AHA (Agile Hardware): Visual Computing

Clark Barrett, Kayvon Fatahalian, Pat Hanrahan,
Mark Horowitz, Priyanka Raina

Briefing Prepared for Tom Rondeau

4/18/2019

Stanford | ENGINEERING

Project Goals

Create:

- A new way to create DSSoCs
  - Use an agile design and a end-to-end flow
  - Create accelerators through specialization of CGRA fabric

Deliver:

- Improved video analysis algorithms
- A DSSoC generator for visual computing
- A framework for mapping these algorithms to the DSSoC

The Advantages of an Agile Design Approach

- Complex problems are hard to get your hands around
  - What you think are your goals are often wrong
  - What you think are the hard problems often aren't

- Best way to understand the goals / problems
  - Try to build a simple solution quickly
  - Learn a lot from that effort
  - Update approach/goal and repeat

After ~ 9 Months

- We completed our first cycle, and working on Gen 2
  - Have a chip and a software system that can map applications to it
    (A framework for mapping these algorithms to the DSSoC)

- Learn a lot from the first version of the system
  - Our initial goal was not quite right
  - Encounter hard problems we hadn't considered

Plan for the Day

1

9:00	Intro (what you are listening to now)
9:15	Alex Carsello, Max Strange Jade – Our first chip (with demo)
9:30	Pat Hanrahan New Approach to Tools - Gemstone
10:00	Priyanka Raina Garnet / Lassen – Next Gen Chip

Plan for the Day

10:30	Break
10:45	Kayvon Fatahalian
	Halide Auto-schedulers
11:15	Jeff Setter
	Halide to CoreIR
11:35	Ross Daly Caleb Donovick
	Mapping CoreIR to Lassen/ …
11:55	Keyi Zhang
	Canal – Tying it all Together

Plan for the Day

12:15 Lunch

12:30 Mark Horowitz
AHA Programmatic Review

1:00 Feedback Session

CGRA Rev 1: Jade

Alex Carsello
Max Strange

DSSoC Site Visit

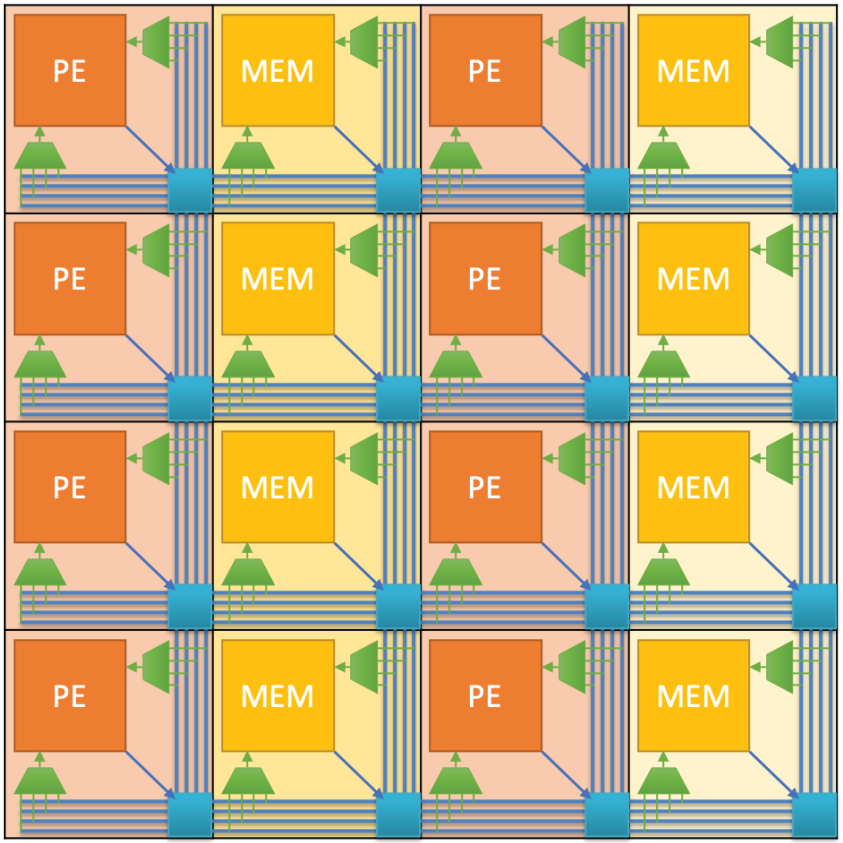
April 18, 2019

- **Goal:** create the simplest possible system that could map an application to hardware

- Iterate on this for future chips

CGRA Architecture

- 16x16 Island-style design with two core types:
  - Processing Element (PE)
  - Memory
- 16-bit and 1-bit wiring tracks
- Connection boxes (CBs) connect wiring tracks to core inputs
- Switch boxes (SBs) connect tracks and core outputs to each other
- Configuration over JTAG



Core Details

PE Core:

- 16-bit ALU (add, multiply, logical operators, etc.)
- LUT
- Some registers
- No floating point support

Memory Core:

- 2 KB SRAM
- 3 modes of operation: SRAM, FIFO, line buffer

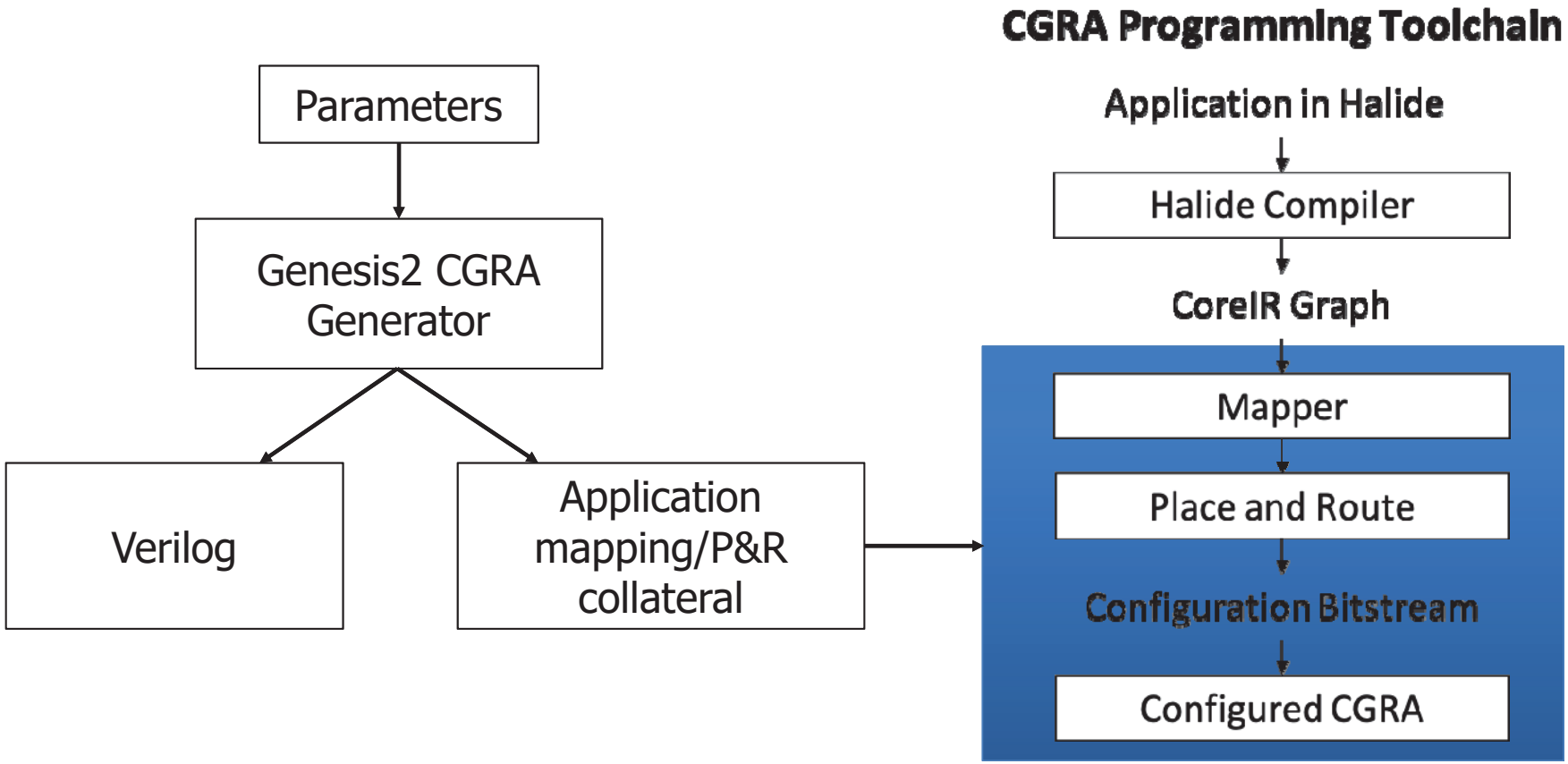
Design Methodology

- Genesis2 generators
- HW generation framework that uses Perl to metaprogram SystemVerilog
- Generators responsible for generating both the Verilog itself and collateral for our application toolchain

Genesis2 (Perl + Verilog)

```
//; for (my $h=0; $h<$cgra_grid_height; $h++) {
//;   for (my $w=0; $w<$cgra_grid_width; $w++) {
//;     my $tile_type = $tile_grid{$key};
//;     if ($tile_type eq "mem") {
//;       my $data_bus = $tile_config->
//;           { $tile_type }{ 'gen_mem_for_busname' };
//;       my $bus_width = $bus_width_hash { $data_bus };
   wire [`$bus_width-1`:0] mem_chain_`$h`_`$w`;
   wire  mem_chain_valid_`$h`_`$w`;
//;     }
//;     if (($tile_type eq "mem") || ($tile_type eq "pe")) {
//;       for (my $i=0; $i<$global_signal_count; $i++) {
//;         if ((${w}%2==0) && (${h}%2==0)) {
   wire global_wire_h2l_1_`${i}`_`${w}`_`${h}`;
//;         }
//;       }
   wire global_wire_l2h_0_`${w}`_`${h}`;
//;     }
//;   }
//; }
```

Interaction between HW Generation and Application Flows



- **Jade works!**

- But we made some mistakes (and learned some lessons) along the way

- Problems not only with hardware generation, but also with collateral generation

Problems with Collateral Generation

- Far too much manual effort required
  - Very long iteration times for design changes
- Parameters don't contain all information required
- No guarantee of consistency between collateral and actual hardware

```
assign mode = config_mem[1:0];
assign tile_en = config_mem[2];
assign depth = config_mem[15:3];
assign almost_count = config_mem[19:16];
assign enable_chain = config_mem[19];

// ... 216 lines later ...

//;my $filename = "MEM".$self->mname();
//;open(MEMINFO, ">$filename") or die "Couldn't open file $filename, $!";
//;print MEMINFO "       <mode bith='1' bitl='0'>00</mode>\n";
//;print MEMINFO "       <tile_en bith='2' bitl='2'>0</tile_en>\n";
//;print MEMINFO "       <depth bith='15' bitl='3'>0</depth>\n";
//;print MEMINFO "       <almost_count bith='19' bitl='16'>0</almost_count>\n";
//;print MEMINFO "       <chain_enable bith='20' bitl='20'>0</chain_enable>\n";
//;close MEMINFO;
```

Problems with Physical Design

- Jade had a lot of global signals that had to be routed to every tile in the array
- Late in our design process, we needed to amend our logical description to make it more amenable to physical implementation
- Error-prone and made RTL harder to understand
  - Poor separation of concerns
- In general, we want to keep logical and physical issues more separate

Automating Testing

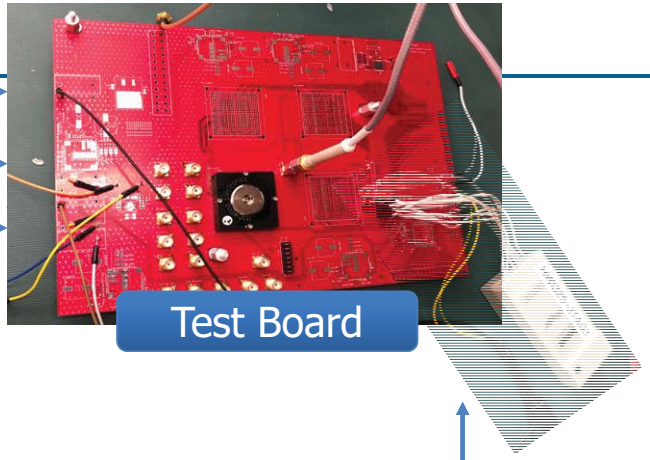
Manual testing (Analog + Digital) is a pain…

- Physical Discomfort
- **HUMAN ERROR**

Quality of Life Improvements with Automation

- Robust
  - Data collection is straightforward and accurate
  - Output text to a file
  - Parameterized control
- Remote Controlled
  - No longer required to be onsite
  - Control from home
- Modular
  - Plug in any arbitrary compatible testing equipment

Test System Topology



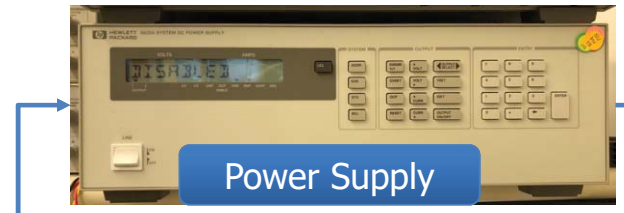
JTAG Driver

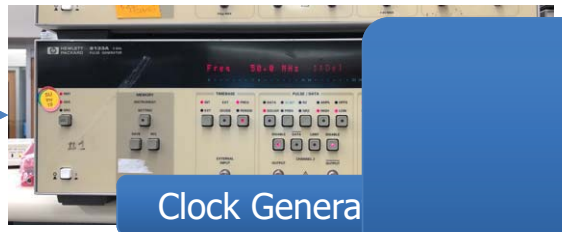
PyVISA

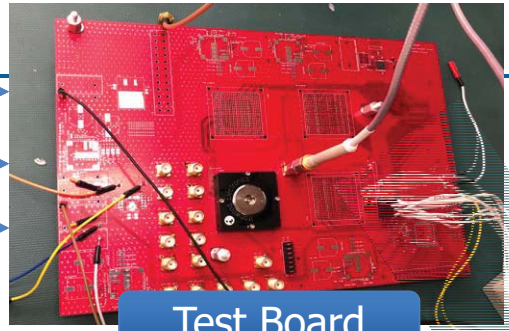
HOST PC

HPIB

Test System Topology







Modular ✓

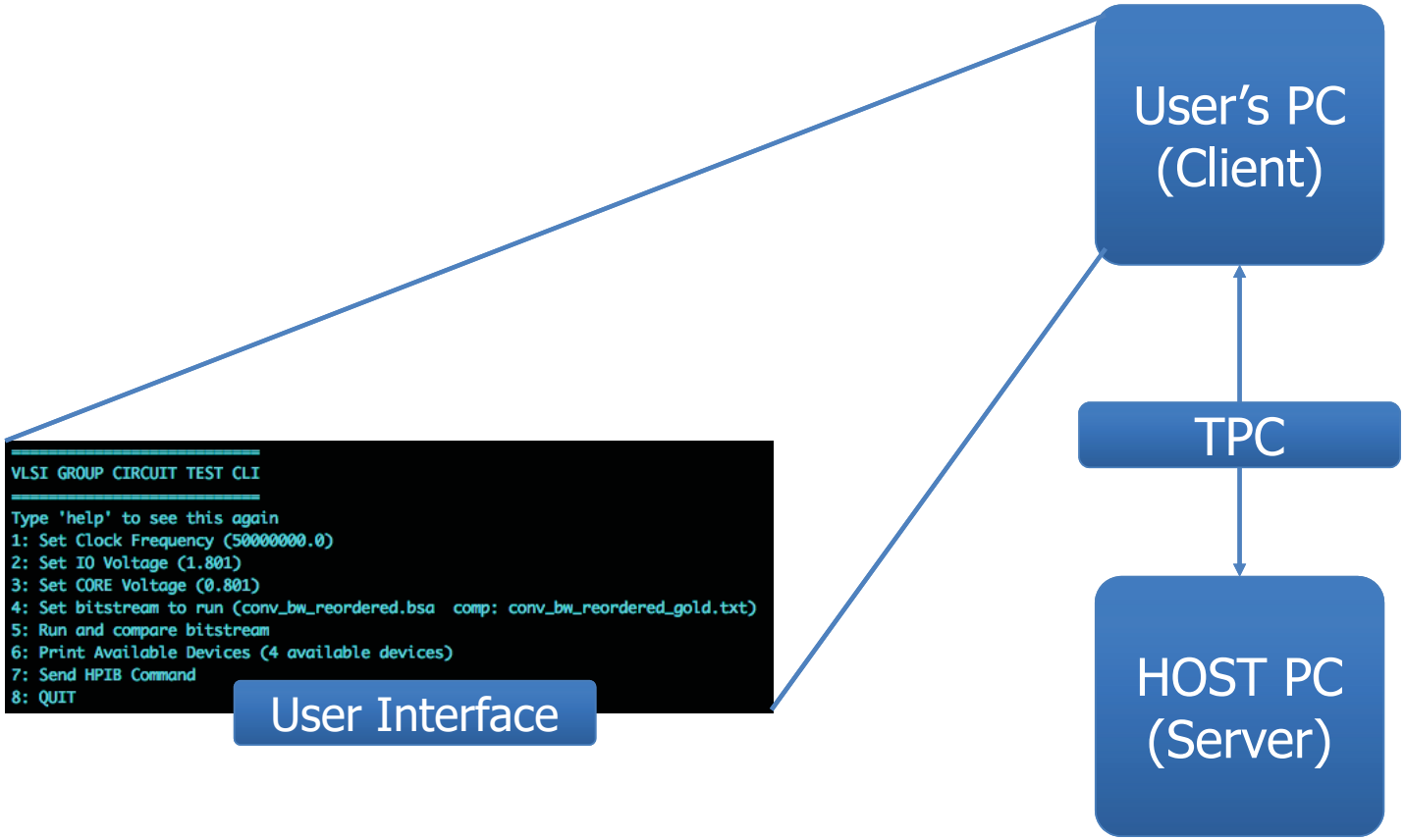
JTAG Driver

PyVISA

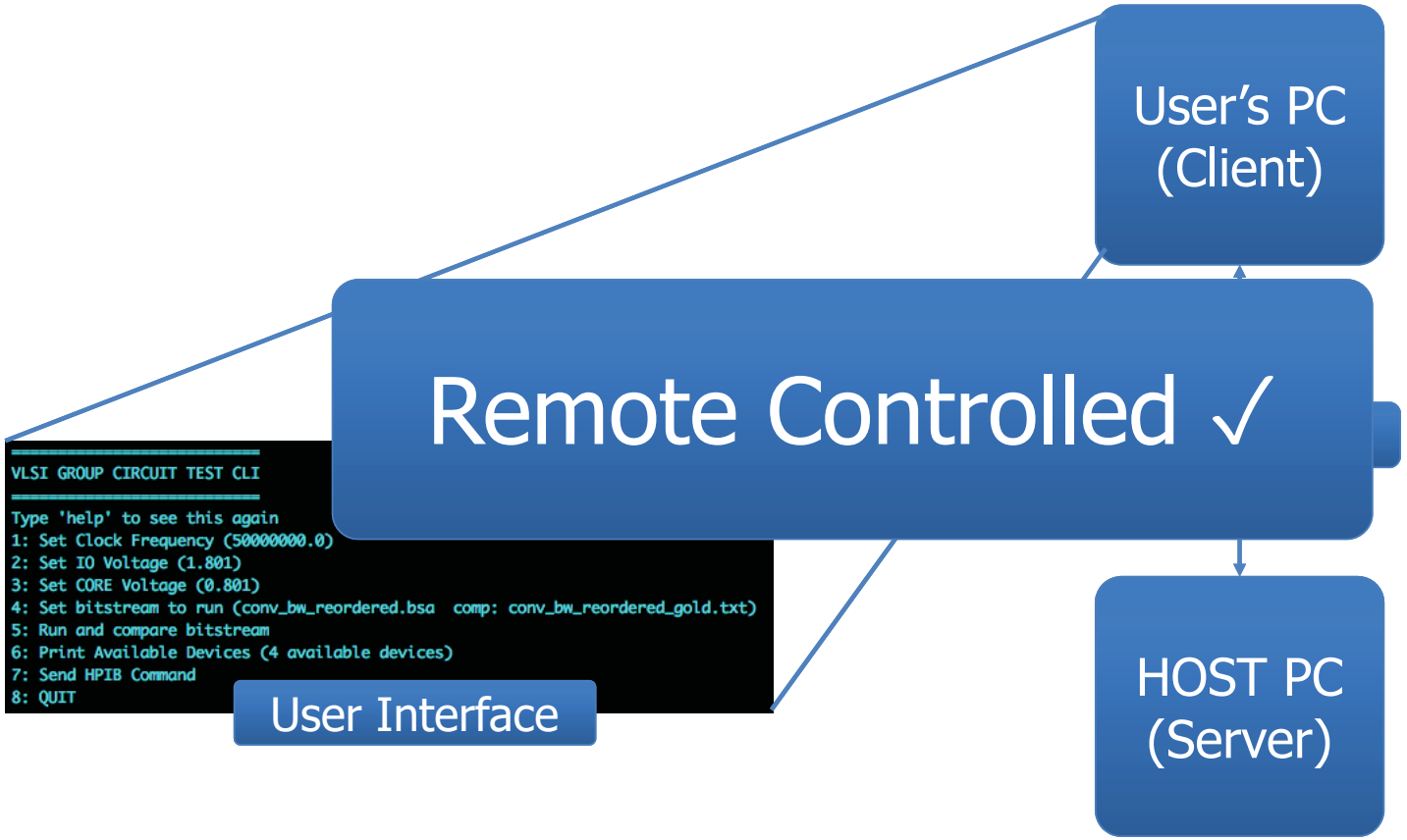
HOST PC

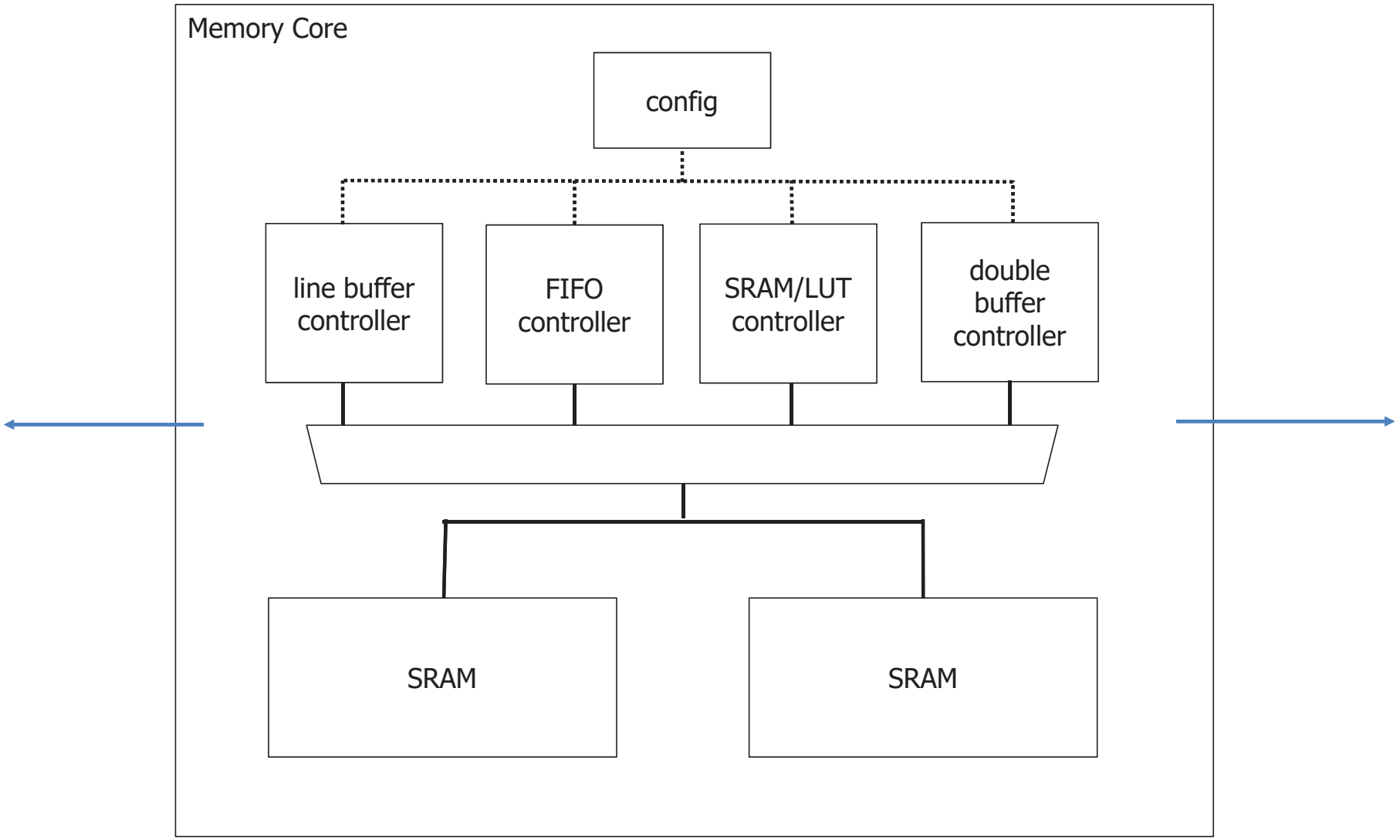
HPIB

User Interface



User Interface




Memory Core
config
line buffer controller
FIFO controller
SRAM/LUT controller
double buffer controller
SRAM
SRAM

Demo…

Demo…

Robust ✓

Next Steps

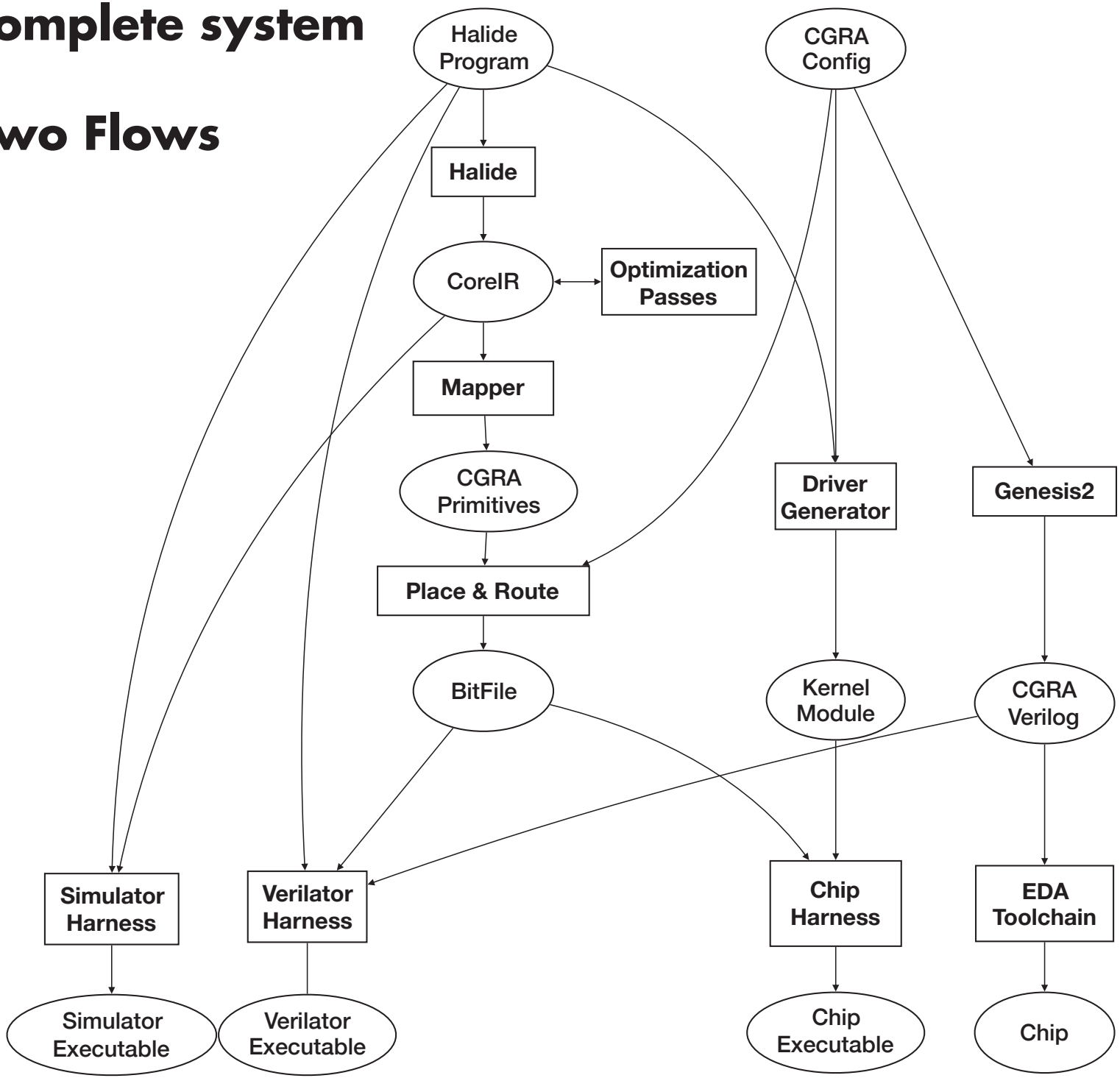
- Connect into CGRAFlow (end-to-end testing flow)
  - Automatically generate memory images as gold comparison files
  - Optionally target actual CGRA
    - Real-time power profiling
    - Iteratively find corner cases for operation (per app profiling)
- Fully Connected to FPGA
  - Allow for more interesting testing – bring in real I/O

How We are Building the Next CGRA

Pat Hanrahan

Generate complete system

Last chip: Two Flows



Gemstone Vision

Overall goal: Accelerate Halide applications using rapid hardware design-space exploration and evaluation

1. Generates complete working hw and sw system

2. Single/consistent source for all design collateral

3. Enhance text-templating approach (Genesis2) with semantic hardware components/DSLs (PEs, interconnect, memory, ...)

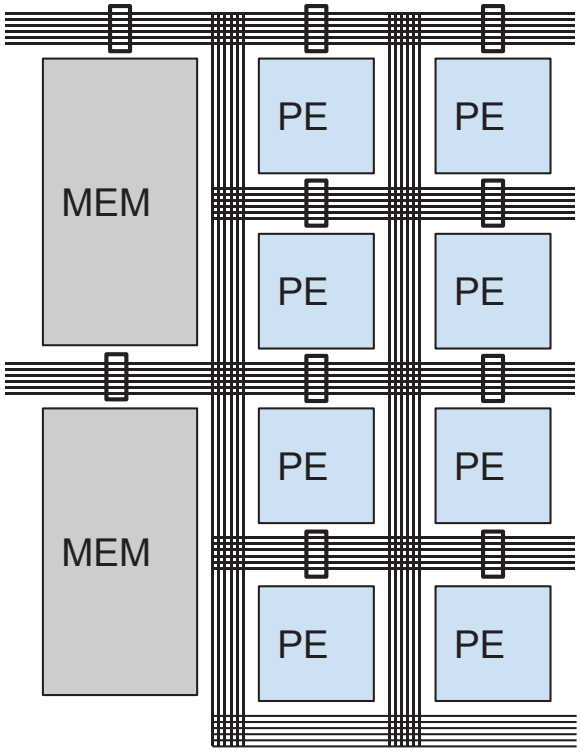
4. High-level representation of the chip that supports analysis and transformation of the design: passes such as inserting clock trees, power domains, configuration registers

Gemstone Components

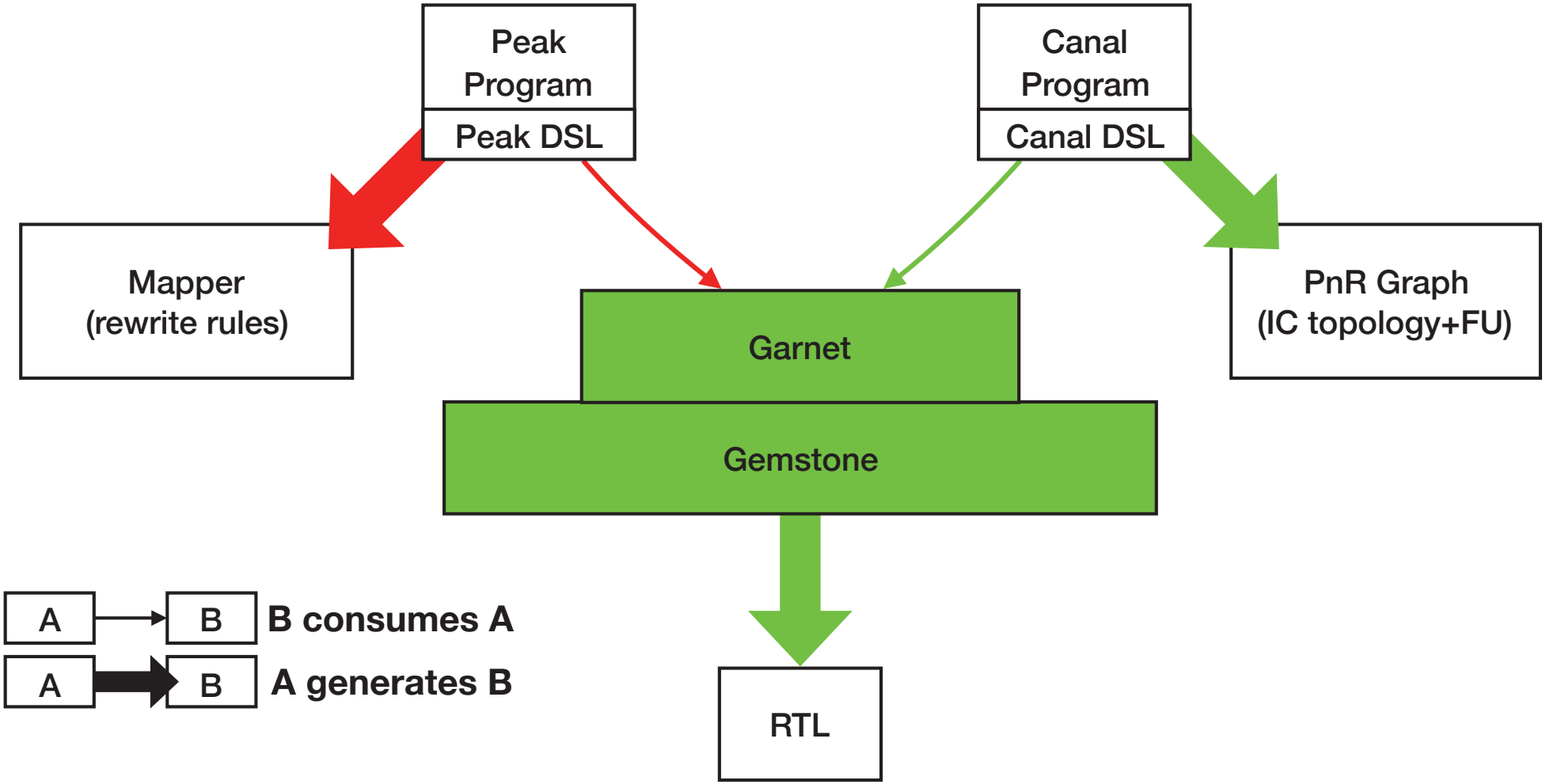
High-level model of a SOC containing a CGRA

Hardware "generators"

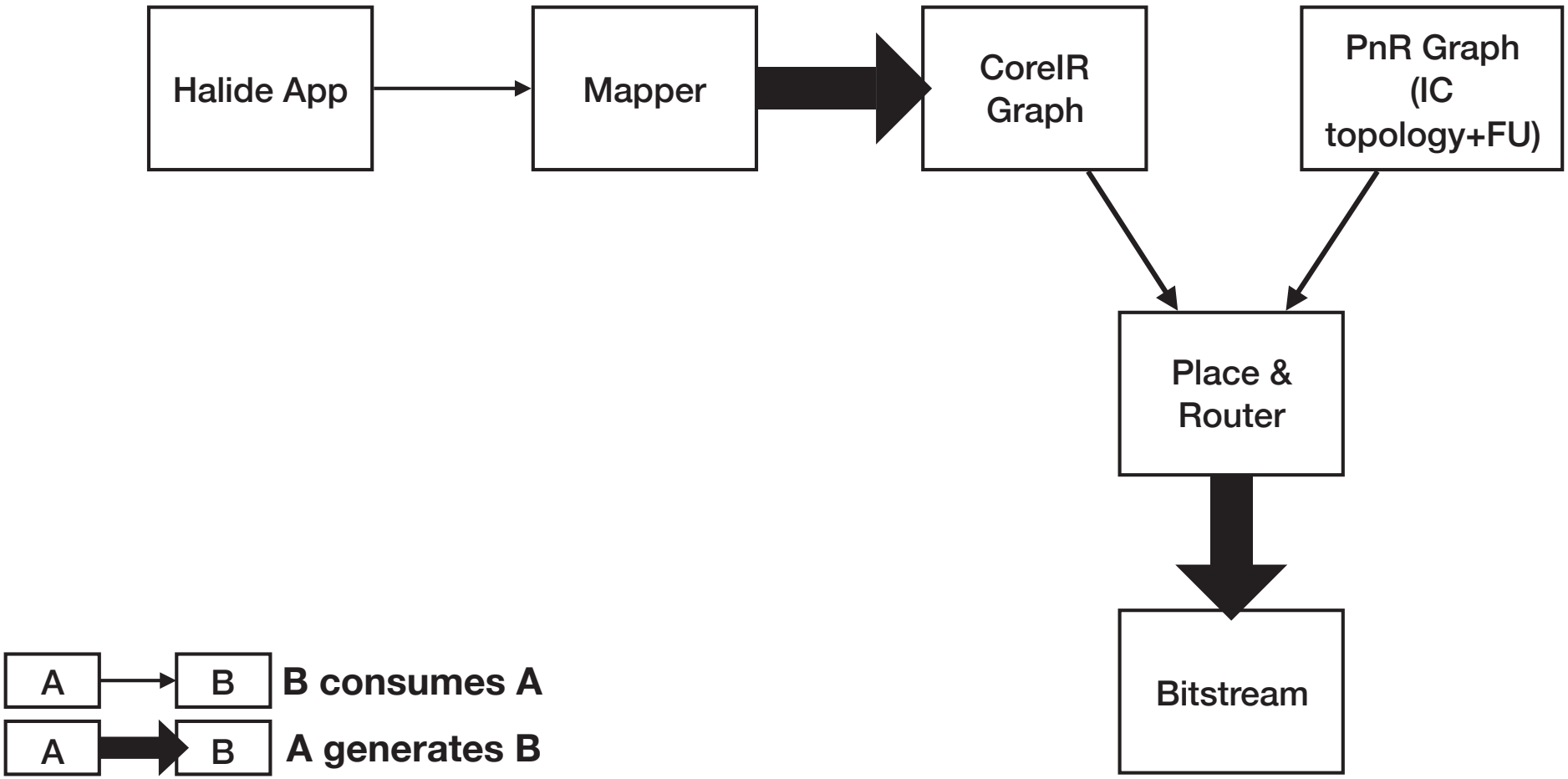
- **PE in *Peak***
- **Interconnect in *Canal***
- **Memory**
- **IO**
- **...**



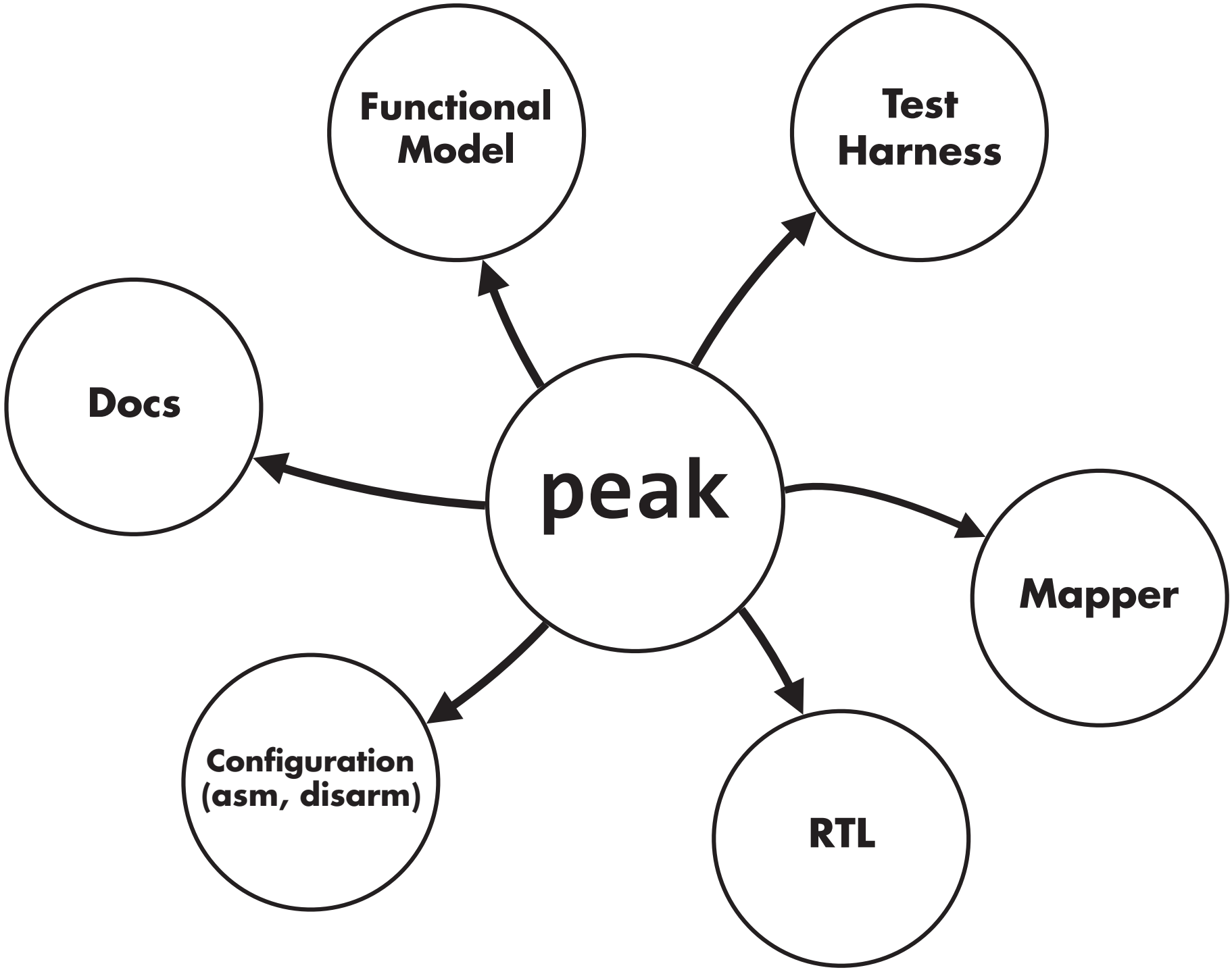
Raj, Alex, and Keyi



Generate both RTL and the Mapper/PnR info



Always able to compile Halide to CGRA



https://github.com/phanrahan/peak

```
class Lassen(Peak):
    def __init__(self):
        # Declare PE state

        # Data registers
        self.rega = Register(Data)
        self.regb = Register(Data)

        # Bit Registers
        self.regd = Register(Bit)
        self.rege = Register(Bit)
        self.regf = Register(Bit)
```

Module inputs and outputs

```
# one clock cycle
def __call__(self, inst: Inst, \
    data0: Data, data1: Data = Data(0), \
    it0: Bit = Bit(0), bit1: Bit = Bit(0), bit2: Bit = Bit(0), \
    clk_en: Bit = Bit(1)) -> (Data, Bit):

    # pass input data through register (BYPASS, DELAY, …)
    ra = self.rega(inst.rega, inst.data0, data0, clk_en)
    rb = self.regb(inst.regb, inst.data1, data1, clk_en)
    rd = self.regd(inst.regd, inst.bit0, bit0, clk_en)
    re = self.rege(inst.rege, inst.bit1, bit1, clk_en)
    rf = self.regf(inst.regf, inst.bit2, bit2, clk_en)

    # calculate alu results
    alu_res, alu_res_p, Z, N, C, V = \
         alu(inst.alu, inst.signed, ra, rb, rd)
    # calculate lut results
    lut_res = lut(inst.lut, rd, re, rf)
    # calculate 1-bit result
    res_p = cond(inst.cond, alu_res, lut_res, Z, N, C, V)

    return alu_res, res_p
```

Peak DSL

Functional simulation in python (we use peak to generate test vectors for the current design); also supports formal methods

Generate RTL (coreir/verilog via magma)

Generate mapper (coreir -> isa using SMT)

Generate configuration (assembler)

Magma

Hardware Construction Language

Lenny

https://github.com/phanrahan/magma

Chisel in Scala

```
//A n-bit adder with carry in and carry out
class Adder(val n:Int) extends Module {
  val io = IO(new Bundle {
    val A    = Input(UInt(n.W))
    val B    = Input(UInt(n.W))
    val Cin  = Input(UInt(1.W))
    val Sum  = Output(UInt(n.W))
    val Cout = Output(UInt(1.W))
  })
  //create an Array of FullAdders
  val FAs   = Array.fill(n)(Module(new FullAdder()).io)
  val carry = Wire(Vec(n+1, UInt(1.W)))
  val sum   = Wire(Vec(n, Bool()))

  //first carry is the top level carry in
  carry(0) := io.Cin

  //wire up the ports of the full adders
  for (i <- 0 until n) {
    FAs(i).a := io.A(i)
    FAs(i).b := io.B(i)
    FAs(i).cin := carry(i)
    carry(i+1) := FAs(i).cout
    sum(i) := FAs(i).sum.toBool()
  }
  io.Sum := sum.asUInt
  io.Cout := carry(n)
}
```

Magma in Python

```
class FullAdder(Circuit):
    IO = ["a", In(Bit), "b", In(Bit), "cin", In(Bit),
          "out", Out(Bit), "cout", Out(Bit)]
    @classmethod
    def definition(io):
        _sum = io.a ^ io.b ^ io.cin
        wire(_sum, io.out)
        carry = io.a & io.b | io.b & io.cin | io.a & io.cin
        wire(carry, io.cout)

def DefineAdder(N):
    T = UInt(N)
    class Adder(Circuit):
        name = f"Adder{N}"
        IO = ["a", In(T), "b", In(T), cin", In(Bit),
              "out", Out(T), "cout", Out(Bit)]
        @classmethod
        def definition(io):
            adders = col(FullAdder, N)
            circ = fold(adders, {"cin":"cout"})
            wire(io.a, circ.a); wire(io.b, circ.b)
            wire(io.cin, circ.cin)
            wire(io.cout, circ.cout)
            wire(io.out, circ.out)
    return Adder

Adder4 = DefineAdder(4)
adder = Adder4()
wire(main.I0, adder.a)
wire(main.I1, adder.b)
```

Python

- Dynamically-typed, type generators
- Popular, with extensive libraries

Magma

- Static product data types
- Circuits, instances, and wiring

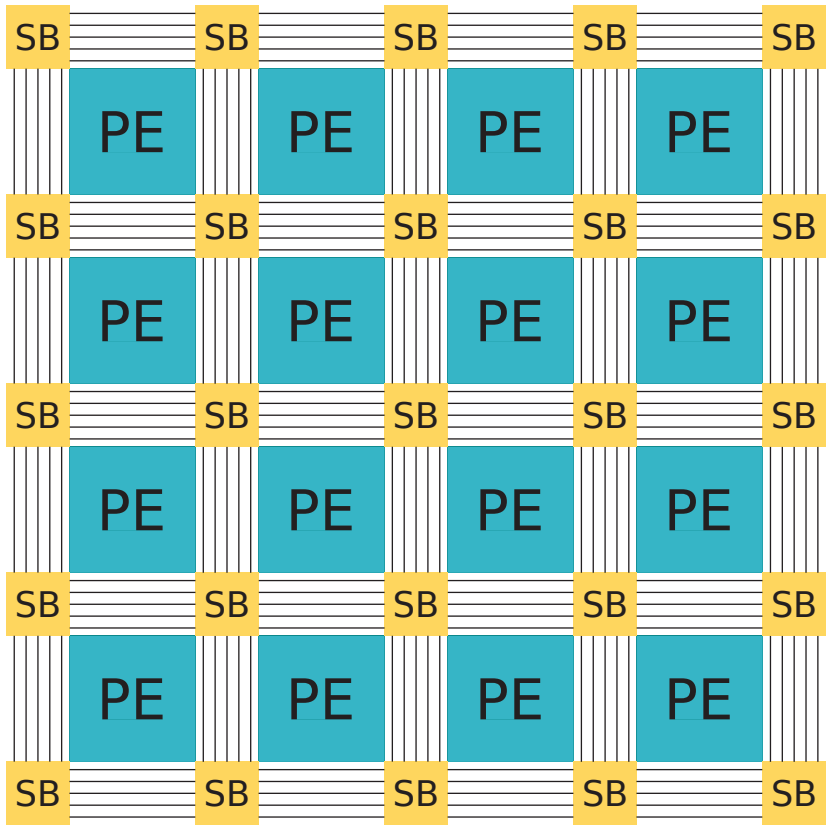
Python-Magma system

- Python meta-programs Magma
- Enables embedded DSLs

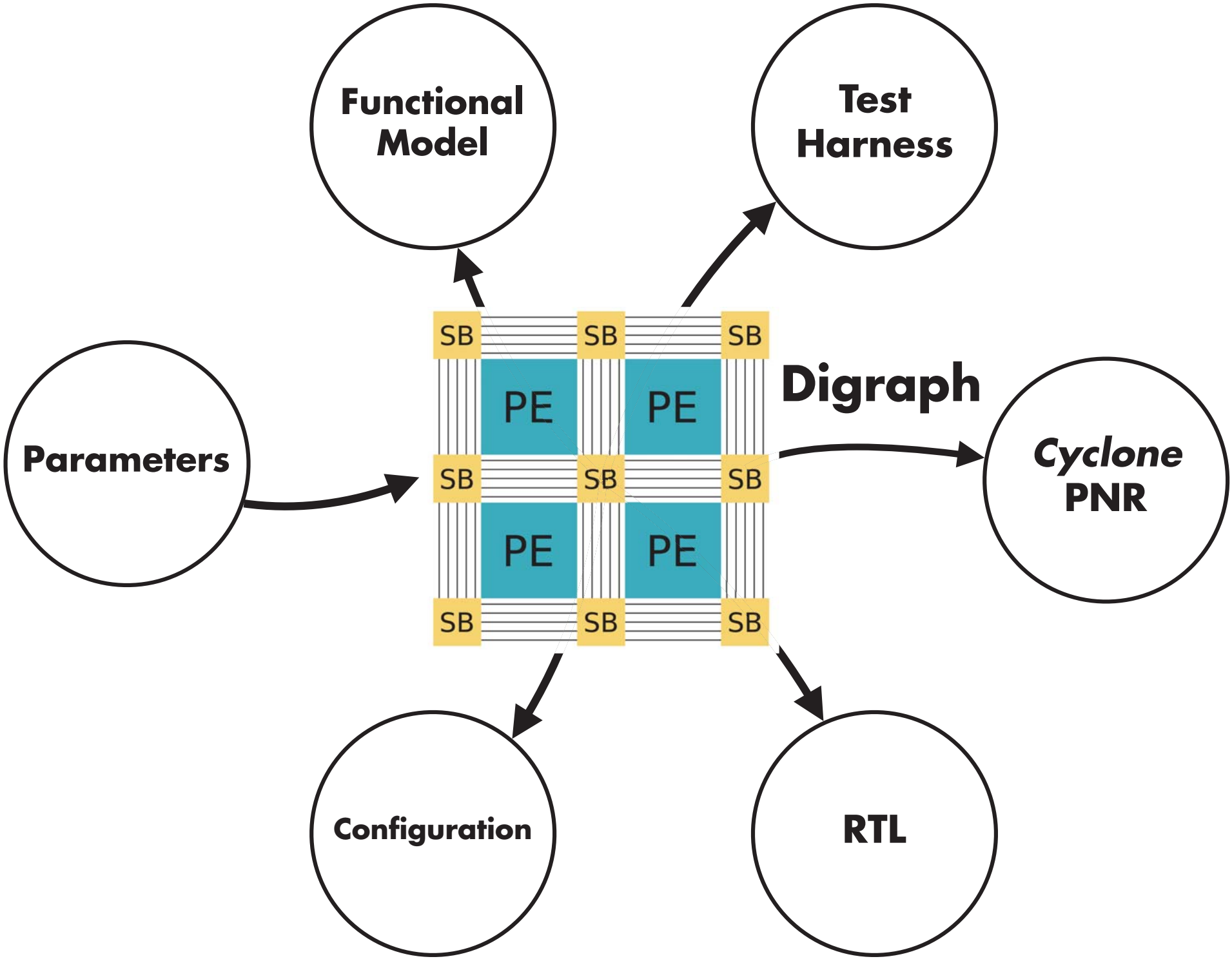
Canal

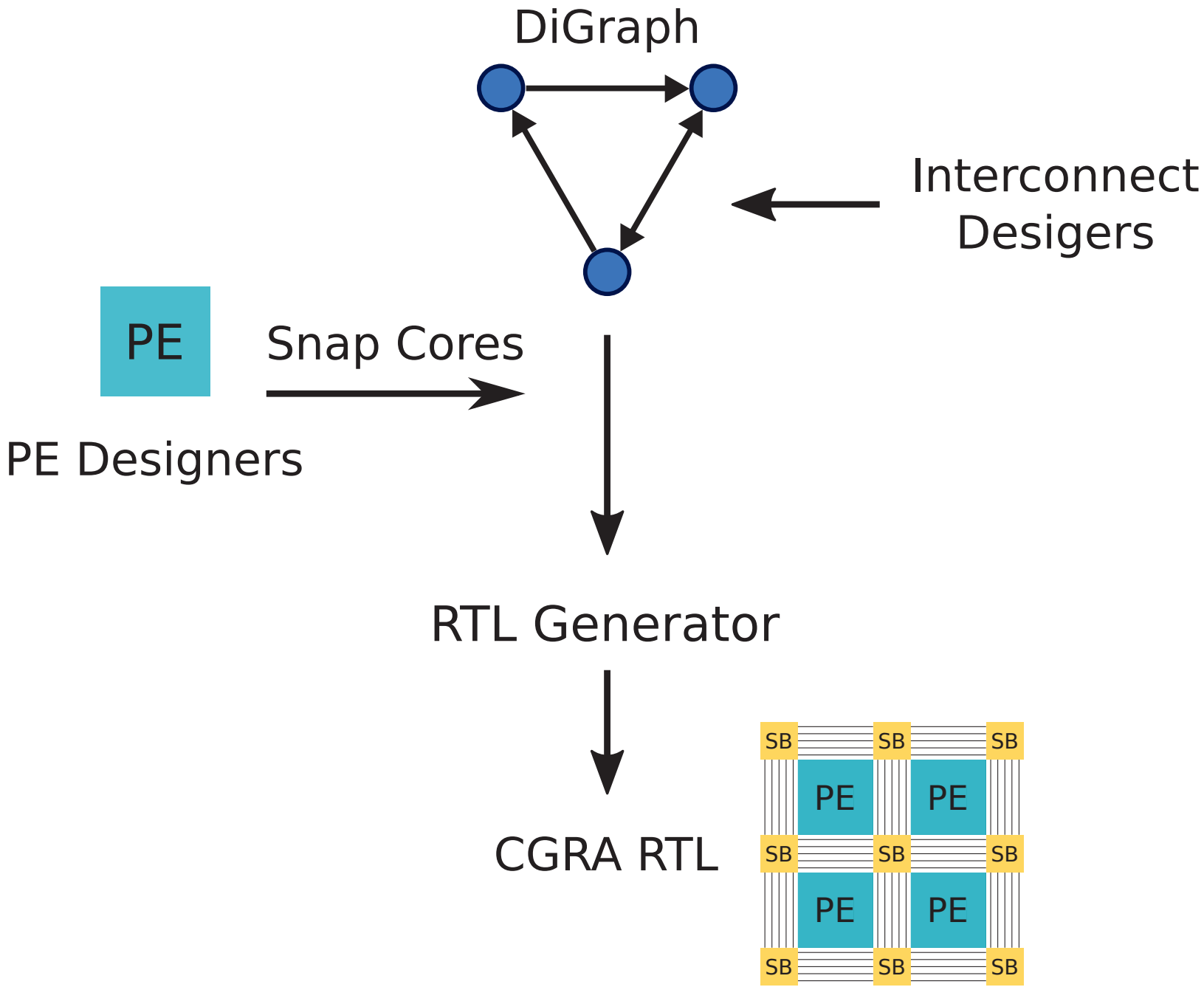
Parameterized island-style interconnects (ala VPR)

- **Connection box (CB)**
- **Switch box (SB)**
  - **Disjoint**
  - **Wilton**
  - **Imran**
- **Pipeline registers**
- **Configuration**



Keyi


Functional Model
Test Harness
Parameters
SB
PE
Digraph
Cyclone PNR
Configuration
RTL

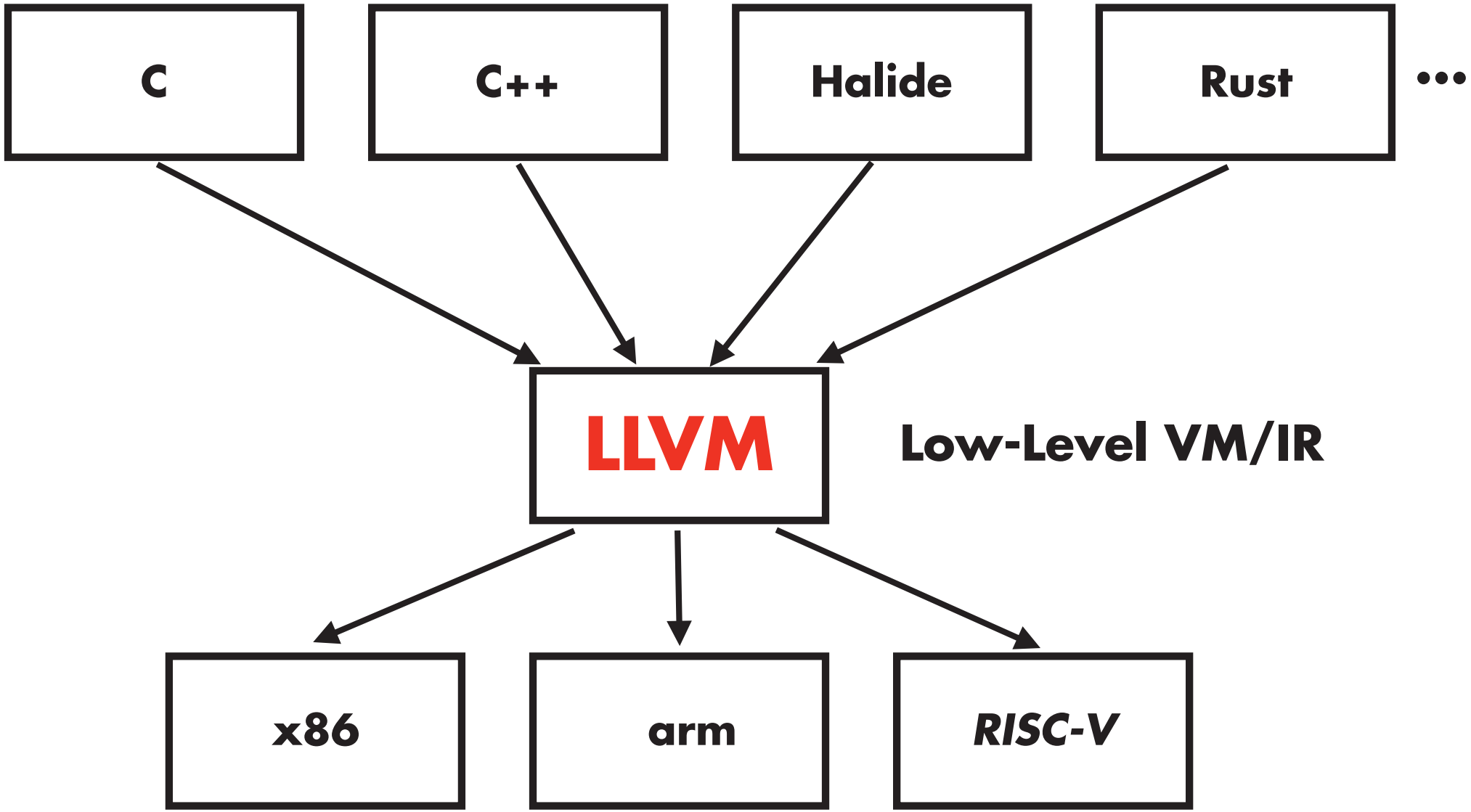

DiGraph
Interconnect
Desigers
PE
Snap Cores
PE Designers
RTL Generator
CGRA RTL
SB
PE

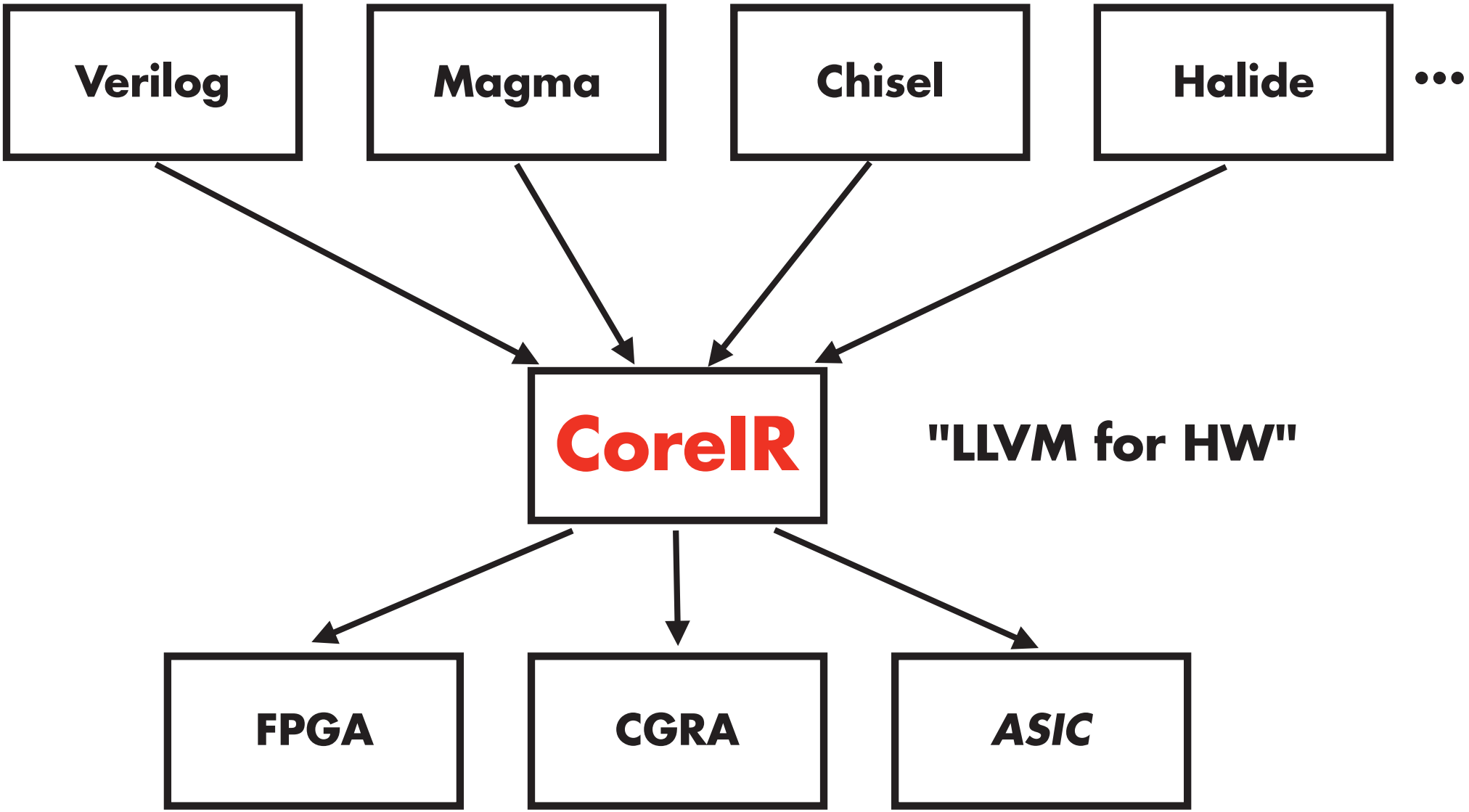
CoreIR

"LLVM for HW"

Ross

https://github.com/rdaly525/coreir


C
C++
Halide
Rust
...
LLVM
Low-Level VM/IR
x86
arm
RISC-V


Verilog
Magma
Chisel
Halide
...
CoreIR
"LLVM for HW"
FPGA
CGRA
ASIC

Different Abstractions

LLVM - Processor + memory

- **ALU, registers, memory, ...**

CoreIR - RTL

- **Combination and sequential logic**
- **Distributed state / registers and memories**
- **Structural : DAG of operations (always synthesizable)**
- **Hierarchical modules**
- **Generators (parameterized modules)**

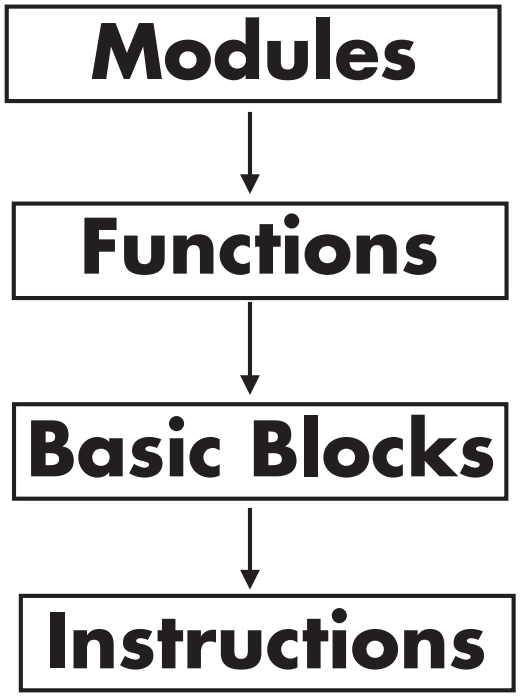
CoreIR Primitives

Bitwise	buf, not, and, or, xor, andr, orr, xorr, shl, lshr, ashr
Arithmetic	neg add, sub, mul udiv, urem, sdiv, srem, smod
Comp	eq, neq, slt, sgt, sle, sge, ult, ugt, ule, uge
Stateful	reg, regrst, mem
Other	mux, slice, concat, zext, sext

hwtypes: python bitvector, smt-lib

https://github.com/StanfordAHA/Primitives/blob/master/coreirprims.csv

LLVM Pass Types



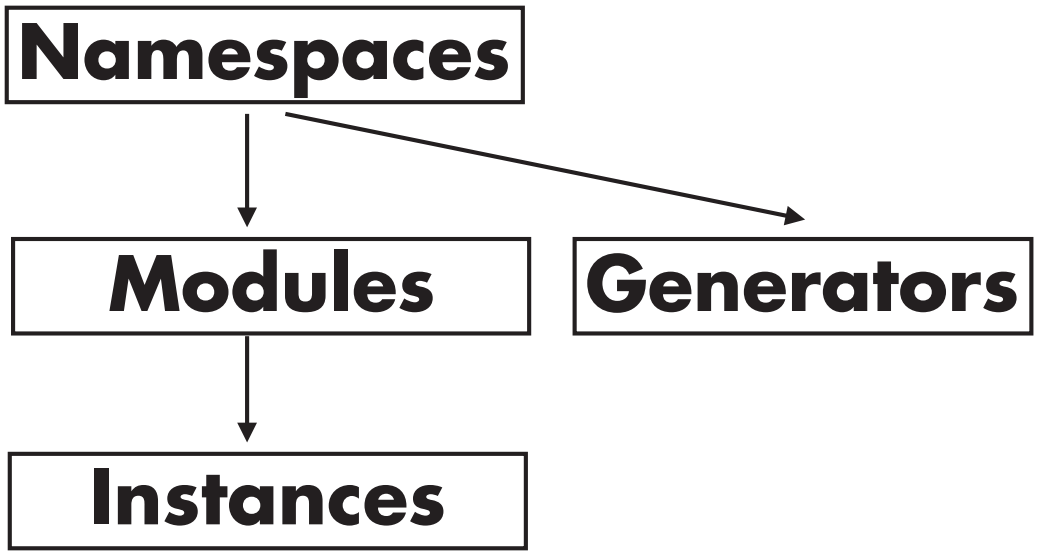
IR passes

- **ModulePass**
- **CallGraphPass**
- **CallGraphSCCPass**
- **FunctionPass**
- **LoopPass**
- **RegionPass**
- **BasicBlockPass**
- **InstructionPass**

Pass manager

- **Executes passes in order**
- **Caches intermediates**

CoreIR Pass Types



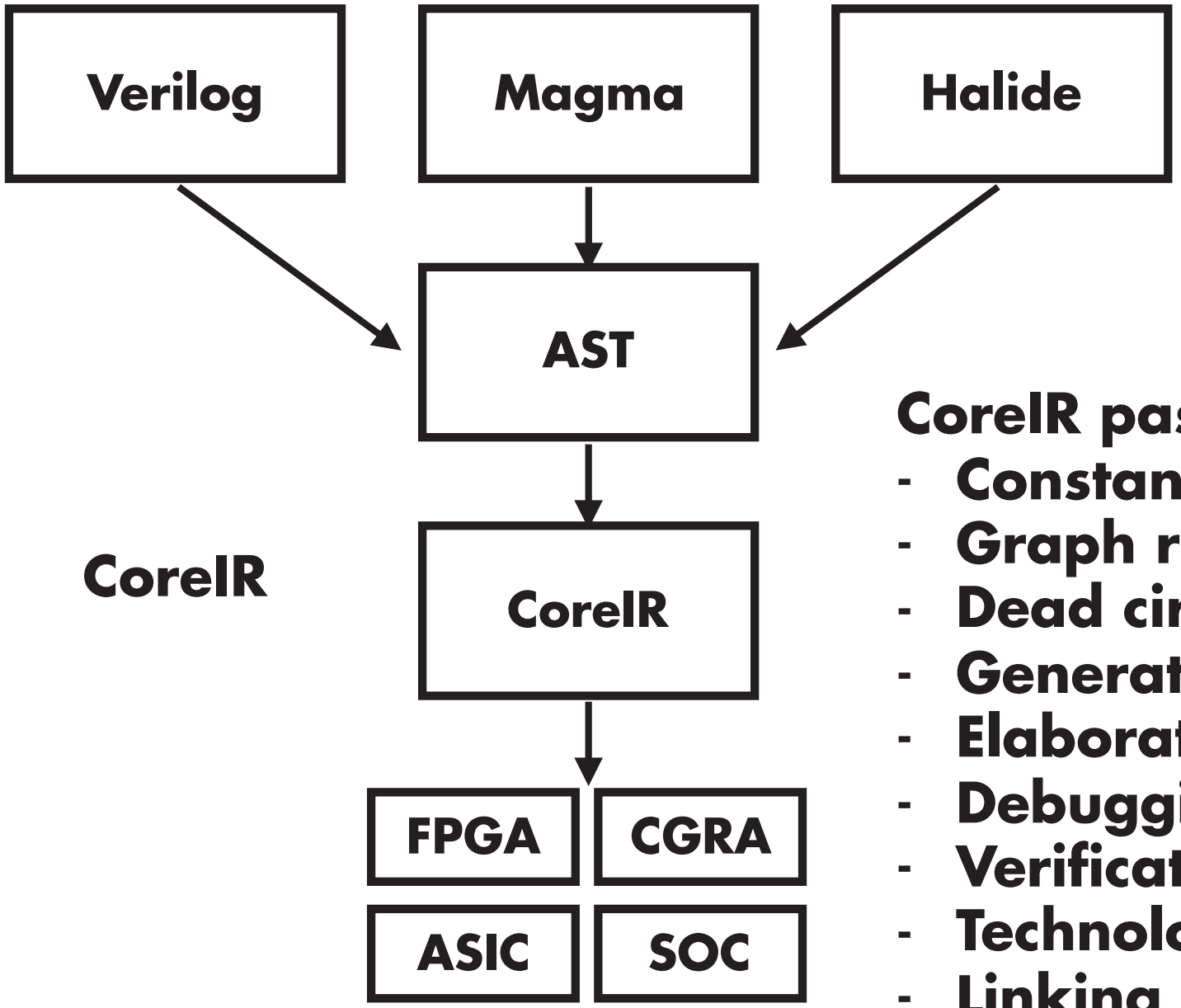
Direct Pass Types

- NamespacePass
- ModulePass
- InstancePass

Other Pass Types

- InstanceGraphPass
- InstanceVisitorPass

CoreIR Passes



CoreIR passes

- Constant folding
- Graph rewrites
- Dead circuit elimination
- Generators
- Elaboration
- Debugging support
- Verification / model ch.
- Technology mapping
- Linking

Gemstone Framework

Gemstone components interoperate

- Generator base class
- Library of basic pass types
- Staged generation
- Configuration manager
- Provides hardware components like registers, muxes

Supporting Physical Design

Layout Information

- **Floor planning**
- **Pin placement**

Global Signals

- **Distribute clock and other global signals**
- **Experiment with different distribution strategies**

Power domains

- **Adding power switches**
- **Adding always-on buffers**
- **Providing boundary protection**

Raj, Alex, Ankita, Keyi, ...

Status

Finished prototypes: Gemstone, Peak, Canal, ...

Lassen PE modeled in Peak (includes FP)

Generate Garnet CGRA with Lassen and new interconnect in Canal (old memory generator)

Automatically builds mapper, place-and-route graph, and configuration engine

Full system simulation in verilog of Garnet running a halide program using floating point

Evaluates area and power

Plans for Tapeout

Gemstone

- **Tighter integration with physical design**
- **Pass abstractions, Magma circuit edits**

Peak

- **Mapper rules for floating point**

Magma

- **Consistent hardware types, support for method interfaces**

Memory

Fault

DHLS

Garnet: The Next Generation CGRA Architecture

Priyanka Raina

DARPA Review Meeting

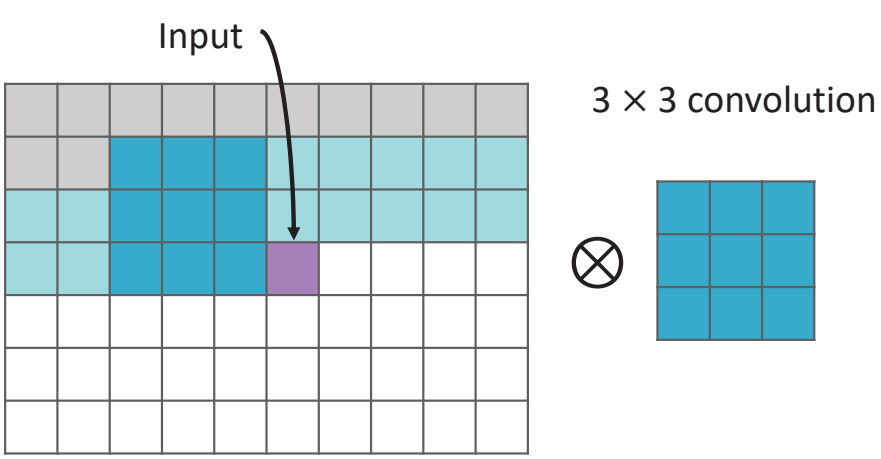
April 18, 2019

Challenges with Jade CGRA

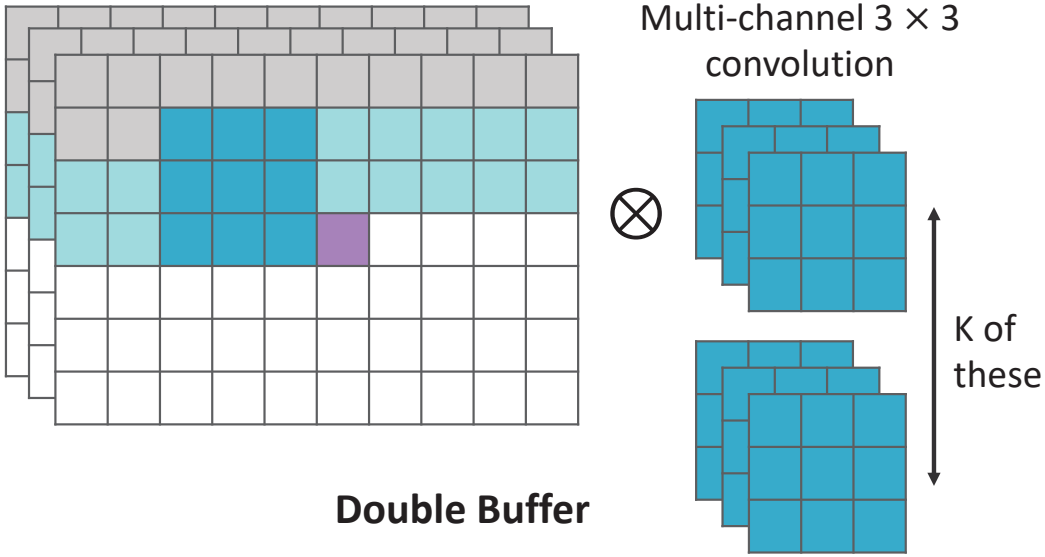
1. PE supports only simple integer operations
   - Porting applications that use floating point requires application expertise and manual effort

Challenges with Jade CGRA

2. Memory supports only line buffered pipelines
   - Most new applications even for imaging and vision use neural networks
   - Need a memory hierarchy with double buffers for energy-efficiency



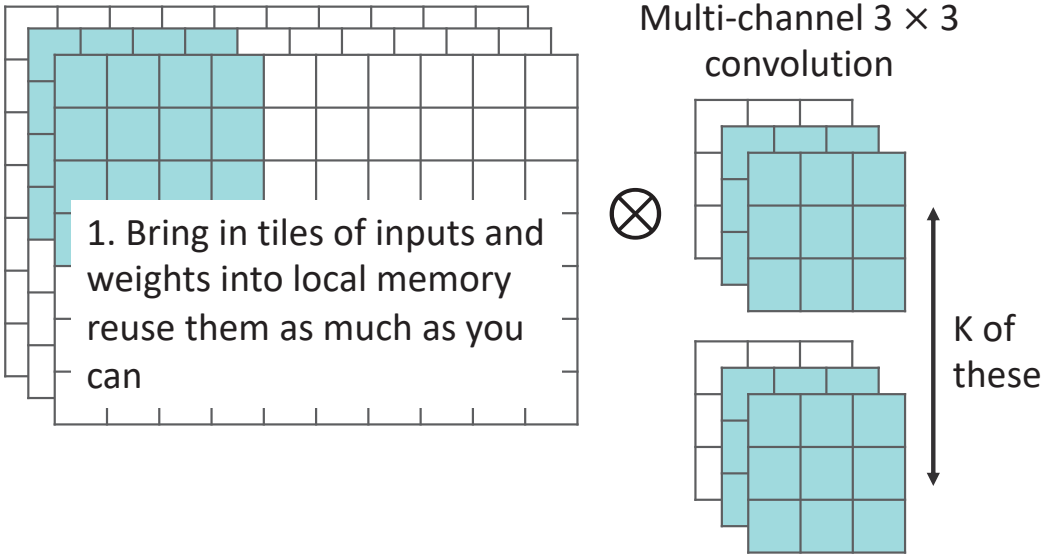
Line Buffer



Double Buffer

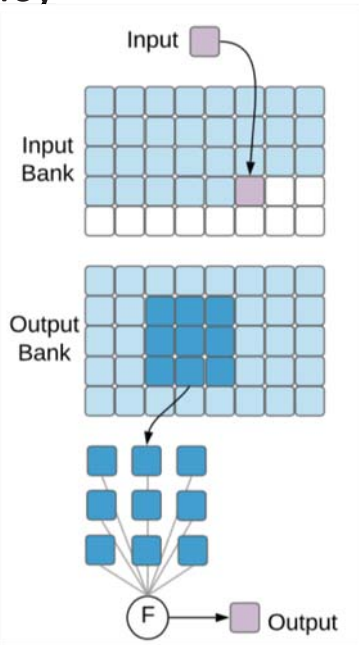
Challenges with Jade CGRA

2. Memory supports only line buffered pipelines
   - Most new applications even for imaging and vision use neural networks
   - Need a memory hierarchy with double buffers for energy-efficiency



2. Use a double buffer to overlap compute on current tile and fetching of next tile

3. Use a hierarchy of double buffers to maximize energy-efficiency



Challenges with Jade CGRA

3. Configuration over JTAG is slow
   - Large applications with multiple kernels don't fit on the CGRA – need fast reconfiguration

Key (application driven!) architectural changes in Garnet CGRA

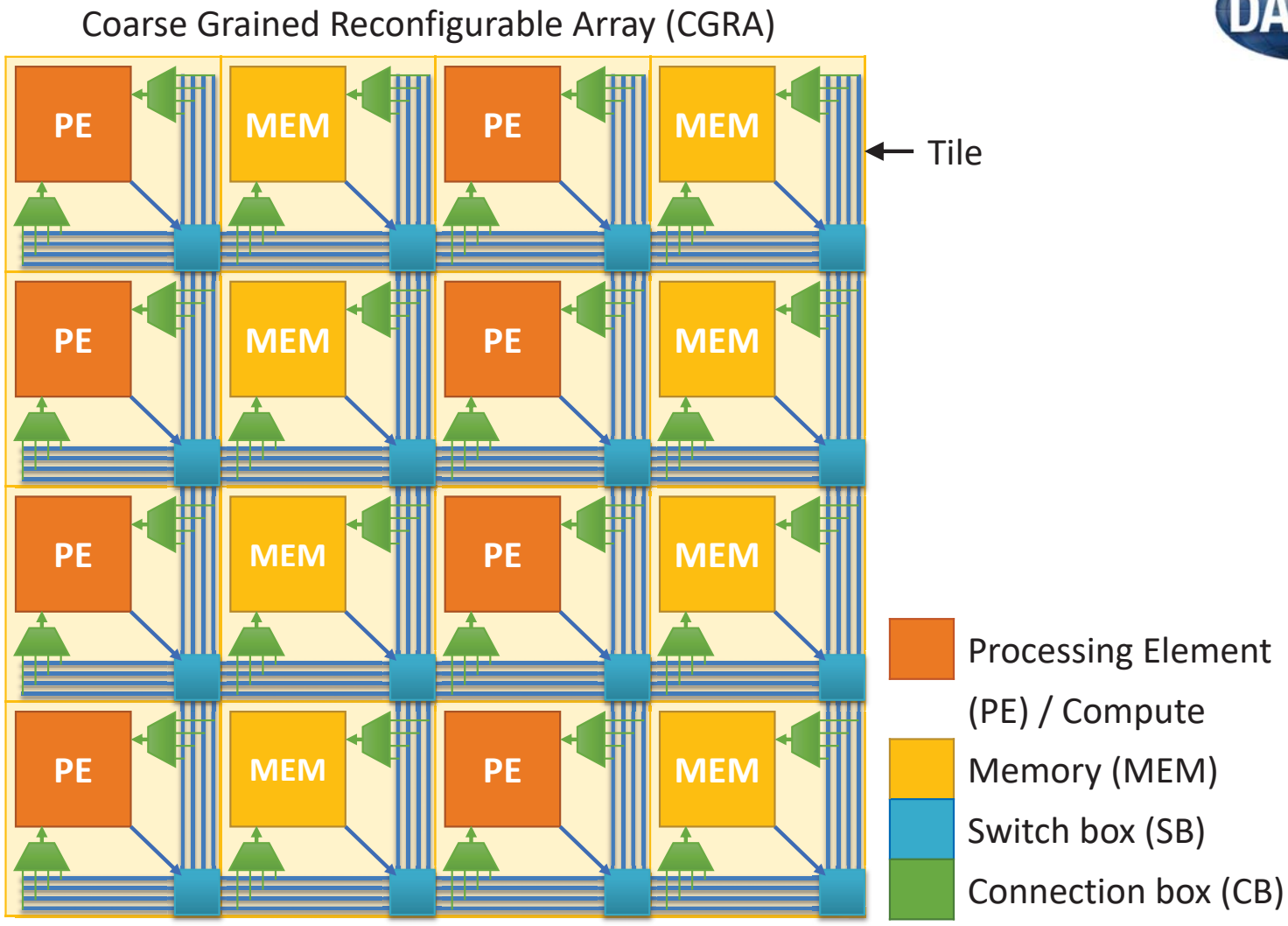
1. Support for Bfloat16, and for executing complex operations like divide using multiple PEs

2. Addition of a global buffer to create a memory hierarchy for efficiently executing neural networks, and double buffer support in all memories

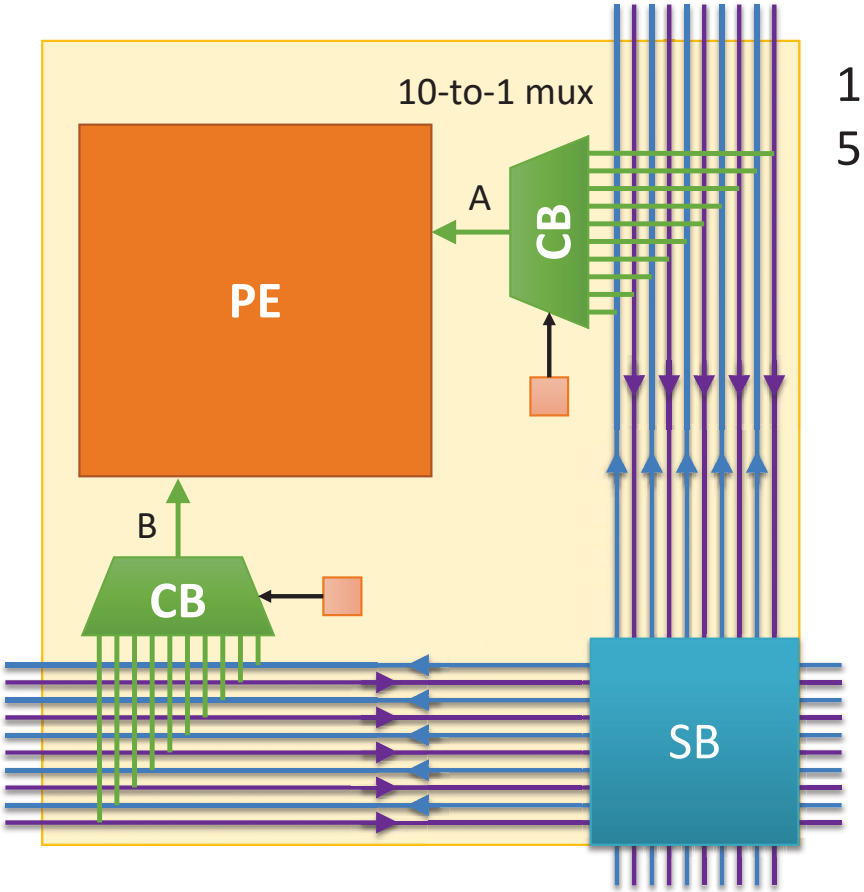
3. Fast reconfiguration support using global buffer and control processor

4. Addition of configurable power domains

CGRA



Connection Box (CB)

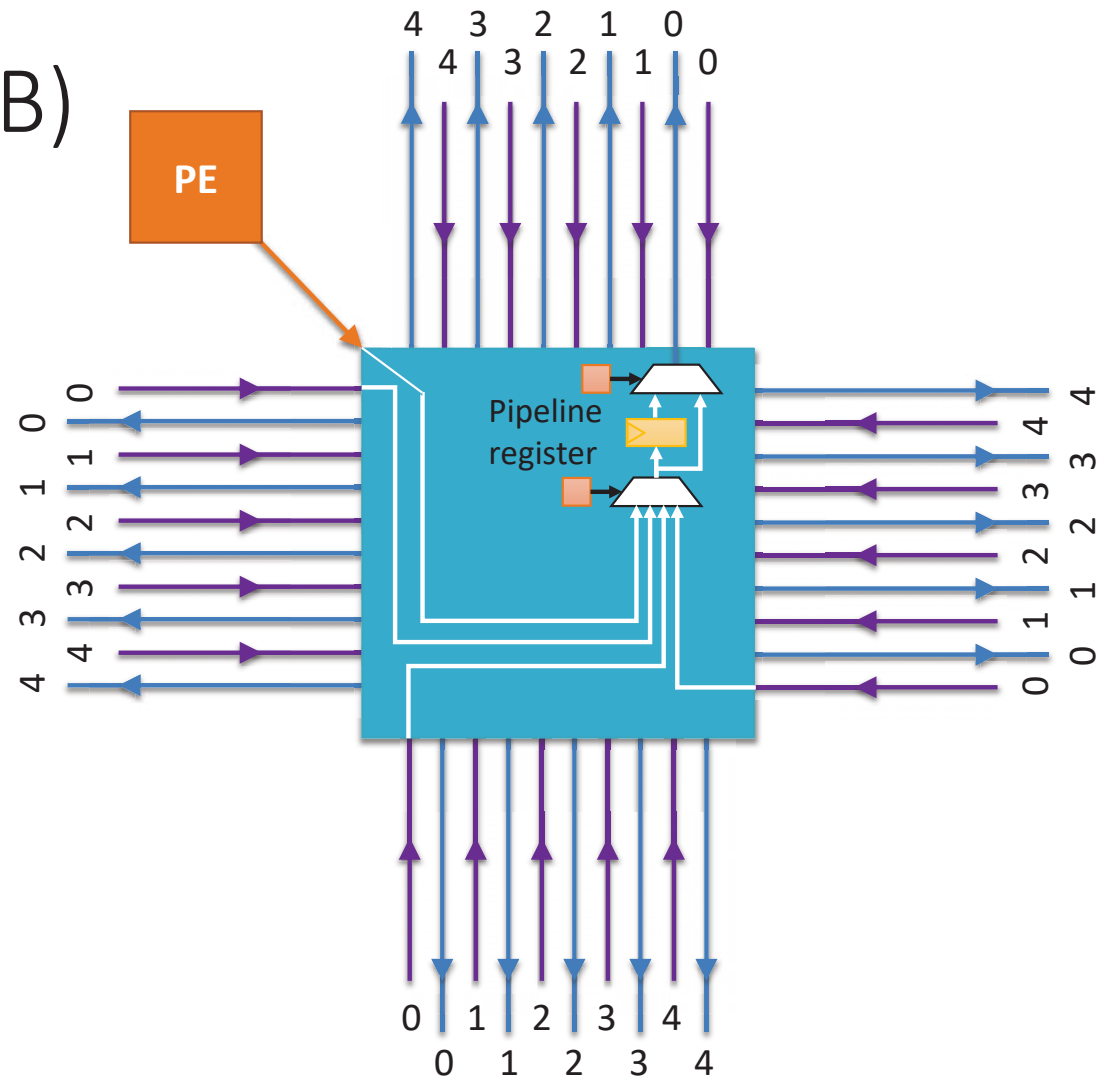


10 vertical 16 bit tracks
5 driven up, 5 driven down

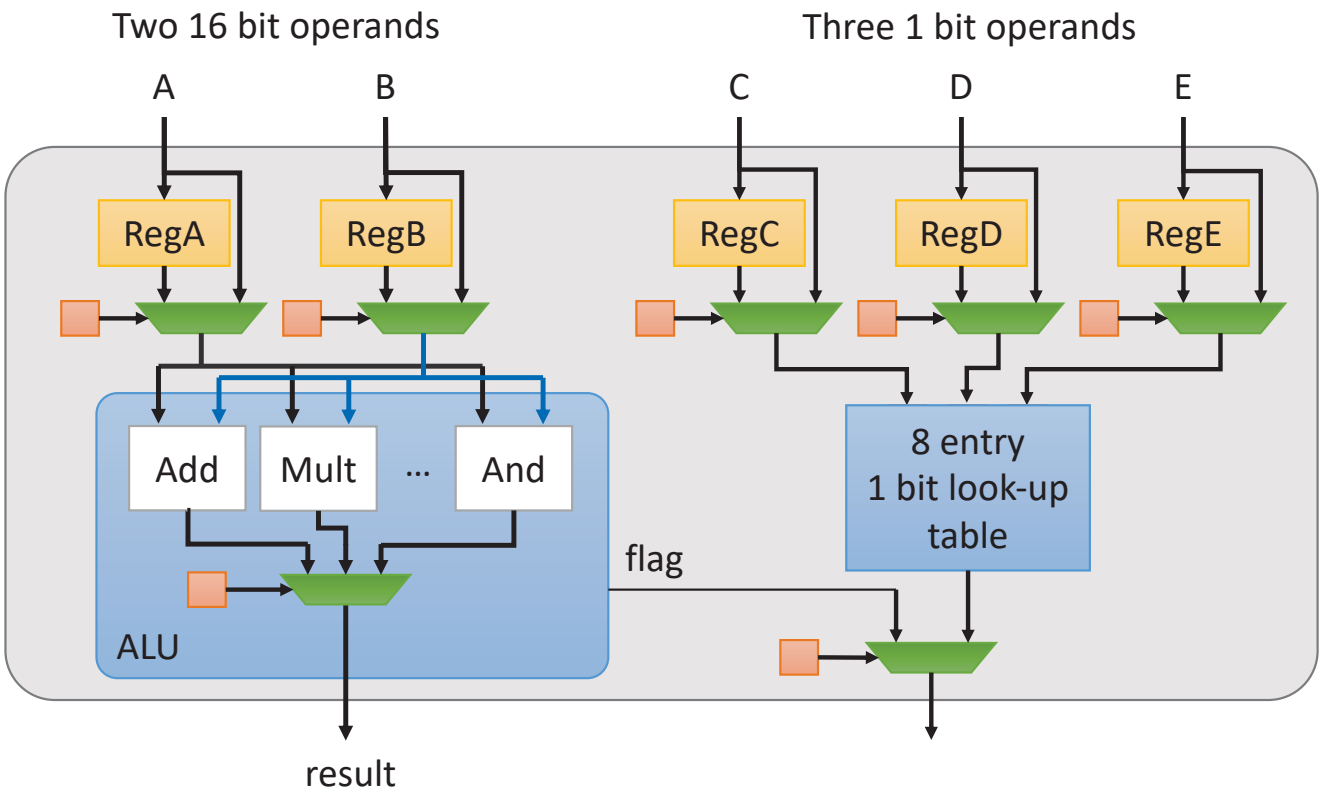
10 horizontal 16 bit tracks
5 driven left, 5 driven right

There are similar 1 bit CBs for feeding 1 bit inputs from 10 horizontal and vertical 1 bit tracks

Switch Box (SB)



Processing Element (PE)



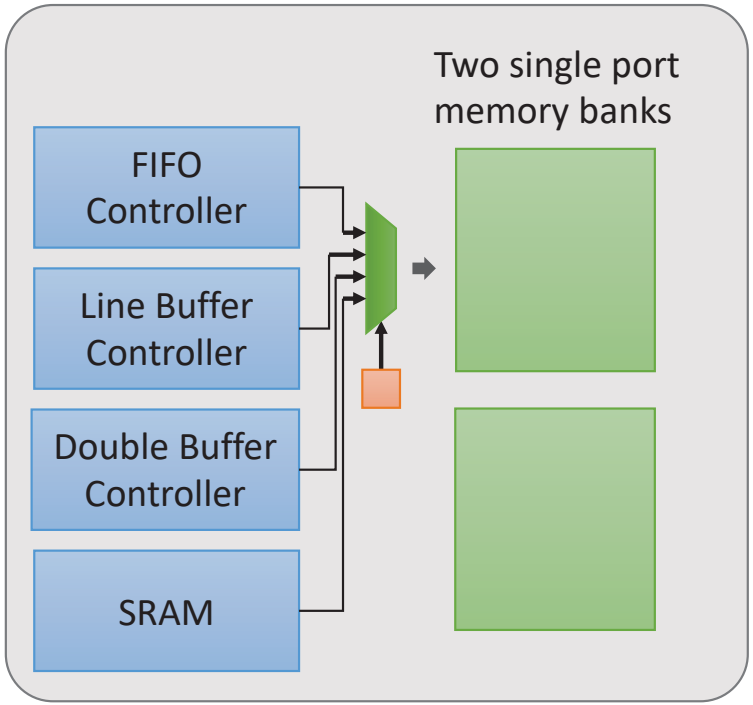
Diablo vs Lassen

Diablo Ops	New Ops in Lassen
Add	FPAdd
Sub	FPMult
Abs	FGetMant
GTE_Max	FAddIExp
LTE_Min	FSubExp
Sel	FCnvExp2F
Mult0	FGetFInt
Mult1	FGetFFrac
Mult2	
SHR	
SHL	
Or	
And	
XOr	

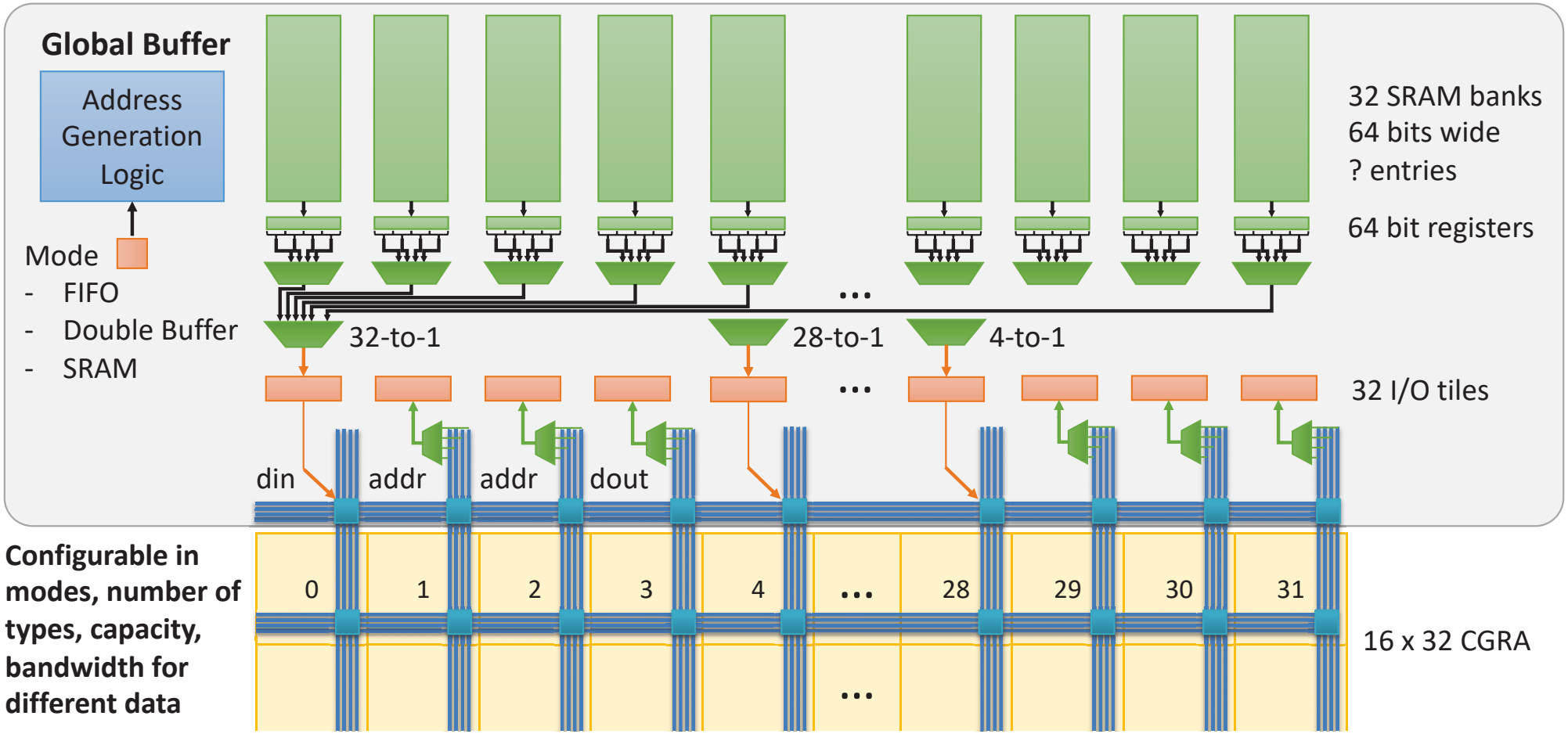
- Lassen supports 16 bit Bfloat type
  - 1 b sign, 8 b exponent, 7 bit mantissa
  - Same range as Float32
- Bfloat add and multiply natively
- Complex ops like divide, exp, power, log, sin, cos using multiple PE and memory tiles

1. **Allows easy porting of applications!**
2. **Enables new applications that need a large range like DNN training or complex ops like divide**

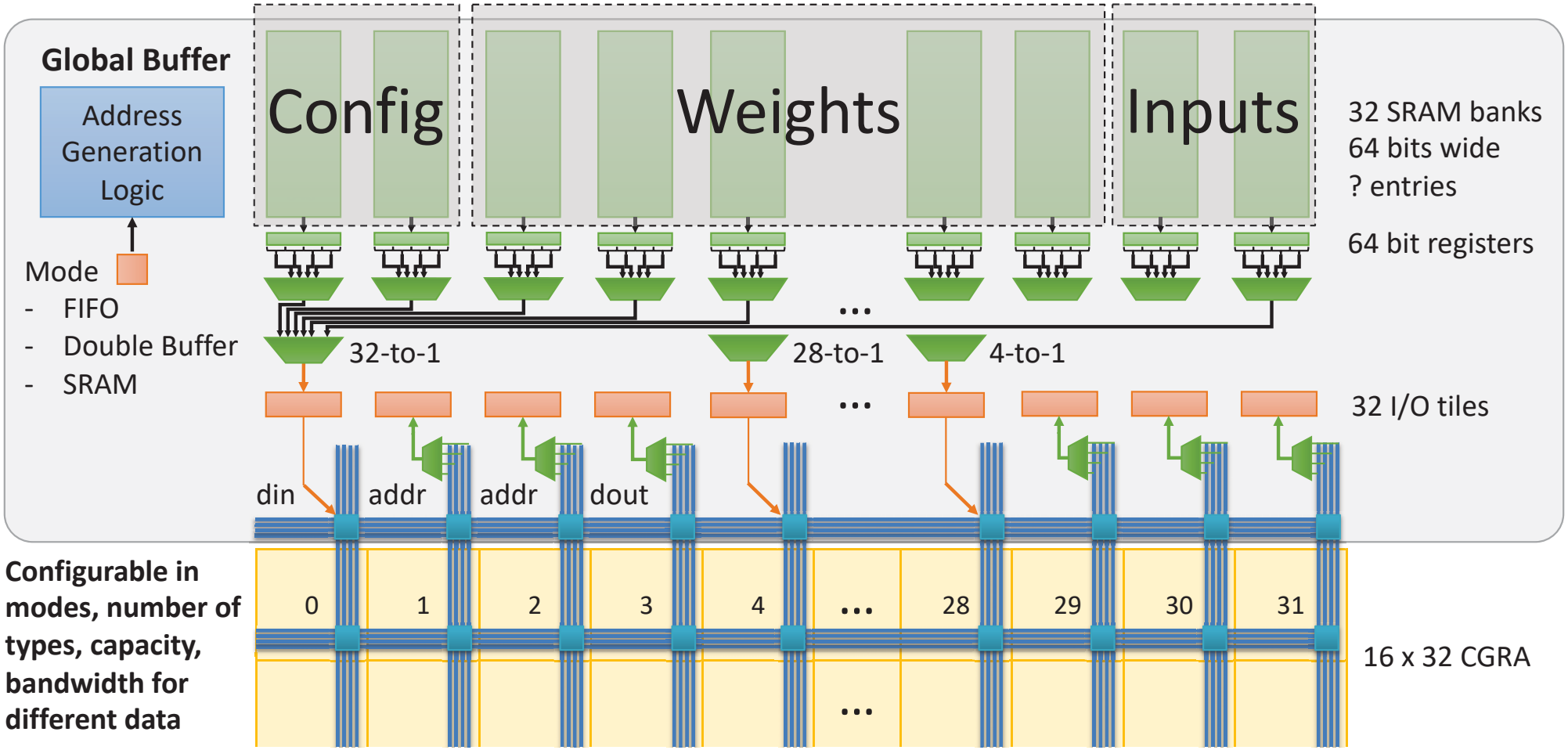
Memory



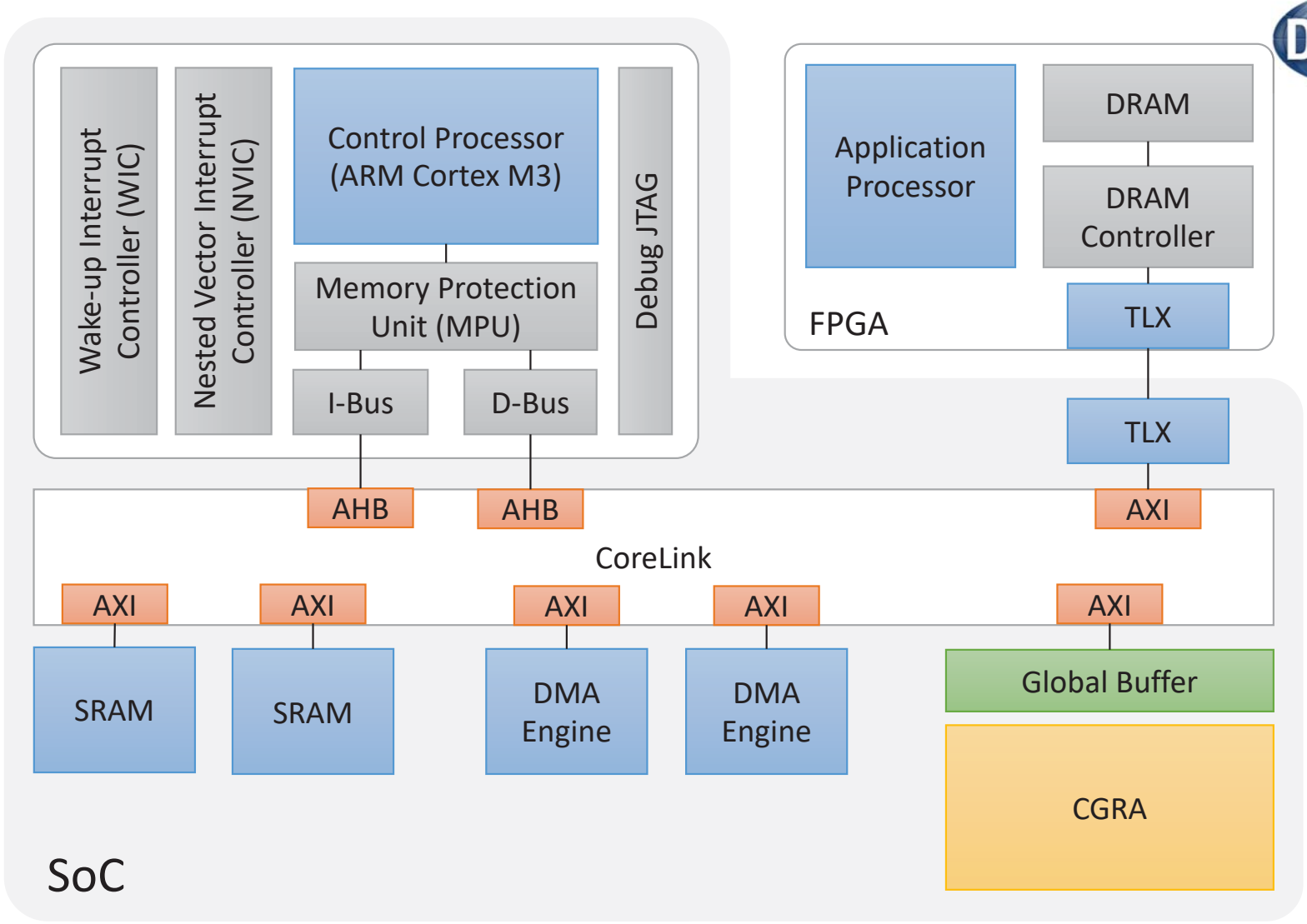
Highly banked and configurable global buffer



Highly banked and configurable global buffer



SoC



Summary of Garnet architectural features

1. Support for Bfloat16, and for executing complex operations like divide using multiple PEs

2. Addition of a global buffer to create a memory hierarchy for efficiently executing neural networks

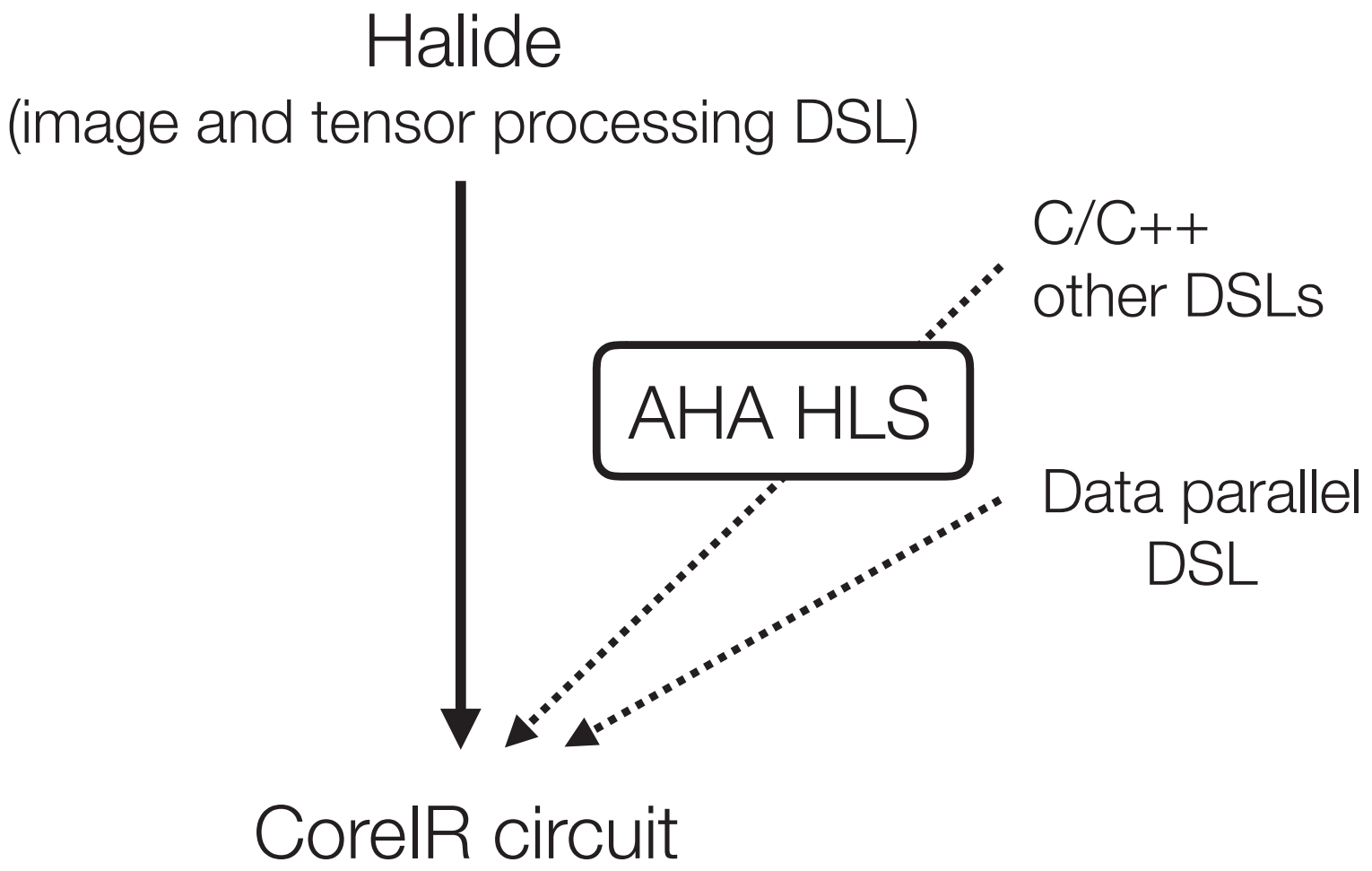
3. Fast reconfiguration support using global buffer and control processor

4. Addition of configurable power domains

Scheduling [Halide] data flow graphs

KAYVON FATAHALIAN

AHA application language(s)



Topics

- Automatically scheduling programs written in the Halide programming language

- Alternative compilation paths:
  - AHA HLS: LLVM to CoreIR
  - Aetherling: data-parallel primitives to CoreIR

- Next talk: Halide-to-CoreIR compiler implementation

Crash course: what is Halide

- Domain-specific language for applications that can be expressed as DAGs of dense tensor computations
- Open source, but industrial strength: in daily production use at Google, Adobe, Instagram, etc.
- Use in industry is primarily for imaging/computational photography applications, some use for deep learning

Example Halide application

- Implementation of key algorithms in photos app in Google Pixel phones

Halide: image processing DSL

[Ragan-Kelley, Adams et al. 2012]

Raised level of abstraction for developing high-performance image processing algorithms

```
blurx(x,y) = (in(x-1,y)    + in(x,y)    + in(x+1,y))    / 3;
out(x,y)   = (blurx(x,y-1) + blurx(x,y) + blurx(x,y+1)) / 3;
```

in

blurx

out

Halide loop nest example


```
blurx(x,y) = (in(x–1, y) + in(x,y) + in(x+1,y)) / 3.0f;
out(x,y)   = (blurx(x,y–1) + blurx(x,y) + blurx(x,y+1)) / 3.0f;
```

```
allocate WIDTHxHEIGHT buffer for blurx
for y=0 to HEIGHT:
   for x=0 to WIDTH:
      blurx(x,y) = …

allocate WIDTHxHEIGHT buffer for out
for y=0 to HEIGHT:
   for x=0 to WIDTH:
      out(x,y) = …
```

Optimized C++ code: 3x3 image blur

Good: ~10x faster on a quad-core CPU than naive code
Bad: specific to SSE (not AVX2), CPU only, code hard to understand…

```
void fast_blur(const Image &in, Image &blurred) {
 __m128i one_third = _mm_set1_epi16(21846);
 #pragma omp parallel for
 for (int yTile = 0; yTile < in.height(); yTile += 32) {
  __m128i a, b, c, sum, avg;
  __m128i tmp[(256/8)*(32+2)];
  for (int xTile = 0; xTile < in.width(); xTile += 256) {
   __m128i *tmpPtr = tmp;
   for (int y = -1; y < 32+1; y++) {
    const uint16_t *inPtr = &(in(xTile, yTile+y));
    for (int x = 0; x < 256; x += 8) {
     a = _mm_loadu_si128((__m128i*)(inPtr-1));
     b = _mm_loadu_si128((__m128i*)(inPtr+1));
     c = _mm_load_si128((__m128i*)(inPtr));
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(tmpPtr++, avg);
     inPtr += 8;
   }}
   tmpPtr = tmp;
   for (int y = 0; y < 32; y++) {
    __m128i *outPtr = (__m128i *)(&(blurred(xTile, yTile+y)));
    for (int x = 0; x < 256; x += 8) {
     a = _mm_load_si128(tmpPtr+(2*256)/8);
     b = _mm_load_si128(tmpPtr+256/8);
     c = _mm_load_si128(tmpPtr++);
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(outPtr++, avg);
}}}}}
```

Multi-core execution (partition image vertically)

Modified iteration order: 256x32 tiled iteration (to maximize cache hit rate)

use of SIMD vector intrinsics

two passes fused into one: tmp data read from cache

Halide is actually two DSLs

Functional algorithm description:

```
blurx(x,y) = (in(x-1,y)    + in(x,y)    + in(x+1,y))    / 3;
out(x,y)   = (blurx(x,y-1) + blurx(x,y) + blurx(x,y+1)) / 3;
```

Schedule: DSL for describing mapping of pipeline stages to a parallel machine

```
output.tile(x, y, xi, yi, 256, 32);

output.vectorize(xi, 8);
output.parallelize(y);

blurx.compute_at(xi);
blurx.vectorize(x, 8);
```

compute output in tiled order

vectorize innermost loop

parallelize loop across cores

loop fusion

vectorize innermost loop

Halide's scheduling DSL is really a mini language for transforming and parallelizing loop nests

Halide loop nest example


```
blurx(x,y) = (in(x–1, y) + in(x,y) + in(x+1,y)) / 3.0f;
out(x,y)   = (blurx(x,y–1) + blurx(x,y) + blurx(x,y+1)) / 3.0f;
```

```
allocate WIDTHxHEIGHT buffer for blurx
for y=0 to HEIGHT:
   for x=0 to WIDTH:
      blurx(x,y) = …

allocate WIDTHxHEIGHT buffer for out
for y=0 to HEIGHT:
   for x=0 to WIDTH:
      out(x,y) = …
```

Halide loop nest example


```
blurx(x,y) = (in(x–1, y) + in(x,y) + in(x+1,y)) / 3.0f;
out(x,y)   = (blurx(x,y–1) + blurx(x,y) + blurx(x,y+1)) / 3.0f;

out.tile(x, y, xi, yi, 256, 32);
```

```
allocate WIDTHxHEIGHT buffer for blurx
for y=0 to HEIGHT:
   for x=0 to WIDTH:
      blurx(x,y) = …

allocate WIDTHxHEIGHT buffer for out
for y=0 to num_tiles_y:
   for x=0 to num_tiles_x:
      for yi=0 to 32:
         for xi=0 to 256:
            idx_x = x*256+xi;
            idx_y = y*32+yi
            out(idx_x, idx_y) = …
```

Halide producer/consumer locality example


```
blurx(x,y) = (in(x-1, y) + in(x,y) + in(x+1,y)) / 3.0f;
out(x,y)   = (blurx(x,y-1) + blurx(x,y) + blurx(x,y+1)) / 3.0f;

out.tile(x, y, xi, yi, 256, 32);
```

```
blurx.compute_at(out, xi);
```

Compute necessary elements of blurx within out's xi loop nest

```
for y=0 to num_tiles_y:
   for x=0 to num_tiles_x:
      for yi=0 to 32:
         for xi=0 to 256:
            idx_x = x*256+xi;
            idx_y = y*32+yi

            allocate 3-element buffer for tmp_blurx

            // compute 3 elements of blurx needed for out(idx_x, idx_y) here
            for (blur_x=0 to 3)
                tmp_blurx(blur_x) = …

            out(idx_x, idx_y) = …
```

Note: Halide compiler performs analysis that the output of each iteration of the xi loop required 3 elements of blurx

Halide producer/consumer locality example


```
blurx(x,y) = (in(x-1, y) + in(x,y) + in(x+1,y)) / 3.0f;
out(x,y)   = (blurx(x,y-1) + blurx(x,y) + blurx(x,y+1)) / 3.0f;

out.tile(x, y, xi, yi, 256, 32);
```

```
blurx.compute_at(out, x);
```

Compute necessary elements of blurx within out's x loop nest (all necessary elements for one tile of out)

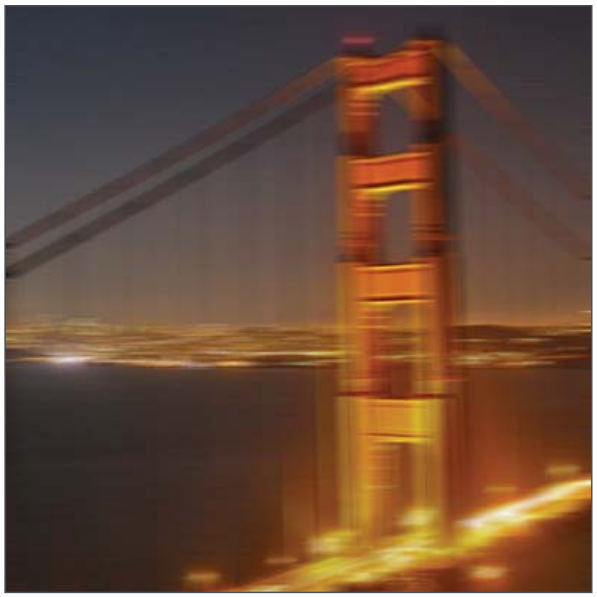
```
for y=0 to num_tiles_y:
   for x=0 to num_tiles_x:

      allocate 258x34 buffer for tile blurx
      for yi=0 to 32+2:
         for xi=0 to 256+2:
            tmp_blurx(xi,yi) = // compute blurx from in

      for yi=0 to 32:
         for xi=0 to 256:
            idx_x = x*256+xi;
            idx_y = y*32+yi
            out(idx_x, idx_y) = …
```

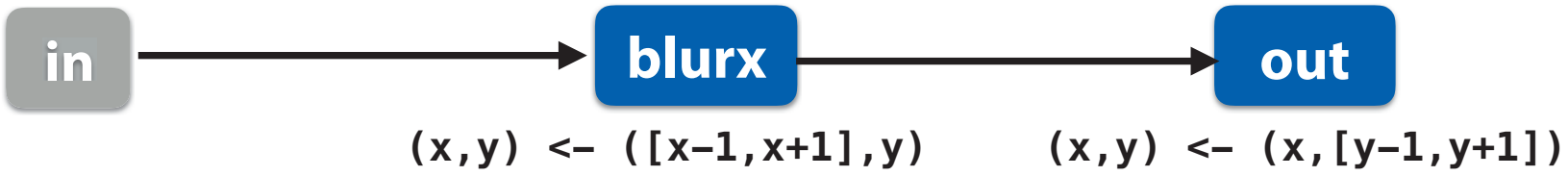
tile of blurx is computed here

tile of blurx is consumed here

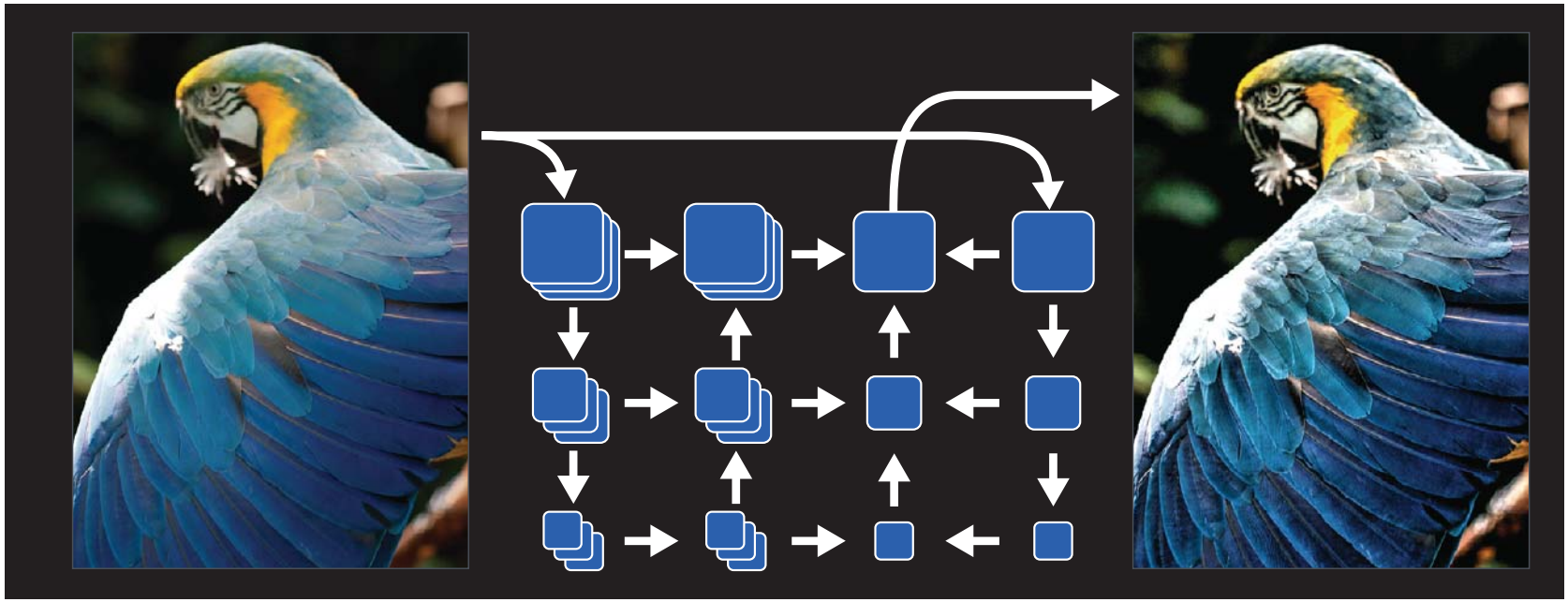
A Halide program is a dataflow graph of operations on tensors

```
blurx(x,y) = (in(x-1,y)    + in(x,y)    + in(x+1,y))    / 3;
out(x,y)   = (blurx(x,y-1) + blurx(x,y) + blurx(x,y+1)) / 3;
```

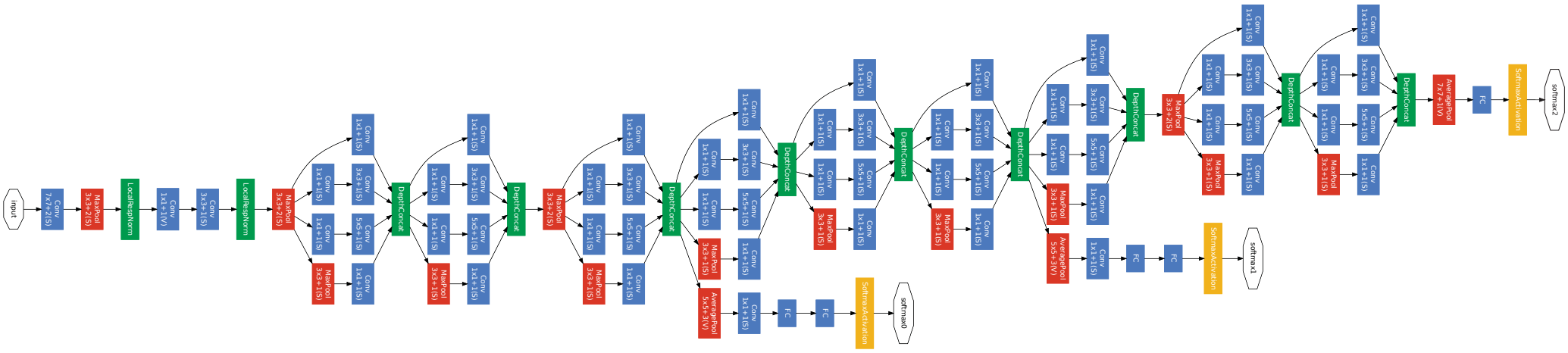



More data flow graphs



100 stages

Local Laplacian filter
[Paris 2010, Aubry 2011]



Inception (GoogleLeNet) — 27 total layers

Real-world image processing pipelines feature complex sequences of functions

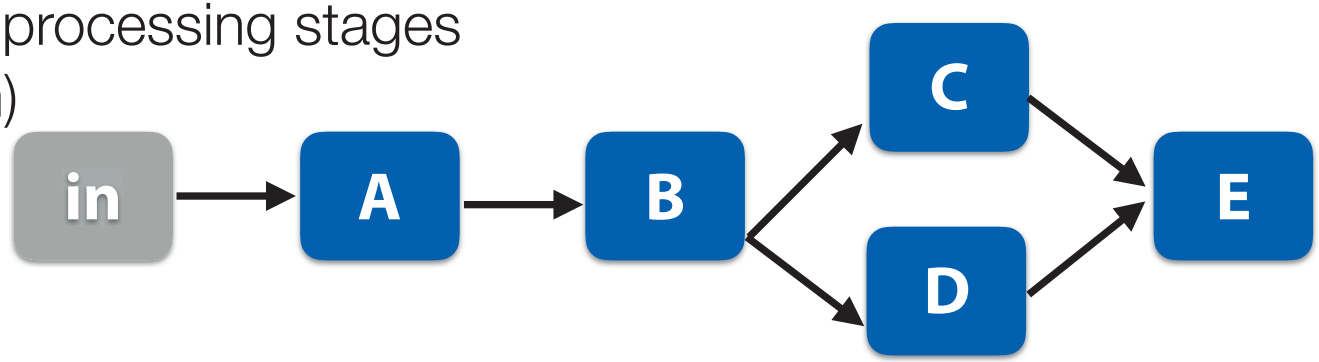
Benchmark	Number of Halide functions
Two-pass blur	2
Unsharp mask	9
Harris Corner detection	13
Camera RAW processing	30
Non-local means denoising	13
Max-brightness filter	9
Multi-scale interpolation	52
Local-laplacian filter	103
Synthetic depth-of-field	74
Bilateral filter	8
Histogram equalization	7
VGG-16 deep network eval	64

Real-world production applications may features hundreds to thousands of functions!
Google HDR+ pipeline: over 2000 Halide functions.

Halide auto scheduling (at the beginning of last year)

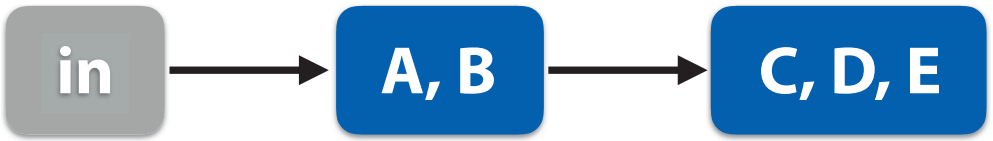
Problem definition

Input: DAG of image processing stages
(from Halide program)



Output: optimized schedule for Halide program

```
for each 8x128 tile in parallel
    compute required pixels of A
    compute pixels in tile of B

for each 8x8 tile in parallel
    compute required pixels of C
    compute required pixels of D
    compute pixels in tile of E
```



Fill parallel machine + make sub-dags “fit” in cache.
Largest tiles possible, subject to constraint the sub-DAG intermediates fit in cache
And subject to constraint that enough tiles exist to fill all cores

The autoscheduler generates schedules that match a single pattern


```
for each tile_y:  // multi-core parallel
   for each tile_x: // multi-core parallel

      allocate tmpC; // buffer for C for tile
      allocate tmpD; // buffer for D for tile

      for each y in required region of C
        for each x in required region of C
          tmpC = ….  // DRAM accesses to B

      for each y in required region of D
        for each x in required region of D
          tmpD = …. // DRAM accessed to B

    for each y in tile:
      for each x in tile:

          x' = tile_x * TILE_WIDTH + x;
          y' = tile_y * TILE_HEIGHT + y;
          E(x',y') = …
```

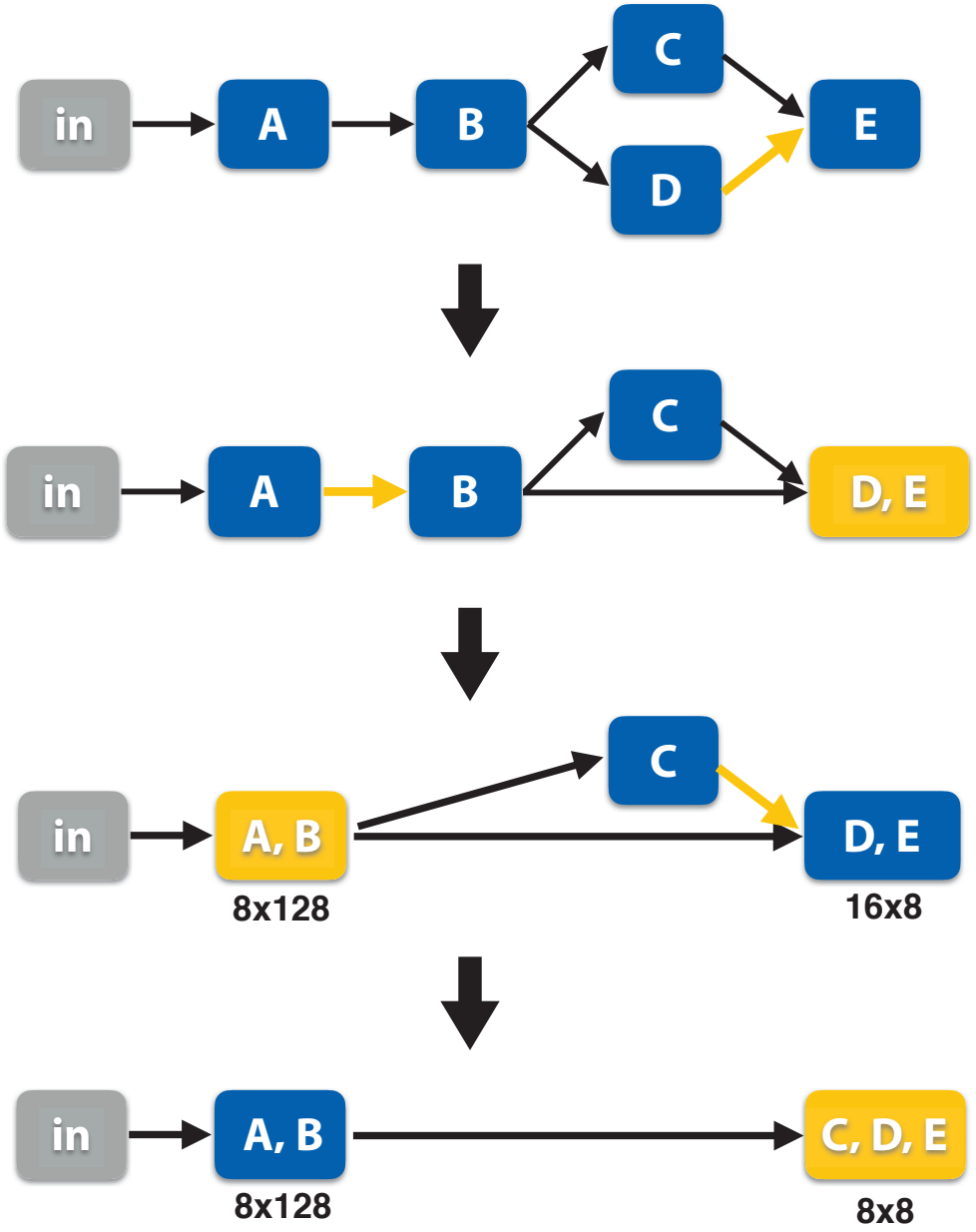
"overlap tiling":

1. compute output in tiles
2. compute and store necessary input values at tile loop

Greedy grouping strategy

Greedily form groups (take best merge)

But re-evaluate local optimization to determine tile size each step.



for each 8x128 tile
compute required pixels of A
compute required pixels of B
compute pixels in tile of D

for each 8x8 tile
compute required pixels of C
compute pixels in tile of E

Another important pattern ("line buffering")


```
for each tile_y:  // multi-core parallel
   for each tile_x: // multi-core parallel

      allocate tmpC; // sliding window buffer for C
      allocate tmpD; // sliding window buffer for D

     for each y in tile:

       for each x in row:
          tmpC = ….  // compute new line of C, add to sliding window buffer
       for each x in row:
          tmpD = ….  // compute new line of D, add to sliding window buffer

      for each x in tile:
           x' = tile_x * TILE_WIDTH + x;
           y' = tile_y * TILE_HEIGHT + y;
           out(x',y') = …  // accesses end of C and D buffers
```

Improved auto scheduler goals

- Goal: generate higher performance code
  - Search a larger space of schedules
  - Be extensible to support new Halide scheduling primitives (e.g., sliding window, async, etc) and future primitives

- Cooperation between researchers at Facebook/UCB/MIT/Stanford [Adams et al 2019]

Two ideas

- Adopt general search algorithm (not specific to any patterns or Halide primitives)
  - Use variant of backtracking beam search

- Use machine learning to learn an accurate cost model for a target architecture (to intelligently guide search)

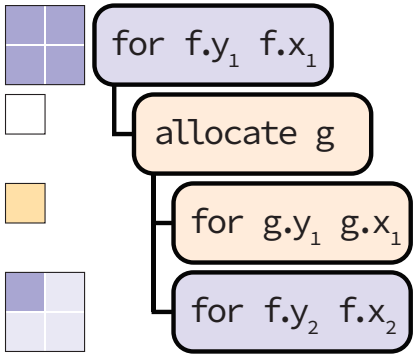
Scheduling as a sequence of choices

Model DAG scheduling as a sequence of choices that build a loop nest, where scheduling proceeds "from end to start" (output to input) in the DAG

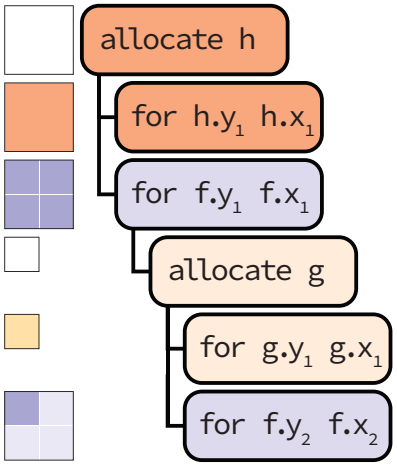
Code:

```
h(x, y) = ...;
g(x, y) = pow(h(x, y), 1.8);
f(x, y) = g(x, y−1) + g(x, y+1);
```

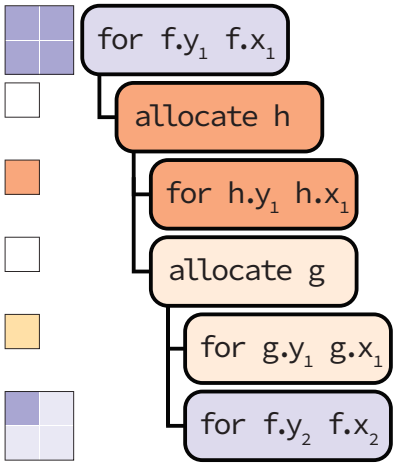
Current state:



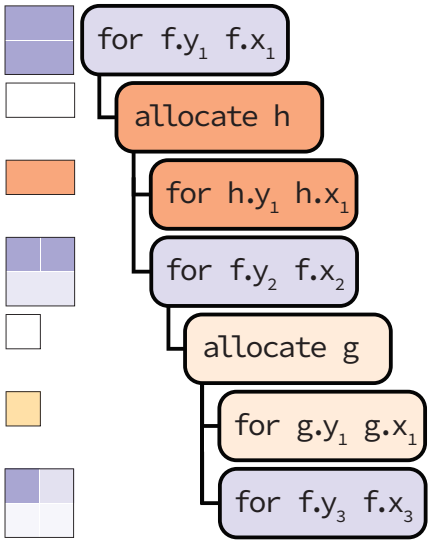
Successor states



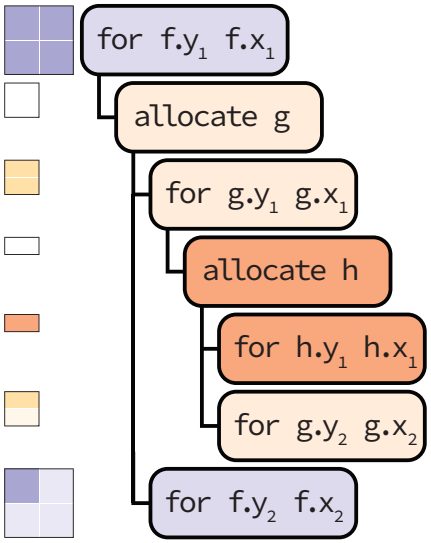
1) compute h at root



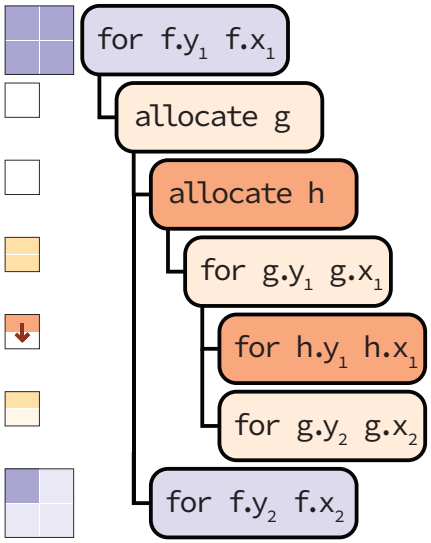
2) compute h at an existing tiling



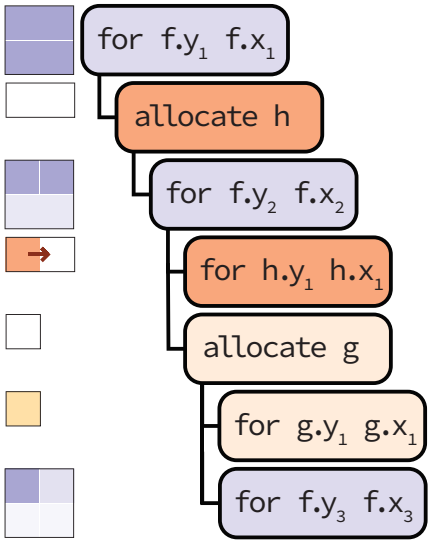
3) compute h at a new outer tiling of f



4) compute h at a new sub-tiling of g



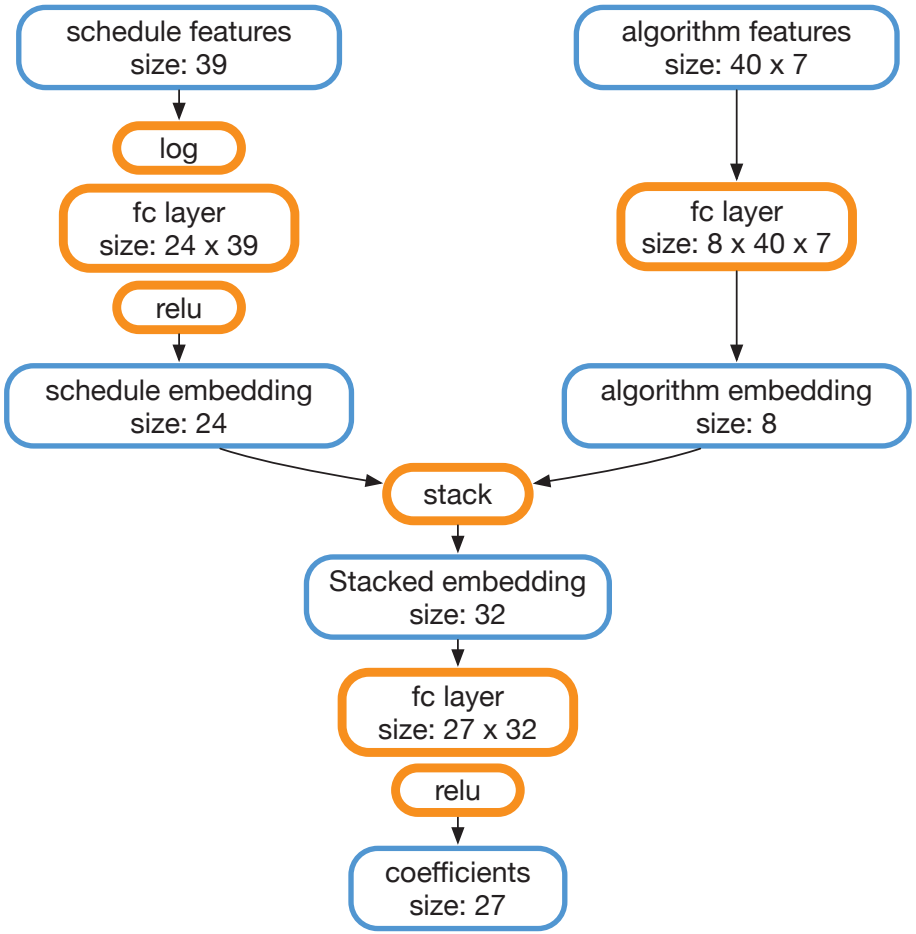
5) store h at tiles of f,
compute h at a new sub-tiling of g



6) store h at a new outer tiling of f,
compute h at sub-tiles of f

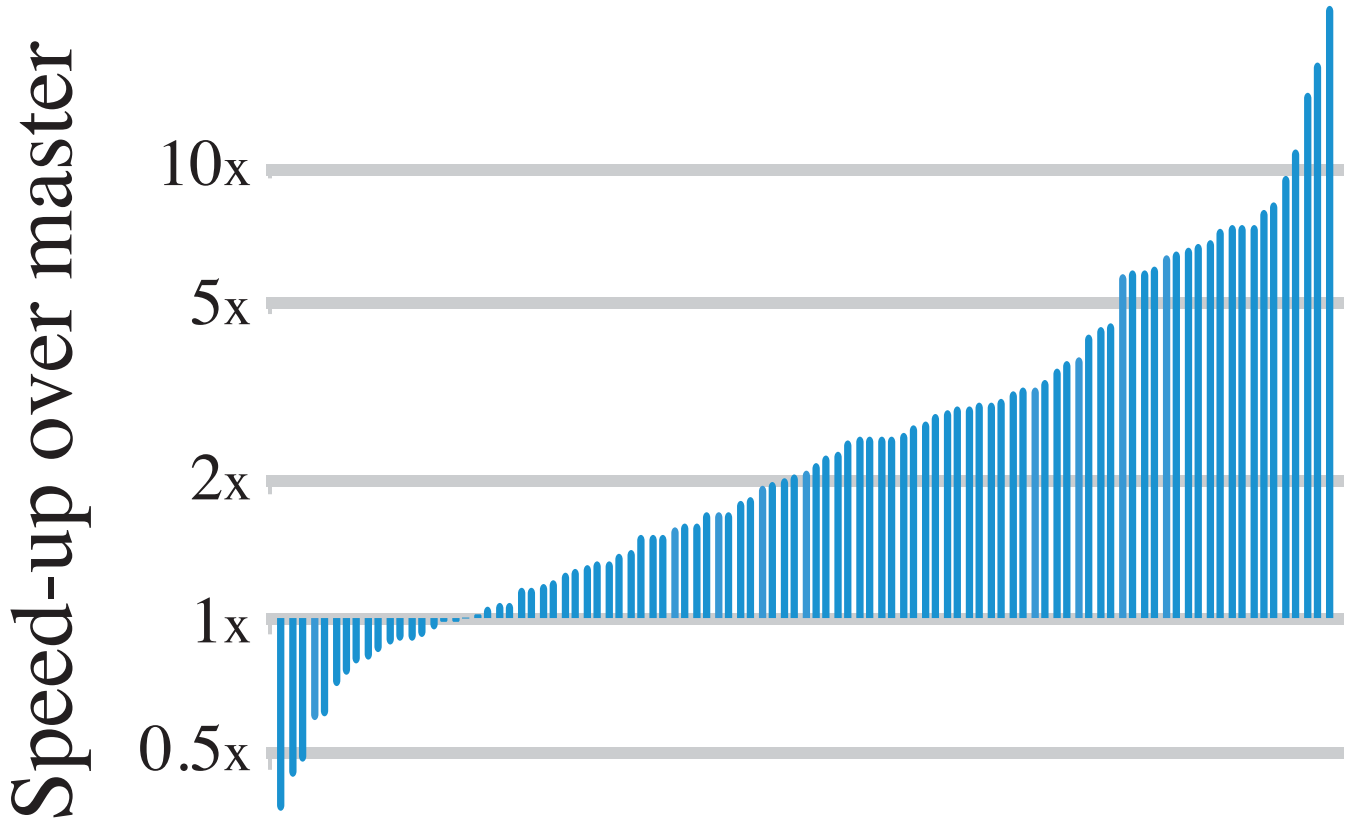
Guide search with learned cost model

- Randomly generate Halide programs as a training set
- Featurize each DAG stage
  - Based on data access pattern (pointwise, stencil, broadcast, etc.)
  - Amount of data allocated, arithmetic intensity, etc.
- Use DNN to predict 27 parameters of hand-designed cost model from features



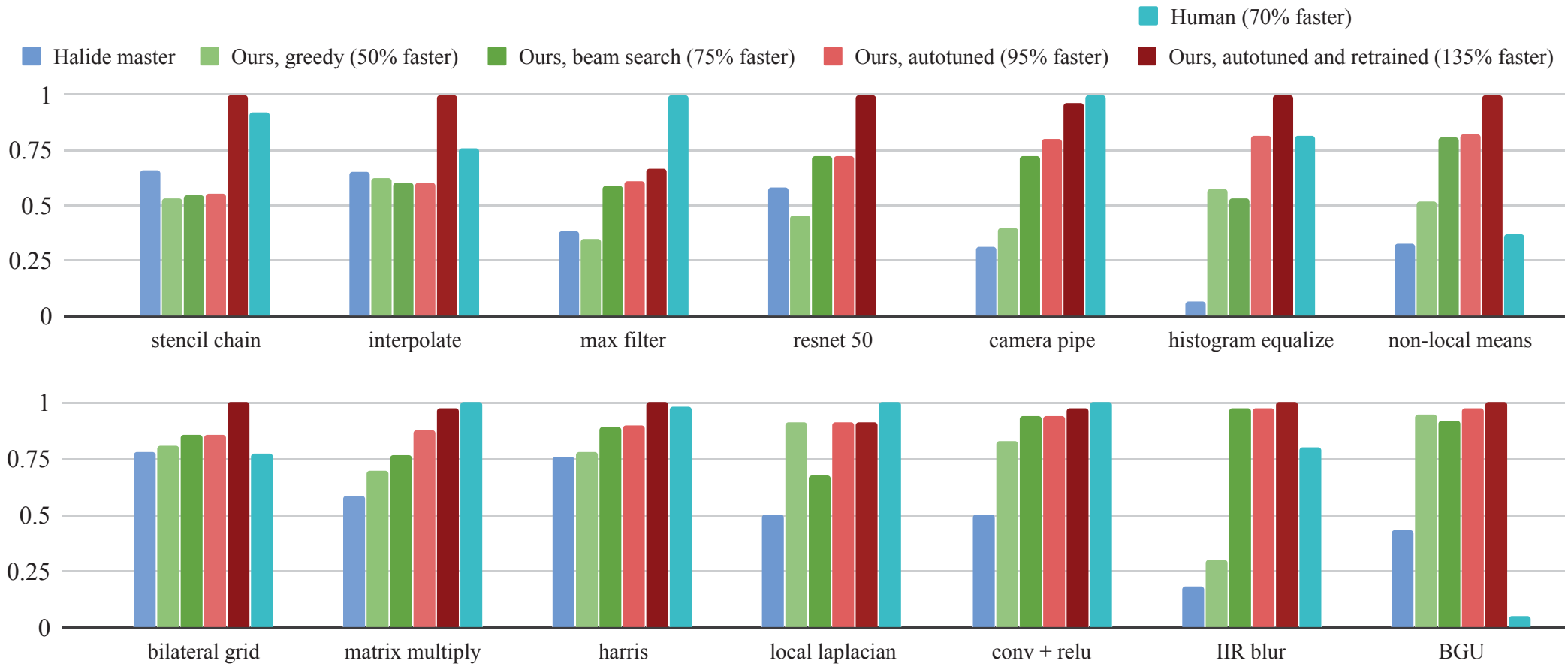
Results (on random programs)

Overall: 2.29X geo mean speedup vs. Halide master



target machine: Core i9 1760X (32 hyperthread CPU)

Application results



target machine: Core i9 1760X (32 hyperthread CPU)

(Have also targeted ARM CPUs and CUDA GPUs.)

Summary

- Resign of auto-scheduler sothat:
  - Optimization (search) process is agnostic to scheduling primitives available for use
  - Leverage ML to learn target-specific cost models
- In general: “near” expert-human performance
- Running on real hardware:
  - Intel CPUs, ARM CPUs, NV GPUs

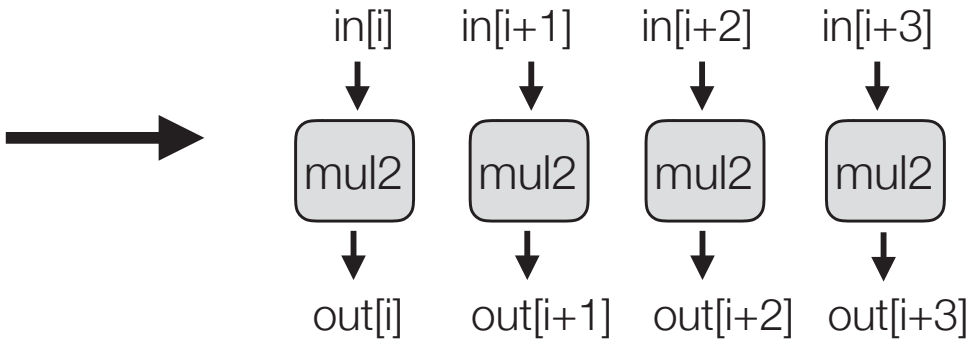
Halide-to-CoreIR backend

- Reinterpret Halide's CPU-centric scheduling directives as hardware units

- Examples:

  1. unrolling loop = generating multiple hardware units

```
out(x) = 2 * in(x)
out.unroll(4)
```



  2. specific store_at/compute_at patterns
     = emit line buffer memory units

Supporting additional input languages

- AHA HLS: high-level synthesis backend for LLVM to CoreIR
- Aetherling: scheduling functional data parallel primitives to CoreIR circuits

AHA HLS: motivation

- Domain specific languages are great (for the applications they are intended for

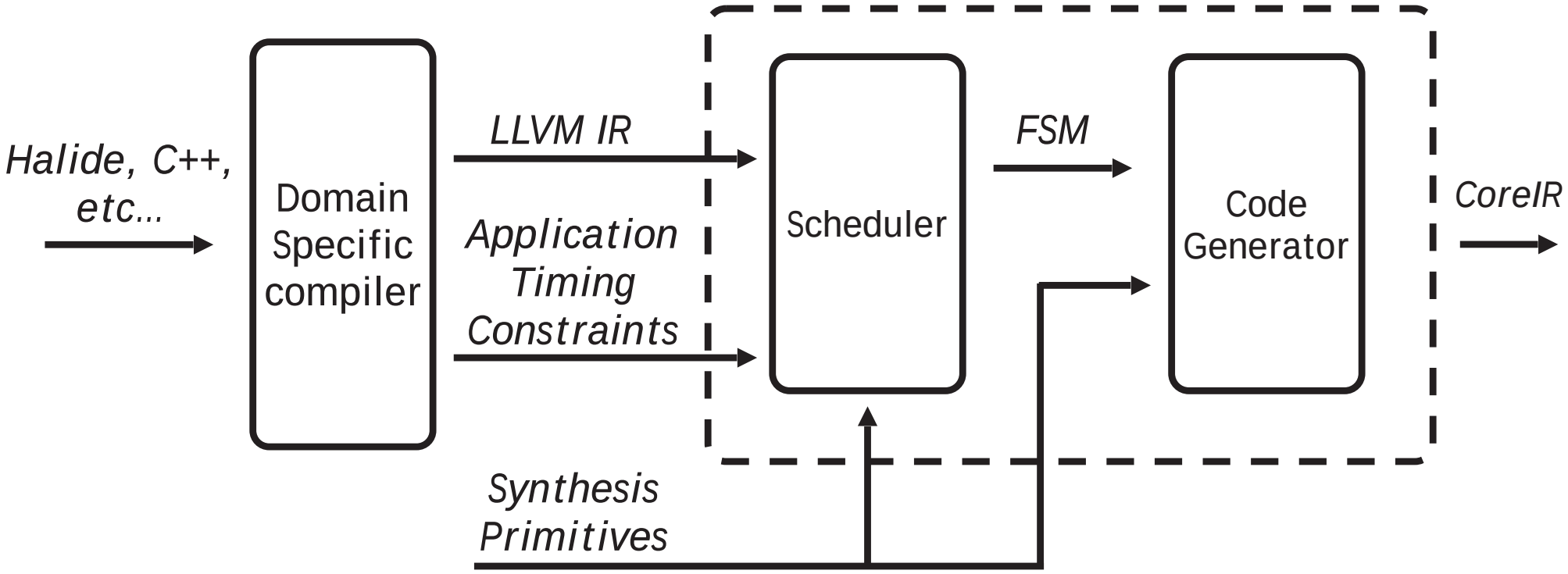
- Many hardware synthesis optimizations are domain agnostic, so prefer to not re-implement them for every input DSL

- Ideal: Halide —> HLS —> RTL
  - Problem: no existing HLS tool supports CGRA/ custom HW architectures

Design goals for AHA HLS tool

- Re-use LLVM
- Provide domain agnostic hardware optimizations
- Allow user defined synthesis back-ends (no existing HLS for CGRAs)
- Scheduling API rather than pragmas and directives

Compiling applications



Status

- Can map simple applications to the CGRA (multiply an image by two)
- Pipelining working for inner loops
- Linebuffers and more complicated primitives are coming

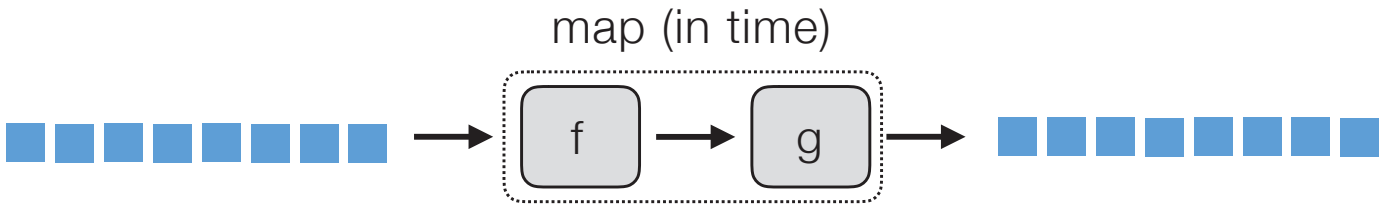
Halide code as data parallel operations on sequences


```
out(x,y) = 2 * in(x,y)
     let f x = 2*x
     map f

out(x,y) = in(x,y-1) + in(x,y) + in(x,y+1)
     map (reduce 3 +) . stencil(1,3)

out(x,y) = in(x/2, y/2)
     upsample(2,2)

out(x,y) = in(2x, 2x)
     downsample(2,2)

out(x,y) = in(2x, 2 * lookup(x,y))
     downsample in x, gather in y
```

Scheduling data-parallel programs to hardware pipelines

```
let N = 32

let f x = 2x              :: int -> int
let g x = x+1             :: int -> int

let in = // sequence of size N
let tmp1 = (map N f) tmp2          // tmp1 is sequence of len N
let out =  (map N g) tmp1          // out  is sequence of len N
```

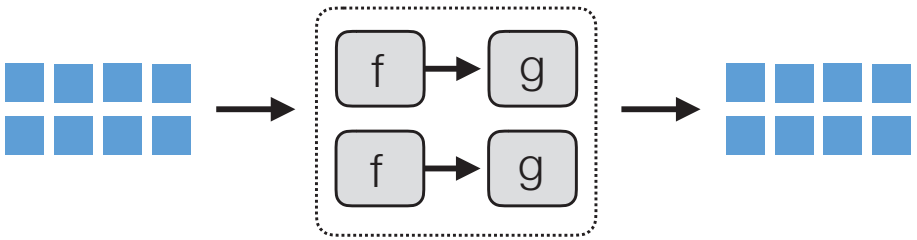
```
let fog x = g . f
let out = (map N fog) in
```



Scheduling data-parallel programs to hardware pipelines


```
let N = 32

let f x = 2x              :: int -> int
let g x = x+1             :: int -> int

let in = // sequence of size N
let tmp1 = (map N f) tmp2          // tmp1 is sequence of len N
let out =  (map N g) tmp1          // out  is sequence of len N
```

```
let fog x = g . f              :: int -> int
let par_fog = map 2 fog        :: int2 -> int2

let tmp1 = partition 2 in     // create sequence of len N/2
let tmp2 = (map N/2 par_fog) tmp1
let out  = unpartition 2 tmp2
```



Aetherling

1. Begin with a simple, data-parallel language on sequences

2. Transform programs in this language into “space-time” programs that have a streaming hardware interpretations

3. Autoschedule space-time programs: manipulate space-time programs to adjust their throughput

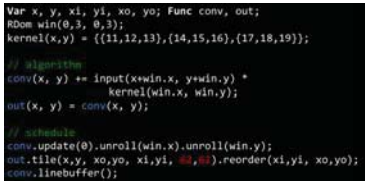
Summary

- Three paths from source to AHA toolchain
  - **Halide to CoreIR via new Halide compiler backend**
  - LLVM to CoreIR via HLS
  - Data-parallel primitives to CoreIR via Aetherling

Halide to Hardware

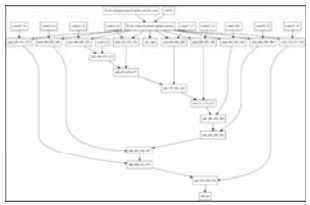
Jeff Setter

Rough overview of CGRA application flow



Halide

Compiler

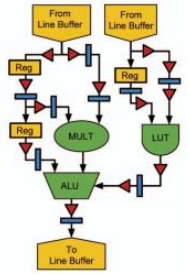


CoreIR

Tech Mapper

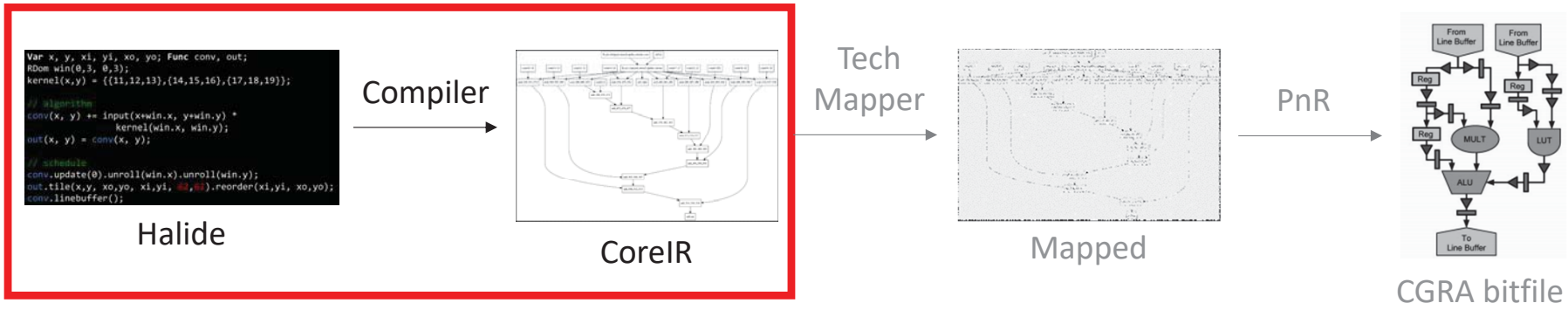
Mapped

PnR



CGRA bitfile

Rough overview of CGRA application flow



Halide Frontend

Algorithm

```
Var x, y, xi, yi, xo, yo;
ImageParam input;
Func conv, output;
RDom win(0, 3, 0, 3);

conv(x, y)  = 0.0;
conv(x, y) += input(x + win.x, y + win.y) *
              weights(win.x, win.y);
output(x, y) = conv(x, y);
```

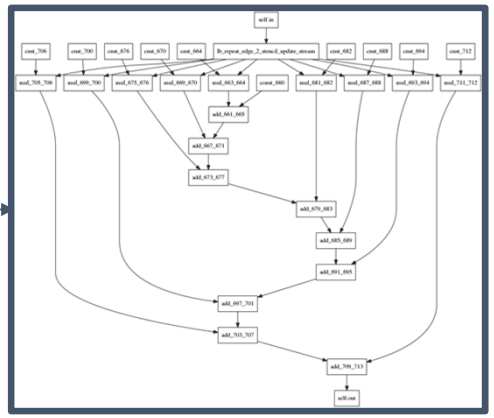
Schedule

```
output.tile(x, y, xo, yo, xi, yi, 64, 64);
output.hw_accelerate(xo, xi);
conv.compute_at(output, xi)
conv.update()
    .unroll(win.x)
    .unroll(win.y);

input.in()
     .store_at(output, xo)
     .compute_at(output, xi)
     .stream_to_accelerator();
```

Halide IR loop nest

```
GENERATED_HARDWARE(xo, yo):
   StorageUnitToBeSpecified weights;
   Register<int> tmp_conv;
   LineBuffer<size=2x64, stencil=3x3, thruput=1>
         tmp_input;

   for output.yi = 0 to 64:
     for output.xi = 0 to 64:

       Load one new pixel into tmp_input

       tmp_conv = tmp_input[3x3 stencil] * weights

       // streaming store to memory
       STORE tmp_conv to output(xo*64+xi, yo*64+yi)
```

CoreIR circuit



Halide Example: 3x3 convolution

```
// Algorithm
RDom win(0, 3, 0, 3);
conv(x, y)  = 0.0;
conv(x, y) += input(x + win.x, y + win.y) *
              weights(win.x, win.y);
output(x, y) = conv(x, y);

// Schedule
output.tile(x, y, xo, yo, xi, yi, 64, 64);
output.hw_accelerate(xo, xi);
input.in()
     .stream_to_accelerator();
     .store_at(output, xo)
     .compute_at(output, xi)

conv.update()
    .unroll(win.x)
    .unroll(win.y);
```

Defines a 3x3 convolution using input and weights.

Halide Example: 3x3 convolution

```
// Algorithm
RDom win(0, 3, 0, 3);
conv(x, y)  = 0.0;
conv(x, y) += input(x + win.x, y + win.y) *
              weights(win.x, win.y);
output(x, y) = conv(x, y);

// Schedule
output.tile(x, y, xo, yo, xi, yi, 64, 64);
output.hw_accelerate(xo, xi);
input.in()
     .stream_to_accelerator();
     .store_at(output, xo)
     .compute_at(output, xi)

conv.update()
    .unroll(win.x)
    .unroll(win.y);
```

Defines a 3x3 convolution using input and weights.

Creates an accelerator from input to output operating on 64x64 image tiles.

Halide Example: 3x3 convolution

```
// Algorithm
RDom win(0, 3, 0, 3);
conv(x, y)  = 0.0;
conv(x, y) += input(x + win.x, y + win.y) *
              weights(win.x, win.y);
output(x, y) = conv(x, y);

// Schedule
output.tile(x, y, xo, yo, xi, yi, 64, 64);
output.hw_accelerate(xo, xi);
input.in()
     .stream_to_accelerator();
     .store_at(output, xo)
     .compute_at(output, xi)

conv.update()
    .unroll(win.x)
    .unroll(win.y);
```

Defines a 3x3 convolution using input and weights.

Creates an accelerator from input to output operating on 64x64 image tiles.

Specifies that input should be stored in a line buffer.

Halide Example: 3x3 convolution

```
// Algorithm
RDom win(0, 3, 0, 3);
conv(x, y)  = 0.0;
conv(x, y) += input(x + win.x, y + win.y) *
              weights(win.x, win.y);
output(x, y) = conv(x, y);

// Schedule
output.tile(x, y, xo, yo, xi, yi, 64, 64);
output.hw_accelerate(xo, xi);
input.in()
     .stream_to_accelerator();
     .store_at(output, xo)
     .compute_at(output, xi)

conv.update()
    .unroll(win.x)
    .unroll(win.y);
```

Defines a 3x3 convolution using input and weights.

Creates an accelerator from input to output operating on 64x64 image tiles.

Specifies that input should be stored in a line buffer.

Unrolls the implicit RDom FOR loops; thus 9 multipliers are created.

Halide Compiler

Algorithm

```
Var x, y, xi, yi, xo, yo;
ImageParam input;
Func conv, output;
RDom win(0, 3, 0, 3);

conv(x, y)  = 0.0;
conv(x, y) += input(x + win.x, y + win.y) *
              weights(win.x, win.y);
output(x, y) = conv(x, y);
```

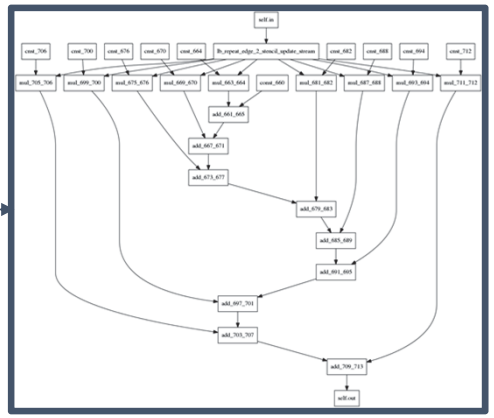
Schedule

```
output.tile(x, y, xo, yo, xi, yi, 64, 64);
output.hw_accelerate(xo, xi);
conv.compute_at(output, xi)
conv.update()
    .unroll(win.x)
    .unroll(win.y);

input.in()
     .store_at(output, xo)
     .compute_at(output, xi)
     .stream_to_accelerator();
```

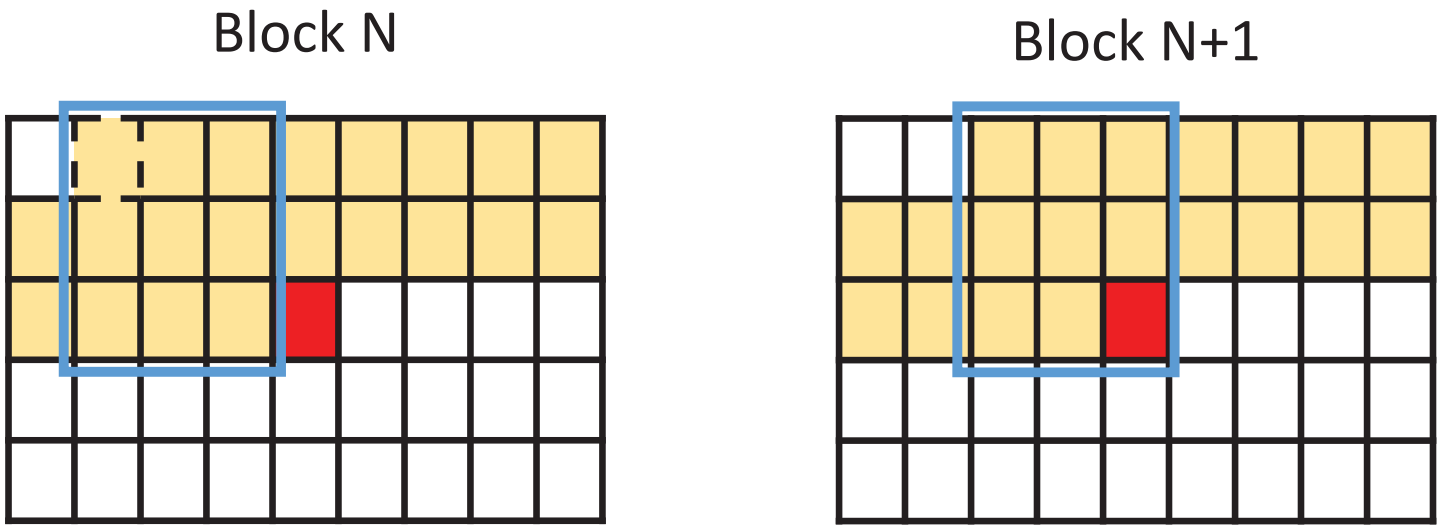
Halide IR loop nest

```
GENERATED_HARDWARE(xo, yo):
   StorageUnitToBeSpecified weights;
   Register<int> tmp_conv;
   LineBuffer<size=2x64, stencil=3x3, thruput=1>
         tmp_input;

   for output.yi = 0 to 64:
     for output.xi = 0 to 64:

       Load one new pixel into tmp_input

       tmp_conv = tmp_input[3x3 stencil] * weights

       // streaming store to memory
       STORE tmp_conv to output(xo*64+xi, yo*64+yi)
```

CoreIR circuit

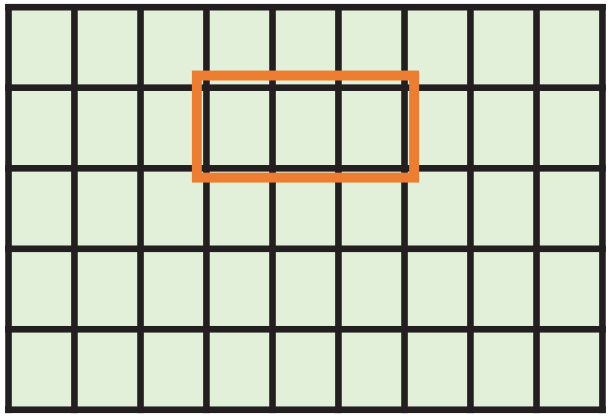


Line buffer: key property is overlap

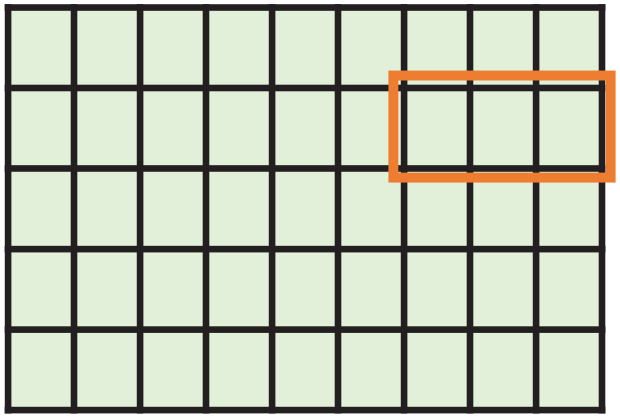


Double buffer: access buffer over multiple cycles

Block N, cycle 5

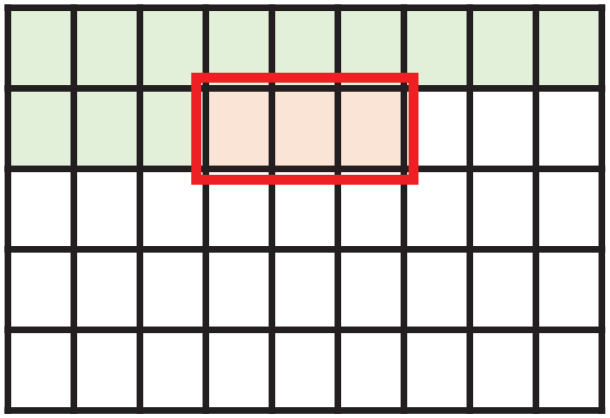


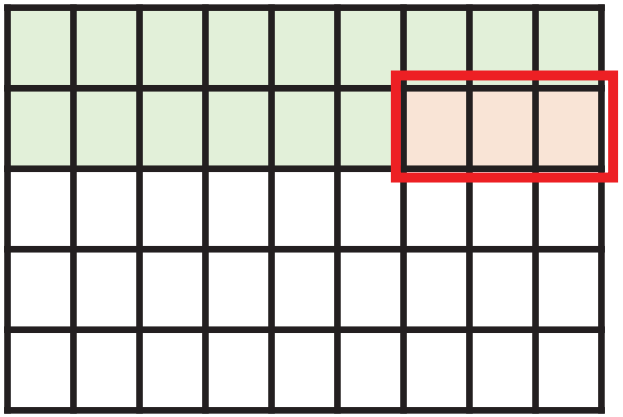
Block N, cycle 6



Double buffer: 1 buffer to read, 1 buffer to write

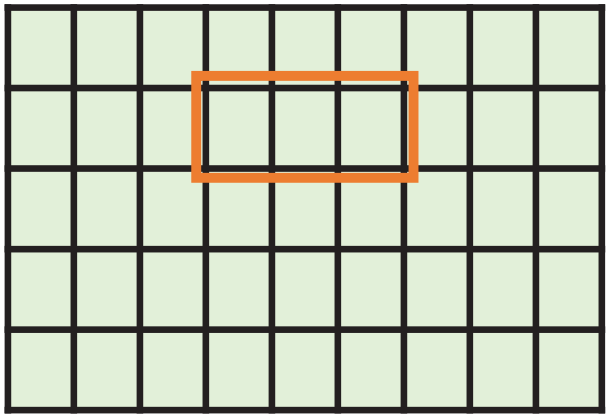
Write
Block N+1
Cycle 5

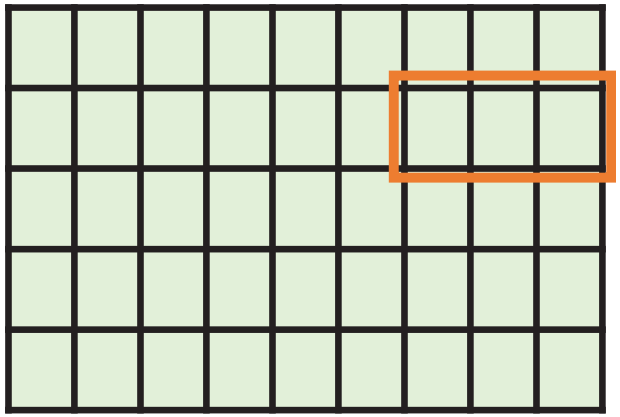




Write
Block N+1
Cycle 6

Read
Block N
Cycle 5

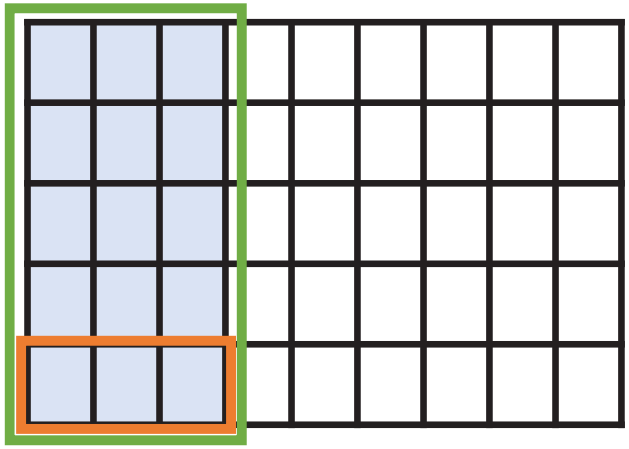




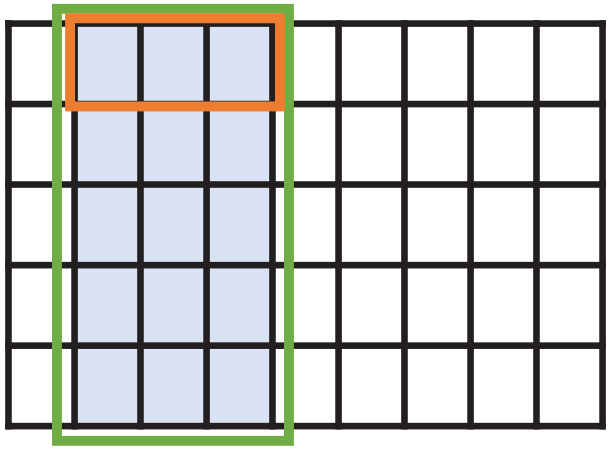
Read
Block N
Cycle 6

Unified hardware buffer: combining line buffer and double buffer properties

Block N, cycle 5



Block N+1, cycle 1



Overlapping output stencils +
multiple cycles to execute each iteration

IR modifications specific to hardware

- Memory parameter extraction
  - Output stencil size: calculate working set at loop level
  - Input stencil size: difference in working sets between two iterations
- Modify IR based on hardware execution
  - Insert buffers based on memory extraction parameters
  - Record how memory elements connect to track valid data
  - Convert data processing based on streaming stencils

CoreIR Codegen

Algorithm

```
Var x, y, xi, yi, xo, yo;
ImageParam input;
Func conv, output;
RDom win(0, 3, 0, 3);

conv(x, y)  = 0.0;
conv(x, y) += input(x + win.x, y + win.y) *
              weights(win.x, win.y);
output(x, y) = conv(x, y);
```

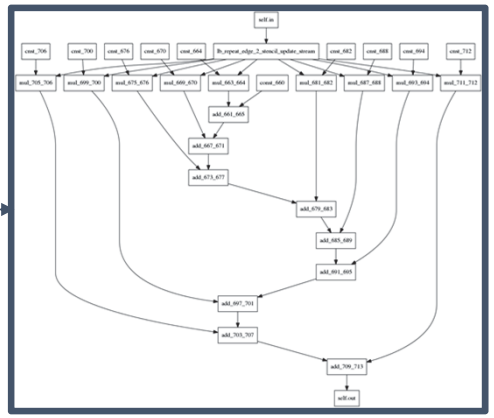
Schedule

```
output.tile(x, y, xo, yo, xi, yi, 64, 64);
output.hw_accelerate(xo, xi);
conv.compute_at(output, xi)
conv.update()
    .unroll(win.x)
    .unroll(win.y);

input.in()
     .store_at(output, xo)
     .compute_at(output, xi)
     .stream_to_accelerator();
```

Halide IR loop nest

```
GENERATED_HARDWARE(xo, yo):
   StorageUnitToBeSpecified weights;
   Register<int> tmp_conv;
   LineBuffer<size=2x64, stencil=3x3, thruput=1>
         tmp_input;

   for output.yi = 0 to 64:
     for output.xi = 0 to 64:

       Load one new pixel into tmp_input

       tmp_conv = tmp_input[3x3 stencil] * weights

       // streaming store to memory
       STORE tmp_conv to output(xo*64+xi, yo*64+yi)
```

CoreIR circuit

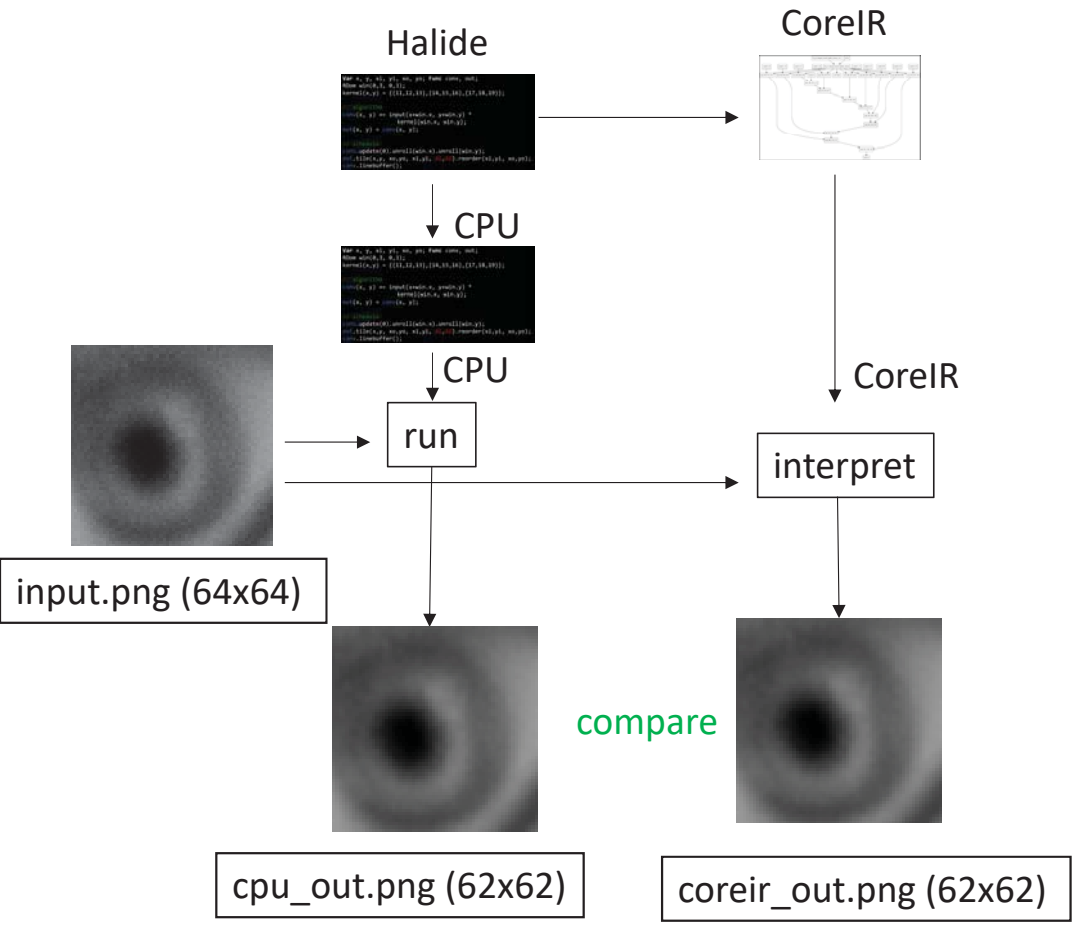


Codegen: produce hardware output files

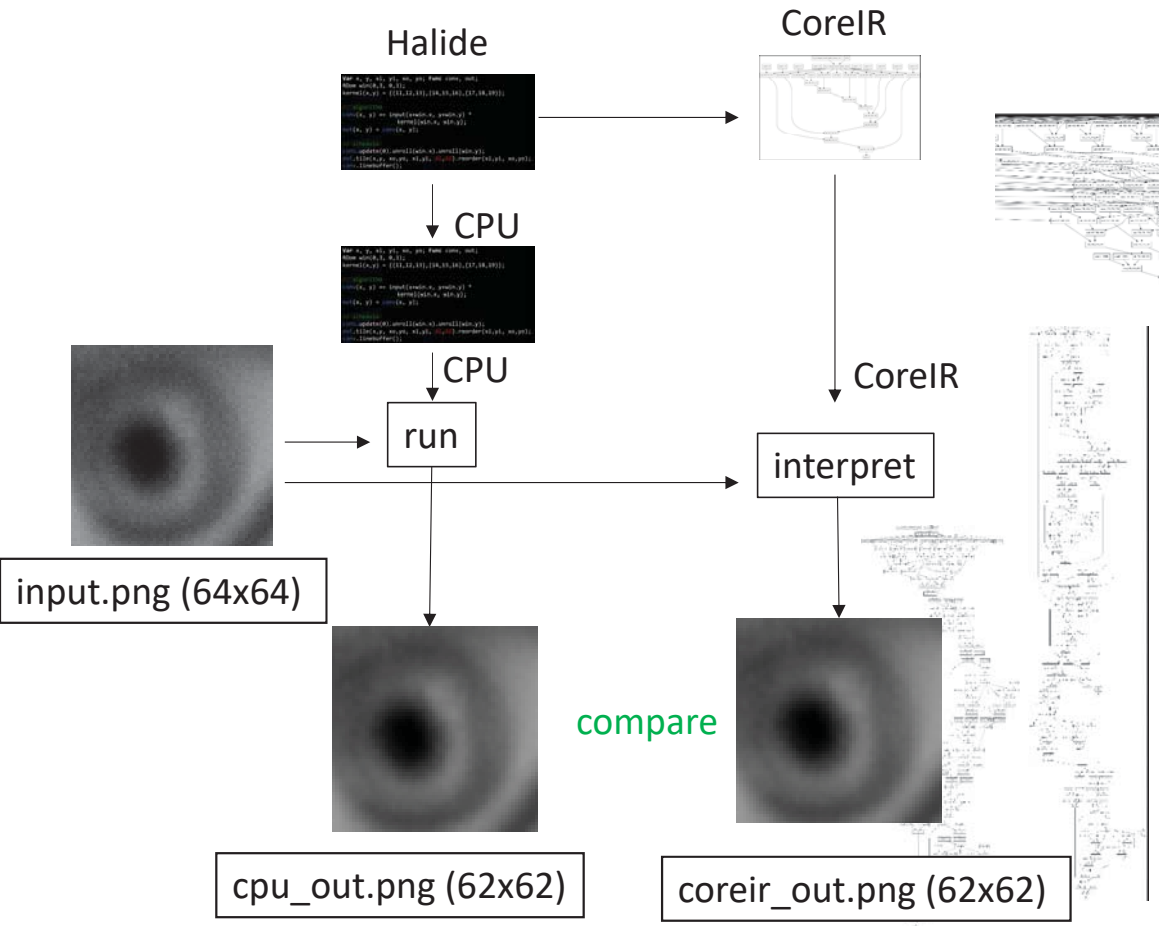
- For each operator, find corresponding set of operators in CoreIR
- More complex operators (counter, line buffer, etc.) use a generator

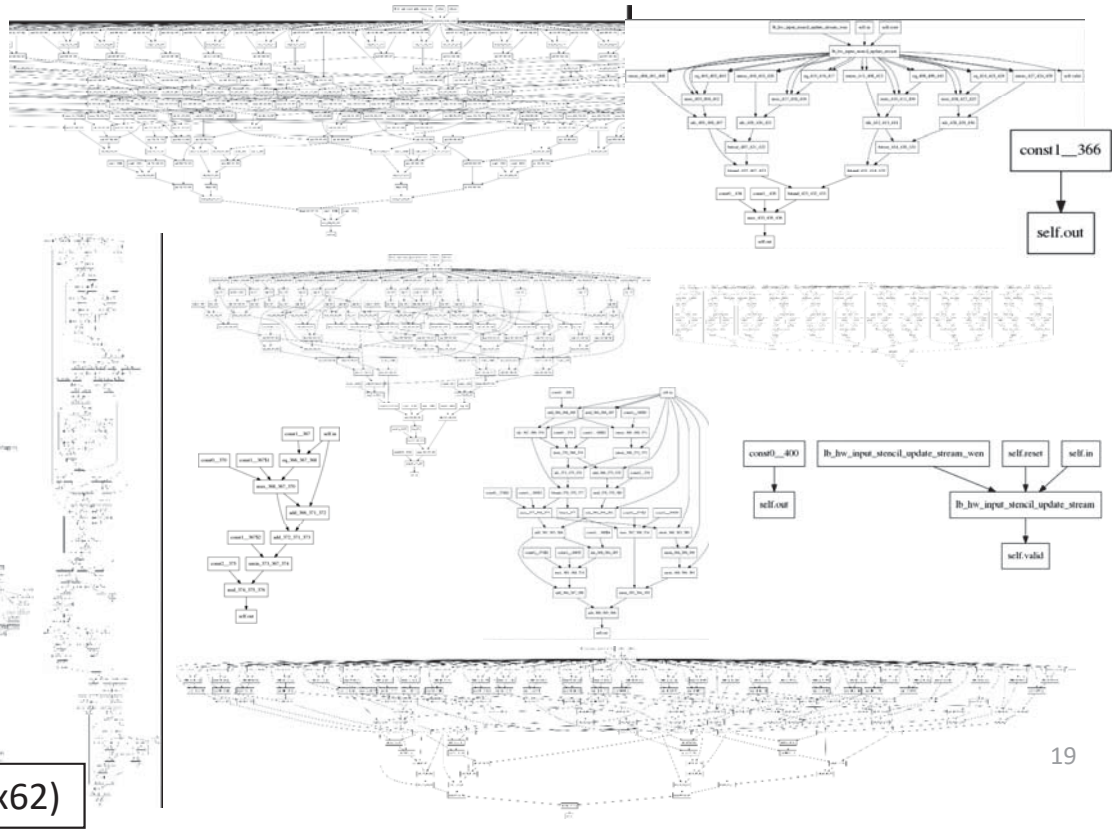
	Halide representation	CoreIR instances
Input / Output	InputParam, Param	def.input, const (set of configuration)
	const	const
Algorithm functions	* / + - %	mul, ashr, add, sub, and
	!= == < <= > >=	neq, eq, {u,s}lt, {u,s}le, {u,s}gt, {u,s}ge
	&& || !	and, or, not
	& | ~ ^ >> <<	and, or, not, xor, ashr, shl
	select max min	mux, {u,s}max, {u,s}min
	absd, * +	absd, mad
Floating Point	*[float]* * / + - %	fmul, fdiv, fadd, fsub, frem
	[float] != == < <= > >=	fneq, feq, flt, fle, fgt, fge
	select min max floor ceil	fmux, fmin, fmax, fflr, fceil
	log exp pow sqrt	log, exp, pow, sqr
	sin cos tan asin acos atan2	sin, cos, tan, asin, acos, atan
Control flow	for, if	counter, *enable wire*
	var load linebuffer stencil, *var load array*	*input* => muxn, *const* => muxn
Schedule primitives	accelerate	Create circuit between input and output
	linebuffer = comp+store_at	Create linebuffer (memories and registers)
	RDom	Define stencil input size for linebuffer
	unroll	Duplicate algorithm operators by amount. Can be used to remove counters / var load.

Testing: image comparison and random pipelines

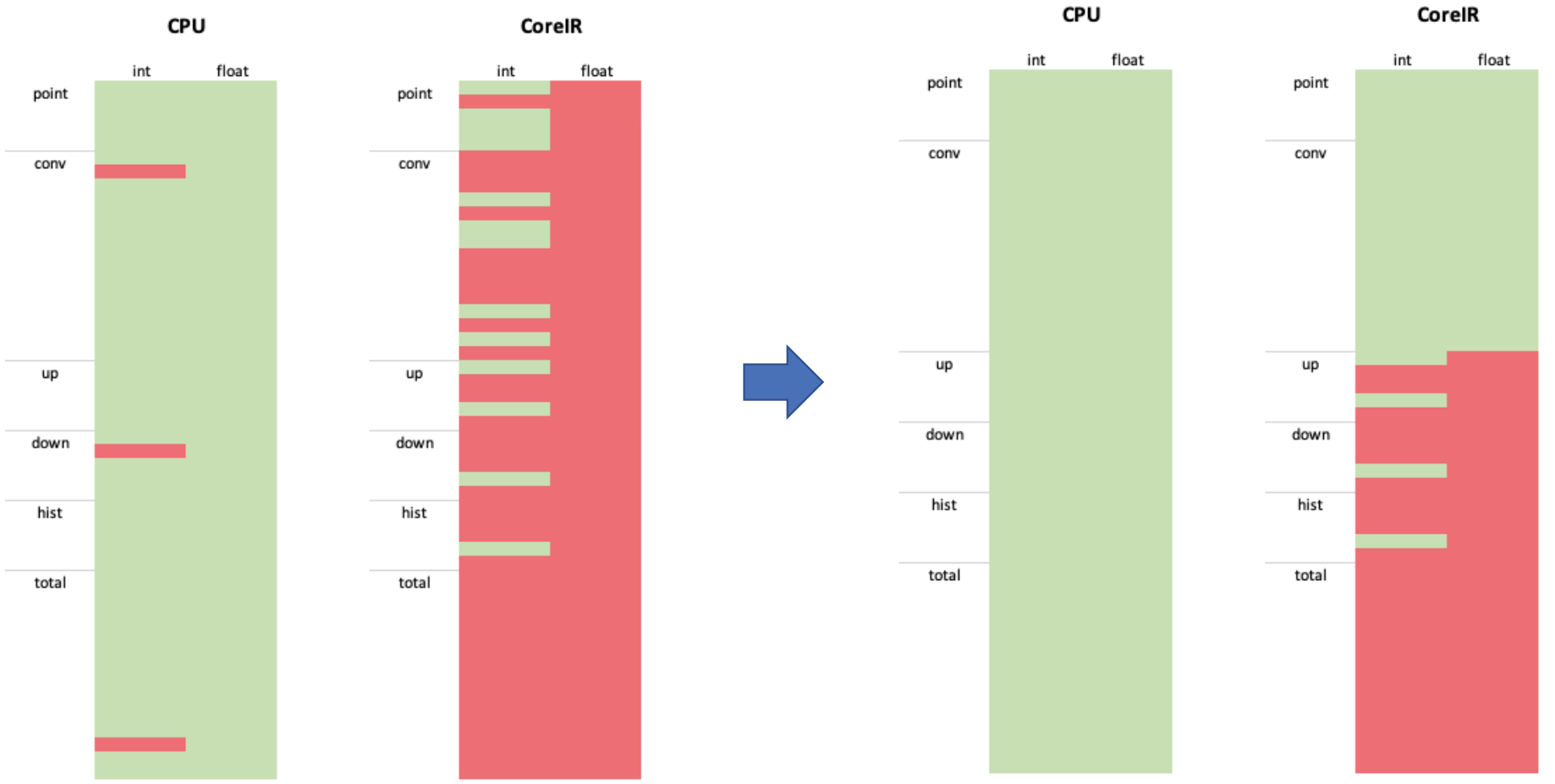


Testing: image comparison and random pipelines





Random Pipeline Generator – overall progress so far



Halide Frontend

Algorithm

```
Var x, y, xi, yi, xo, yo;
ImageParam input;
Func conv, output;
RDom win(0, 3, 0, 3);

conv(x, y)  = 0.0;
conv(x, y) += input(x + win.x, y + win.y) *
              weights(win.x, win.y);
output(x, y) = conv(x, y);
```

Schedule

```
output.tile(x, y, xo, yo, xi, yi, 64, 64);
output.hw_accelerate(xo, xi);
conv.compute_at(output, xi)
conv.update()
    .unroll(win.x)
    .unroll(win.y);

input.in()
     .store_at(output, xo)
     .compute_at(output, xi)
     .stream_to_accelerator();
```

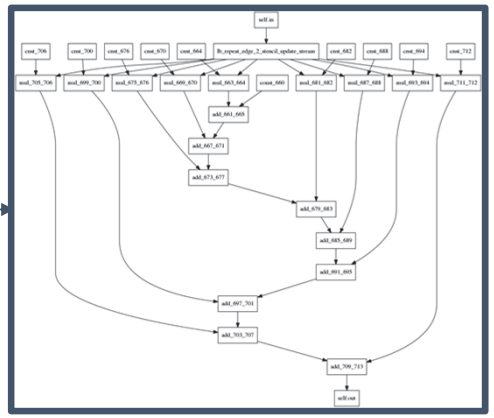
Halide Compiler

Halide IR loop nest

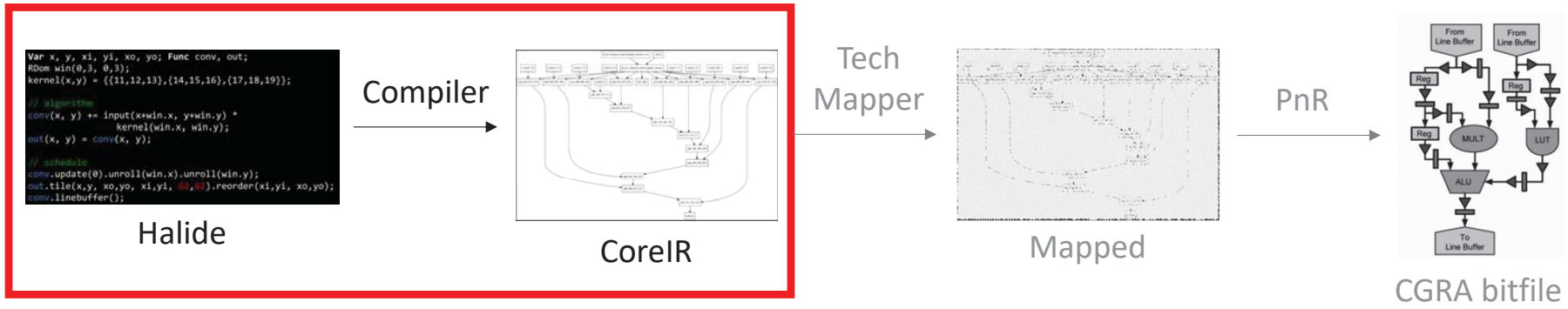
```
GENERATED_HARDWARE(xo, yo):
   StorageUnitToBeSpecified weights;
   Register<int> tmp_conv;
   LineBuffer<size=2x64, stencil=3x3, thruput=1>
         tmp_input;

   for output.yi = 0 to 64:
     for output.xi = 0 to 64:

       Load one new pixel into tmp_input

       tmp_conv = tmp_input[3x3 stencil] * weights

       // streaming store to memory
       STORE tmp_conv to output(xo*64+xi, yo*64+yi)
```

CoreIR Codegen

CoreIR circuit



Questions?



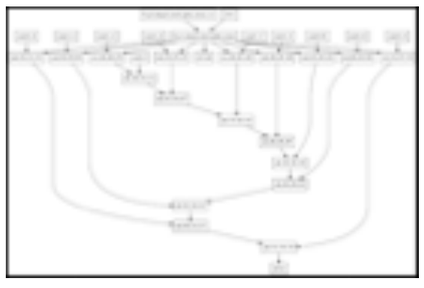
Automatic Generation of a Technology Mapper

Ross Daly
Caleb Donovick

Overview of the Application Compilation Flow

Halide

Compiler

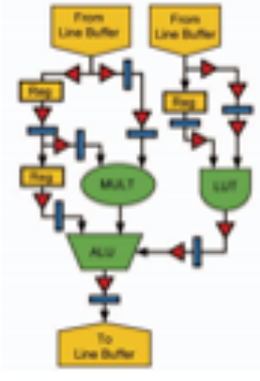


CoreIR

Tech Mapper

Mapped

PnR



CGRA bitfile

Basic CoreIR Overview

- CoreIR is an LLVM-like Intermediate Representation and compiler framework.

- Can represent any Hardware Circuit or Dataflow Graph

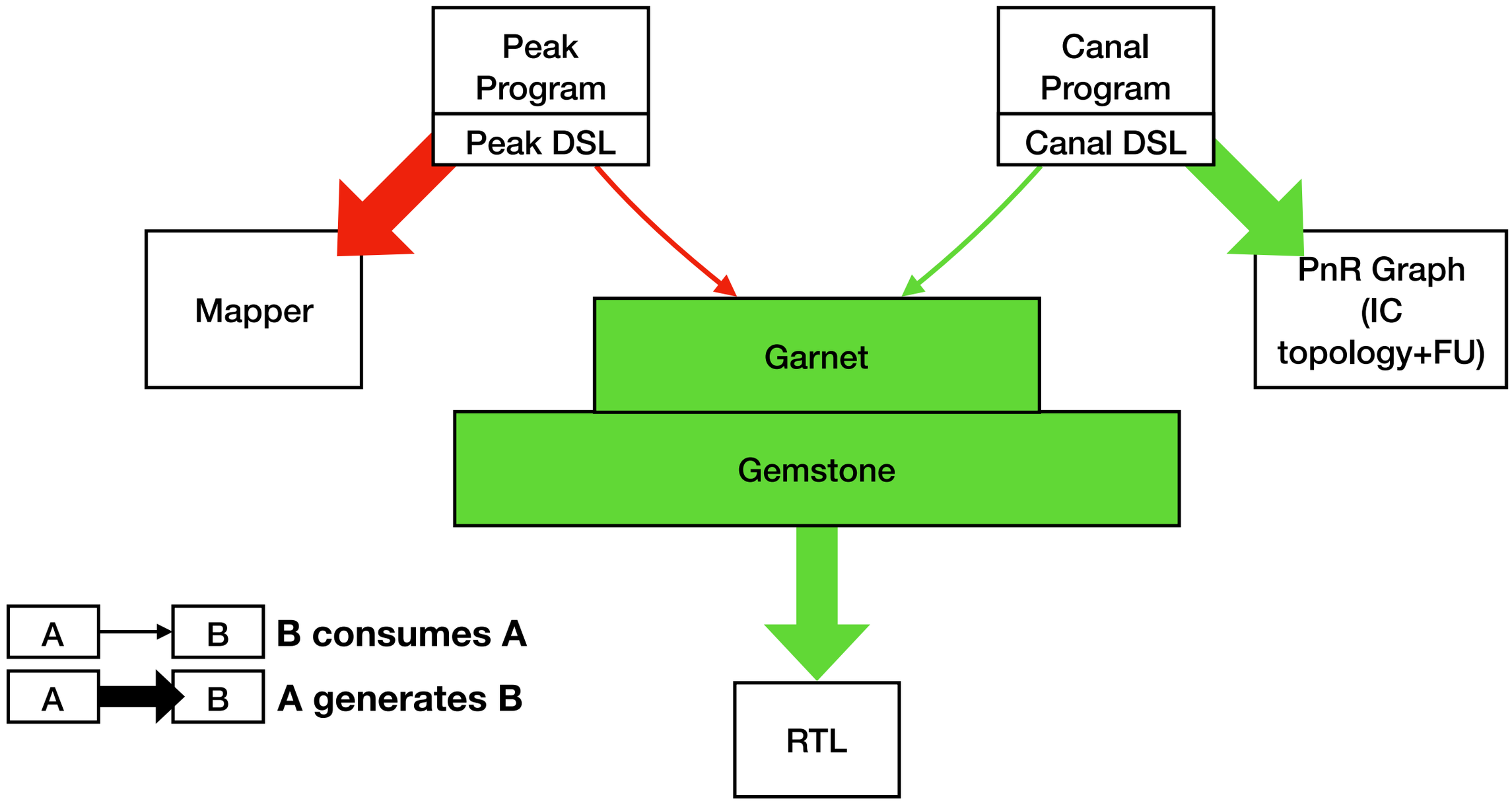
- Primitives have **precise formal semantics;** identical to SMT BitVectors

PEak: A DSL for describing Processing Elements (PEs)

- Python-embedded DSL for describing the ISA and functional model of a PE.
- Has **precise formal semantics**
- Generate correct-by-construction RTL
- Generate collateral for bitstream generation
- Goal: Also generate a mapper

A more Agile approach to Architecture Design

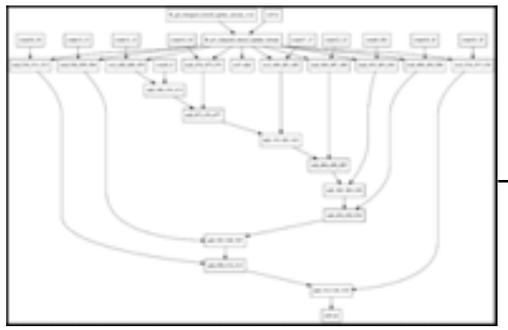
- The Problem: How can we explore different CGRA architectures
    - Any change to the CGRA architecture requires manual changes to the Application Compilation Flow
        - Specifically Technology Mapping and PnR
- Architects do not want to update the compiler for every change they make.
- This is tedious, time consuming, and error-prone
- Agile approach would be to remove the human in the loop



"Compile" design to RTL *and* mapper/pnr-graph

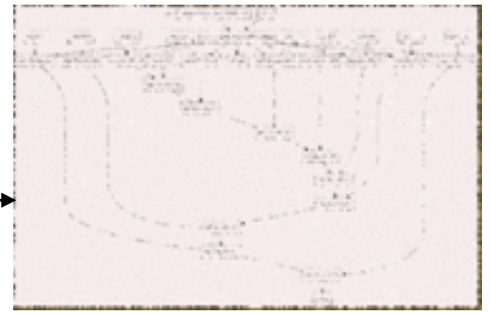
Problem Description: Technology Mapping

CoreIR Graph



Mapper

Mapped Graph



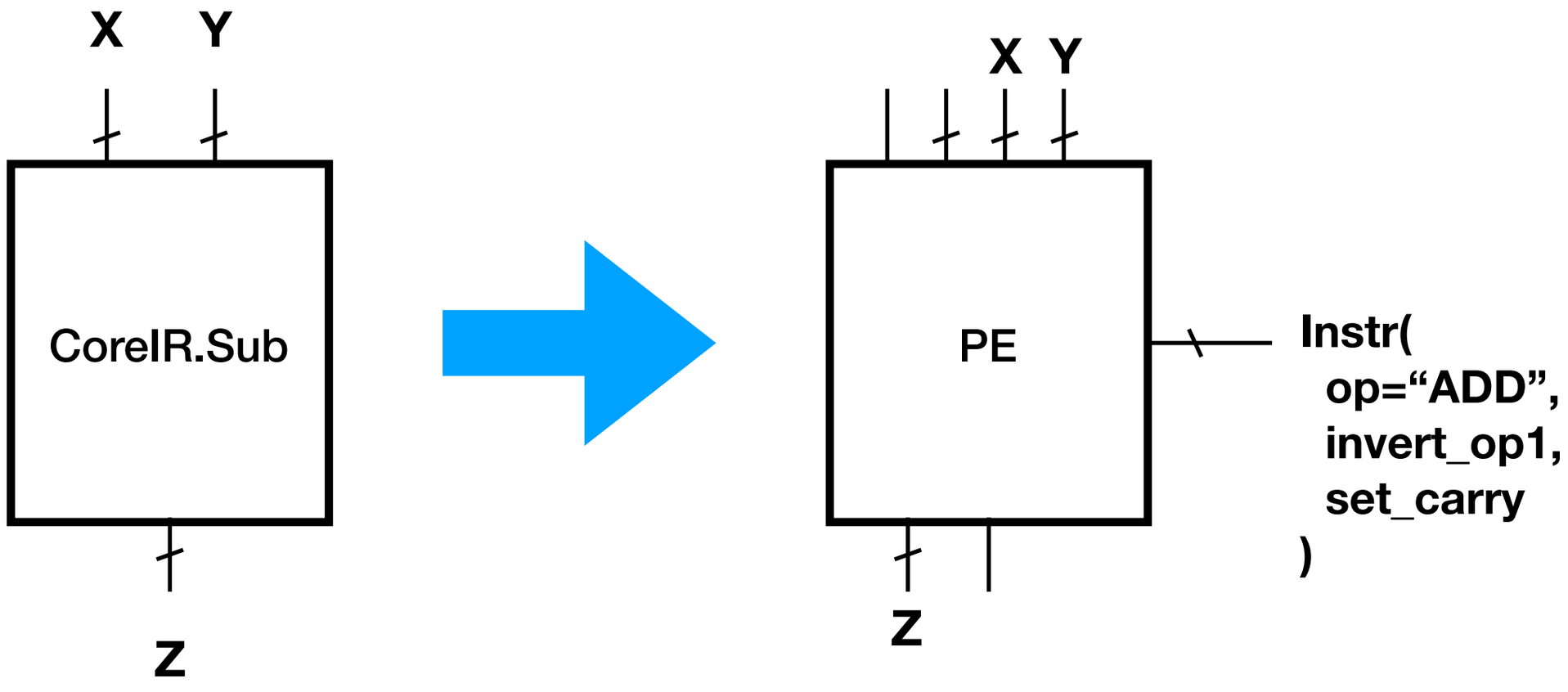
Uses: CoreIR operations

Uses: PEs Mem tiles, etc…

Simple design of a mapper

- Foreach primitive in CoreIR:
    - Manually write one (or more) **Rewrite Rule** from that primitive to a PE instruction (and an interface binding)
- Given an application graph of CoreIR primitives
    - Foreach node in application graph:
        - Apply appropriate rewrite rule to generate PE graph

‘Sub’ Rewrite Rule



How do we know the rewrite rules are correct?

- Because CoreIR and Peak are formally specified
- Foreach Rewrite Rule:
  - Generate SMT formula for CoreIR primitive
  - Generate SMT formula for PE (given an Instruction)
  - Use SMT solver to prove the two formulas are the same

Problem: What if the PE changes? (Agile!)

- Examples:
  - Design space exploration
  - Changing implementation of operations
  - Changing bitwidths
  - Adding ALU Flags
  - etc…

Algorithm for Automatically Generating Rewrite Rule from PEak

```
Given a CoreIR primitive:
  coreir_smt = generate_smt_formula(primitive)
foreach PEak instruction:
  pe_smt = generate_smt_formula(instruction)
  if coreir_smt is equivalent to pe_smt:
    Found a rewrite rule for the primitive!
```

Does this work?

- Yes! We can generate most CoreIR Rewrite rules currently for Lassen (our current PEak specification)

- Using this approach we have already found and fixed bugs from Lassen

Example: Can we generate a ‘Sub’?

isa.py

```
class OP(Enum):
    Add = new_instruction()
    Or =  new_instruction()
    And = new_instruction()
    XOr = new_instruction()
    #etc...

class Instr(Product):
    op : OP
    invert_in0 : Bit
    invert_in1 : Bit
    set_carry : Bit
```

functional_model.py

```
class SimplePE(Peak):
    def __call__(self, instr, in0, in1):
        if instr.invert_in0:
            in0 = ~in0
        if instr.invert_in1:
            in1 = ~in1

        carry = instr.set_carry
        if instr.op == Op.Add:
            res = in0 + in1 + carry
        elif instr.op == OP.And:
            res = in0 & in1
        elif instr.op == OP.Or:
            res = in0 | in1
        #etc…
        return res
```

Work in progress

- Mapping memories
  - Current work on a DSL for specifying Memories
  - Will generate rewrite rules using that DSL
    - Exploring specifying using SMT Array semantics
- Registers and Pipelining
  - Our applications are dataflow graphs. Can apply automatic pipelining and retiming during our compilation.
  - Registers with backedges can be allocated separately
  - Exploring formal routes like model checking

Work in progress

- Merging operations
  - example: Add(Multiply(X,Y),Z) -> FMA(X,Y,Z)
    - Use a similar SMT-based generative technique to create more complicated rules like “fused multiply-add”
- Splitting operations
  - example: Sub(X,Y) -> Add(Add(X,~Y),1)

Questions?

Canal, a new interconnect generator

Keyi Zhang
keyi@cs.stanford.edu

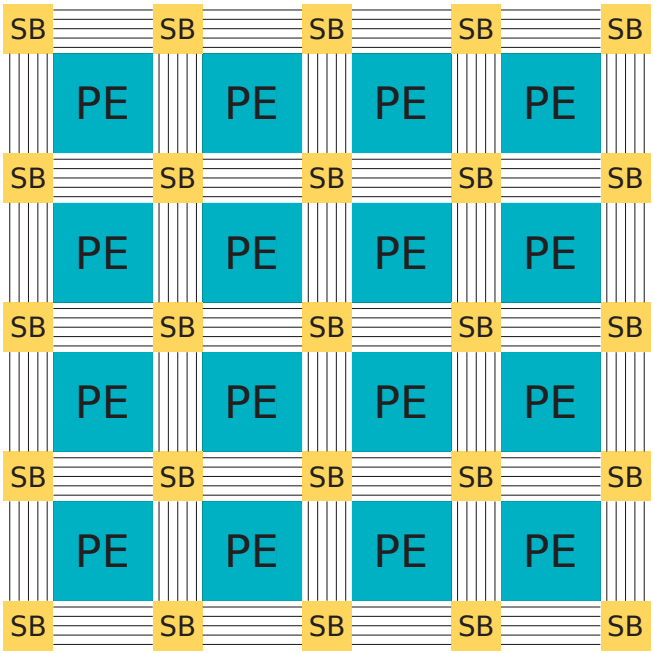
Stanford AHA! Agile Hardware Center

Background and Motivation

What is Interconnect

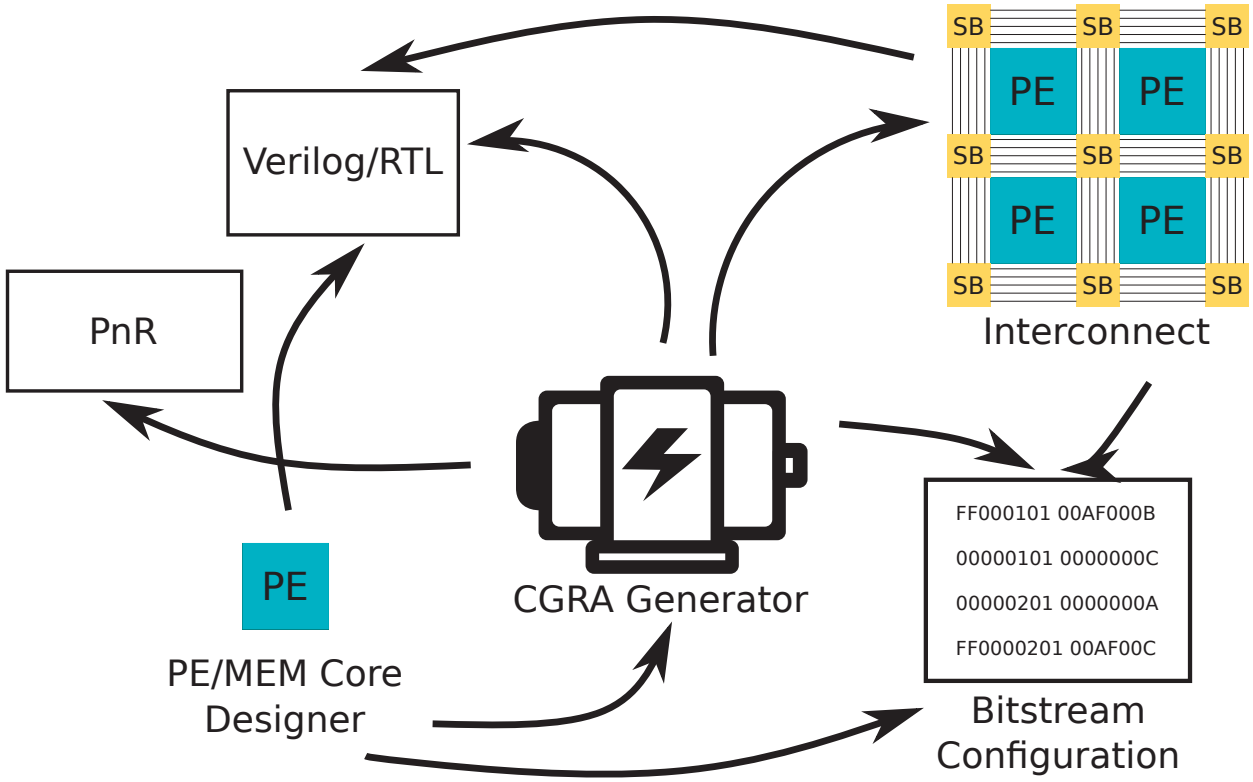
- Programmable logic to connect inputs to and outputs from logic blocks, e.g. PE.
- Consists of unidirectional wires, configuration registers, and multiplexer (SB and CB).
- Contribute to most of the critical path delay and area.

We will focus on "island-style" mesh-based interconnect.



Simplified CGRA interconnect with PE array and SBs.

Current Mess in CGRA Generator



A tangled state of CGRA Generator

Change in one place and manual edit in a thousand other places

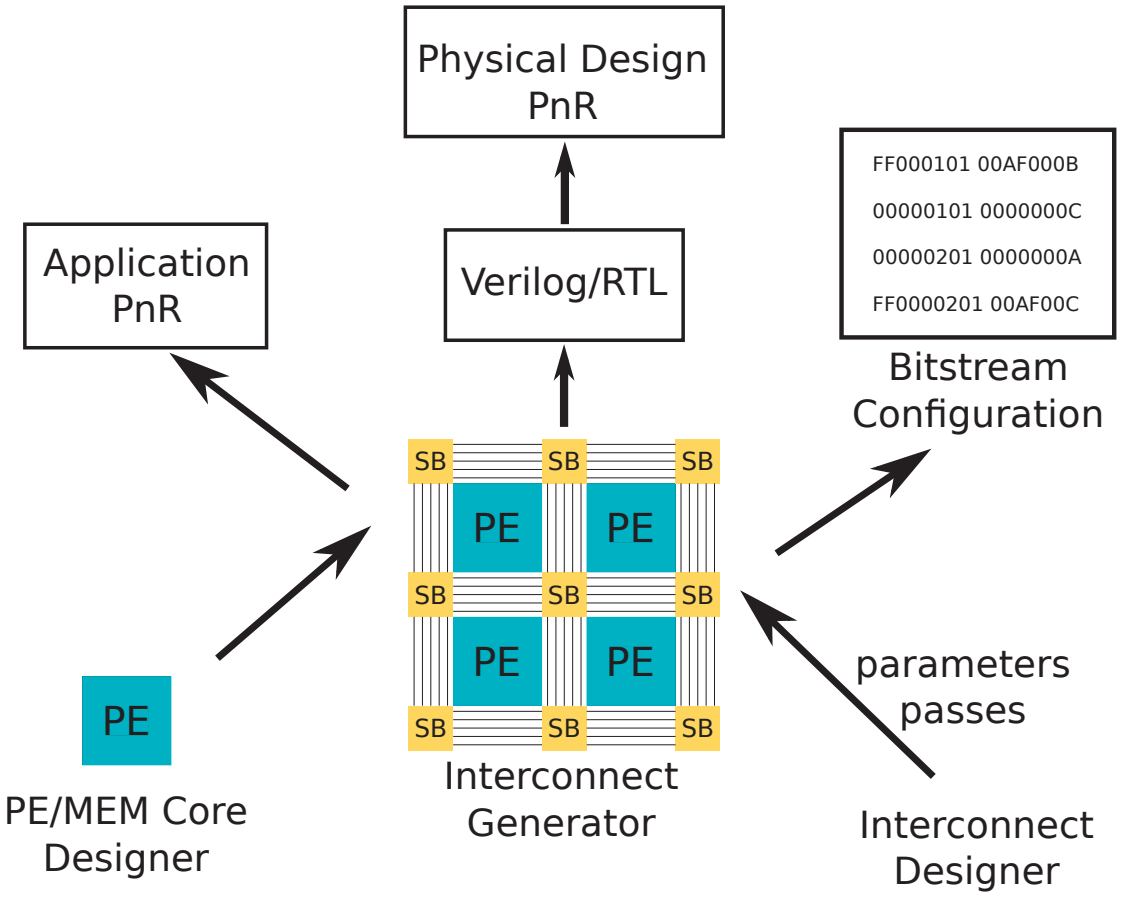
1. New PE/MEM requires changes in SB and CB designs, as well as configuration/bitstream
2. New interconnect design requires changes in
   - 2.1 configuration/bitstream
   - 2.2 Application PnR
   - 2.3 RTL + physical design PnR

A New Interconnect Generator, Canal

The goals of Canal

1. Separation of concerns of different components.
2. Provide a high-level representation to allow complex transformation passes.
3. Single-source of truth for RTL, PnR, and bitstream configuration.
4. Interconnect architecture research tool, i.e. the ability to model various interconnect design.
5. Flexible RTL generator and testing infrastructure.

The structure of Canal

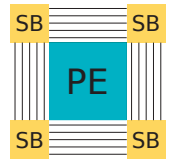


An interconnect-centered design

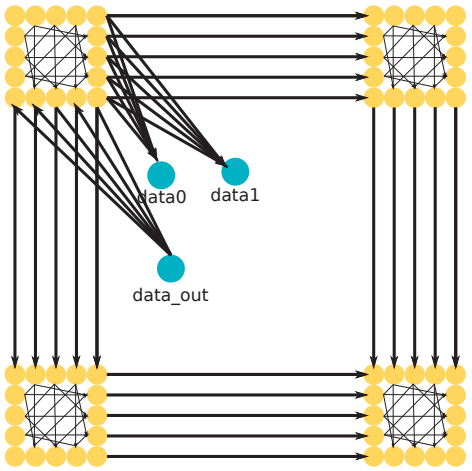
Canal internal representation: directed graph

A different way to model interconnect.

1. Anything connectable in the RTL is a node in the DiGraph
2. A uni-directional wire is an edge connection.
3. Based on its functionality, each node will have different attributes
   - 3.1 NodeType: PortNode, SwitchBoxNode, RegisterNode...
   - 3.2 Coordinate: x, y
   - 3.3 Name, for PortNode
   - 3.4 ...
4. Different bit width tracks have their own graphs.



Traditional Representation

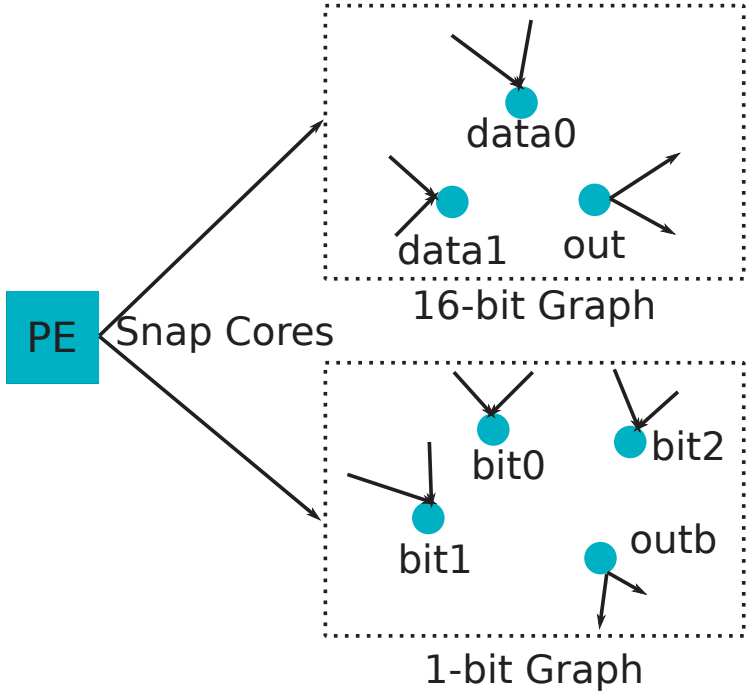


DiGraph Representation

Directed graph representation of the interconnect connections

Snapping in cores (PE or MEM) is very easy

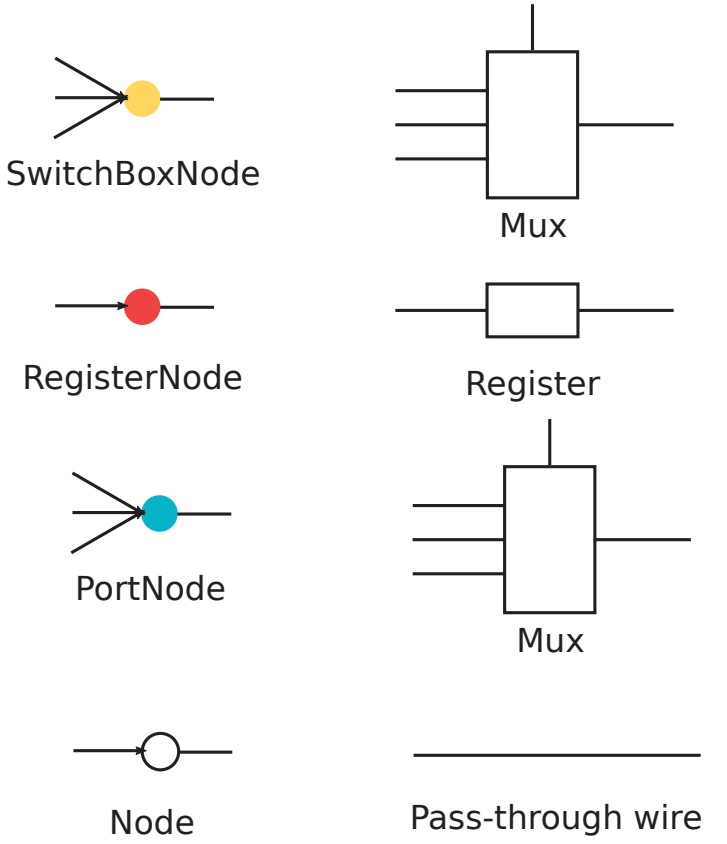
PE/MEM designers can snap in cores in the interconnect through a well-defined core interface. Canal will introspect the core and generate corresponding port node in the DiGraph.
Canal also allocates configuration address space for the cores as well as the interconnect. As a result, Canal removes the contention between PE/MEM and interconnect designers.



A PE snaps into 2 interconnect graphs with different bit widths

DiGraph in Canal can be transformed into RTL directly

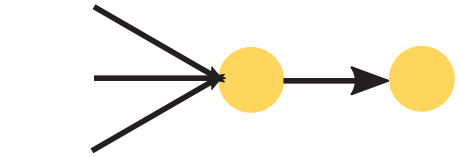
Depends on the connections and node attributes, different types of hardware are created for different nodes.



Transform rules from DiGraph to RTL

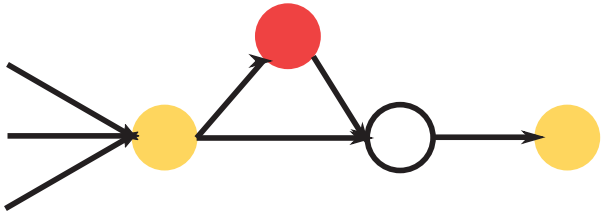
Canal can have multiple passes to transform the graph

Example



SwitchBoxNode

RegisterNode

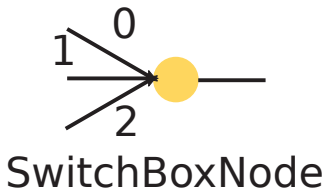


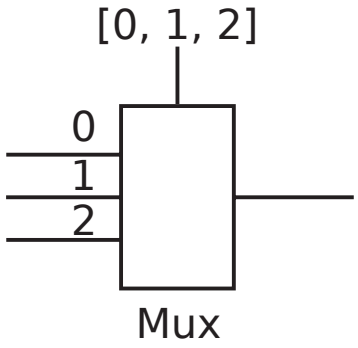
SwitchBoxNode

Pipeline register insertion by transforming the edge into a group of nodes and connections.

Canal can facilitate bitstream generation out of the box

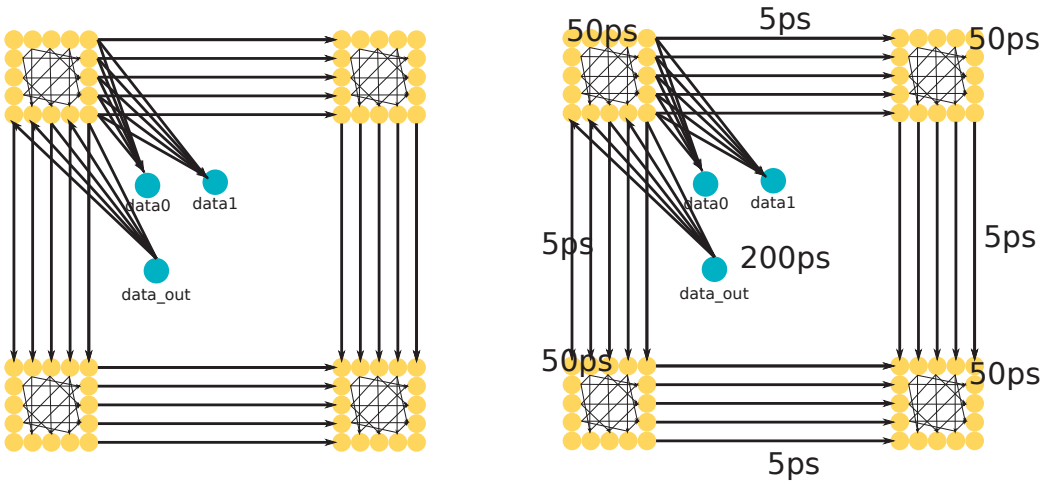
Thanks to the DiGraph representation, mux configuration can be generated from the connections directly. Based on the incoming edges connections (in orders such creation or naming), we can assign index to each edge, which is the configuration data for the mux.





PnR tools can get information from Canal for free

Canal's DiGraph representation is originally designed for PnR[1] tools. After adding technology-dependent delay information, the graph can be consumed directly in the downstream PnR tool[2].



Adding node and edge cost to the DiGraph and using it in PnR directly.

[1] We refer PnR as place and route CGRA primitives on the chip, as in FPGA.
[2] We have an efficient PnR tool designed for the CGRA.

Working in Progress

Features we are working on for the new chip

- Leverage the unique role of Canal to provide a functional model of the entire CGRA (almost done).
- Add power domain to the interconnect. This need to be co-designed with PnR tools to prevent x propagation.

Thank you

AHA (Agile Hardware): Visual Computing

Program Review

Mark Horowitz

Briefing Prepared for Tom Rondeau

4/18/2019

Stanford | ENGINEERING

What Are We Trying to Do?

Create:

- A new way to create DSSoCs
  - Use an agile design and a end-to-end flow
  - Create accelerators through specialization of CGRA fabric

Deliver:

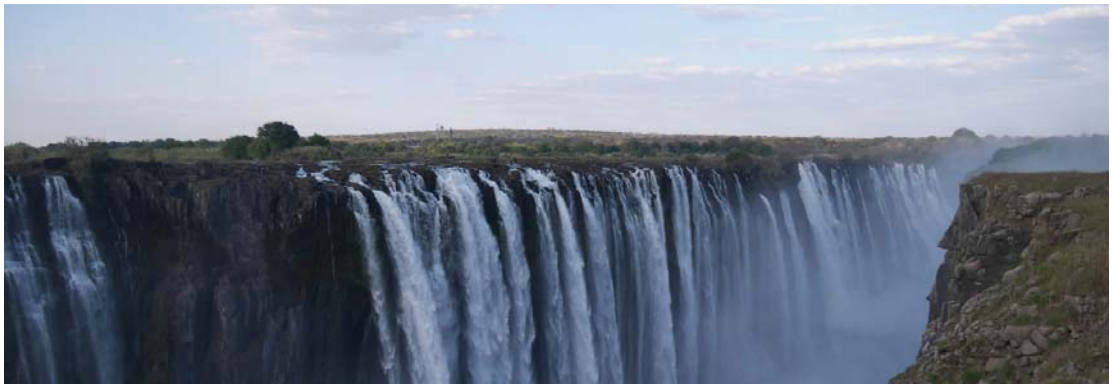
- Improved video analysis algorithms
- A DSSoC generator for visual computing
- A framework for mapping these algorithms to the DSSoC

How is it Done Today?

- Study the application
- Build the hardware
- Then write the software for the machine
  - And that is the only application this hardware will ever support…

- Repeat

- Problems
  - Application moves
  - Don’t address real issues
  - Long, expensive design



What is New in Your Approach?

- Changing the design approach
  - From Waterfall to Agile

- Forces you to create an end-to-end system sooner
  - Makes you see the whole problem early
  - Allow you to track applications
  - The software system is needed from day 1.

- Allow you to:
  - Experience issues sooner (and we can vouch for that!)
  - Use current software system to create your solution

Why Do You Think It Will Work?

- Widely used in software
- Stopper in hardware is the long fabrication cycle
  - Can address that by creating prototypes that are configurable
- Only successful approach to hardware/software co-design
  - Standardized ISA, incrementally improve the processor and application

Who Cares?

- About a visual DSSoC?
  - Anyone who wants to do image processing/analysis (nearly everyone)
- About a better approach to hardware/software design
  - Everyone who is trying to scale system performance
- Unless we care a new design approach
  - Innovation in VLSI components will rapidly decrease

How Much Will It Cost? Milestones?

- See proposal …

Application Scheduling Phase 1 Goals

To allow us to auto-schedule a wide collection of applications for optimal performance/energy

- Deliverable: Halide auto-scheduler

- Created a general Halide auto-scheduler (Kayvon's talk)
  - Used on a number of different processors
    - In cellphones and data centers (Google, Facebook, etc)
  - We will use for our hardware as well

- But it didn't target Hexagon …

Software Tools Phase 0 Goal

Create a tool chain that takes applications in a HLL (currently Halide) and map them to a FPGA or CGRA

- Deliverable: Halide to core IR compiler, and the coreIR tool set

- We have a working Halide to CoreIR compiler (Jeff's talk)
  - We are working on extending its application range

- We have a working CoreIR system (Ross/Caleb's talk)
  - Can automatically generate mapping from hardware information
  - Working on better optimizations, handling more complex PEs

Software Tools Additional Work

Create a tool chain that takes applications in a HLL (currently Halide) and map them to a FPGA or CGRA

- Creating the Gemstone framework (Pat's talk)
  - Peak, Canal, Lake?
  - To allow end-to-end system to function with changing components

Application Study Phase 0 Goal

Explore different types of visual computing to get a better sense of types of computation and communication demands. Start to create a set of benchmark applications.

- Deliverable: Image processing taxonomy

- Have studied modern imaging applications (Kayvon's Feb talk)
  - Motivated new memory model in our Halide to CoreIR compiler
  - Working to get ML applications running through the system

Application Mapping Phase 0 Goals

Create our first generation CGRA in silicon with test and debug capabilities

- Deliverable: CGRA Chip
- The Jade CGRA works! (Max's talk)
  - It has a few unanticipated features
  - Really a pipe cleaner (and it did find lots of gunk)
- Working on Garnet SoC next

Hardware Integration Phase 0 Goals

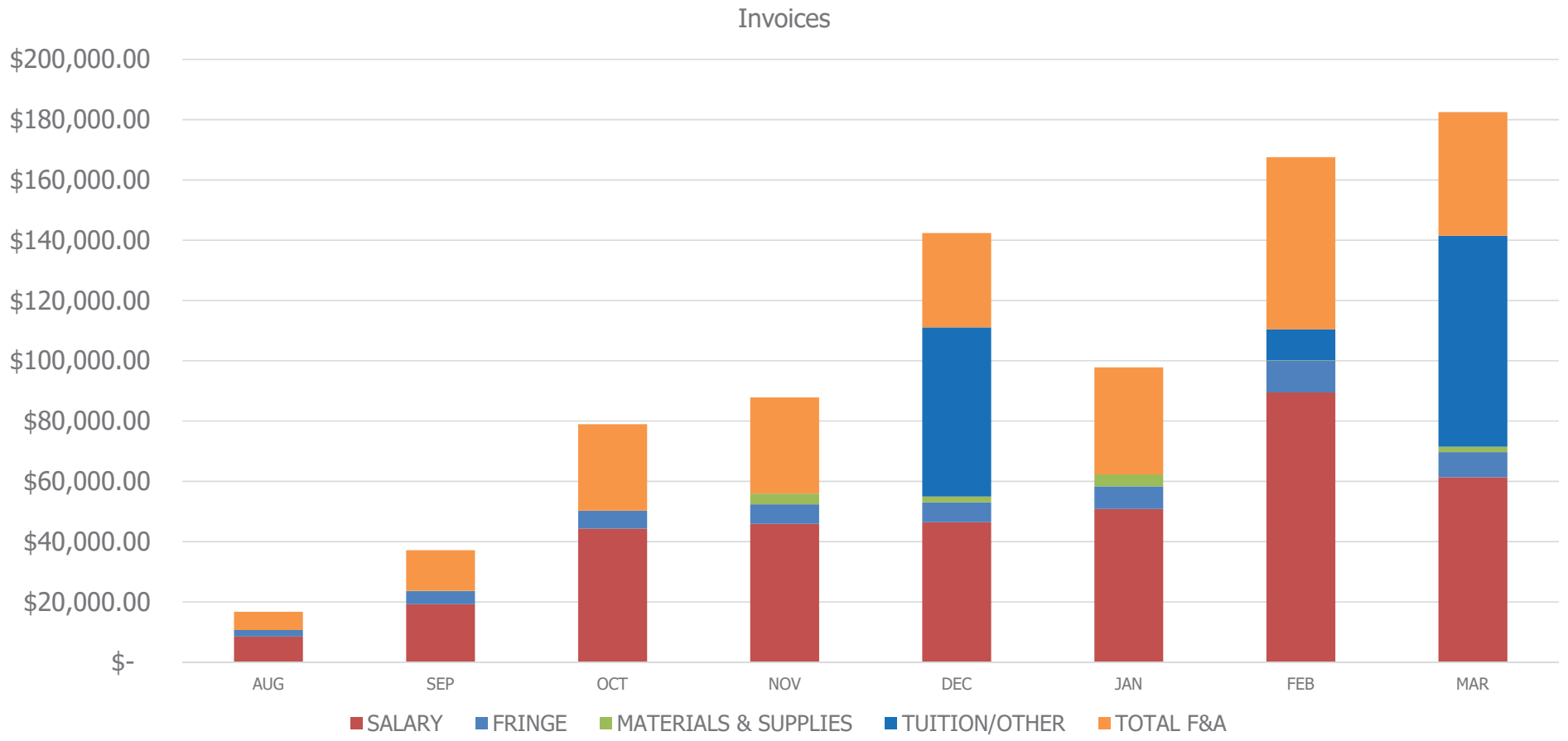
Take the output from the compiler system and place and route the desired hardware on the CGRA.

- Deliverable: Place and Route system for CGRA
- Created an application place and route system (Keyi's talk)
  - Connected to the interconnect generator, Canal
  - Requires mapped CoreIR
- Working on retiming next

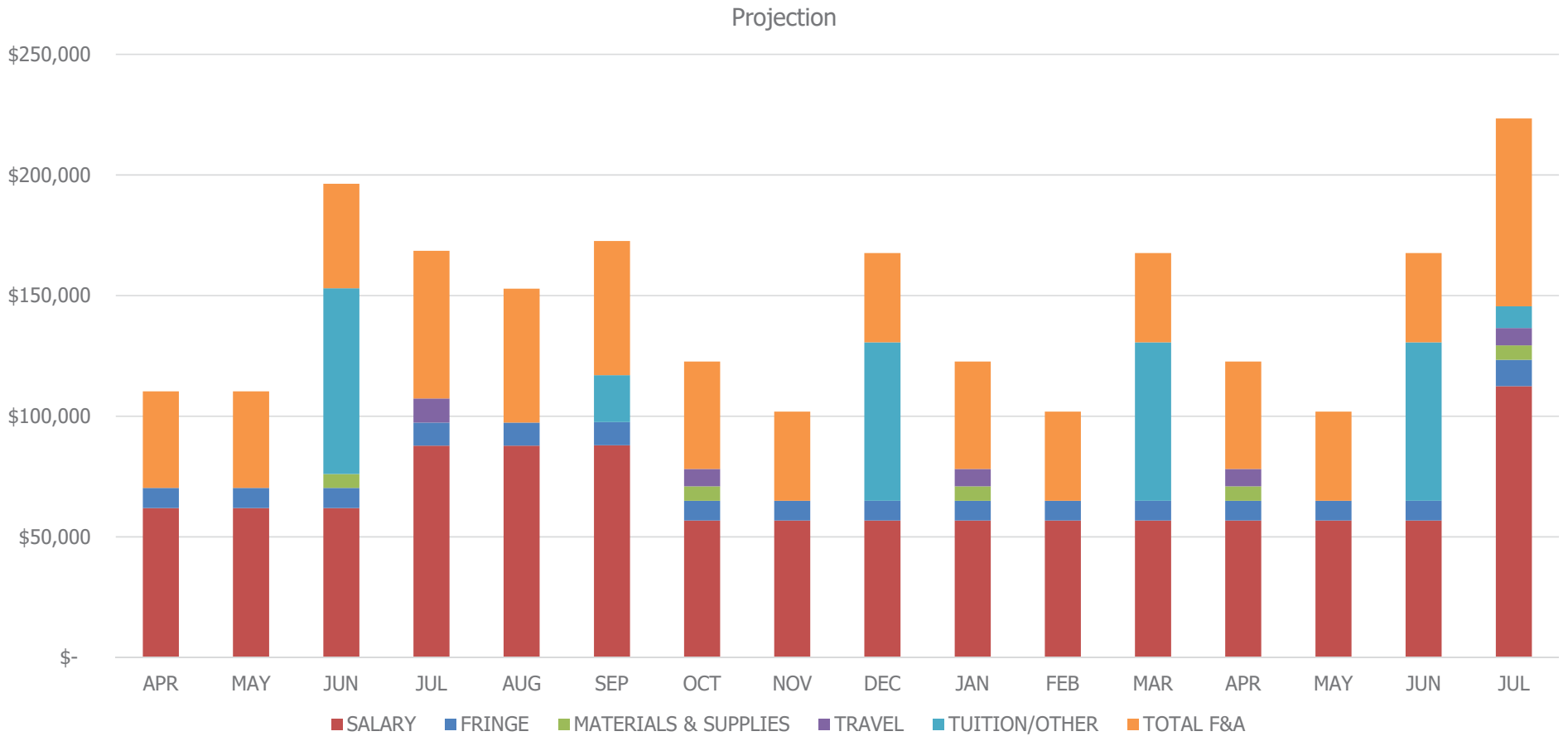
Risk and Issues

- Tape out is in 2.5 months
  - And we have a lot of work to do
- Just got the IP issues settled in ARM for SoC components
  - We think we have time to get that setup
- We need to use Zynq as "host" for this version
  - And we haven't started working on that yet
- We are late on technical reports
  - No one mentioned we had to file them (oops)

Budget and Finance



Budget Projections



Transition Planning

- Working with 5 main partners on this project
  - Plus Xilinx and Apple

Transitions

- The Halide scheduling work
  - Collaboration between Facebook, Google, Stanford, UCB
  - Being used at those companies

- There is a lot of interest in our design approach
  - Have "consultants" from our partners working with us
  - Watching out approach for possible spin outs, and internships

MAC for DSSoC

- Working on this subsystem now
- Two level approach
  - Simple ARM controller for the CGRA
  - Unix processor talks with that controller to use device
- We will know much more in Summer

Emulation Approach?

- Have working prototype for initial testing
- Next generation system
  - Using Zynq as the host processor
    - Using an AXI bridge between Zynq and SoC
  - ARM embedded controller for CGRA