Creating an Agile Hardware Design Flow

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

Stanford AHA! Agile Hardware Center
About Me

Priyanka Raina
Assistant Professor of Electrical Engineering
Stanford University

Ph.D. in EECS, MIT (2018), S.M. in EECS, MIT (2013)
and B.Tech. in EE, IIT Delhi (2011)

I am one of the faculty leads in Stanford’s AHA!
Agile Hardware Center and work on domain-specific
hardware architectures and design methodology
Motivation

• Digital design tools and methodology have improved dramatically letting us create large SoCs with accelerators

Dozens of specialized accelerators
  - Machine Learning
  - Computational Photography
  - Video Coding
  - Cryptography
  - Depth Processing

• But completing these designs (with software)
  • Takes years
  • Costs hundreds of millions of dollars

Apple’s A13 SoC
8.5 billion transistors
7 nm

https://www.anandtech.com/show/14892/the-apple-iphone-11-pro-and-max-review/2
Waterfall Approach to Accelerator Design

- A **waterfall** approach is still used for most accelerator designs

  - Application Analysis
  - ResNet
  - MobileNet
  - ...  
  - Architectural Specification
  - RTL Design and Test
  - Physical Design
  - Software / Compiler Design

- Fails when
  - Changing application requirements
  - Incomplete knowledge/understanding of the problem
Agile Approach to Accelerator Design

• We explore an **agile** hardware/software design flow
  • Incrementally update the hardware accelerator and software to map to it

```
| Base Hardware Accelerator v0 | Compiler Toolchain v0 | Application 1
|-------------------------------|----------------------|---------------
|                               |                      | Application 2
|                               |                      | Incremental Updates
|                               |                      | Base Hardware Accelerator v1
|                               |                      | Compiler Toolchain v1
|                               |                      | Application 2.1
|                               |                      | Application 3
```

Power, Performance, Area
Agile Approach to Accelerator Design

1. Accelerator must be **configurable**
   - So we can map new or modified applications to it (although with lower efficiency)

2. Hardware and compiler must **evolve together**
   - Any change in hardware must propagate to compiler automatically
SoC with a Coarse-Grained Reconfigurable Array

- Our accelerator is an island-style CGRA
  - Processing element (PE) tiles – potentially heterogeneous
  - Memory (MEM) tiles
  - Statically configured interconnect
- Programmable, but allows exploiting parallelism and locality
Halide Program for a 3x3 Convolution

Algorithm:
RDom r(0, 3, 0, 3);
output(x, y) += input(x + r.x, y + r.y) * weight(r.x, r.y)

Schedule:
input.in().store_at(output, y).compute_at(output, x);
output.accelerate({input}, y);
output.unroll(r.x).unroll(r.y);

Loop tiling, ordering, fusion
Which loops to parallelize

Memory hierarchy
Scope of accelerator
Software Compiler

Application Halide Program \(\xrightarrow{\text{Lower}}\) CoreIR Graph

\(\xrightarrow{\text{CPU Code}}\)

Unified Buffer

\(\xrightarrow{\text{input}}\)

CoreIR Dataflow Graph

\(\xrightarrow{\text{output}}\)
Software Compiler

- Application Halide Program
  - Lower
  - CoreIR Graph
    - Map PE and Memory
      - Mapped CoreIR Graph
        - CGRA
          - Place & Route
            - CGRA Bitstream

  - CPU Code

- Mapped CoreIR Graph
  - MEM
    - SR
      - PE
        - Shift Registers
          - Output
Software compiler must evolve with hardware!

- Application Halide Program
- Lower
- CoreIR Graph
- Map PE and Memory
- Mapped CoreIR Graph
- CGRA Place & Route
- CGRA Bitstream

Hardware independent

- Depends on the PE and Memory hardware
- Depends on the interconnect hardware
Our Key Contribution

• Traditionally, designers create parameterized hardware generators that communicate with the software compiler through configuration files.

• We create mini languages whose semantics are sufficiently expressive to communicate both configuration values and how changes to those values impact other layers in the system.

• Our system has three mini-languages or domain-specific languages (DSLs):
  • PEak for PEs, Lake for memories, Canal for interconnect.
Our DSL-based Hardware Generation and Software Compilation Flow

- PEak Program (PE spec)
- PEak Compiler
  - PE HW in Magma
  - Magma Compiler
  - CGRA Verilog
- Application Halide Program
  - Halide Compiler
  - CoreIR Graph
  - PE and MEM Mapper
  - Mapped CoreIR Graph
  - Place & Route Engine
  - CGRA Bitstream
Our DSL-based Hardware Generation and Software Compilation Flow

PEak Program (PE spec) → PEak Compiler → PE HW in Magma

Lake Program (MEM spec) → Lake Compiler → MEM HW in Magma

Magma Compiler → CGRA Verilog

Halide Program → Halide Compiler → CoreIR Graph

PE and MEM Mapper → Mapped CoreIR Graph

Place & Route Engine → CGRA Bitstream

Rewrite Rules
Our DSL-based Hardware Generation and Software Compilation Flow

- PEak Program (PE spec) → PEak Compiler → PE HW in Magma → Magma Compiler → CGRA Verilog
- Lake Program (MEM spec) → Lake Compiler → MEM HW in Magma
- Canal Program (Interconnect spec) → Canal Compiler → Interconnect HW in Magma

Routing Graph:
- Halide Program → Halide Compiler → CoreIR Graph → PE and MEM Mapper → Mapped CoreIR Graph → Place & Route Engine → CGRA Bitstream
PEak: PE DSL

**PE ISA Specification**

```python
class Opcode(Enum):
    Add = 0
    And = 1

class Instruction(Product):
    op = Opcode
    invert_A = Bit
    scale_B = Bit
    reg_out = Bit

# Data is a BitVector
Data = Unsigned[16]
```

**PE Functional Specification**

```python
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)
        else:
            res = A & B
            flag = (res == 0)
        if inst.reg_out:
            res = self.o_reg(res)
            flag = self.f_reg(flag)
        return res, flag
```

Define sub-components and state

Define desired behavior of each instruction

Specific types (or composition of types) for operands and instructions
PE Python Execution

```python
pe = PE()

inst = Instruction(
    Opcode.Add,
    Bit(0),  # invert_A
    Bit(1),  # scale_B
    Bit(0))) # reg_out

out, flag = pe(
    inst,
    Data(2),  # A
    Data(3),  # B
    Data(5),  # C
    Bit(0))  # c_in

assert out == Data(17)
assert flag == Bit(0)
```
PEak Compiler

Functional Model

Serves as

PEak Specification

Generates Using SMT

Rewrite Rules for Mapper

Tests

Generates Using magma

Single Source of Truth

Hardware RTL Verilog

Single Source of Truth
Multiple Interpretations of PEak Specification

- PEak program uses abstract types provided by the PEak DSL such as Bit, BitVector etc.
- Each component of the PEak compiler provides a separate concrete implementation of these abstract types
- Multiple interpretations of a PEak specification in different contexts
PE Rewrite Rule Generation from Symbolic Representation

\[ \exists \text{inst} \, \forall \text{inputs} : \text{CoreIR.Op}(\text{inputs}) = \text{PE}(\text{inst}, \text{inputs}) \]

Instruction (  
\begin{align*}
\text{op} &= \text{Add}, \\
\text{invert}_A &= \text{True}, \\
\end{align*}
...  
)
Lake: Memory DSL

- Lake memory modules contain
  - One or more memory units
  - Blocks that select or combine inputs to create an output
  - Graph interconnecting these units to each other and the ports
Lake: Memory DSL

**Hardware RTL Verilog** → **Lake Specification** → Using an SMT solver → **Hardware Configuration**

High-level specification for a polyhedral rewrite system that uses hardware parameters to optimize unified buffers in the application.
High-Level Specification for Polyhedral Rewrite System

- For each memory unit, the rewrite system needs to know the
  - Memory capacity
  - Number of ports
  - Port width
  - Read/write delay
  - Capability of the address generators
    - Like nested affine loops (number of loops and constraints on the loop values)

- All extracted from the hardware specification
Polyhedral Rewrite System

(a) Reuse analysis
Reduces memory bandwidth, capacity

(b) Banking
Increases bandwidth

(c) Chaining
Increases capacity

(d) Vectorization
Matches interface width
Canal Interconnect DSL

- Canal takes a set of (heterogeneous) PE and memory cores and a directed **graph-based specification of the interconnect**, and generates the hardware, the routing graph and the configuration bitstream.
Garnet SoC

- 32×16 array of PE and memory tiles
  - Each PE tile has a 16-bit, two-input, fixed point ALU, and some registers
  - Each memory tile contains 2 KB of SRAM and flexible address generators
- An interconnect with five 16-bit tracks and five 1-bit tracks connects the tiles
- Second level memory called global buffer
- ARM Cortex M3 processor
Results: Energy per operation

The CGRA consumes 6.92× to 25.3× less energy than the FPGA in the same TSMC 16 nm technology.
Summary

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

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