Halide and Clockwork Compiler

Jeff Setter

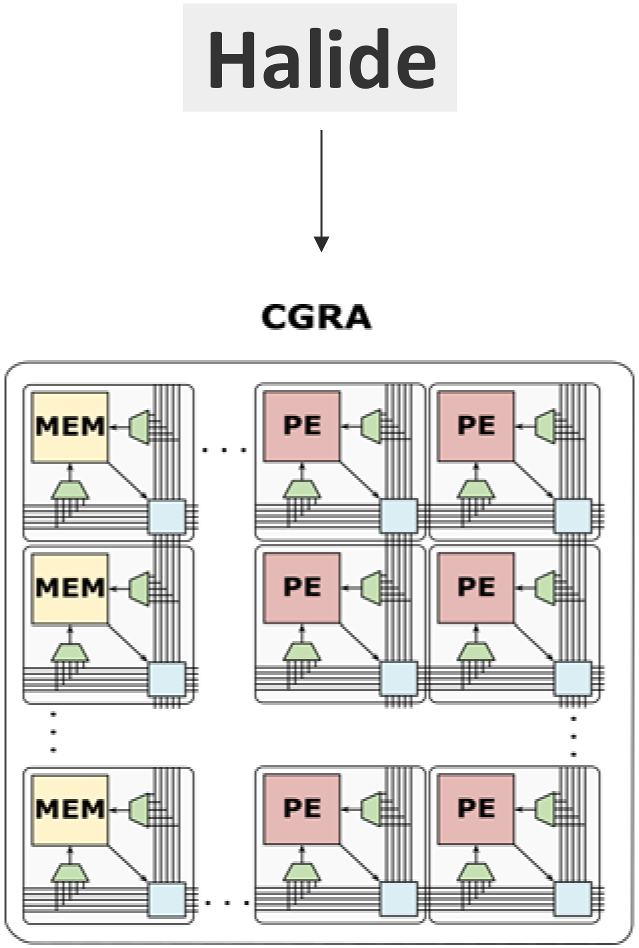
Stanford University

Overview of our System

Applications: in Halide DSL (stencils + DNNs)

Hardware: CGRAs

- Coarse-Grained Reconfigurable Array

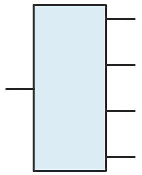


Talk Overview:

1) **Halide Scheduling**

```
app += algorithm;
app.scheduling();
```

2) Clockwork to map Unified Buffers



Halide

Motivation: need a concise way to describe specializations for each hardware target

=> Separate algorithm from schedule

Algorithm: description of the computation (*what* output values)

- Mathematical operations

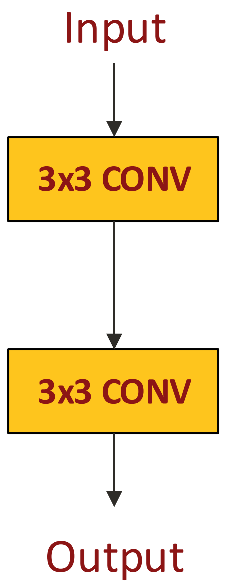
Schedule: optimization decisions (*how* to compute)

- Loop optimizations to run fast

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Fredo Durand, and Saman Amarasinghe. *Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines*. In PLDI 2013, page 519-530.

Example Algorithm: cascade for CPU

```
// Algorithm:
// Define 3x3 window of kernel weights
RDom r(0, 3, 0, 3); // Reduction
kernel(0,0) = 1;  kernel(0,1) = 2;  ... kernel(2,2) = 1;

// First convolution
conv1(x, y) += kernel(r.x, r.y) * input(x + r.x, y + r.y);
conv1_norm(x, y) = conv1(x,y) / 16;

// Second convolution
conv2(x, y) += kernel(r.x, r.y) * conv1_norm(x + r.x, y + r.y);
conv2_norm(x, y) = conv2(x,y) / 16;

output(x, y) = conv2_norm(x, y);
```



Halide Scheduling: parallelism and memory

Efficient and fast application execution relies on **parallelism** and good **memory management**.

Optimizations for CPU

- Loop parallelism: unrolling loops, vectorization, threading
- Memory: tiling, memory granularity (fusion)

Halide Scheduling: parallelism and memory

Efficient and fast application execution relies on **parallelism** and good **memory management**.

Optimizations for CPU

- Loop parallelism: unrolling loops, vectorization, threading
- Memory: tiling, memory granularity (fusion)

Scheduling for **CGRAs**

- Hardware parallelism: PEs simultaneously executing
- Memory: tiling, memory hierarchy, specialized for streaming

Jeff Setter. *Compiling Image Processing and Machine Learning Applications to Reconfigurable Accelerators*. In Stanford Doctoral Dissertation, 2023.

Halide Scheduling to CGRA

Reused the existing Halide compiler toolchain as much as possible without introducing new primitives.

Hardware scheduling:

- Accelerator: input, output, image size
- Loops: reorder loops, tiling
- Memory: memory temporaries, hierarchy
- Compute: duplication

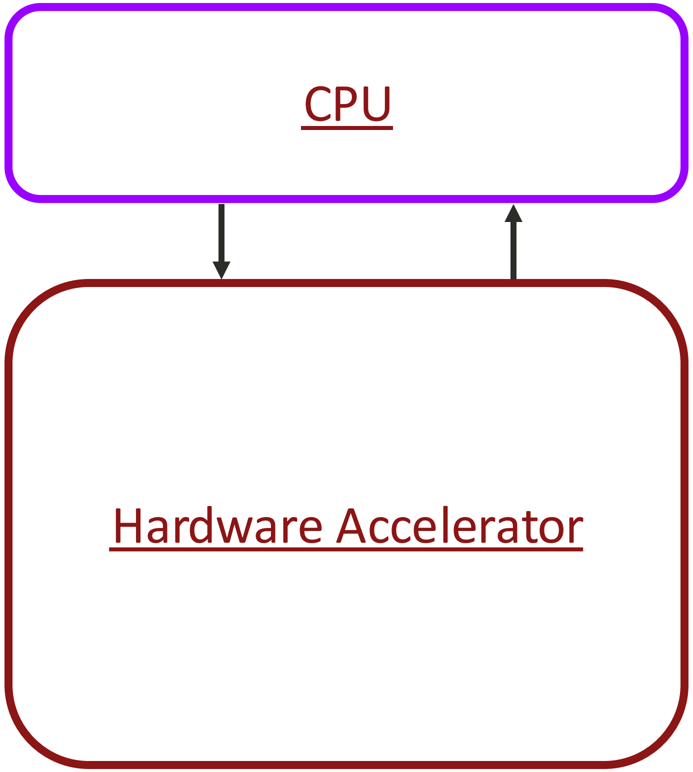
Define Accelerator's Scope

```
hw_accelerate(xi, xo)
```

- Defines the accelerator output
- Argument specifies outermost loop

```
stream_to_accelerator()
```

- Defines an accelerator input



Define Loops: strip mining, reordering, tiling

```
split(x, xo, xi, 64)
```

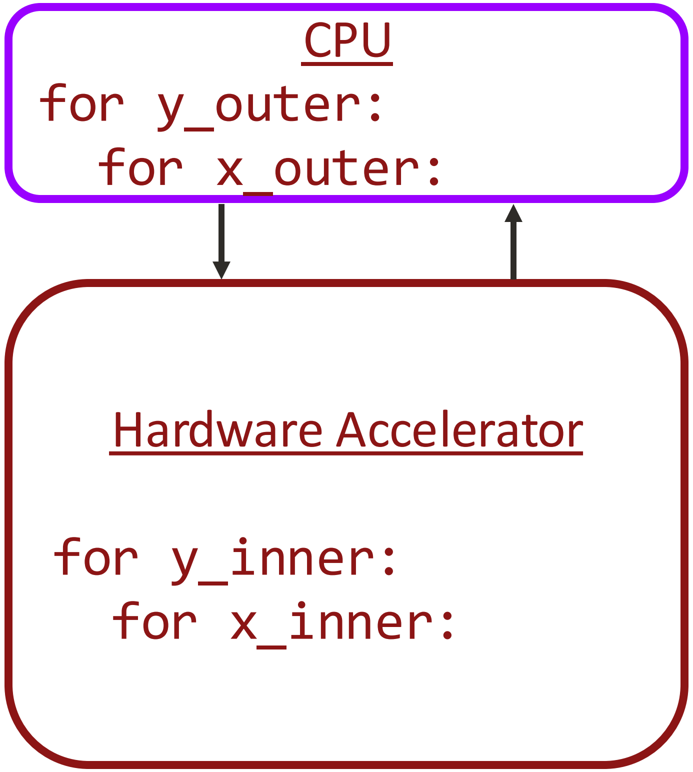
- create nested loops
- specified size for inner loop

```
reorder(xi, yi, xo, yo)
```

- new loop order (from inner to outer)

```
tile(x,y, xo,yo, xi,yi, 64,64)
```

- split two loops and reorder
- common for images



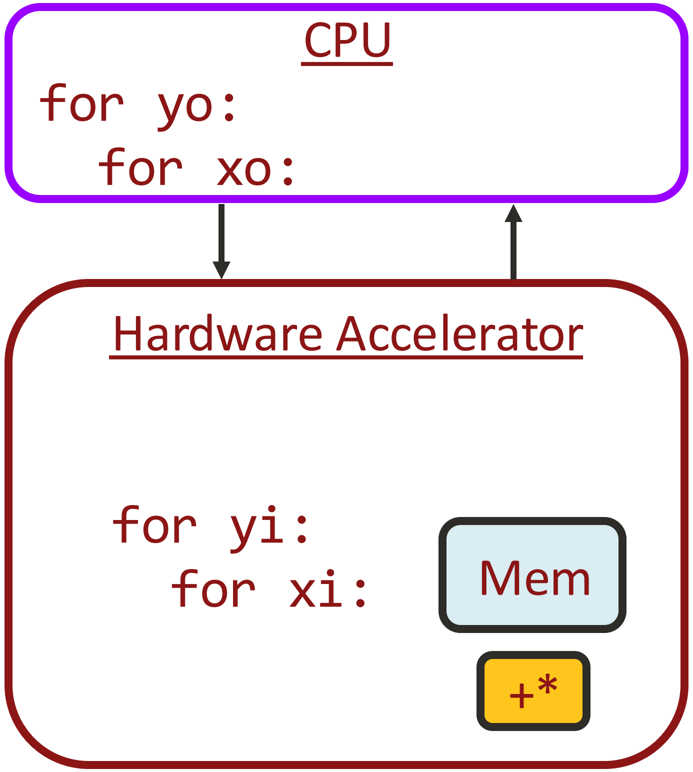
Define Memories: temporaries

`store_at(consumer, xo)`

- create buffers between compute
- at which loop level to create buffer

`compute_at(consumer, xo)`

- for CGRA, we compute at tile level and leave loop fusion for Clockwork
- not: choosing line buffer or double buffer



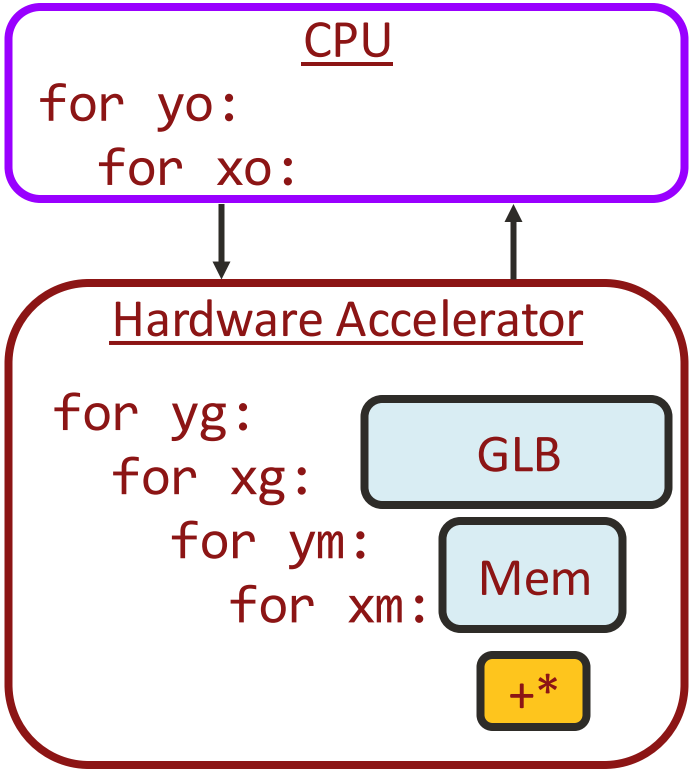
Define Memories: hierarchy

`in()`

- creates a copy (identity function)
- Needed to move data between memory hierarchy levels with `tile` and `compute_at`

`store_in(MemoryType::GLB)`

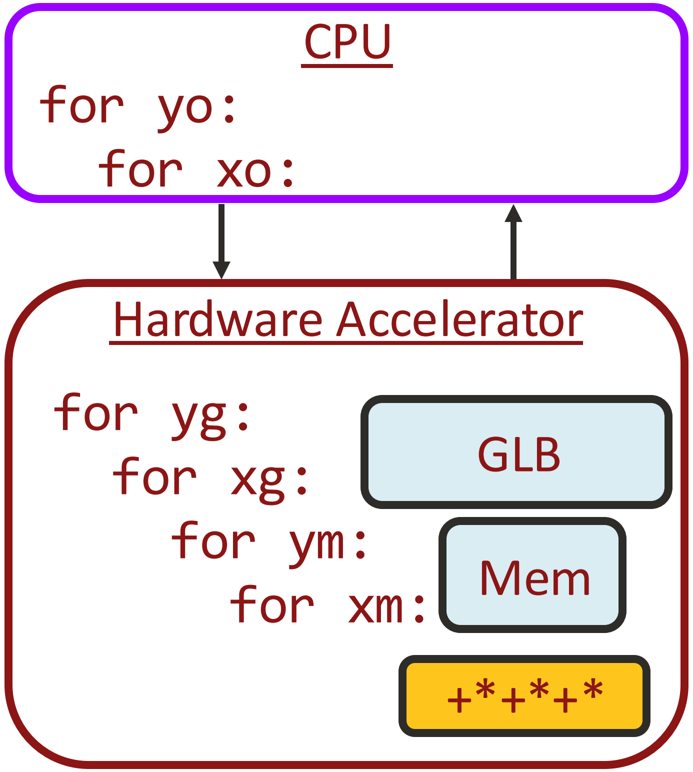
- specify what storage type to use
- Examples: Host, GLB, Memory, Pond



Define Compute: duplication and sharing

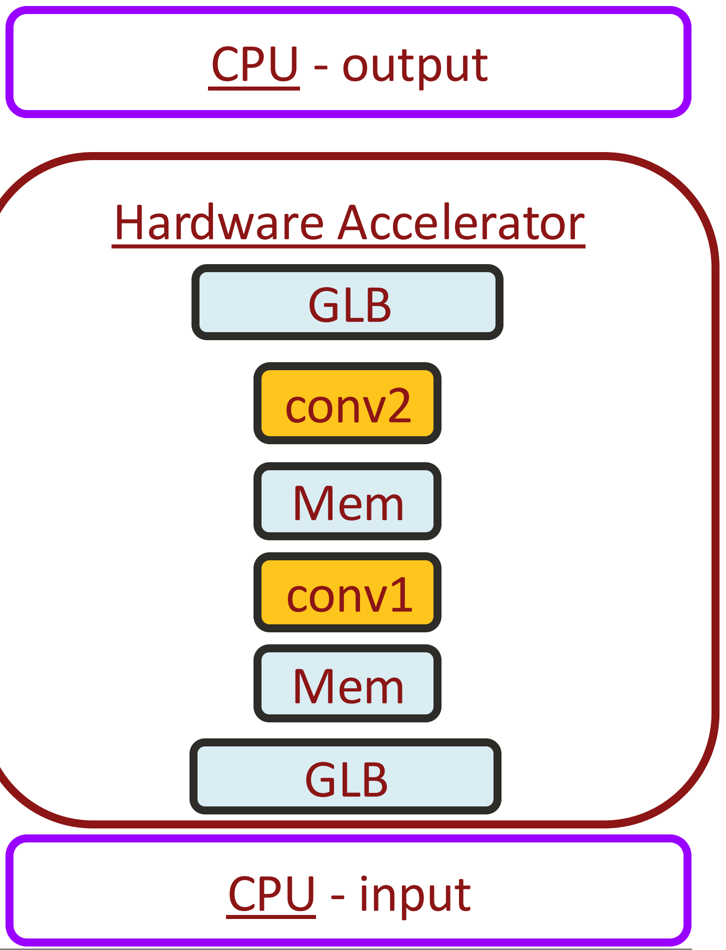
```
unroll(r.y, 3)
```

- increases hardware parallelism
- in Halide IR duplicates loop, because multiple statements = utilizes more existing hardware



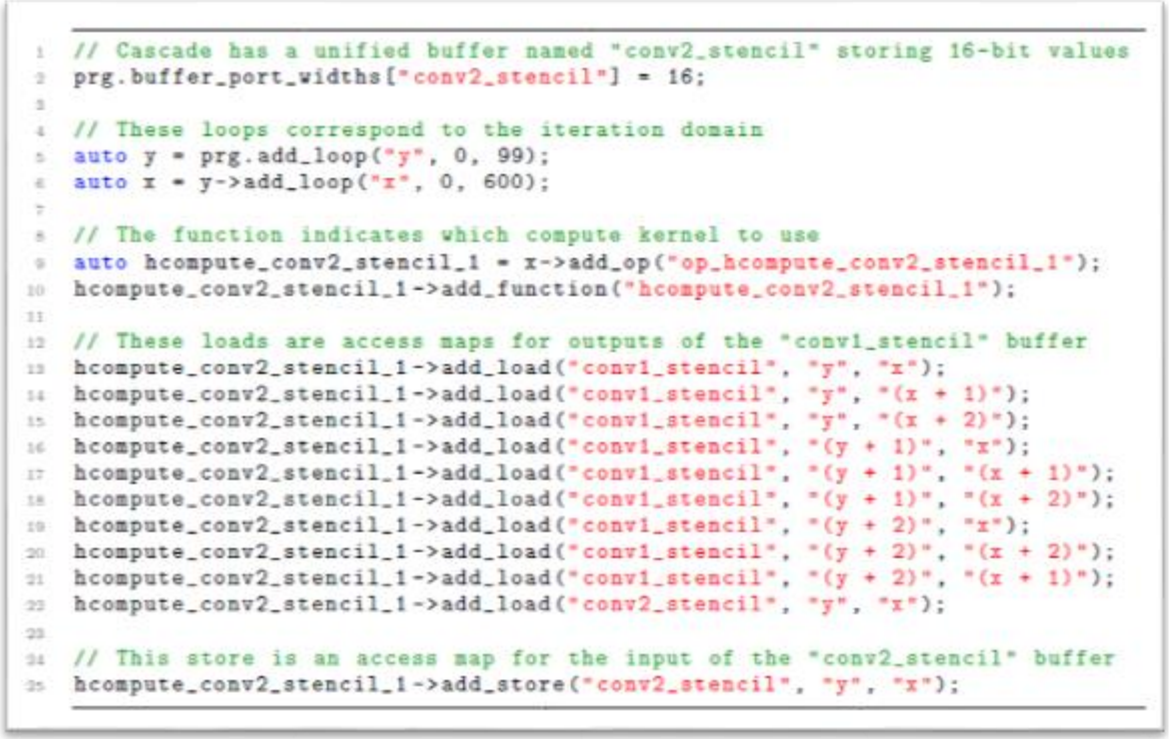
CGRA schedule - cascade

```
/* CGRA Schedule */
Var xii, xio, yii, yio, xi, xo, yi, yo;
output.bound(x, 0, outImgSizeX)
      .bound(y, 0, outImgSizeY);

// Accelerate 360x360 tiles in GLB
hw_output.in().compute_root()
    .tile(x,y, xo,yo, xi,yi, 360,360)
    .hw_accelerate(xi, xo);

// Send 60x60 tiles to CGRA fabric
hw_output
    .tile(x,y, xio,yio, xii,yii, 60,60)
    .compute_at(hw_output.in(), xo)
    .store_in(MemoryType::GLB);

// conv2 kernel with unrolled reduction
conv2_norm.compute_at(hw_output, xio);
conv2.update()
    .unroll(r2.x).unroll(r2.y);

// conv1 kernel with unrolled reduction
conv1_norm.compute_at(hw_output, xio);
conv1.update()
    .unroll(r.x).unroll(r.y);

// MemoryTile and GLB for input stream
hw_input.in().in()
    .compute_at(hw_output, xio);
hw_input.in()
    .compute_at(hw_output.in(), xo)
    .store_in(MemoryType::GLB);
hw_input.accelerator_input();

kernel.compute_at(hw_output, xio);
```



Halide Compiler and Codegen

- Compiler: performs loop transformations
- Codegen to Clockwork: generates files of **compute kernels** and buffers defined as **sequences of memory accesses**



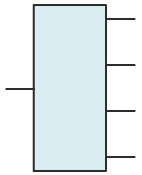
```
// Cascade has a unified buffer named "conv2_stencil" storing 16-bit values
prg.buffer_port_widths["conv2_stencil"] = 16;

// These loops correspond to the iteration domain
auto y = prg.add_loop("y", 0, 99);
auto x = y->add_loop("x", 0, 600);

// The function indicates which compute kernel to use
auto hcompute_conv2_stencil_1 = x->add_op("op_hcompute_conv2_stencil_1");
hcompute_conv2_stencil_1->add_function("hcompute_conv2_stencil_1");

// These loads are access maps for outputs of the "conv1_stencil" buffer
hcompute_conv2_stencil_1->add_load("conv1_stencil", "y", "x");
hcompute_conv2_stencil_1->add_load("conv1_stencil", "y", "(x + 1)");
hcompute_conv2_stencil_1->add_load("conv1_stencil", "y", "(x + 2)");
hcompute_conv2_stencil_1->add_load("conv1_stencil", "(y + 1)", "x");
hcompute_conv2_stencil_1->add_load("conv1_stencil", "(y + 1)", "(x + 1)");
hcompute_conv2_stencil_1->add_load("conv1_stencil", "(y + 1)", "(x + 2)");
hcompute_conv2_stencil_1->add_load("conv1_stencil", "(y + 2)", "x");
hcompute_conv2_stencil_1->add_load("conv1_stencil", "(y + 2)", "(x + 2)");
hcompute_conv2_stencil_1->add_load("conv1_stencil", "(y + 2)", "(x + 1)");
hcompute_conv2_stencil_1->add_load("conv2_stencil", "y", "x");

// This store is an access map for the input of the "conv2_stencil" buffer
hcompute_conv2_stencil_1->add_store("conv2_stencil", "y", "x");
```

Talk Overview:

1) Halide Scheduling

```
app += algorithm;
app.scheduling();
```

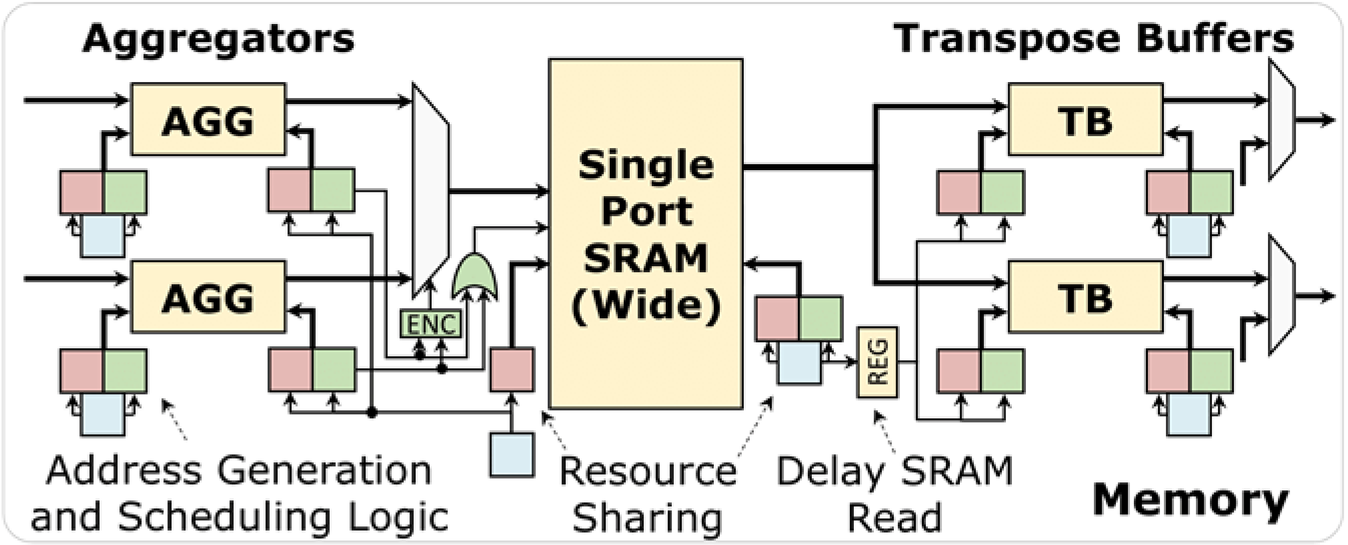
2) **Clockwork to map Unified Buffers**



Unified Buffer Motivation

Motivation:

need abstraction for memories in streaming accelerators



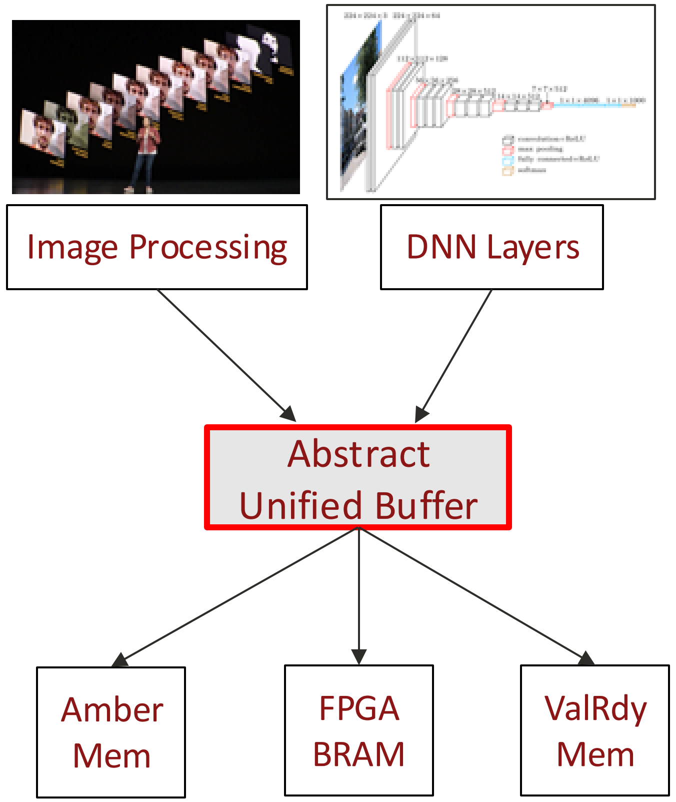
Push memories are **efficient**, but also **complex**

Unified Buffer Abstraction

Create an abstraction to **interface** between the Halide **application** and the **hardware** motifs

- Domain: fully describes streaming applications (stencils, DNNs)

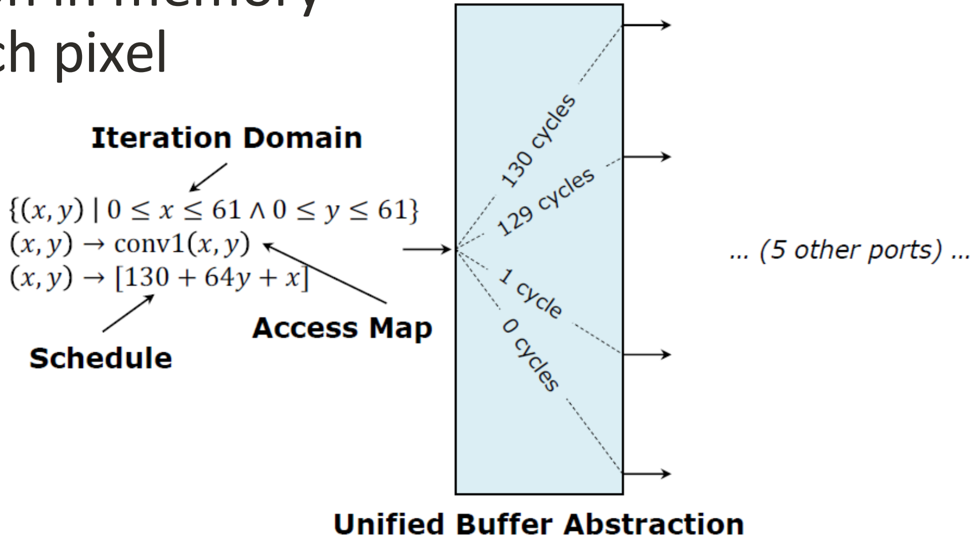
- Hardware Mapping:
  - Sufficiently general to map to dataflow accelerators classes (FPGA, CGRA; statically scheduled, ready-valid)
  - with efficiency of common hardware stream memories (linebuffers, double buffers)



Unified Buffer Properties

Properties for each memory port:

- dependencies: relation to other memories
- iteration domain: valid indices
- access map: relative location in memory
- op schedule: cycle time each pixel executes



Iteration Domain
$\{(x,y) \mid 0 \le x \le 61 \wedge 0 \le y \le 61\}$
$(x,y) \rightarrow \text{conv1}(x,y)$
$(x,y) \rightarrow [130 + 64y + x]$
Schedule
Access Map
130 cycles
129 cycles
1 cycle
0 cycles
... (5 other ports) ...
Unified Buffer Abstraction

Task for Memory Mapper: Clockwork

Optimize

- Improve schedule: reduce latency as much as possible using loop fusion and loop pipelining

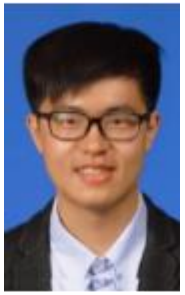
Map to memories

- Occurs after schedule optimization
- Account for hardware constraints

Implemented by Qiaoyi

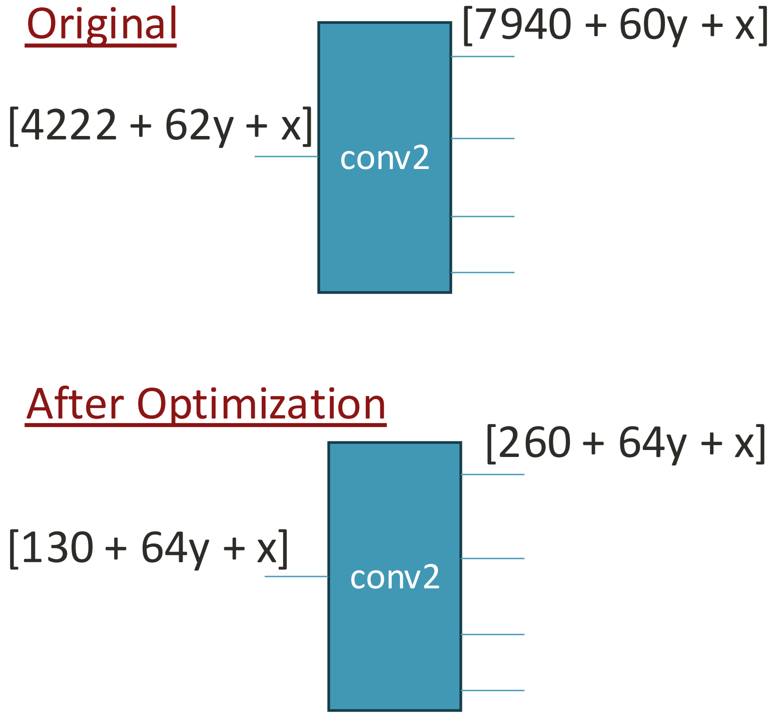
Qiaoyi Liu. *Compiling Applications to Reconfigurable Push Memory Accelerators*. In Stanford Doctoral Dissertation, 2023.

Unified Buffer: Clockwork scheduling

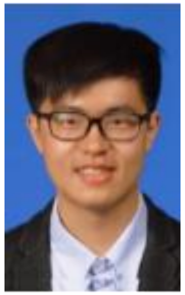


Clockwork scheduling:

- Polyhedral analysis in Clockwork, created by Dillon + Qiaoyi
- Purpose: statically determine cycle times when all stores/loads occur
- Calculation: determined by dependencies, then minimize latency
- Buffer size: Number of cycles between store and last load for each pixel

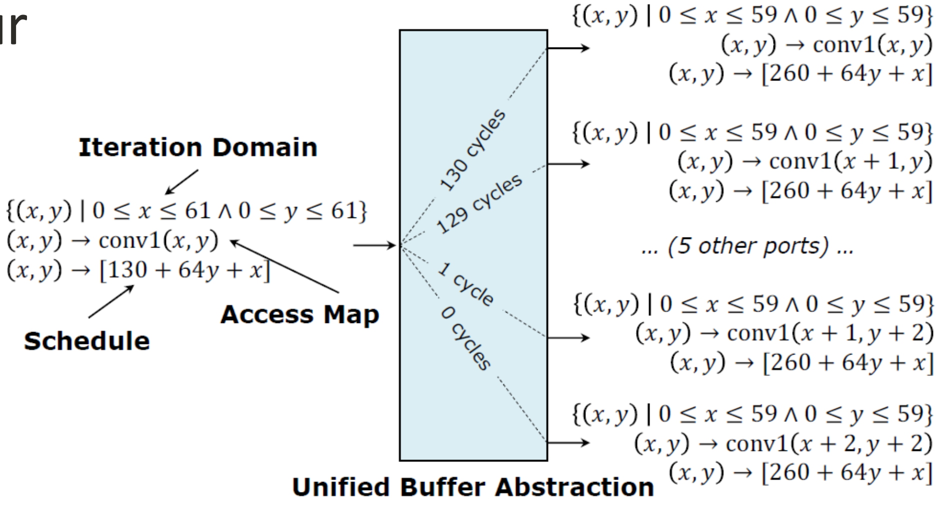


Unified Buffer: Clockwork scheduling



Clockwork scheduling:

- Polyhedral analysis in Clockwork, created by Dillon + Qiaoyi
- Purpose: statically determine cycle times when all stores/loads occur
- Calculation: determined by dependencies, then minimize latency
- Buffer size: Number of cycles between store and last load for each pixel


Iteration Domain
$\{(x,y) \mid 0 \le x \le 61 \wedge 0 \le y \le 61\}$
$(x,y) \to \mathrm{conv1}(x,y)$
$(x,y) \to [130 + 64y + x]$
Schedule
Access Map
130 cycles
129 cycles
1 cycle
0 cycles
$\{(x,y) \mid 0 \le x \le 59 \wedge 0 \le y \le 59\}$
$(x,y) \to \mathrm{conv1}(x,y)$
$(x,y) \to [260 + 64y + x]$
$\{(x,y) \mid 0 \le x \le 59 \wedge 0 \le y \le 59\}$
$(x,y) \to \mathrm{conv1}(x+1,y)$
$(x,y) \to [260 + 64y + x]$
… (5 other ports) …
$\{(x,y) \mid 0 \le x \le 59 \wedge 0 \le y \le 59\}$
$(x,y) \to \mathrm{conv1}(x+1,y+2)$
$(x,y) \to [260 + 64y + x]$
$\{(x,y) \mid 0 \le x \le 59 \wedge 0 \le y \le 59\}$
$(x,y) \to \mathrm{conv1}(x+2,y+2)$
$(x,y) \to [260 + 64y + x]$
Unified Buffer Abstraction

Dillon Hu, Steve Dai, and Pat Hanrahan.
Clockwork: Resource-efficient static scheduling for multi-rate image processing applications on FPGAs. In FCCM 2021.

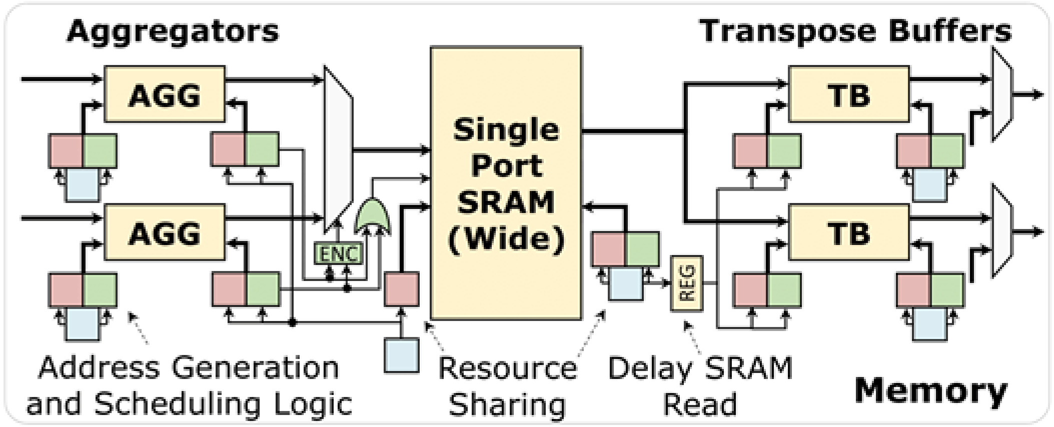
Unified Buffer: mapping to MEM tiles

Mapping to hardware

- Occurs after unified buffer abstraction and scheduling
- **One unified buffer != one memory tile**

Amber CGRA Memory Tile:

- 2048 B capacity
- 2 inputs, 2 outputs
- 6-level affine loopnest
- 4-wide SRAM reads/writes

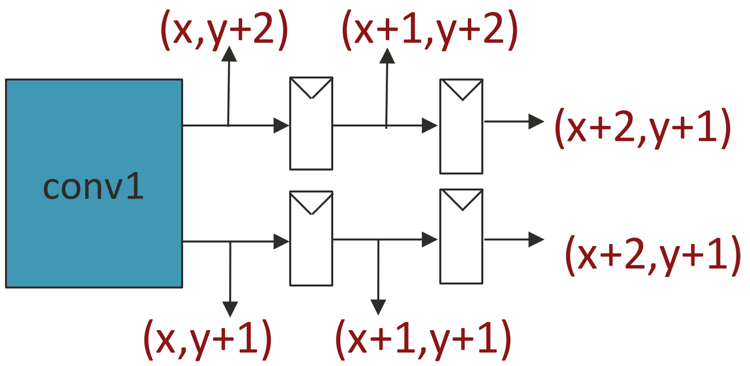


Clockwork: mapping steps

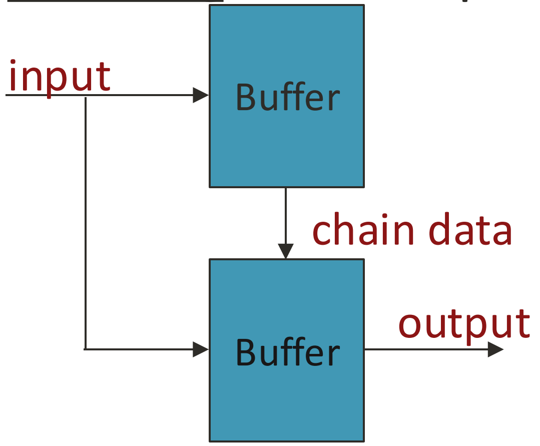
Key steps:

- Wide fetch: align with 4-wide single-port SRAM
- Banking: more IOs

- Shift register optimization: identify locality opportunities



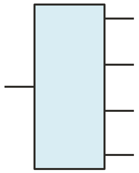
- Chaining: more capacity



Summary

```
app += algorithm;
app.scheduling();
```

Used **Halide** to specify an application using an algorithm, and applied a CGRA schedule to create an efficient application execution on hardware.



Clockwork extracted Unified Buffers to enable us to map to the memories on the CGRA.

Thank You

Thank You

Thank You

Demo notes

Commands

- aha map --apps apps/cascade
- Generated Log
  - Full application (CPU enclosed around accelerator = _hls_target.hw_output)
  - HalideIR printout of cascade for CGRA
  - Runs CPU and clockwork versions and compares generated output images
  - Runs clockwork memory compilation (with logging in mem_cout)
  - Runs MetaMapper

Demo notes

Important files

- Halide generator: algorithm+schedule
  - /aha/Halide-to-Hardware/apps/hardware_benchmarks/apps/cascade/cascade_generator.cpp
- Generated files from compiler
  - /aha/Halide-to-Hardware/apps/hardware_benchmarks/apps/cascade/bin/*
- Input to Clockworks
  - /aha/Halide-to-Hardware/apps/hardware_benchmarks/apps/cascade/bin/cascade_memory.cpp
  - /aha/Halide-to-Hardware/apps/hardware_benchmarks/apps/cascade/bin/cascade_compute.json
- Output of Clockwork and MetaMapper: mapped design with memories and compute
  - /aha/Halide-to-Hardware/apps/hardware_benchmarks/apps/cascade/bin/design_top.json