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

Stanford University

About Me

Priyanka Raina

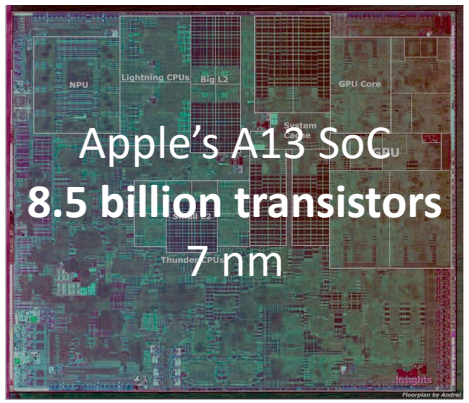
Assistant Professor of Electrical Engineering
Stanford University

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



https://www.anandtech.com/show/14892/the-apple-iphone-11-pro-and-max-review/2

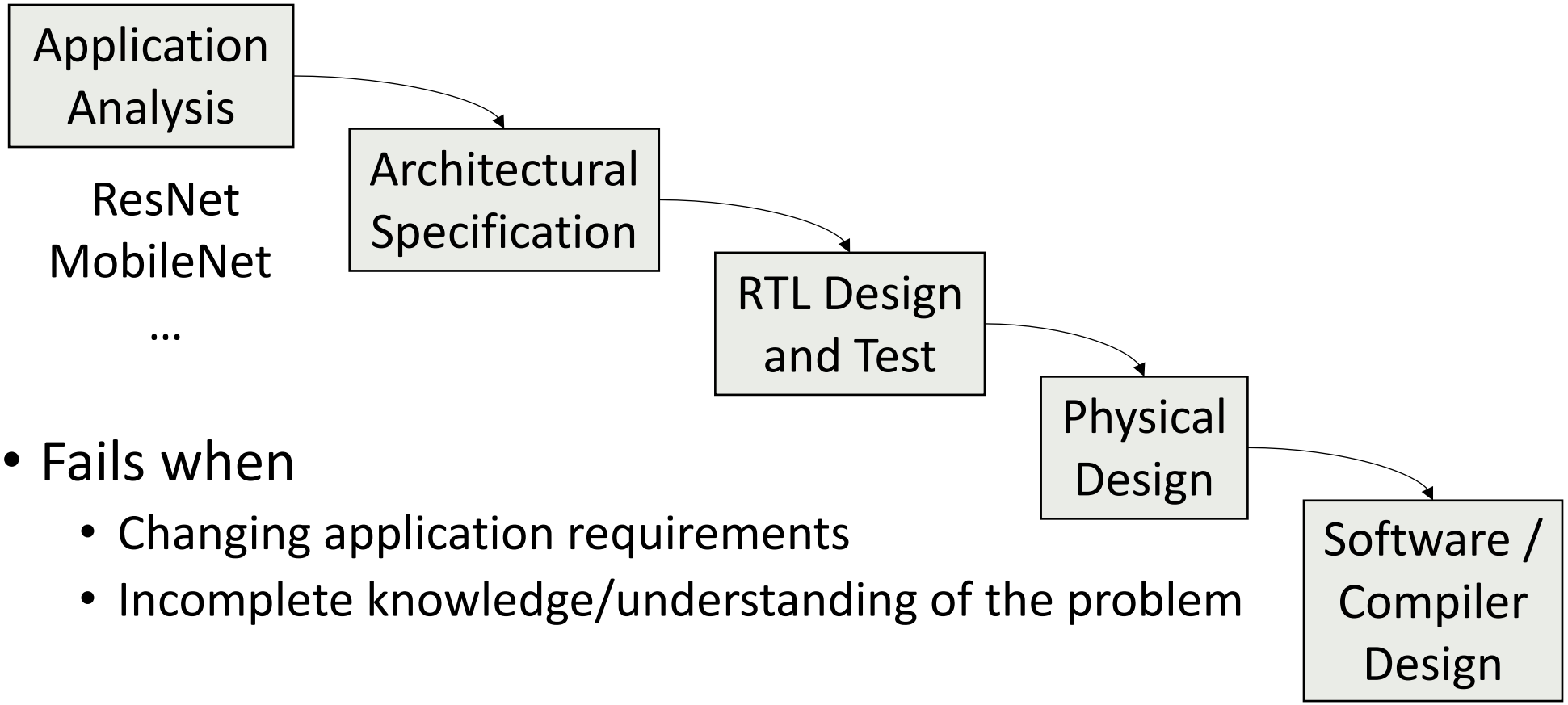
Dozens of specialized accelerators

- Machine Learning
- Computational Photography
- Video Coding
- Cryptography
- Depth Processing

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

Waterfall Approach to Accelerator Design

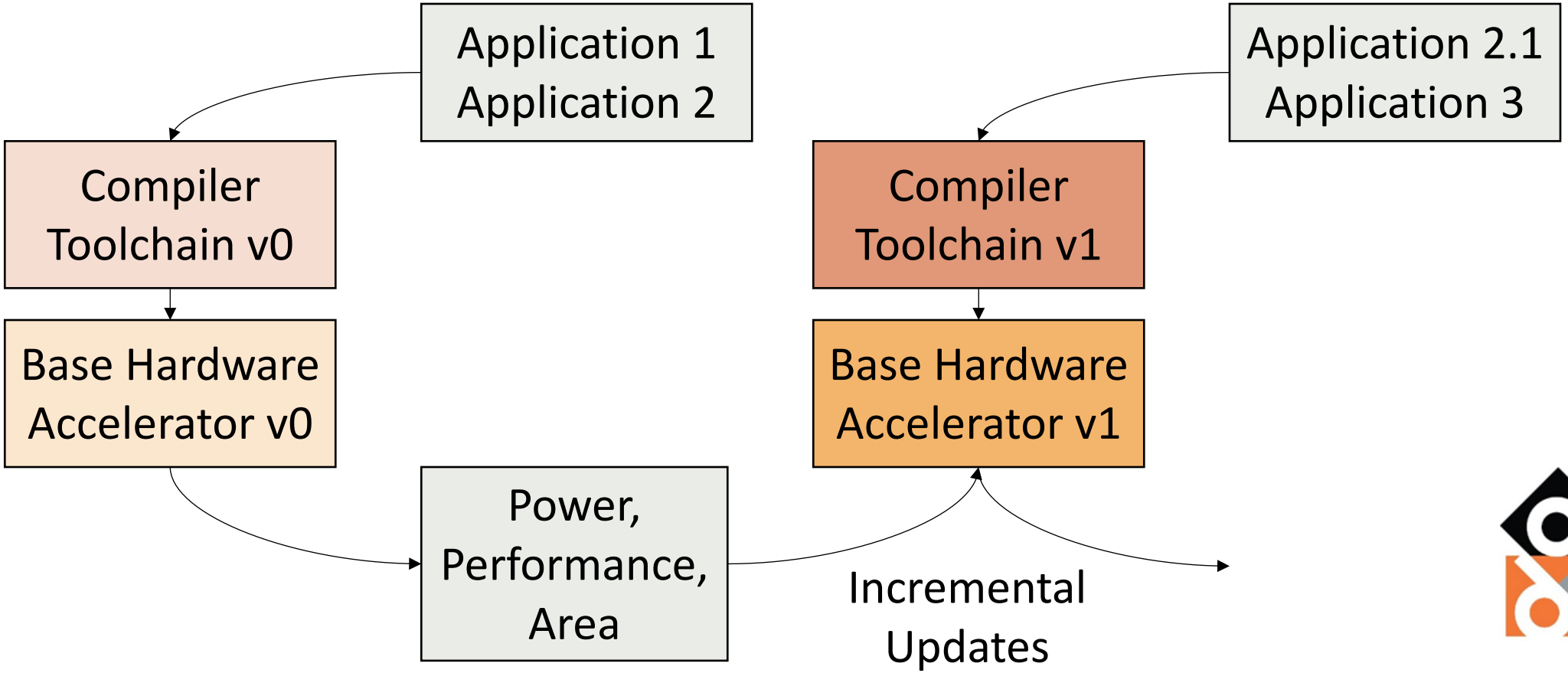
- A **waterfall** approach is still used for most accelerator designs



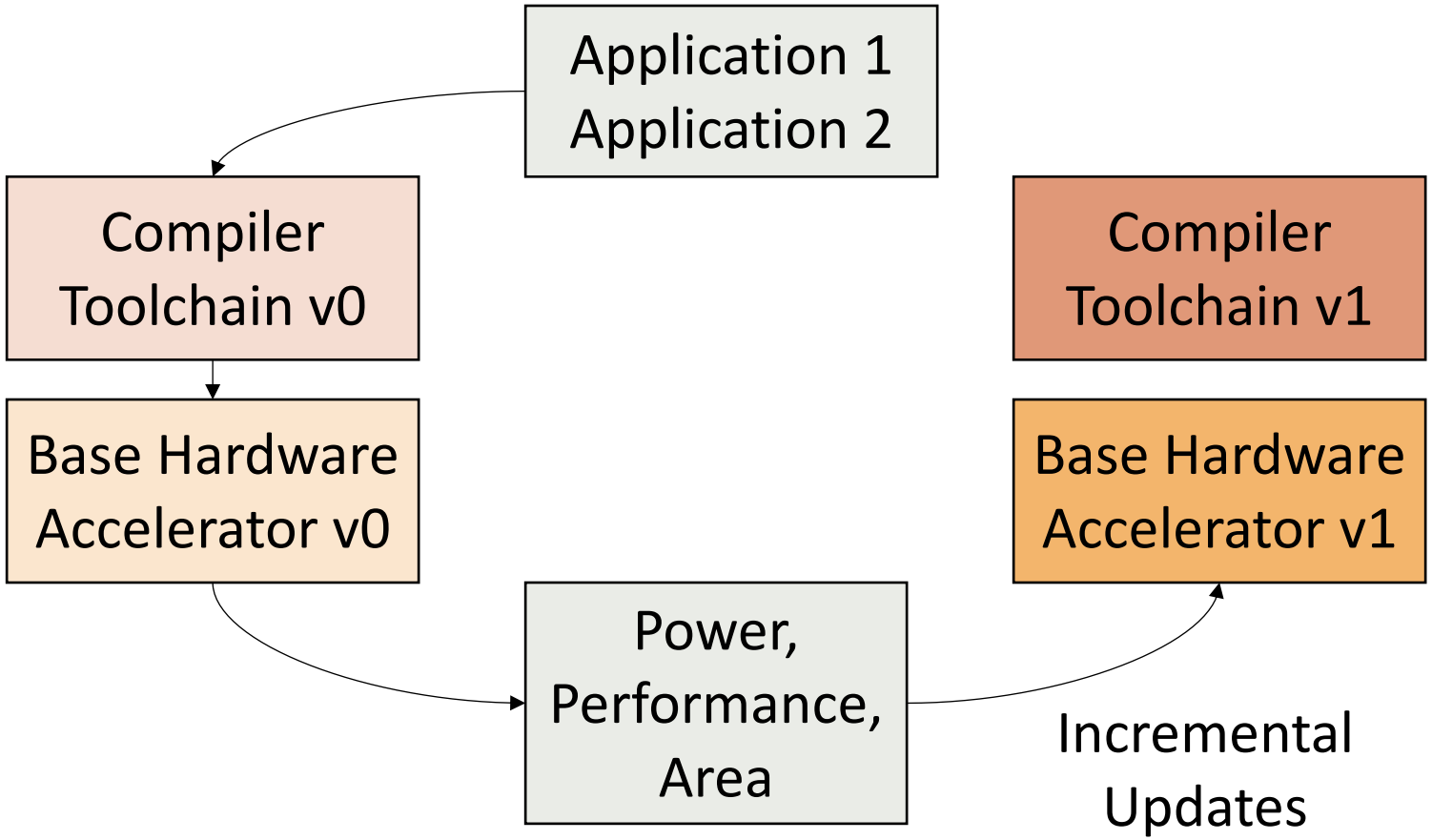
- Fails when
  - Changing application requirements
  - Incomplete knowledge/understanding of the problem

Agile Approach to Accelerator Design

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

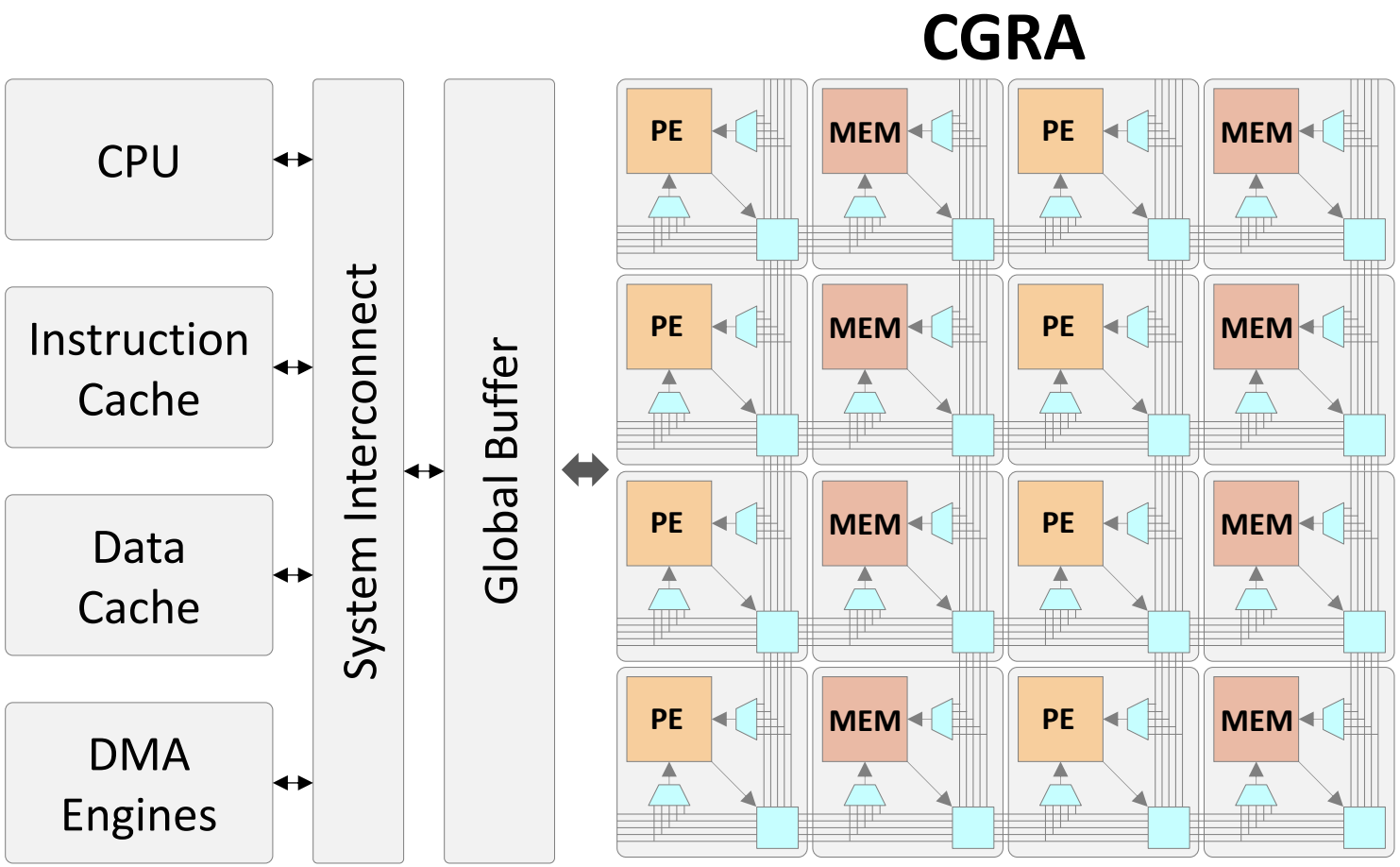


Agile Approach to Accelerator Design



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

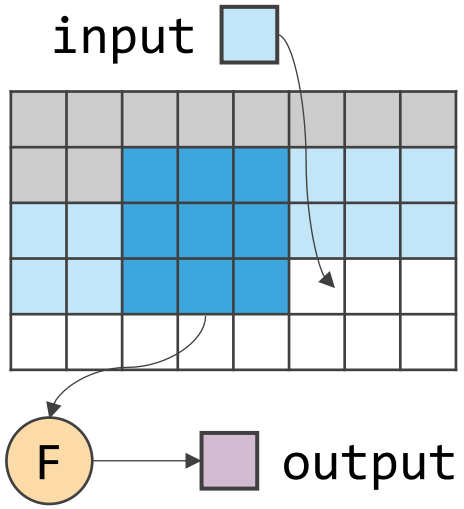
SoC with a Coarse-Grained Reconfigurable Array



- 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

Software Compiler

Application Halide Program



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);
```

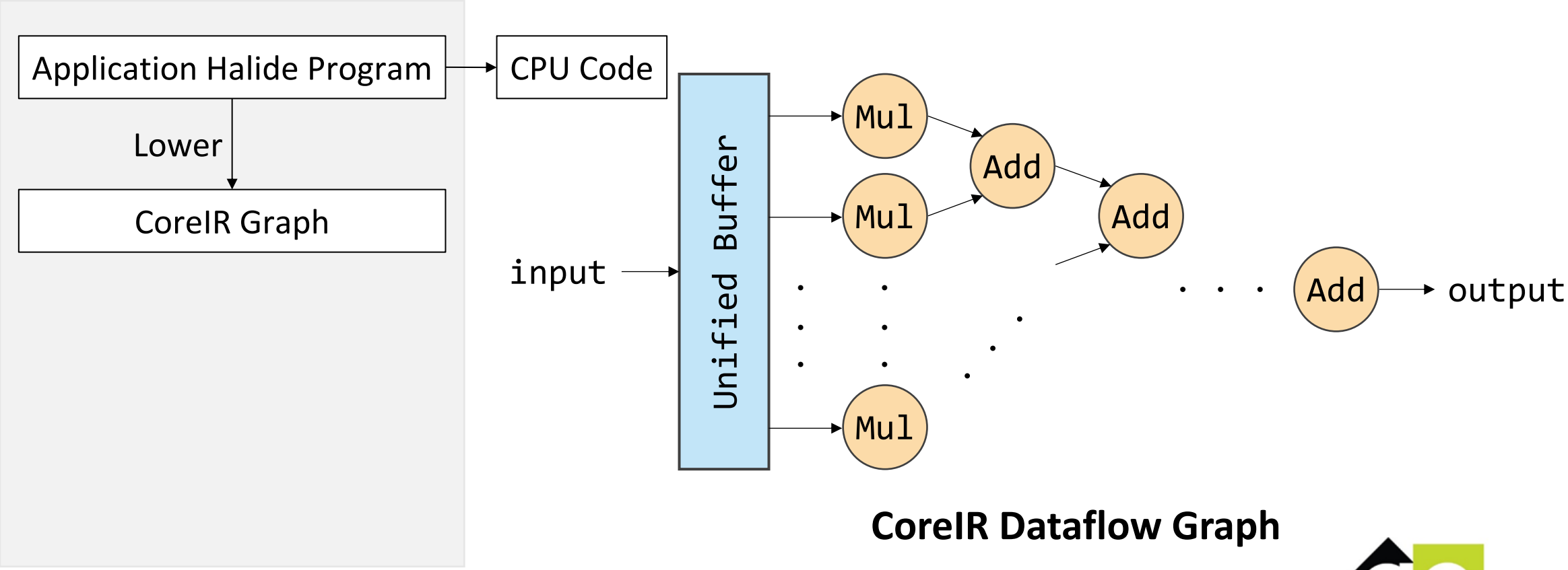
Memory hierarchy

Scope of accelerator

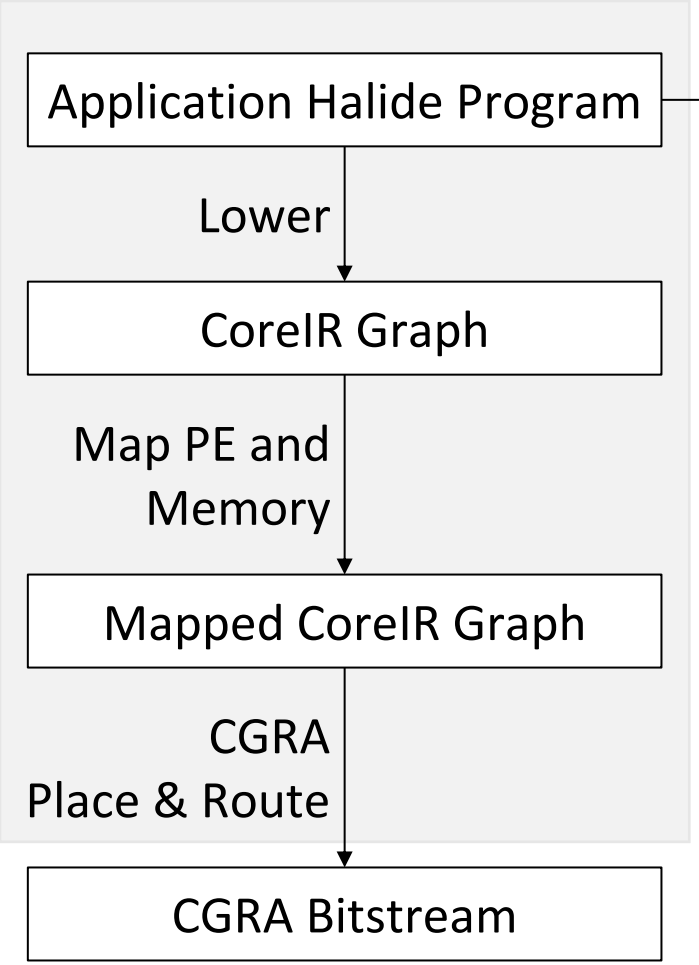
Loop tiling, ordering, fusion

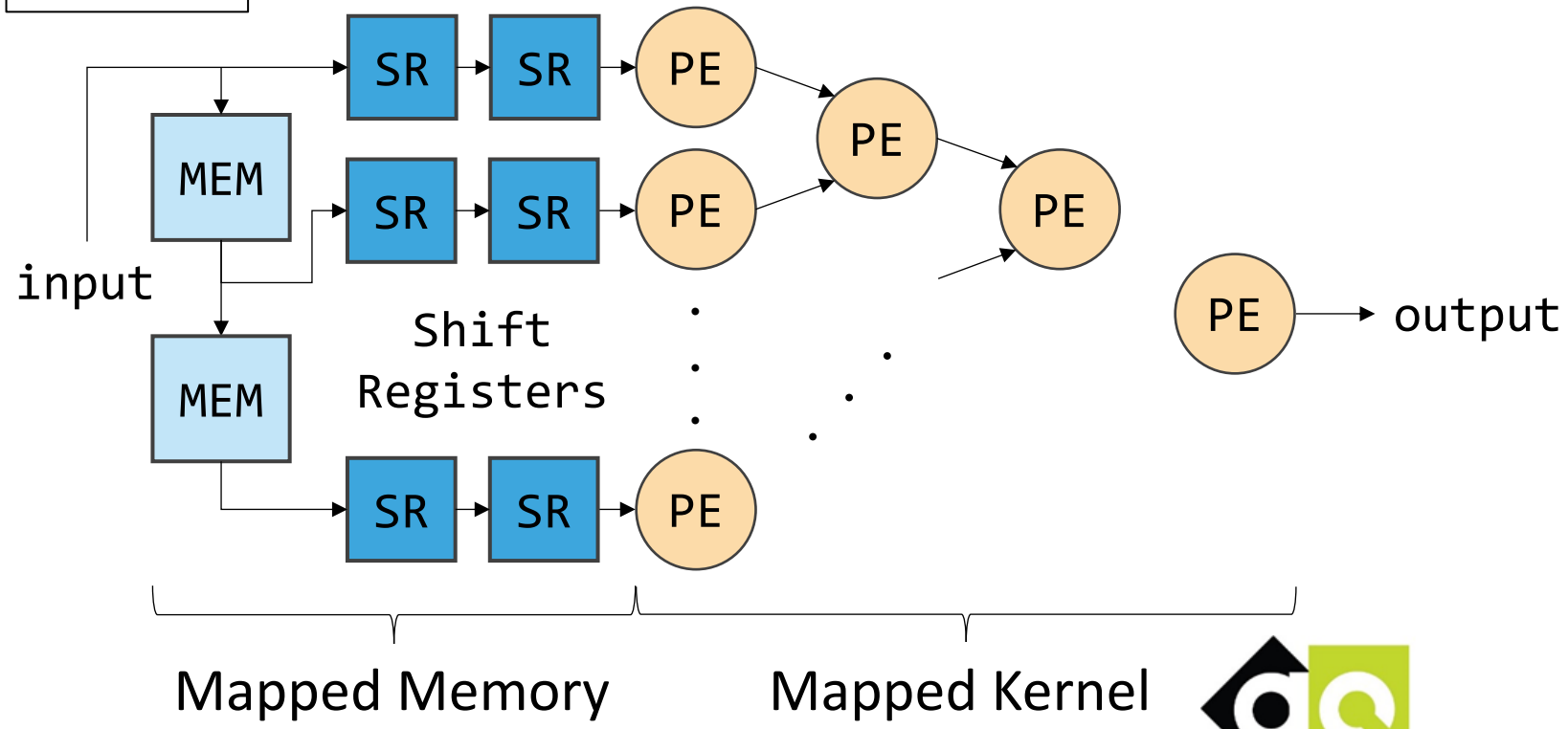
Which loops to parallelize

Software Compiler



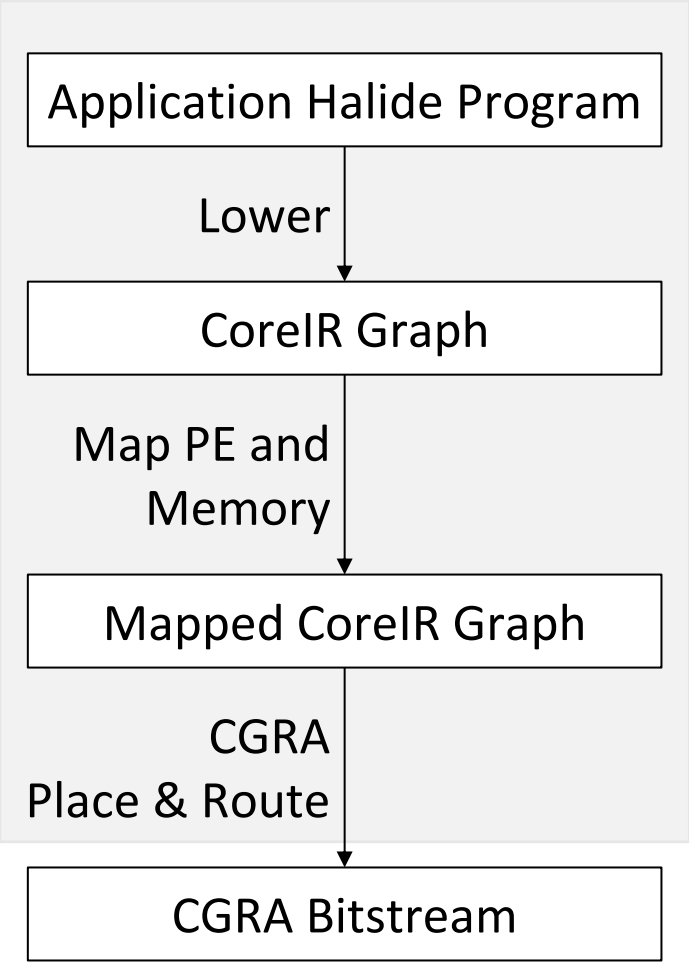
Software Compiler





Mapped CoreIR Graph

Software compiler must evolve with hardware!



Hardware independent

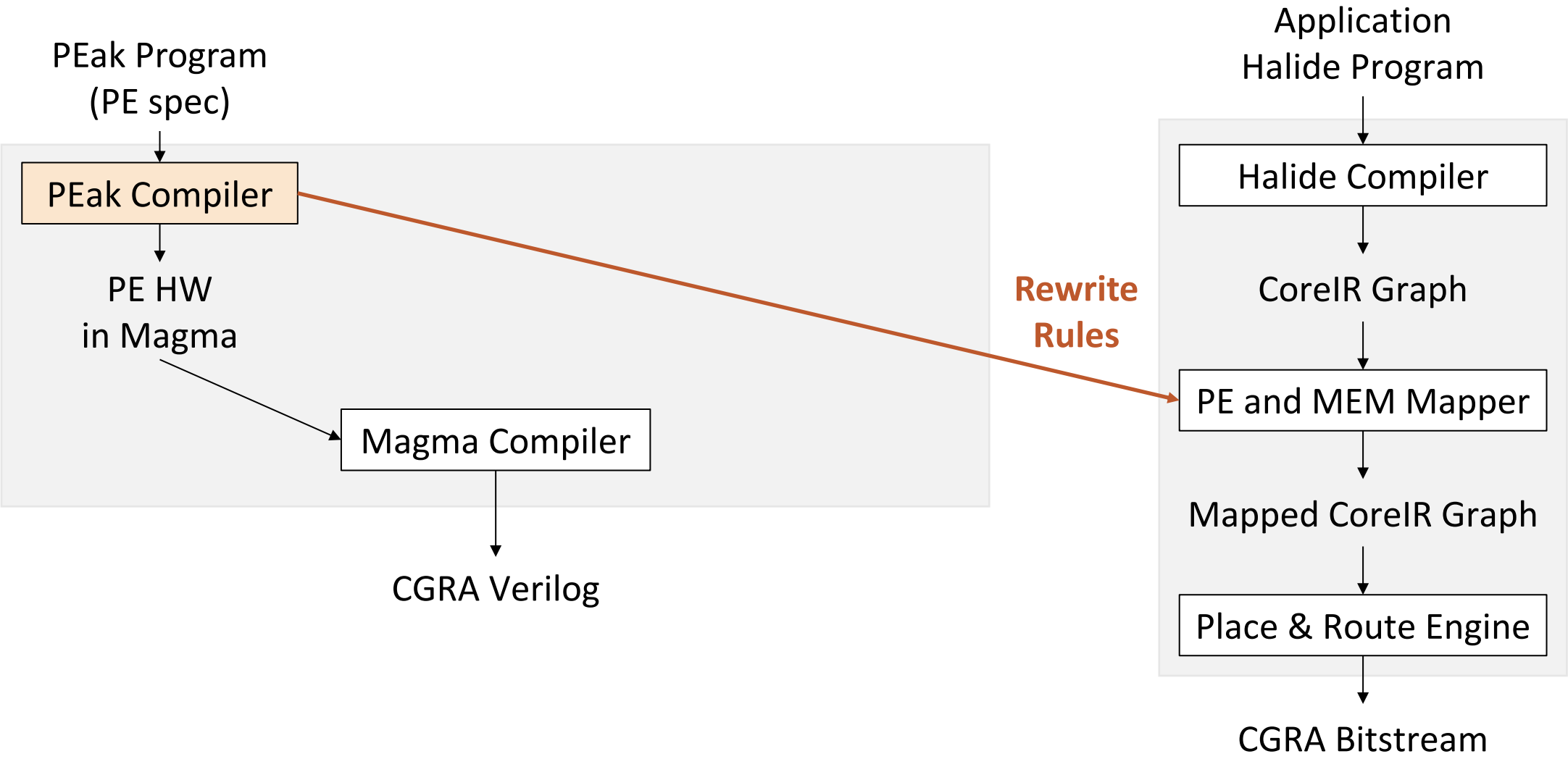
Depends on the PE and Memory hardware

Depends on the interconnect hardware

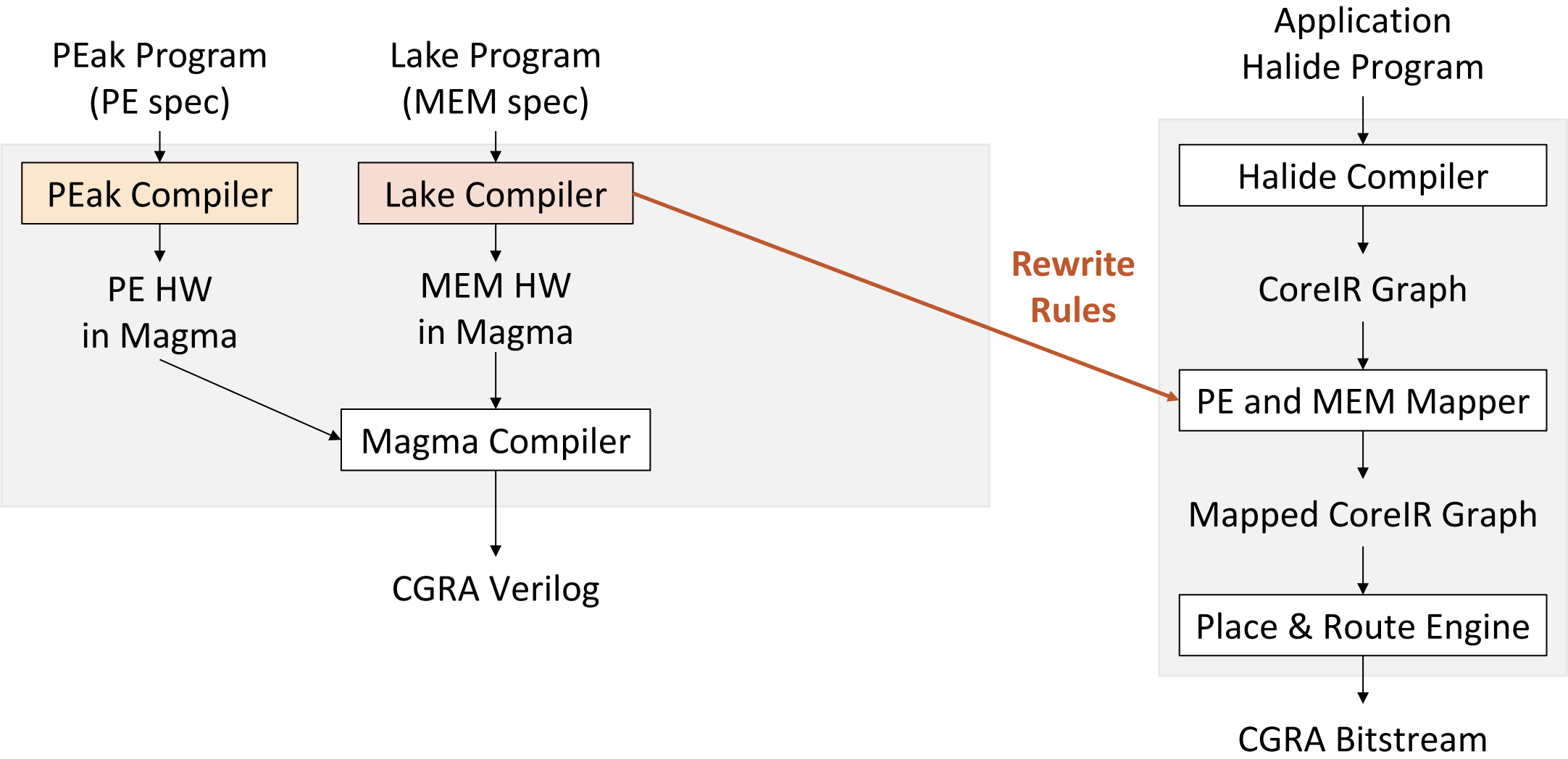
Our Key Contribution

- 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

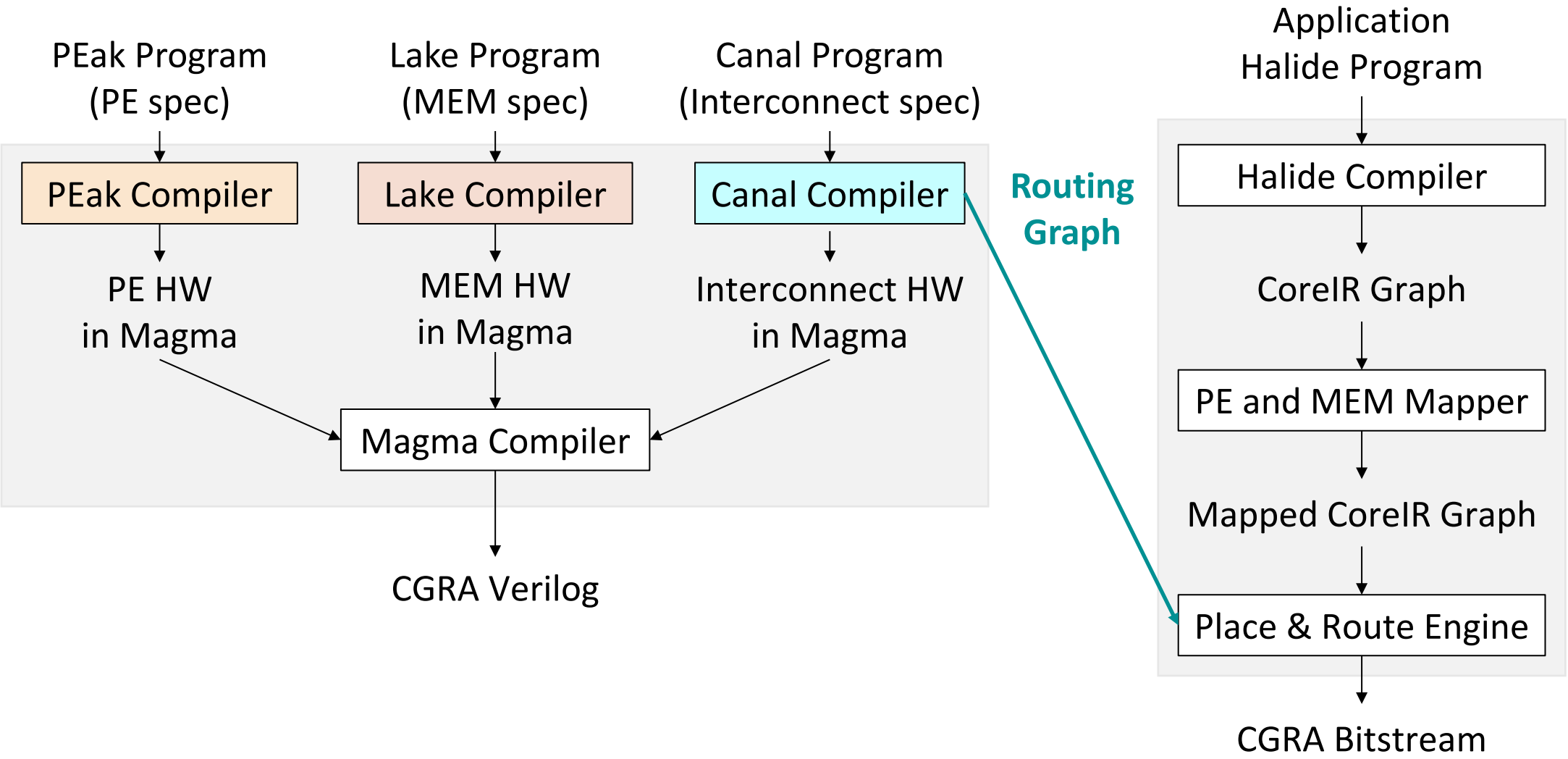
Our DSL-based Hardware Generation and Software Compilation Flow



Our DSL-based Hardware Generation and Software Compilation Flow



Our DSL-based Hardware Generation and Software Compilation Flow



PEak: PE DSL

PE ISA Specification

```
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]
```

Specific **types** (or composition of types) for operands and instructions

PE Functional Specification

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

Define sub-components and state

```
    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: # 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
```

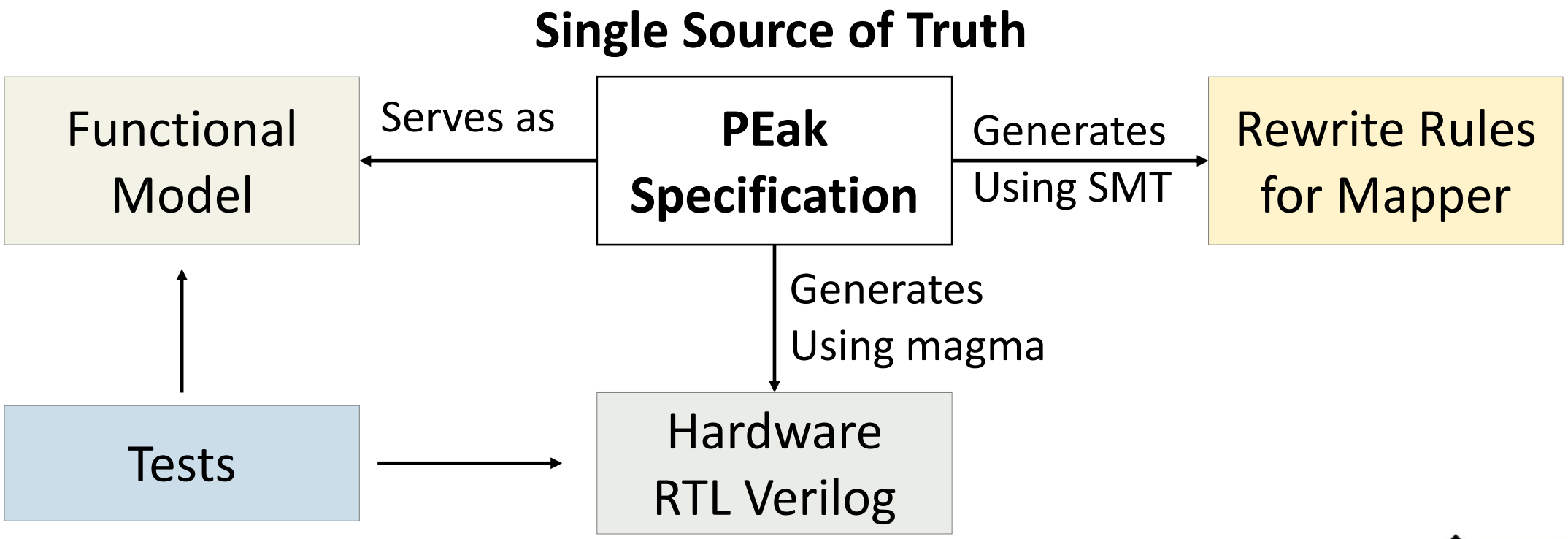
Define desired behavior of each instruction

PEak Specification

PE Python Execution

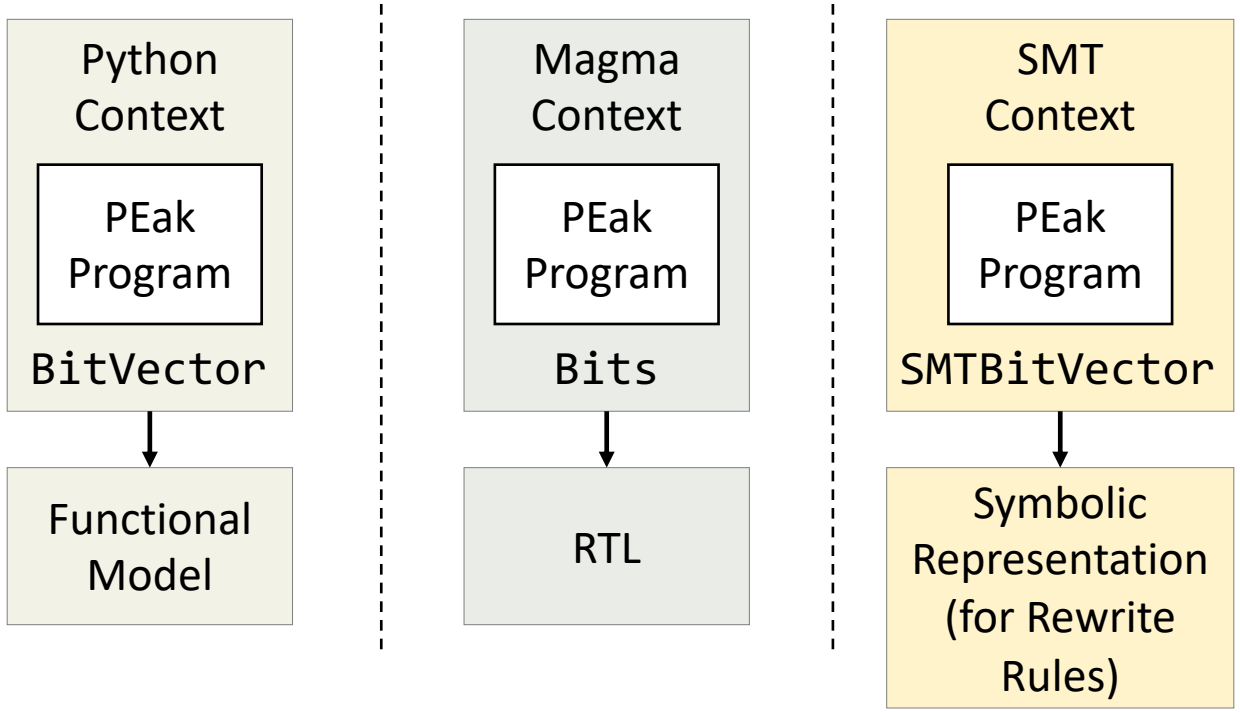
```
pe = PE()

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



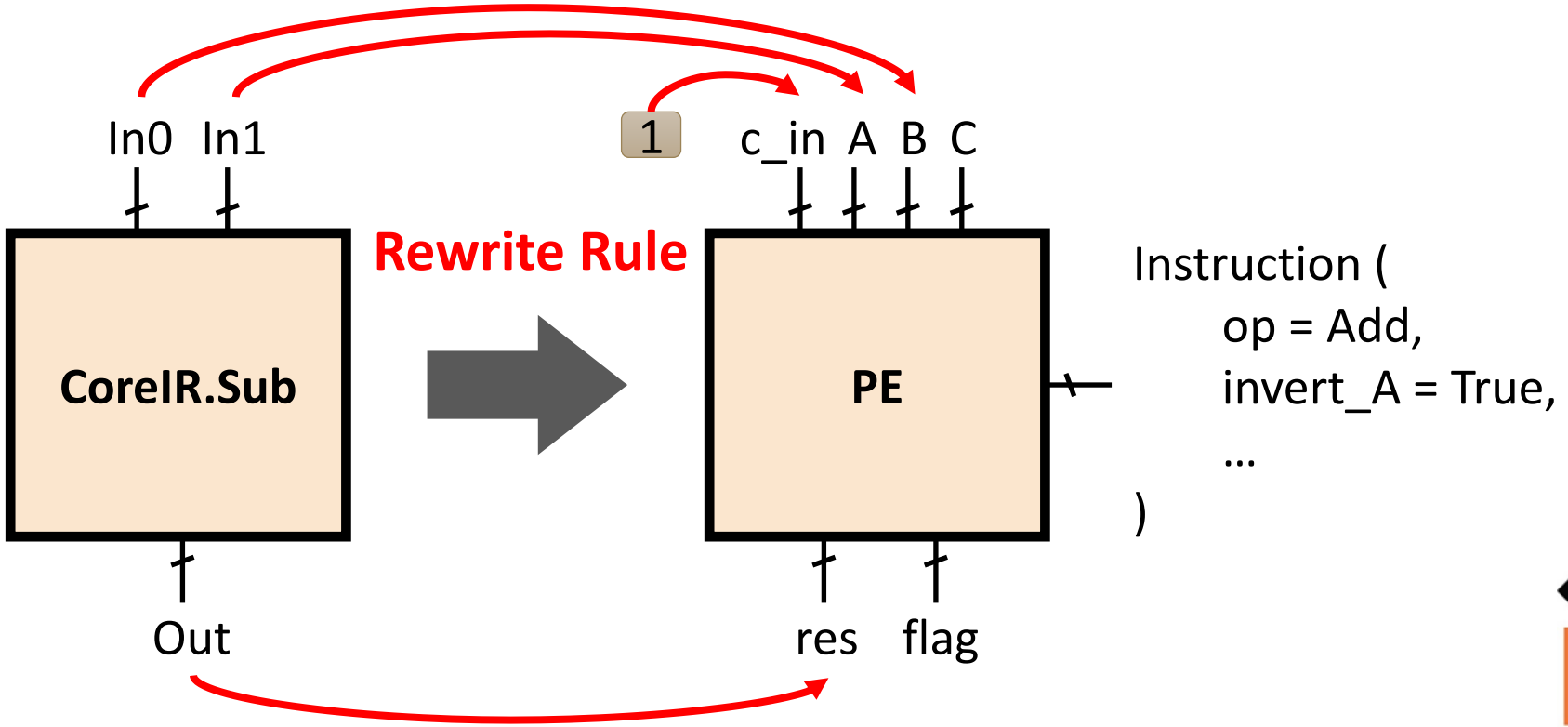
Multiple Interpretations of PEak Specification

- 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



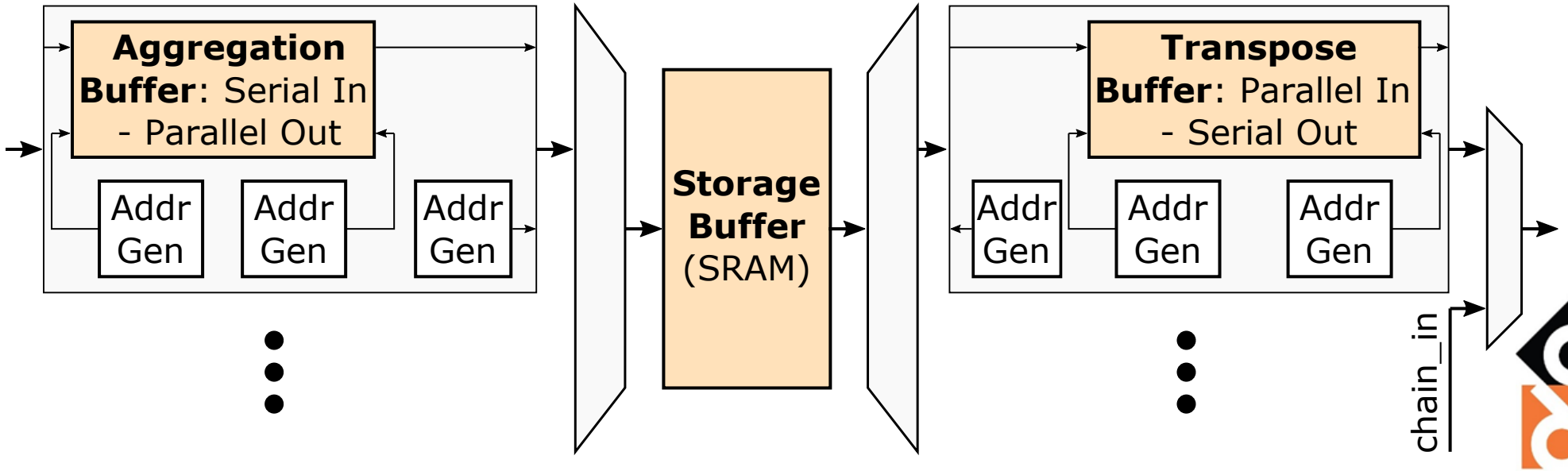
PE Rewrite Rule Generation from Symbolic Representation

∃inst ∀inputs : CoreIR.Op(inputs) == PE(inst,inputs)

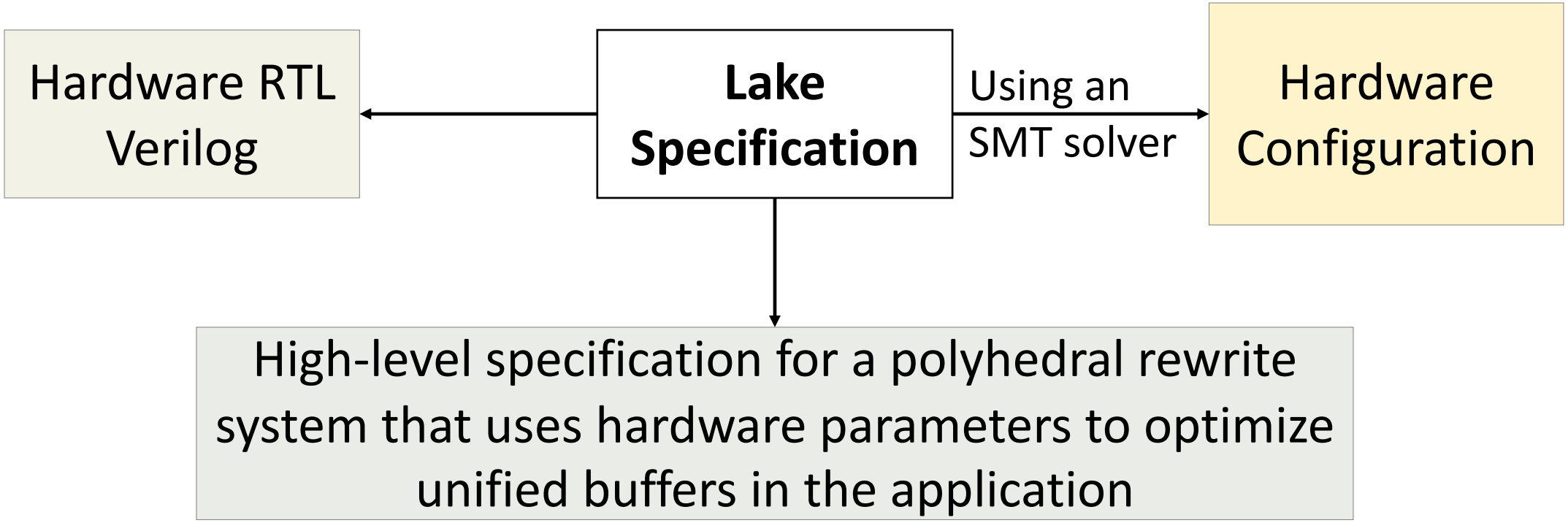


Lake: Memory DSL

- 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



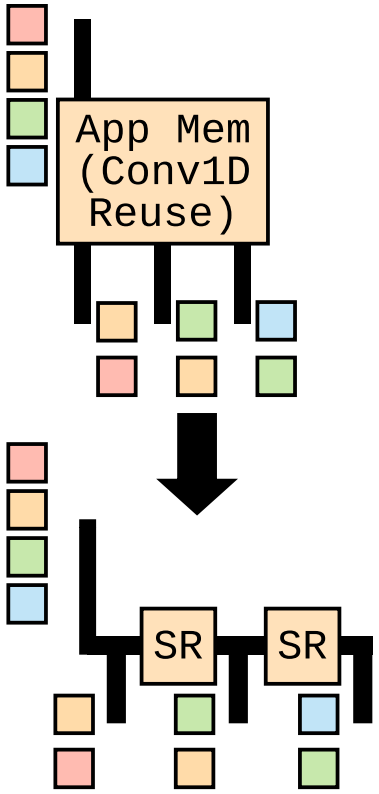
High-Level Specification for Polyhedral Rewrite System

- 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

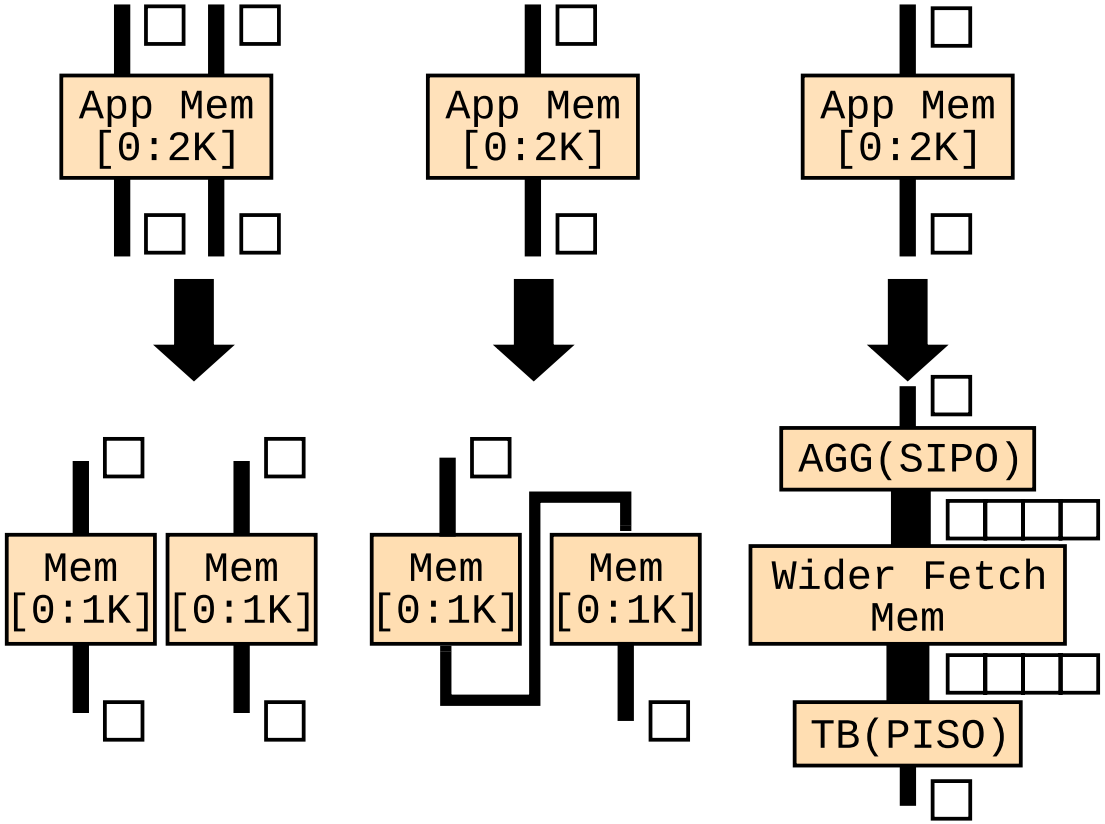
Polyhedral Rewrite System

Hardware Independent



(a) **Reuse analysis**
Reduces memory bandwidth, capacity

Hardware Dependent



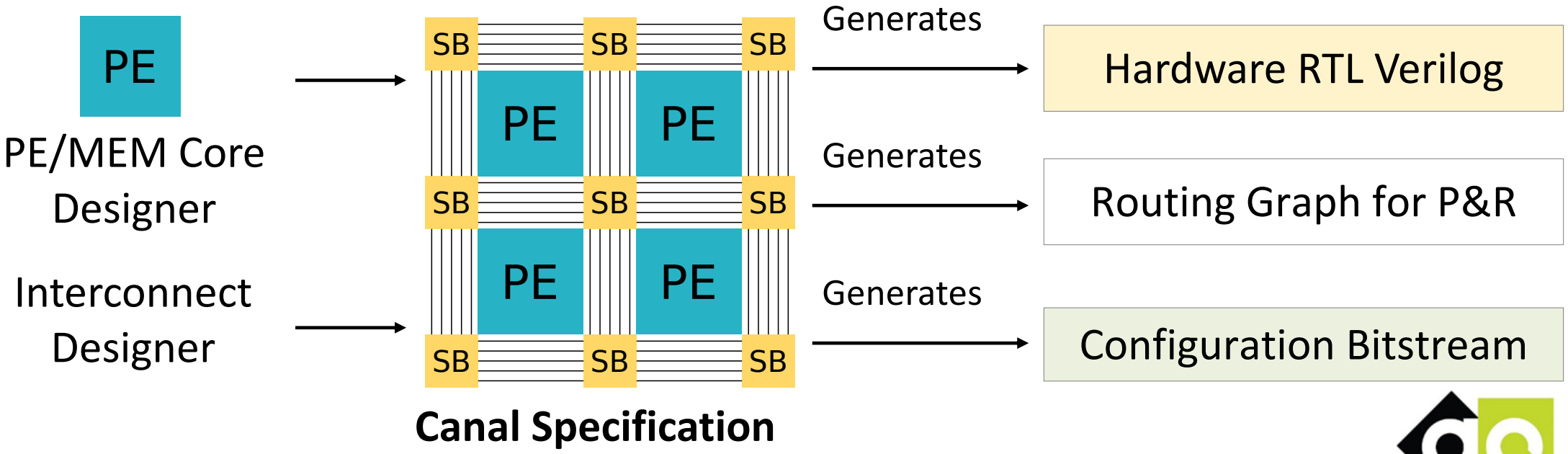
(b) **Banking**
Increases bandwidth

(c) **Chaining**
Increases capacity

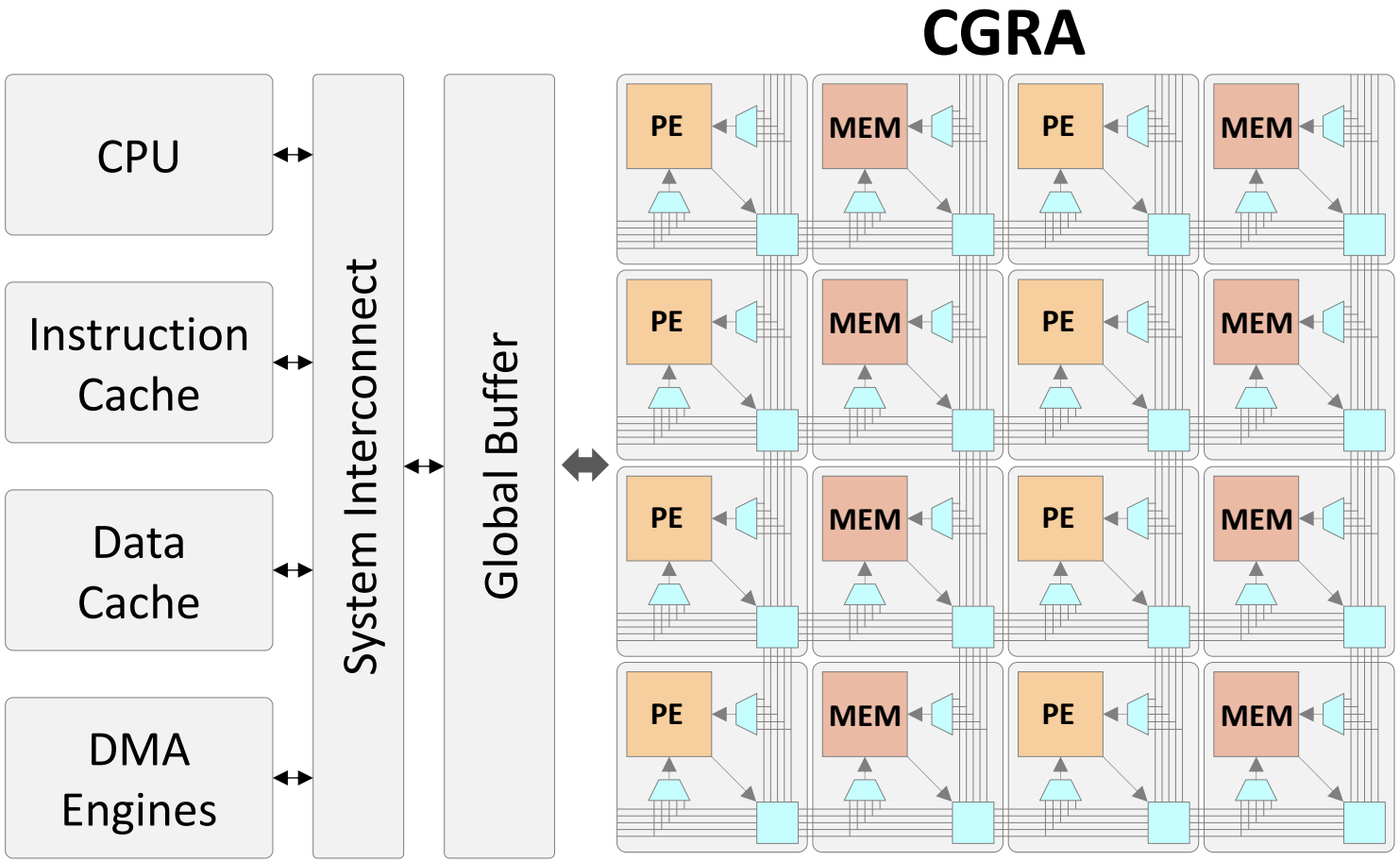
(d) **Vectorization**
Matches interface width

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

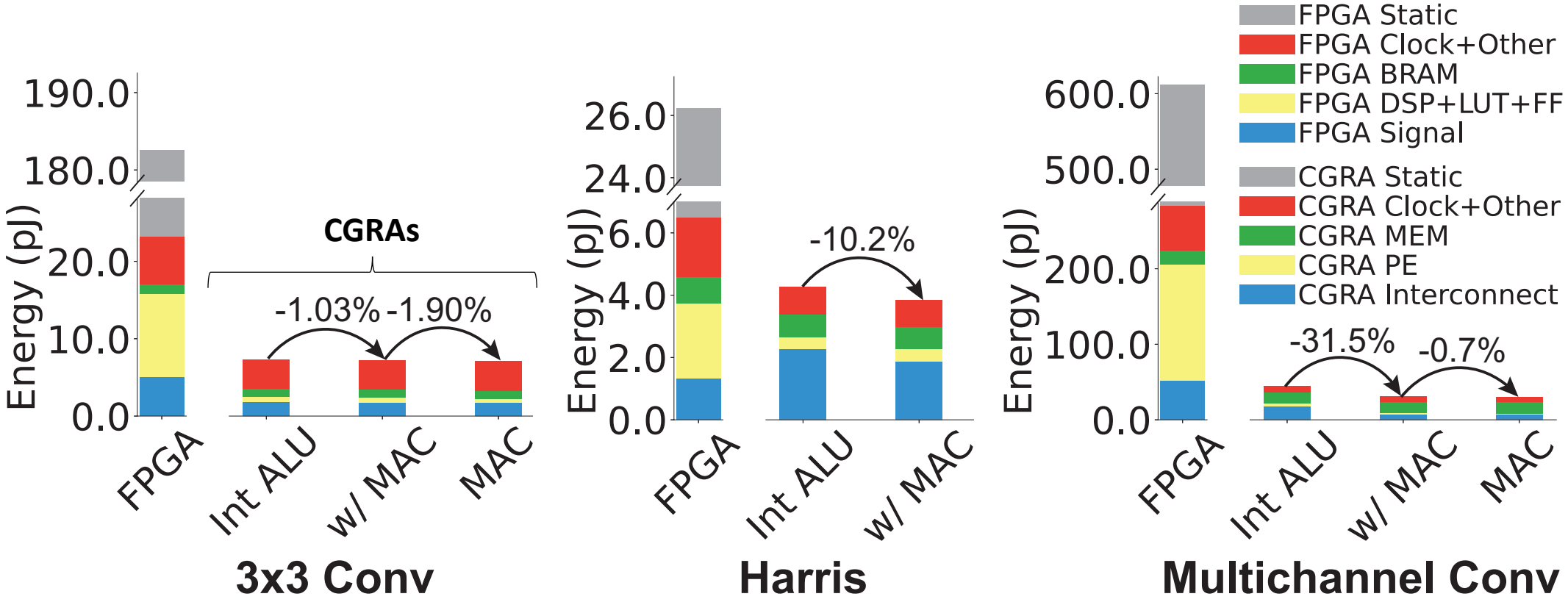


Garnet SoC



- 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.