are available in the interconnect. For example, if the packer analyzes the application graph and detects a pipeline register in front of a PE configured to perform an arithmetic operation, it will pack the register into the PE (i.e., the register available within the PE, shown in Figure 8). Additionally, instead of creating PE tiles for constants in applications, we pack those constants into the registers available in the PEs that consume the constants.

5.3.2 Placement. Placement is performed in two parts: global placement and detailed placement. We perform analytical global placement using the standard conjugate gradient method (similar to APlace [23]) on the summation of the cost of each net (Equation (1)), where each net is a connection between two tiles. The first part of the cost function is the half-perimeter wirelength (HPWL) between two tiles. In our algorithm, L2 distance is used as an approximation for HPWL. This approximation is used to reduce tool runtime. The second part of the cost function ensures that the memory tiles are placed correctly. Since the CGRA consists of both PE and MEM tiles, the memory placement after each iteration might not be legal and must be legalized to the closest memory column. Such legalization produces suboptimal results and slows down the convergence. To speed up the solving process, we apply potential functions as described in [7] on the MEM columns so that MEM nodes can be more easily placed into MEM tiles.

$$Cost_{net} = HPWL_{net,estimate} + MEM_{potential} \tag{1}$$

After global placement, we perform simulated annealing (SA)–based detailed placement [52]. We cannot use SA at the global placement stage because it does not converge. However, after global placement, SA is a good fit for power-aware placement because a power-related cost can be directly added to the total annealing cost. In particular, because tiles that only serve to implement pass-through routing must still be powered *on*, potentially long routes must be taken into consideration during placement. For the cost function for SA, we modify the global placement cost function by calculating the real HPWL, removing the potential function, and adding a new factor (Equation (2)) that, at each step, approximates the number of tiles used for routing a net (as an area) and penalizes the addition of new pass-through tiles. γ is a hyperparameter. The intuition

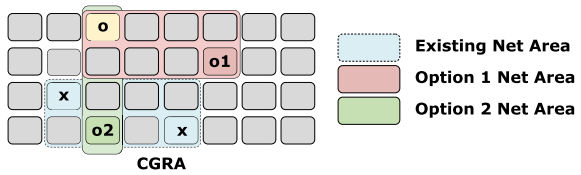


Fig. 15. Example shows an existing blue net area connecting two tiles marked "x." A new tile "o" may connect to either "o1" (option 1 net area) or "o2" (option 2 net area). The intersection of the existing net area with option 1 has an area of 0, and the intersection with option 2 has an area of 2, making option 2 the better choice.

behind the new adjusted term is that after each new placement, if the net covers a new area on the grid that has not been covered by any existing nets yet, then additional tiles are likely to be used as pass-through tiles. Using this cost function allows us to reduce the number of pass-through tiles and, thus, the number of powered-on tiles by encouraging the intersection of a new net area with existing net areas. An example is shown in Figure 15.

$$Cost_{net} = HPWL_{net} - \gamma \times (Area_{net} \cap Area_{existing}) \tag{2}$$

5.3.3 Routing. We use an iteration-based routing strategy described in [47] where overcongested nets are ripped up and rerouted to achieve a final legal result. In each iteration, we first compute the slack on each net and determine its timing criticality given global timing information. Next, we perform a routing pass based on the A* algorithm, a widely used path-search algorithm for weighted graphs. In this algorithm, costs factors are adjusted dynamically based on historical usage, net slack, and current congestion. The intuition is to balance routing congestion and timing criticality. A net with positive slack can use less-congested routes to provide space for timing-critical nets. We also adjust the wire cost functions to discourage utilizing wire tracks in unused tiles and to favor tracks in existing tiles. For example, when the first track in a tile has been chosen for routing, we dynamically reduce the cost of other wiring tracks in the same tile. These modified wiring costs decrease the number of tiles used. We finish iterating when a legal routing is produced.

5.4 Pipelining

The application compute kernels consist entirely of combinational logic, spanning from inputs to outputs. Adding pipeline registers along these long paths can greatly increase performance. An effective pipelining approach will balance delays across all pipeline stages and account for propagation delays (derived from static timing analysis in the physical design flow) through both the functional units and the interconnect components.

We add registers at the compute mapping stage and retime them during P&R. First, registers are added before each PE in the compute kernel. Registers are also added before and after the input tiles. However, naïvely adding pipeline registers before each PE results in imbalanced data waves (i.e., some paths through the compute kernel now take more cycles than others). To correct for this, every path in a compute kernel must span the same number of stages. We implement an algorithm that traverses every path in the compute kernel from input nodes to output nodes. For each path, the algorithm tracks the number of registers, which represents the number of cycles to reach a particular PE. For each PE, if one input arrives in fewer cycles than another input, the algorithm inserts registers to match the delays. When complete, all paths in the compute kernel are traversed in the same number of cycles.

Fully pipelining every functional unit in the compute kernel ensures that the critical path is never more than the delay through one PE. However, detailed VLSI modeling of our architecture suggests that the delay through a PE is roughly 3.5× the delay of one hop in the interconnect. Therefore, long routes (i.e., more than three hops) can easily become critical. As a result, retiming is necessary during P&R to minimize these long routes. As described in Section 5.3, registers in the application graph are packed into PEs whenever possible; thus, most pipeline registers are automatically packed into PEs. However, chains of additional pipeline registers must be placed in the routing fabric itself (see registers available in the switchbox in Figure 10). P&R models the delay along each route and retimes (i.e., moves) each register along the route until the critical path is minimized. Section 8 shows how pipelining significantly improves the performance of our applications.

5.5 CGRA Bitstream Generation

CGRA bitstream generation takes the output of the P&R stage and converts it into the corresponding configuration bits for the CGRA. The bitstream is sent to the CGRA through the GLB as described in Section 4.4. The steps for this conversion are as follows.

(1) Generate the routing result (automated by Canal, see Section 3.3.3).
(2) Perform a pass over the routing result to reduce switching power, utilizing a routing fix that alters the default routing to make sure unnecessary data is not routed over the CGRA. Specifically, this routing fix ensures that unused SBs do not unnecessarily forward switching data by configuring them to forward non-switching nets.
(3) Convert this altered routing result into a bitstream.
(4) Introspect each tile's core to determine whether it is a PE or a MEM core and produce the bitstream for the tile.
(5) Finally, do another pass to select unused tiles in the array and generate the power-gating configuration to disable them in the bitstream.

5.6 Application Processor Code Generation

The application processor invokes the accelerator (CGRA) for each kernel in an application and within each kernel for each tile of an image. The accelerator executes the target kernel but is not designed to efficiently run the remaining general-purpose code. Therefore, in the Halide schedule, we designate portions of the code to be accelerated on the CGRA; for the remaining portions, we generate application processor code using the base Halide compiler.

To identify the boundaries between the application processor and the accelerator, the user adds `stream_to_accelerator` and `hw_accelerate` calls within the Halide schedule. `stream_to_accelerator` is used at each input to designate input values to be copied from the DRAM to the accelerator's global buffer. The `hw_accelerate` call determines the loop level to offload to the accelerator. An example of these calls is shown in Figure 13, in which both the `brighten` and `blur` kernels are executed on the accelerator and the input image is streamed into the accelerator in tiles of 64×64 pixels.

6 RUNTIME

Three main components participate in end-to-end execution of an accelerated application: (1) an application processor capable of running the non-accelerated portion of an application and managing the accelerator life cycle and memory, (2) the CGRA fabric that is responsible for running hardware-accelerated functions, and (3) a control processor for the CGRA that configures the CGRA and orchestrates intermediate data movement. To allow interactions between the



Fig. 16. The three layers in RDAI: RDAI API (interfacing with the host code), host runtime, and platform runtime.

Linux-capable application processor and the RTOS-based control processor, we provide a set of software libraries and operating system drivers, collectively referred to as the runtime. From an implementation perspective, this system-level runtime is split into two smaller runtimes, one for the application processor and the other for the control processor.

Both the application and control processor runtimes are similarly architected: they consist of two pieces: (1) a static portion that does not need to change across accelerated applications and (2) a dynamic and typically application-specific configuration. The static portions of the runtimes provide services such as buffer allocation, memory transfer, application life cycle management, and so on. In essence, these static parts of the runtimes provide the supported "functions" in the system whereas the dynamic/application-specific configurations provide the parameters to those functions. Application-specific configurations for the application processor include the interface description of the generated accelerator (number of input/output ports, sizes of the buffers backing those inputs/outputs, and the accelerator identification information). On the control processor side, the application-specific configuration contains the accelerator bitstream, a description of the loop-nest containing the accelerated kernels from the Halide program, and some system-level metadata used to configure communication channels between the application processor and the control processor. The accelerated-kernel loop-nest is used by the control processor to schedule data movement between the system memory and the CGRA.

6.1 Application Processor and RDAI

The application processor runtime interfaces with both the non-accelerated portion of a Halide program—in the remainder of this text, we will refer to this code as the Halide C/C++ code—and the control processor. The interface with the Halide C/C++ code is standardized using an application programming interface—the reconfigurable device access interface (RDAI). The RDAI can be viewed as a subset of the OpenCL runtime API v3 [24] that can be decomposed into three essential layers: a C/C++ API used by the application to target the host runtime, a host runtime, and one or more platform runtimes. This layering of the RDAI system supports agility in two ways: (1) Device-specific functionality is pushed down the hierarchy as much as possible. This minimizes the number of sites where changes are needed when a new device/accelerator is available. (2) Reuse of the upper-layer services essentially means that no changes are needed at the interface between the host Halide C/C++ code and the runtime.

As shown in Figure 16, the RDAI API decouples the application code from the runtime. Each execution environment contains a single host runtime. This host runtime is a shim layer that is responsible for dispatching API calls to appropriate platform runtimes. The RDAI API itself is made of three groups of API calls: (1) management and life cycle APIs used to identify the execution environment and to start/stop the accelerator, (2) memory management APIs used to provision and migrate data buffers, and (3) tracing and error management APIs. The host runtime is a collection of software libraries and operating system drivers responsible for safely crossing the operating system boundary, allocating/deallocating contiguous memory buffers in the shared memory space, mapping/unmapping buffers in user space, and managing queues for platform runtimes.

6.2 Platform Runtime

Platform runtimes are the true workers in the RDAI world. They have intimate knowledge of the target platform on which accelerated functions run. Examples of platforms are CGRAs, FPGAs, and GPUs. While there can be at most one host runtime in an execution environment, multiple platforms can coexist in the same environment, each being uniquely designated by a vendor-name-version triple. Platform runtimes provide a unified abstraction over the differing underlying hardware platforms. Most of the services/APIs exposed by the platform runtimes are naturally similar to those found in OpenCL: command queues, management of kernel objects, data movement between shared system memory and device memory, and so on. In the two-processor configuration (application processor and control processor) for our CGRA-based SoC, the platform runtime forms yet another indirection layer standardizing different communication schemes with the underlying control processor. Communication between the platform runtimes and the control processor allows actions to be triggered in the CGRA subsystem. The supported channels for this communication are hardware mailboxes, software mailboxes, memory-based command queues, and custom protocols carried over serial links (e.g., UART).

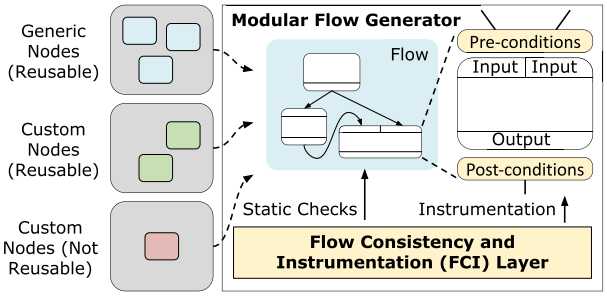
6.3 Control Processor

The ARM Cortex-M3 control processor turns the CGRA subsystem into a master device capable of initiating memory transactions at the system level. On one hand, this relieves the application processor of the task of multiplexing and orchestrating all of the different execution schedules for the various hardware platforms in the system. On the other hand, it also simplifies the design of the control processor task scheduler. That is, the control processor is fed the loop nest schedule as a dynamic configuration information and it uses that information to (1) schedule DMA transfers between global system memory and CGRA's global buffer, (2) prefetch data buffers, and (3) report execution status back to its designated platform runtime. The control processor's runtime is split into a generic scheduler and a CGRA-specific support library. The scheduler is static (i.e., it does not change across different CGRA architectures). Software patches to hardware bugs or architectural changes in the CGRA are directly implemented in the control processor's CGRA support library.

7 ACCELERATOR IMPLEMENTATION

Within an evolving accelerator-compiler co-design flow, each silicon prototype is not an isolated effort and is likely to be followed by future iterations. Unfortunately, tapeouts in advanced technologies involve tremendous effort, and non-recurring engineering costs have only continued to rise. Physical design (PD) methodologies for evolvable flows must prioritize support for efficient code reuse from previous iterations, even as scripts are aggressively customized for specific technologies, vendor libraries, and design considerations.

We build our design with mflowgen [6], a new tool that adopts an agile approach designed to support high code reuse based on *modular flow generators* coupled with a *flow consistency and instrumentation (FCI) layer* embedded in Python. Unlike existing parameterized Tcl templates and generators offered by commercial EDA vendors, the goal of a modular flow generator is not to emit Tcl but to provide the necessary abstractions to compose and reuse code. Figure 17(a) shows how a modular flow generator composes modular nodes from both generic sources (both technology- and design-agnostic) and custom sources. Nodes may be parameterized and can capture large code blocks (e.g., synthesis, power) or lightweight operations (e.g., glue scripts) with no restriction on the language used. Because nodes from different sources can be inconsistent with each other when composed, we introduce an FCI layer that statically analyzes the constructed graph and includes mechanisms to check for properties across a distributed code base (e.g., heights of different tiles



(a)

SoC Physical Design Parameters	
% of codebase (LoC) reused from common library	30%
% of codebase (LoC) reused from previous designs	50%
Months to tapeout	6
Static check runtime	2.2 sec
Physical hierarchy levels	3
Hierarchical blocks	7
Number of power domains	512
Number of clock domains	3
Tile abutment	Yes

(b)

Fig. 17. A modular approach to physical design flows – (a) Traditional Tcl-based scripts make code difficult to reuse. Modularity allows for reuse, while flow assembly in a high-level language (e.g., Python) enables new language features and the chance to augment the flow with agile mechanisms. (b) Our SoC includes non-trivial physical design complexity that encourages reuse across blocks within a generation as well as future iterations of the evolvable SoC.

in the CGRA must match) that must hold for nodes to compose. The layer also instruments nodes to support runtime assertions and to enable sharing prebuilt nodes across a team.

Figure 17(b) lists the physical design parameters of our silicon prototype. The SoC was constructed hierarchically bottom-up with tiles (i.e., PE, memory, and global buffer tiles) abutted to form arrays (i.e., CGRA compute array, global buffer array), which were then placed into the full chip along with other blocks (i.e., processor subsystem, global controller, peripherals). Physical design scripts and features—for example, code to implement power domains—were modularized into nodes, and each hierarchical block constructed a Python graph representing its physical design flow. We achieved significant code reuse with 80% of lines of code reused either from custom nodes designed for previous iterations of the SoC (50% of codebase) or from a common library of technology- and design-agnostic nodes (30% of codebase).

8 EVALUATION

In this section, we make the case that an agile approach to compiler-accelerator co-design can deliver competitive efficiency and performance for modern workloads. We evaluate the performance and energy consumption of our end-to-end system as well as the iteration time to run the complete agile toolchain. Finally, in the last subsection, we explore different PEs, MEMs, and interconnects and evaluate them, demonstrating the flexibility of our system. For evaluation, we design the complete SoC shown in Figure 1 with the parameters listed in Table 1 and implement it in a commercial 16 nm technology.

8.1 Applications

We evaluate the image processing and computer vision applications listed in Table 2. The applications are written in Halide [39]; we apply a set of techniques including scheduling optimizations, unrolling, and pipelining to demonstrate how each application can be progressively tuned to improve performance, energy, and resource utilization on our end-to-end system.

8.2 Impact of Scheduling Optimizations

We first quantify the impact of the scheduling optimizations described in Section 5.2.2 that leverage polyhedral analysis to transform loops and create more efficient schedules. Without these

Table 1. SoC Parameters

SoC Parameters	
Global Buffer Size	4 MB
Global Buffer Word Width	64 bit
CGRA Dimensions	32 × 16
Number of PE Tiles	384
Number of MEM Tiles	128
MEM Tile Buffer Size	4 KB
PE/MEM Word Width	16 bit

Table 2. Applications Used for SoC Evaluation

Application	Description	Image Size	Tile Size
Harris	Harris corner detection	1,530 × 2,554 × 3	64 × 64 × 3
Blur	Separable 3 × 3 image blur	6,400 × 4,800	56 × 62
Camera	Processes raw image into RGB image using hot-pixel suppression, demosaicing, color correction, gamma correction, and contrast enhancement	2,568 × 1,928	64 × 64
Unsharp	Enhances local contrast by isolating the high-frequency content of an image, and combining it with the original image	1,536 × 2,560 × 3	64 × 64 × 3

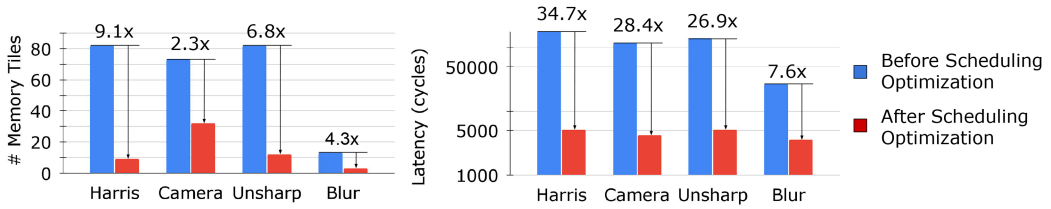


Fig. 18. Impact of scheduling optimizations from Section 5.2.2 on memory tile usage and application latency.

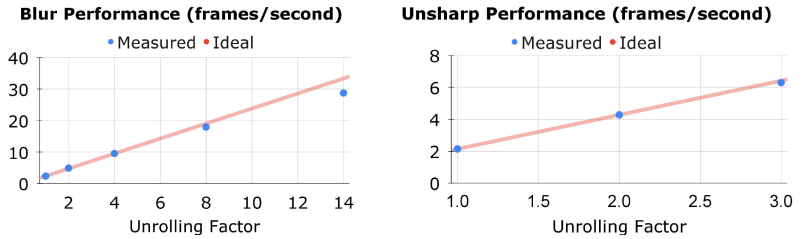


Fig. 19. Measured versus ideal application performance with increasing parallelization (loop unrolling).

optimizations, we obtain mappings with high latencies and a large number of MEM tiles, and these inefficiencies are magnified by later optimizations that unroll and pipeline each application kernel. Figure 18 shows significant reduction in both the number of memory tiles (2.3×–9.1×) and the application execution latency (7.6×–34.7×) as a result of fusing loops and scheduling producer and consumer statements closer together in time.

8.3 Impact of Unrolling

Unrolling the outer loop around the application kernel allows for increased utilization of CGRA resources. An input tile is unrolled across the GLB tiles and streamed into the CGRA to produce an unrolled output tile. To show the benefit and overhead of such parallelization, we show application performance with different unroll factors for both Blur and Unsharp in Figure 19. The maximum amount of unrolling is determined by either the I/O (maximum of 16 ports between the GLB and CGRA) or the number of PE and MEM tiles and routing resources needed by the application kernel. For Blur, we unrolled the kernel by 2, 4, 8, and 14. For Unsharp, we unrolled the kernel by 2 and 3. The ideal runtime is calculated by taking the result without unrolling and dividing by the unroll

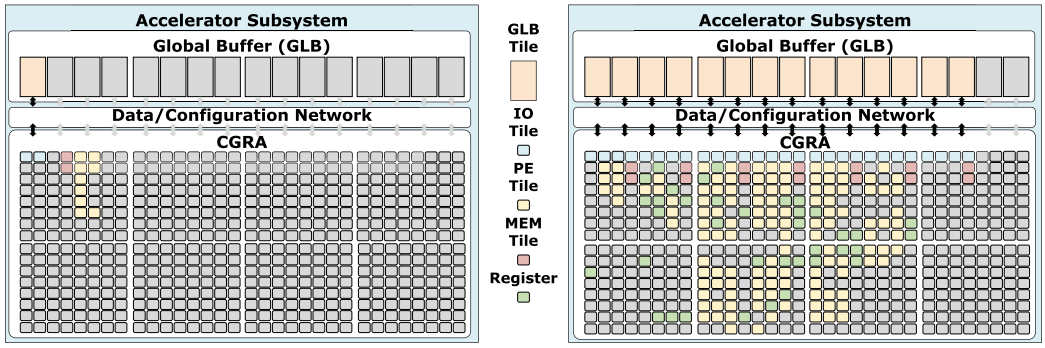


Fig. 20. Visualization of CGRA resource utilization for Blur with unroll = 1 (left) and unroll = 14 (right), showing how a large unroll factor scales utilization.

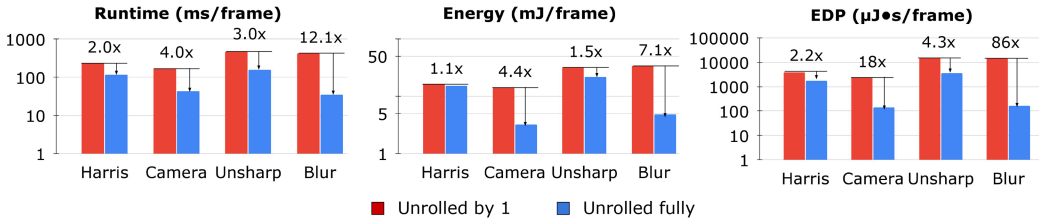


Fig. 21. Runtime, energy, and energy delay product (EDP) for applications unrolled fully versus unrolled by 1.

amount. As kernels are further parallelized, there is a slight scheduling overhead introduced by the compiler, but the measured performance closely tracks ideal performance. Figure 20 visualizes the CGRA resource utilization for Blur with unroll = 1 and unroll = 14, demonstrating how a large unroll factor scales utilization. Figure 21 applies maximum unrolling to each application kernel and shows a significant improvement in runtime over an unroll = 1 baseline. Note that since unrolled applications have higher utilization, the power consumption increases, but there is a significant runtime improvement (2.0×–12.1×) resulting in 1.1×–7.1× lower energy consumption, and a 2.2×–86× improvement in energy delay product (EDP).

8.4 Impact of Pipelining

Pipelining the application graph as described in Section 5.4 allows our design to run at significantly higher frequencies. The potential frequency benefit of pipelining can vary widely and is limited by the need to balance delays across data waves. Specifically, every pipeline register added to the critical path to increase frequency will delay the arrival of the data at the destination tile. All other data arriving at that tile must be similarly delayed by one cycle to preserve functionality. When all available pipeline registers along a path are already allocated and in use (e.g., in switch boxes, in PEs), the application kernel cannot be pipelined any further without performing pipelining and placement and routing iteratively. Such iterative placement and routing with pipelining is our immediate future work, which will allow us to achieve frequencies closer to 1 GHz (as evidenced by the individual operation delays on our CGRA shown in Figure 22). Figure 24 shows how a single pass of pipelining improves runtime by 5.5×–24.2×, energy consumption by 1.7×–9.1×, and EDP by 9.5×–221× for the different applications.

Op	Delay (ns)	Op	Delay (ns)
SwitchBox	0.14	Abs	0.49
Add	0.52	FpAdd	0.61
Sub	0.48	FpMul	0.75
UMult0	0.57	FpGetMant	0.41
UMult1	0.67	FAddIExp	0.45
UMult2	0.67	FSubExp	0.52
SMult0	0.59	FCnvExp2F	0.52
SMult1	0.65	FGetFInt	0.36
SMult2	0.70	FGetFFrac	0.56
And	0.55	FCnvSInt2F	0.43
Or	0.57	FCnvUint2F	0.36
Abs	0.49	FCnvSInt2F	0.43

Fig. 22. PE operation delays.

Application	Blur	Unsharp	Camera	Harris
Target Output Rate (pixels/cycle)	14	9	3	2
Temporal Occupancy	72%	83%	73%	83%
Frequency	360 MHz	260 MHz	320 MHz	380 MHz
# PE / 384	266	303	294	206
# MEM / 128	14	36	34	17
# GLB / 16	14	9	3	6
# 1-bit Routing Tracks / 10240	69	296	417	226
# 16-bit Routing Tracks / 10240	1743	1892	1410	791

Fig. 23. CGRA resource utilization for applications.

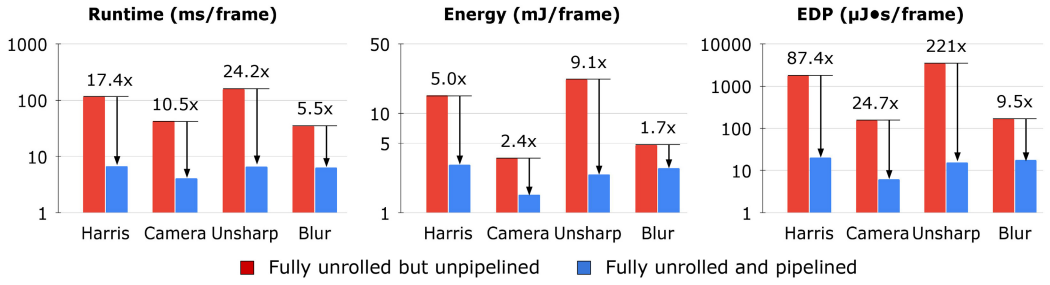


Fig. 24. Effect of pipelining on fully unrolled applications.

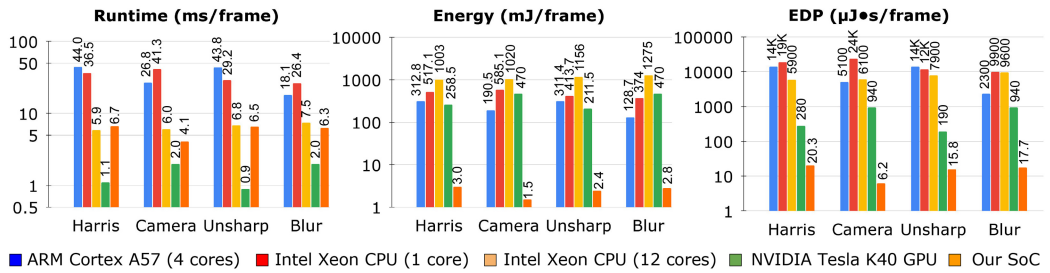


Fig. 25. Comparison with CPU and GPU platforms.

8.5 Comparison with CPU and GPU Platforms

The final resource utilization statistics for each application can be found in Figure 23. We compare our design against implementations from [39] on CPU and GPU platforms (ARM Cortex A57 CPU, a single-core Xeon CPU, a multi-core Xeon (12 cores) CPU, and a Tesla K40 GPU) in Figure 25. [39] uses the Halide auto-scheduler to generate highly optimized code that maximally utilizes SIMD and multi-core capabilities of these platforms. Compared with the ARM Cortex A57, the Intel Xeon CPU (1-core), Intel Xeon CPU (12-core), and the NVIDIA Tesla K40 GPU, our design performs 130×–887×, 561×–3878×, 291×–986×, and 12×–152× better in terms of EDP, respectively. Our work shows that an agile approach to compiler-accelerator co-design can deliver competitive efficiency and performance for modern workloads.

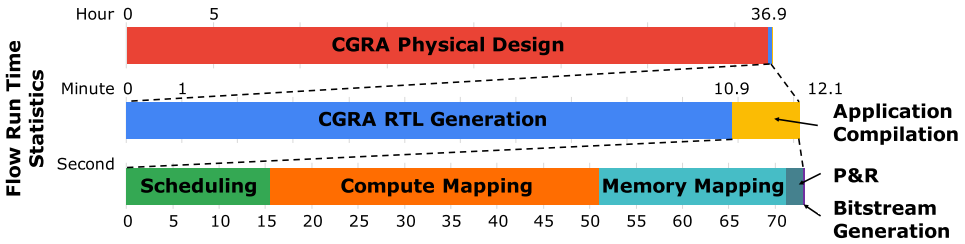


Fig. 26. Toolchain runtimes to regenerate a CGRA accelerator spanning from the application and hardware DSL specifications down to physical layout.

Table 3. Specifications of PE Designs Used for PE Exploration

	PE Design 1	PE Design 2	PE Design 3
Feature 1	16-bit and 1-bit integer ops	16-bit and 1-bit integer ops	16-bit and 1-bit integer ops
Feature 2	+ Floating point ops	+ Floating point ops	+ Floating point ops
Feature 3		+ Register File	+ MAC and 3-input add
Feature 4			+ Register File
PE Area (μm^2)	4,523	8,806	10,095
PE Max Frequency (MHz)	1,428	1,428	909

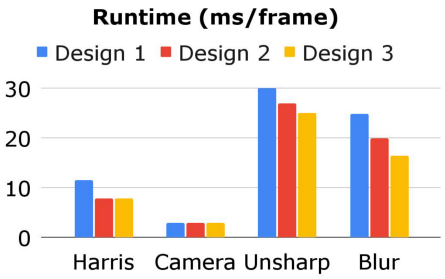
8.6 Toolchain Runtime

Iterating on a design in an agile manner requires fast turnaround times. Figure 26 shows that our toolchain takes less than two minutes for scheduling, compute mapping, memory mapping, P&R, bitstream generation, and CGRA generation. Application P&R has a low runtime compared with the rest of the compiler since CGRAs are coarse grained and our current P&R tool exits on the first routable result. Compared with physical design, our toolchain is extremely fast and allows the quick generation of new designs.

8.7 Design Space Exploration

Our flexible compiler-accelerator generation framework enables quick design space exploration of PEs, MEMs, and interconnects. For each exploration, several design points are generated and benchmarked on a suite of applications. In the following subsections, we demonstrate fast design space exploration on our suite of image processing applications.

8.7.1 PE Exploration. Two important aspects of PE design are what operations they support and internal storage options. Therefore, we explore PEs with register files as well as different operation types: integer, floating point, and specialized operations for image processing applications. We compare the three PEs in Table 3. Design 1 is a baseline PE with 16-bit and 1-bit integer operations, and floating point operations. Design 2 adds a register file to this PE. Design 3 specializes the PE for image processing applications by adding multiply-accumulate and three input add instructions. The area and maximum frequency for each PE are shown in Table 3. The runtime and resource utilization for our set of image processing applications (each with unroll = 1) are shown in Figure 27. The register file in design 2 allows us to improve runtime in cases in which we can use the register files to pipeline data. The only exception is camera pipeline, for which pipelining with register files requires more register files than those on the CGRA. Thus, we use interconnect registers to pipeline this application instead. Specializing the PE in design 3 reduces PE utilization since several operations can occur in a single PE. This experiment shows that by utilizing PEak and our generator/compiler framework, we can quickly compare PE designs on application-level metrics.



Application		Design 1	Design 2	Design 3
Harris	PE	91	91	80
	MEM	6	6	6
	Reg File	0	132	122
	Reg	87	57	47
Camera	PE	306	306	303
	MEM	26	26	26
	Reg File	0	0	0
	Reg	253	253	256
Unsharp	PE	101	101	74
	MEM	11	11	11
	Reg File	0	150	124
	Reg	97	75	71
Blur	PE	20	20	14
	MEM	2	2	2
	Reg File	0	28	22
	Reg	24	18	23

Fig. 27. PE exploration: Application runtime and resource utilization for three different PEs.

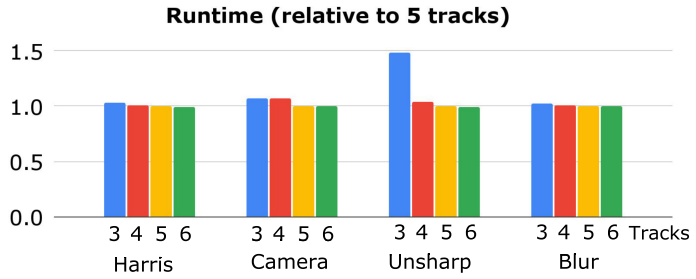
	MEM Area ($1000^* \mu m^2$)	SRAM Area (%)
DP SRAM with Address Generators	20	66
4 Wide SP SRAM with Address Generators	22	25

Application		DP SRAM	4 Wide SP SRAM
Harris	PE	103	97
	MEM	11	5
Camera	PE	331	294
	MEM	28	14
Unsharp	PE	101	101
	MEM	9	6
Blur	PE	19	19
	MEM	3	1

Fig. 28. Memory exploration: Dual port (DP) SRAM vs. 4-wide single port (SP) SRAM-based memory tiles.

8.7.2 MEM Exploration. Our accelerator is designed to take advantage of the affine nature of addressing in image processing applications. However, there are many options on how to implement this in hardware. For MEM exploration, we compare two implementations of a streaming memory abstraction: a dual-port (DP) SRAM with optimized address generators and a single-port (SP) SRAM with a fetch width of 4. Figure 28 compares the area breakdown for the tile and the resource utilization. Swapping out the DP SRAM for a 4 wide SP SRAM increases the area per MEM tile by 2,000 μm^2, but approximately halves the total memories used since there is higher bandwidth with 2 input and 2 output ports. Note that this does not have an effect on runtime. In some cases, the PE count also decreases as the P&R tool is able to pack constants or registers into PEs (as described in Section 5.3.1).

8.7.3 Interconnect Exploration. A key decision in interconnect design is determining the number of tracks per switchbox. In this experiment, we continue to use the Imran switchbox in our fabric and sweep over the number of tracks per switchbox on our set of image processing applications (unroll = 1). Figure 29 shows the runtime and area for each switchbox configuration. As we increase the number of tracks, the tool may find shorter routes, which may lead to higher frequencies and, therefore, lower application runtimes. As expected, at some point, extra tracks are not used and will only add an area overhead. Here, we see that for some applications, going from 3 to 5 tracks reduces runtime. After that, however, there is no significant reduction. This demonstrates our ability to change a hardware knob in our design and quickly evaluate its effect on performance on a suite of applications.



Number of Routing Tracks	Switchbox Area (μm^2)
3	1278
4	1696
5	2158
6	2534

Fig. 29. Interconnect exploration: Runtime and area for designs with different numbers of routing tracks.

9 FUTURE WORK

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, and 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 our hardware-compiler co-design approach, there is a massive opportunity to automate large-scale design space exploration (DSE) of accelerator architectures. We will explore these DSE frameworks in our future work, particularly, automating the PE specification process by analyzing a set of application kernels. Automating the specification of PEs will allow the designer to quickly converge on efficient PE designs for a set of applications.

Another key direction to further develop our hardware-compiler co-design system is accelerating sparse applications. To accelerate sparse applications, a few portions of the framework must change: the front-end language (Halide), static scheduling, and memory design, while the rest of the flow can be reused. To be more specific, by developing new hardware primitives in conjunction with a new spatial dataflow IR targeted by a TACO [26] frontend, we can easily swap in sparse components for their dense counterparts (Halide to TACO, Clockwork to Concrete Index Notation [25], dense MEM to sparse MEM) and leverage the rest of the existing infrastructure to quickly iterate, test, and build new highly reconfigurable chips. We will explore these opportunities in our future work.

10 CONCLUSION

We have demonstrated an agile methodology that automates the co-design of accelerators and compilers by utilizing the key insight that we can automatically update the compiler as the hardware evolves. We use domain-specific languages and formal methods to automatically generate both the accelerator hardware and its compiler from a single source of truth. Our system 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] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. 2020. Chipyard: Integrated design, simulation, and implementation framework for custom SoCs. *IEEE Micro* 40, 4 (2020), 10–21. https://doi.org/10.1109/MM.2020.2996616

[2] 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, and Keyi Zhang. 2020. Creating an agile hardware design flow. In *57th ACM/IEEE Design Automation Conference (DAC'20)*. 1–6. https://doi.org/10.1109/DAC18072.2020.9218553

[3] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2016. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org.

[4] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: High-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 33–36. https://doi.org/10.1145/1950413.1950423

[5] Alex Carsello, Kathleen Feng, Taeyoung Kong, Kalhan Koul, Qiaoyi Liu, Jackson Melchert, Gedeon Nyengele, Maxwell Strange, Keyi Zhang, Ankita Nayak, Jeff Setter, James Thomas, Kavya Sreedhar, Po-Han Chen, Nikhil Bhagdikar, Zachary Myers, Brandon D'Agostino, Pranil Joshi, Stephen Richardson, Rick Bahr, Christopher Torng, Mark Horowitz, and Priyanka Raina. 2022. Amber: A 367 GOPS, 538 GOPS/W 16nm SoC with a coarse-grained reconfigurable array for flexible acceleration of dense linear algebra. In *2022 IEEE Symposium on VLSI Technology and Circuits (VLSI Technology and Circuits)*. 70–71. https://doi.org/10.1109/VLSITechnologyandCir46769.2022.9830509

[6] Alex Carsello, James Thomas, Ankita Nayak, Po-Han Chen, Mark Horowitz, Priyanka Raina, and Christopher Torng. 2022. mflowgen: A modular flow generator and ecosystem for community-driven physical design. In *Design Automation Conference* (*DAC*).

[7] Yu-Chen Chen, Sheng-Yen Chen, and Yao-Wen Chang. 2014. Efficient and effective packing and analytical placement for large-scale heterogeneous FPGAs. In *2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'14)*. IEEE, 647–654. https://doi.org/10.1109/ICCAD.2014.7001421.

[8] Yuze Chi, Jason Cong, Peng Wei, and Peipei Zhou. 2018. SODA: Stencil with optimized dataflow architecture. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'18)*. 1–8. https://doi.org/10.1145/3240765.3240850

[9] S. Alexander Chin, Noriaki Sakamoto, Allan Rui, Jim Zhao, Jin Hee Kim, Yuko Hara-Azumi, and Jason Anderson. 2017. CGRA-ME: A unified framework for CGRA modelling and exploration. In *IEEE 28th International Conference on Application-Specific Systems, Architectures and Processors (ASAP'17)*. 184–189. https://doi.org/10.1109/ASAP.2017.7995277

[10] Nitin Chugh, Vinay Vasista, Suresh Purini, and Uday Bondhugula. 2016. A DSL compiler for accelerating image processing pipelines on FPGAs. In *Proceedings of the International Conference on Parallel Architectures and Compilation*. 327–338. https://doi.org/10.1145/2967938.2967969

[11] Keith D. Cooper, L. Taylor Simpson, and Christopher A. Vick. 2001. Operator strength reduction. *ACM Transactions on Programming Languages and Systems* 23, 5 (2001), 603–625.

[12] Ross Daly, Leonard Truong, and Pat Hanrahan. 2018. Invoking and linking generators from multiple hardware languages using CoreIR. In *Workshop on Open-Source EDA Technology (WOSET'18)*. https://woset-workshop.github.io/PDFs/2018/a11.pdf.

[13] Jules R. Degila and Brunilde Sanso. 2004. A survey of topologies and performance measures for large-scale networks. *IEEE Communications Surveys Tutorials* 6, 4 (2004), 18–31. https://doi.org/10.1109/COMST.2004.5342296

[14] 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 *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'20)*. 408–422. https://doi.org/10.1145/3395633

[15] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. 2012. DySER: Unifying functionality and parallelism specialization for energy-efficient computing. *IEEE Micro* 32, 5 (2012), 38–51. https://doi.org/10.1109/MM.2012.51

[16] James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. 2014. Darkroom: Compiling high-level image processing code into hardware pipelines. *ACM Transactions on Graphics* 33, 4, Article 144 (July 2014), 11 pages. https://doi.org/10.1145/2601097.2601174

[17] James Hegarty, Ross Daly, Zachary DeVito, Jonathan Ragan-Kelley, Mark Horowitz, and Pat Hanrahan. 2016. Rigel: Flexible multi-rate image processing hardware. *ACM Transactions on Graphics* 35, 4, Article 85 (2016), 11 pages. https://doi.org/10.1145/2897824.2925892

[18] Ralf Hinze. 2004. An algebra of scans. In *International Conference on Mathematics of Program Construction*. Springer, 186–210.

[19] Dillon Huff, Steve Dai, and Pat Hanrahan. 2021. Clockwork: Resource-efficient static scheduling for multi-rate image processing applications on FPGAs. In *IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'21)*. 186–194. https://doi.org/10.1109/FCCM51124.2021.00030

[20] Intel Inc. [n. d.]. Altera OpenCL. https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html.

[21] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia* (Orlando, FL) *(MM'14)*. ACM, New York, NY, 675–678. https://doi.org/10.1145/2647868.2654889

[22] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-datacenter performance analysis of a tensor processing unit. *SIGARCH Comput. Archit. News* 45, 2 (June 2017), 1–12. https://doi.org/10.1145/3140659.3080246

[23] Andrew B. Kahng, Sherief Reda, and Qinke Wang. 2005. APlace: A general analytic placement framework. In *Proceedings of the 2005 International Symposium on Physical Design* (San Francisco, CA) *(ISPD'05)*. ACM, New York, NY, 233–235. https://doi.org/10.1145/1055137.1055187

[24] Khronos® OpenCL Working Group. [n. d.]. The OpenCL™ C Specification. Retrieved July 13, 2022 from https://www.khronos.org/registry/OpenCL/specs/3.0-unified/pdf/OpenCL_C.pdf.

[25] Fredrik Kjolstad, Peter Ahrens, Shoaib Kamil, and Saman Amarasinghe. 2019. Tensor algebra compilation with workspaces. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'19)*. 180–192. https://doi.org/10.1109/CGO.2019.8661185

[26] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA, Article 77 (Oct 2017), 29 pages. https://doi.org/10.1145/3133901

[27] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Spatial: A language and compiler for application accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA) *(PLDI'18)*. ACM, New York, NY, 296–311. https://doi.org/10.1145/3192366.3192379

[28] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. 2018. MAERI: Enabling flexible dataflow mapping over DNN accelerators via reconfigurable interconnects. *ACM SIGPLAN Notices* 53, 2 (March 2018), 461–475. https://doi.org/10.1145/3296957.3173176

[29] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. 2019. HeteroCL: A multi-paradigm programming infrastructure for software-defined reconfigurable computing. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'19)* (Seaside, CA). 242–251. https://doi.org/10.1145/3289602.3293910

[30] Jiajie Li, Yuze Chi, and Jason Cong. 2020. HeteroHalide: From image processing DSL to efficient FPGA acceleration. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA) *(FPGA'20)*. ACM, New York, NY, 51–57. https://doi.org/10.1145/3373087.3375320

[31] Qiaoyi Liu, Dillon Huff, Jeff Setter, Maxwell Strange, Kathleen Feng, Kavya Sreedhar, Ziheng Wang, Keyi Zhang, Mark Horowitz, Priyanka Raina, et al. 2021. Compiling halide programs to push-memory accelerators. *arXiv preprint arXiv:2105.12858* (2021).

[32] Muhammad Masud. 2000. *FPGA Routing Structures: A Novel Switch Block and Depopulated Interconnect Matrix Architectures*. Ph. D. Dissertation. University of British Columbia. https://people.ece.ubc.ca/stevew/papers/pdf/imran_masc.pdf.

[33] Maxeler Inc. [n. d.]. MaxCompiler. Retrieved July 13, 2022 from https://www.maxeler.com/products/software/maxcompiler.

[34] Wim Meeus, Kristof Van Beeck, Toon Goedemé, Jan Meel, and Dirk Stroobandt. 2012. An overview of today's high-level synthesis tools. *Design Automation for Embedded Systems* 16, 3 (2012), 31–51. https://doi.org/10.1007/s10617-012-9096-8

[35] Baisha Mei, Mladen Berekovic, and J.-Y. Mignolet. 2007. ADRES & DRESC: Architecture and compiler for coarse-grain reconfigurable processors. Springer. 10.1007/978-1-4020-6505-7_6

[36] Mentor Graphics Inc. [n. d.]. Catapult High Level Synthesis. Retrieved July 13, 2022 from https://www.mentor.com/hls-lp/catapult-high-level-synthesis.

[37] Thierry Moreau, Tianqi Chen, and Luis Ceze. 2018. Leveraging the VTA-TVM hardware-software stack for FPGA acceleration of 8-bit ResNet-18 inference. In *Proceedings of the Reproducible Quality-Efficient Systems Tournament on Co-Designing Pareto-Efficient Deep Learning (ReQuEST)* (Williamsburg, VA). Article 5. https://doi.org/10.1145/3229762.3229766

[38] Thierry Moreau, Tianqi Chen, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. VTA: An open hardware-software stack for deep learning. *arXiv preprint arXiv:1807.04188* (2018).

[39] Ravi Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically scheduling halide image processing pipelines. *ACM Transactions on Graphics* 35 (7 2016), 1–11. https://doi.org/10.1145/2897824.2925952

[40] Ankita Nayak, Keyi Zhang, Raj Setaluri, Alex Carsello, Makai Mann, Stephen Richardson, Rick Bahr, Pat Hanrahan, Mark Horowitz, and Priyanka Raina. 2020. A framework for adding low-overhead, fine-grained power domains to CGRAs. In *2020 Design Automation Test in Europe Conference Exhibition (DATE'20)*. 846–851. https://doi.org/10.23919/DATE48585.2020.9116477

[41] John O'Donnell. 1988. Hydra: Hardware description in a functional language using recursion equations and high order combining forms. *The Fusion of Hardware Design and Verification* (1988), 309–328.

[42] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A reconfigurable architecture for parallel patterns. In *ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA'17)*. 389–402. https://doi.org/10.1145/3079856.3080256

[43] 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 Transactions on Architecture and Code Optimization* 14, 3, Article 26 (Aug. 2017), 25 pages. https://doi.org/10.1145/3107953

[44] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, WA) *(PLDI'13)*. ACM, New York, NY, 519–530. https://doi.org/10.1145/2491956.2462176

[45] Oliver Reiche, Moritz Schmid, Frank Hannig, Richard Membarth, and Jürgen Teich. 2014. Code generation from a domain-specific language for C-based HLS of hardware accelerators. In *2014 International Conference on Hardware/-Software Codesign and System Synthesis (CODES+ISSS'14)*. 1–10. https://doi.org/10.1145/2656075.2656081

[46] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. 2016. From high-level deep neural models to FPGAs. In *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. 1–12. https://doi.org/10.1109/MICRO.2016.7783720

[47] Jordan S. Swartz, Vaughn Betz, and Jonathan Rose. 1998. A fast routability-driven router for FPGAs. In *Proceedings of the ACM/SIGDA 6th International Symposium on Field Programmable Gate Arrays* (Monterey, CA) *(FPGA'98)*. ACM, New York, NY, 140–149. https://doi.org/10.1145/275107.275134

[48] Christopher Torng, Peitian Pan, Yanghui Ou, Cheng Tan, and Christopher Batten. 2021. Ultra-elastic CGRAs for irregular loop specialization. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA'21)*. 412–425. https://doi.org/10.1109/HPCA51647.2021.00042

[49] Lenny Truong and Pat Hanrahan. 2019. A golden age of hardware description languages: Applying programming language techniques to improve design productivity. In *3rd Summit on Advances in Programming Languages (SNAPL'19) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 136)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 7:1–7:21. https://doi.org/10.4230/LIPIcs.SNAPL.2019.7

[50] Lenny Truong, Steven Herbst, Rajsekhar Setaluri, Makai Mann, Ross Daly, Keyi Zhang, Caleb Donovick, Daniel Stanley, Mark Horowitz, Clark Barrett, and Pat Hanrahan. 2020. fault: A python embedded domain-specific language for metaprogramming portable hardware verification components. In *Computer Aided Verification*. Springer International Publishing, 403–414. https://doi.org/10.1007/978-3-030-53288-8_19

[51] Nestan Tsiskaridze, Maxwell Strange, Makai Mann, Kavya Sreedhar, Qiaoyi Liu, Mark Horowitz, and Clark Barrett. 2021. Automating system configuration. In *2021 Formal Methods in Computer Aided Design (FMCAD)*. https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_19

[52] Peter J. M. Van Laarhoven and Emile H. L. Aarts. 1987. Simulated annealing. In *Simulated Annealing: Theory and Applications*. Springer, 7–15.

[53] Artem Vasilyev, Nikhil Bhagdikar, Ardavan Pedram, Stephen Richardson, Shahar Kvatinsky, and Mark Horowitz. 2016. Evaluating programmable architectures for imaging and vision applications. In *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. 1–13. https://doi.org/10.1109/MICRO.2016.7783755

[54] Rangharajan Venkatesan, Yakun Sophia Shao, Miaorong Wang, Jason Clemons, Steve Dai, Matthew Fojtik, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Yanqing Zhang, Brian Zimmer, William J. Dally, Joel Emer, Stephen W. Keckler, and Brucek Khailany. 2019. MAGNet: A modular accelerator generator for neural networks. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'19)*. 1–8. https://doi.org/10.1109/ICCAD45719.2019.8942127

[55] Veripool. [n. d.]. Verilator. Retrieved July 13, 2022 from https://www.veripool.org/verilator/.

[56] Renda Wang, Longjiang Guo, Chunyu Ai, Jinbao Li, Meirui Ren, and Keqin Li. 2013. An efficient graph isomorphism algorithm based on canonical labeling and its parallel implementation on GPU. In *IEEE 10th International Conference on High Performance Computing and Communications IEEE International Conference on Embedded and Ubiquitous Computing*. 1089–1096. https://doi.org/10.1109/HPCC.and.EUC.2013.154

[57] Xilinx Inc. [n. d.]. Vivado High Level Synthesis. Retrieved July 13, 2022 from https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[58] Pengfei Xu, Xiaofan Zhang, Cong Hao, Yang Zhao, Yongan Zhang, Yue Wang, Chaojian Li, Zetong Guan, Deming Chen, and Yingyan Lin. 2020. AutoDNNchip: An automated DNN chip predictor and builder for both FPGAs and ASICs. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA) *(FPGA'20)*. ACM, New York, NY, 40–50. https://doi.org/10.1145/3373087.3375306

[59] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, CA) *(FPGA'15)*. ACM, New York, NY, 161–170. https://doi.org/10.1145/2684746.2689060

[60] Xiaofan Zhang, Junsong Wang, Chao Zhu, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. 2018. DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'18)* (San Diego, CA). Article 56, 8 pages. https://doi.org/10.1145/3240765.3240801

[61] Wei Zuo, Yun Liang, Peng Li, Kyle Rupnow, Deming Chen, and Jason Cong. 2013. Improving high level synthesis optimization opportunity through polyhedral transformations. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA) *(FPGA'13)*. ACM, New York, NY, 9–18. https://doi.org/10.1145/2435264.2435271

Received 18 October 2021; revised 12 March 2022; accepted 30 April 2022