# Halide to Hardware

Jeff Setter

# Halide Basics

Domain Specific Language (DSL)

- Built as library in C++
- Front-end: algorithm and schedule
- Back-end: HalideIR as AST with loops and compute nodes
- Main repo targets: CPU, GPU, Qualcomm Hexagon DSP

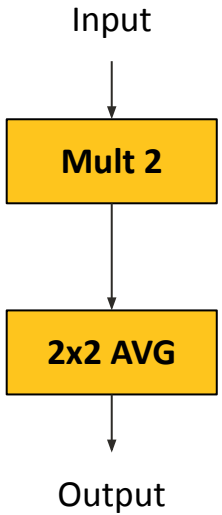
Extension to CGRA

- Github link: https://github.com/StanfordAHA/Halide-to-Hardware/

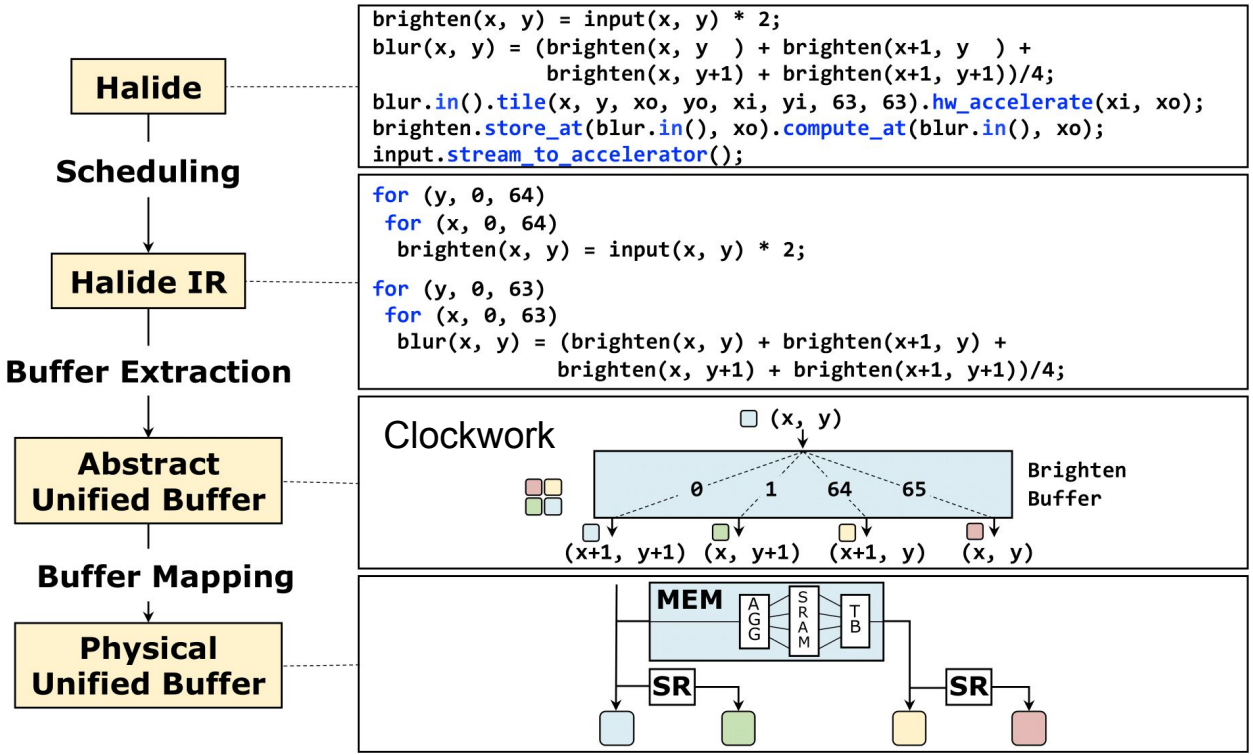
# Example Application

```
// Algorithm
brighten(x, y) = input(x, y) * 2;
RDom r(0, 2, 0, 2);
blur(x, y) += brighten(x + r.x, y + r.y) / 4;

// Schedule
blur.tile(x,y, xi,yi, xo,yo, 63,63)
    .hw_accelerate(xi, xo);
brighten.store_at(blur, xo).compute_at(blur, xo);
brighten.update().unroll(r.x).unroll(r.y);
input.stream_to_accelerator();
```



# Overall Flow



# Applications

Location: apps/hardware_benchmarks/

Categories

- handcrafted: no Halide front-end, just CoreIR (constructed manually)
- tests: simple examples to test operators and patterns
- apps: full examples with relevant input and output

Code

- Single algorithm
- Multiple schedules: CPU, multiple versions of CGRA
- All end-to-end examples

# Compiler Implementation

Location: src/

Important files

- Func.h: location of scheduling primitives
- Lower.cpp: sequence of compilation passes
- Compiler pass files
  - ExtractHWBuffers.cpp
  - HWBufferRename.cpp
  - HWBufferSimplifications.cpp
- Codegen files
  - CodeGen_Clockwork_Target.cpp
  - CoreIRCompute.cpp

# Compiler Passes

Front-end

- Converts algorithm + schedule into HalideIR
- All following passes operate on HalideIR

CGRA/hardware passes

- `extract_hwaccelerators`: identify and modify loops around accelerator bounds
- `rename_hwbuffers`: identify buffers, ROMs, and consts
- `unroll_loops_and_merge`: merges updates to create large compute kernels
- `inline_memory_constants`: removes buffers associated with constants

# Codegen to CGRA

Generated files: CPU file, Clockwork memory file, CoreIR compute file

Clockwork and CPU generation: `src/Codegen_Clockwork_Target.cpp`

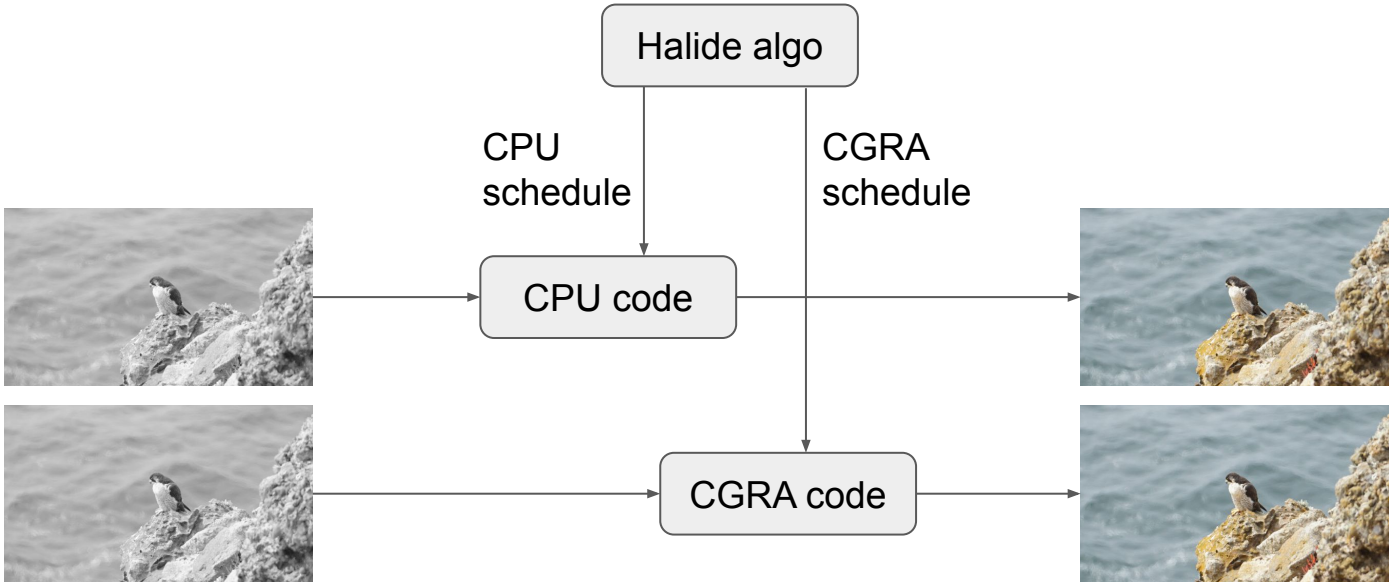
- `For`: adds a for loop
- `Store`: adds load, compute node, and store based on statement

CoreIR generation: `src/CoreIRCompute.cpp`

- Separate codegen for compute nodes
- Creates CoreIR operators based on HalideIR operators
- Generates CoreIR defining each compute kernel (inputs, outputs, ops, wires)

# Testing Strategy

- Performed as end-to-end tests
- Run full image on CPU and CGRA, then compare output values
- Currently, little support for compiler pass unit tests



# Usage

Makefile: set of commands for all target files

Compiles files and runs scripts based on defined dependencies

- make **compiler**: recompile Halide compiler
- make **clockwork**: codegen application files for clockwork
- make **compare**: compare output images from cpu + clockwork
- make **mem**: create CoreIR for application
- make **bin/graph.png**: create visualization from CoreIR graph

# Supported Applications / Halide patterns

Computation

- All Halide operators supported
- Instruction selection: basic implementation supported (MAC)

Addressing

- Perfectly nested loops
- Affine addressing: `mem(2*x-1, y+3)`

# Unsupported Halide patterns

Data-dependent access

- Access pattern: `mem(mem2(x, y)) += 1;`
- add_dynamic_load and add_dynamic_store: Clockwork doesn't support these
- App examples: histogram, bilateral

Iterative Memory access

- Access pattern: `mem(x, y) = mem(x-1, y) + 5;`
- iterative operations: might want special consideration in Clockwork scheduling
- App examples: smith_waterman, fft

# Future Directions: auto-scheduling

Goal: generate CGRA schedule for applications automatically

My suggested implementation strategy

- Declarative scheduling: reframe scheduling syntactic sugar for desired metric
  - For example, tile → create loop nests
  - Labeling to construct memory hierarchy
  - User-specified output rate for entire application (rate-matched for each stage)
- Hardware resource estimator
- User-provided resource constraints
  - How many PE tiles and MEM tiles
- Halide generates its own schedule

# Future Directions: Halide-generated control code

Missing step in push-button flow: control CPU code

- Code between CGRA execution and Host CPU
- Responsibility: DMA, configuring CGRA, executing multiple tiles
- Similar to generated CPU testbench code, extend codegen to create necessary code for control CPU

# Future Directions: Full End-to-End DNN Applications

With control code generated, full end-to-end DNN applications can be created

- Involves multiple configuration phases
- Iterate for multiple DNN layers
- GLB: Must be careful about GLB placement between DNN layer execution
- Halide: currently is able to describe multiple accelerator layers
- Codegen: control CPU codegen would be extended for multiple configurations

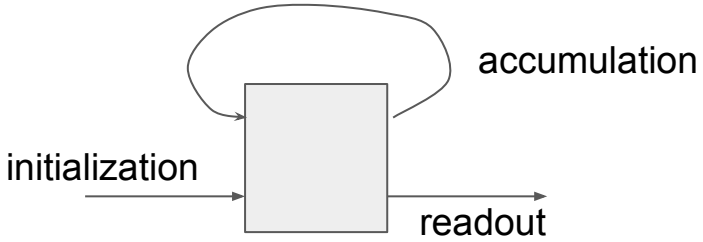
# Future Directions: More Compiler Optimizations

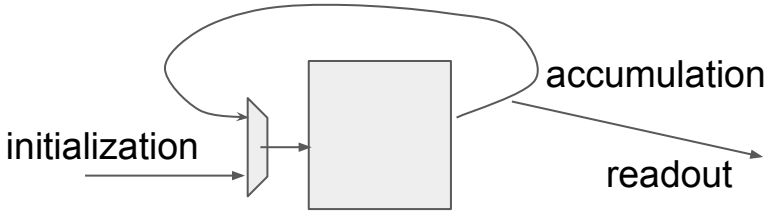
Memory Sharing

- Place multiple buffers in a single memory tile
- Would require hardware change to access generators

DNN Phase Merging

- DNNs use four ports for initialization, accumulation, and output
- Could merge these phases into two total phases if iteration order is the same



# Future Directions: Implement More Halide Scheduling

`parallel()`

- Create multiple instances of an application on the same CGRA fabric
- Helpful for PnR

`vectorize()`

- Only possible if compute units supported 8-bit math (or lower bitwidths)
- Increases utilization of PEs for apps using lower bitwidths