Halide Scheduling for our CGRA

Jeff Setter

Halide Overview

- Halide: a domain-specific language (DSL) for generating fast image and tensor processing applications

- Separation of the algorithm from the schedule
  - Algorithm: specify the arithmetic to calculate the output pixel values
  - Schedule: loop transformations needed to do the application quickly

Example Application: Gaussian Blur

```
Input<Buffer<uint8_t>>  input{"input", 2};
Output<Buffer<uint8_t>> output{"output", 2};

// Declare Funcs, reduction domain, and cast input to u16
Func hw_input, blur_unnormalized, blur, hw_output;
RDom win(0, 3, 0, 3);
hw_input(x, y) = cast<uint16_t>( input(x, y) );

// Use a Reduction Domain to index the Func for an update (convolution)
blur_unnormalized(x, y) = cast<uint16_t>(0);
blur_unnormalized(x, y) += kernel(win.x, win.y) *
                           hw_input(x+win.x, y+win.y);

// Normalize the output value
blur(x, y) = blur_unnormalized(x, y) / 256;
hw_output(x, y) = blur(x, y);
output(x, y) = cast<uint8_t>( hw_output(x, y) );
```

Halide Scheduling Structure

- Embedded DSL: Schedule behaves as its own language that modifies the algorithm

- C++ backend: C++ language can be used
  - `printf` for debugging purpose
  - `if` statements for conditional scheduling

- Output centric: schedules should be written from output to inputs
  - Easier to understand the bounds analysis this way
  - Some scheduling primitives reference output Funcs
  - However, this is visual and the order of scheduling primitives typically does not matter

Hardware Target: Amber CGRA

- Separate CGRA schedule based on the target
    - `if (get_target().has_feature(Target::Clockwork)) {  ...`
- Different scheduling parameters
    - Tiling sizes are different based on the memory hierarchy
    - Memory hierarchy is more explicit
- Scheduling primitives have different interpretations for hardware
    - Based on HLS interpretations
    - Loop iteration is performed on separate cycles
    - Unroll is used for duplication
    - All compute operators correspond to a unique compute resource

Defining Hardware Boundaries

Output: `hw_accelerate(xi, xo)`

- Second arg: loop boundary
- Use `bound()` to set size

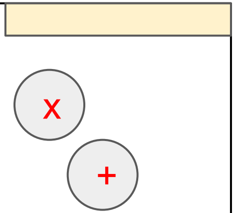
Input: `stream_to_accelerator()`

- Used on each input
- All computation before (or after an accelerator output) is performed on the host processor

```
output.bound(x,0,3840).bound(y,0,2160);

hw_output
  .compute_root()
  .hw_accelerate(x, Var::outermost());

hw_input
  .stream_to_accelerator();
```

```
start_xcel()
```

```
for y:
  for x:
    for win.y:
      for win.x:
```



Strip Mining + Loop Reordering

`split(x, xo, xi, 56)`

- Create two loops from a single loop
- Inner loop has specified size
- Use to fit memories given storage constraints

`reorder(win.x, win.y, xi, xo, y)`

- Interchange loops from innermost to outermost

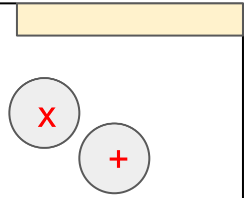
`tile(x, y, xo, yo, xi, yi, 56, 56)`

- Syntactic sugar for `split` and `reorder`
- Used to match target's memory size

```
output.bound(x,0,3840).bound(y,0,2160);

hw_output.compute_root()
  .tile(x, y, xo, yo, xi, yi, 62, 62)
  .reorder(xi, yi, xo, yo)
  .hw_accelerate(xi, xo);

hw_input
  .stream_to_accelerator();
```

```
for yo:
  for xo:
    start_xcel()
```

```
for yi:
  for xi:
    for win.y:
      for win.x:
```



Compute unrolling

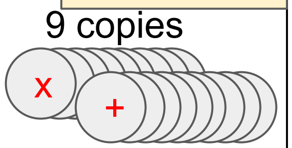
`unroll(win.y, 3)`

- Duplicates the loop body
- Creates more compute hardware, and runs for fewer cycles
- RDoms: unroll works well with reduction domains to go 1 pixel/cycle
- Parallelism: Used on loops to go beyond 1 pixel/cycle
    - Should unroll all loops to match rates

```
output.bound(x,0,3840).bound(y,0,2160);

hw_output.compute_root()
  .tile(x, y, xo, yo, xi, yi, 62, 62)
  .hw_accelerate(xi, xo);

blur_unnormalized.update()
  .unroll(win.y).unroll(win.x);

hw_input
  .stream_to_accelerator();
```

```
for yo:
  for xo:
    start_xcel()
```

```
for yi:
  for xi:
```



Modified Application: 2x Gaussian Blur

```
Input<Buffer<uint8_t>>  input{"input", 2};
Output<Buffer<uint8_t>> output{"output", 2};

// Declare Funcs, reduction domain, and cast input to u16
Func hw_input, blur_unnormalized1, blur1, blur_unnormalized2, blur2, hw_output;
RDom win(0, 3, 0, 3); // starts at 0, width is 3
hw_input(x, y) = cast<uint16_t>( input(x, y) );

// Use a Reduction Domain to index the Func for an update (convolution)
blur_unnormalized1(x, y) = cast<uint16_t>(0);
blur_unnormalized1(x, y) += kernel(win.x, win.y) *
                            hw_input(x+win.x, y+win.y);
blur1(x, y) = blur_unnormalized1(x, y) / 256;

// Insert a second convolution
blur_unnormalized2(x, y) = cast<uint16_t>(0);
blur_unnormalized2(x, y) += kernel(win.x, win.y) *
                            blur1(x+win.x, y+win.y);
blur2(x, y) = blur_unnormalized2(x, y) / 256;

// Normalize the output value
hw_output(x, y) = blur2(x, y);
output(x, y) = cast<uint8_t>( hw_output(x, y) );
```

Creating Memories

`compute_at(hw_output, xo)`

- Creates a temporary memory
- Should be used for all stencils
- Loop level should all be the same; Clockwork will handle loop fusion
    - `store_at` is unnecessary

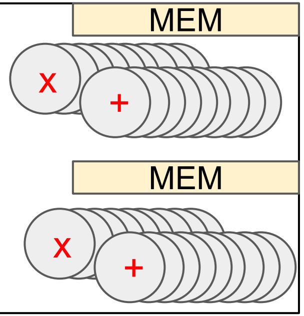
```
output.bound(x,0,3840).bound(y,0,2160);
hw_output.compute_root()
  .tile(x, y, xo, yo, xi, yi, 60, 60)
  .hw_accelerate(xi, xo);
```

```
blur_unnormalized2.compute_at(hw_output, xo);
blur_unnormalized2.update()
  .unroll(win.x).unroll(win.y);
blur_unnormalized1.compute_at(hw_output, xo);
blur_unnormalized1.update()
  .unroll(win.x).unroll(win.y);
```

```
hw_input
  .stream_to_accelerator();
```

```
for yo:
  for xo:
    start_xcel()
```

```
for yi:
  for xi:
```

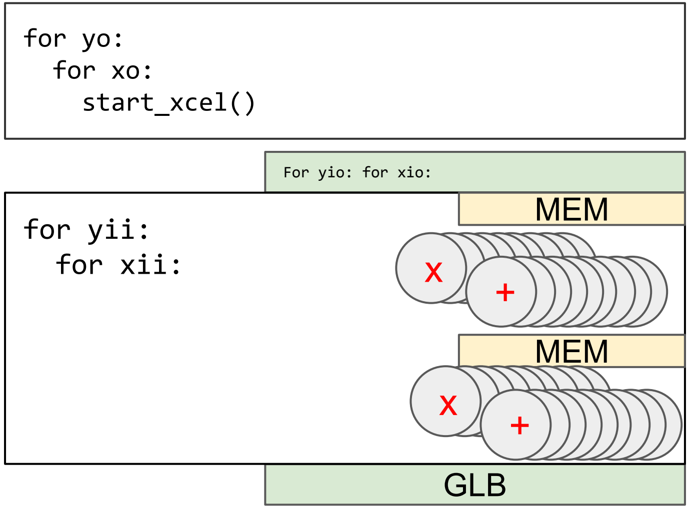


Memory Hierarchy

`hw_output.in().store_in(MemoryType::GLB)`

- `in()` creates a copy
  - Essentially f `f_copy(x, y) = f(x, y)`
- Output and inputs should be buffered
- Hierarchy: `in()` can be cascaded to do multiple copies
  - `store_in()` can label the memory
- Input: use `accelerator_input()`

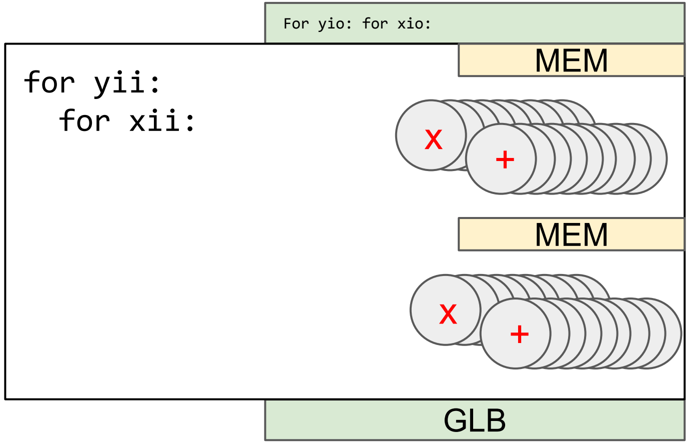
```
hw_output.in().compute_root()
  .tile(x, y, xo, yo, xi, yi, 60, 60)
  .hw_accelerate(xi, xo)
  .store_in(MemoryType::GLB);
hw_output
  .tile(x, y, xio, yio, xii, yii, 60, 60)
  .compute_at(hw_output.in(), xo);
blur_unnormalized{1,2}.compute_at(hw_output, xio);
blur_unnormalized{1,2}.update()
  .unroll(win.x).unroll(win.y);
hw_input.in().in().compute_at(hw_output, xio)
hw_input.in().compute_at(hw_output.in(), xo);
               .store_in(MemoryType::GLB);
hw_input.compute_root().accelerator_input();
```



Memory Hierarchy for Cascade App

```
output.bound(x,0,3840).bound(y,0,2160);

hw_output.in().compute_root()
  .tile(x, y, xo, yo, xi, yi, 60, 60)
  .hw_accelerate(xi, xo)
  .store_in(MemoryType::GLB);
hw_output
  .tile(x, y, xio, yio, xii, yii, 60, 60)
  .compute_at(hw_output.in(), xo);

blur_unnormalized2.compute_at(hw_output, xio);
blur_unnormalized2.update()
  .unroll(win.x).unroll(win.y);

blur_unnormalized1.compute_at(hw_output, xio);
blur_unnormalized1.update()
  .unroll(win.x).unroll(win.y);

hw_input.in().in().compute_at(hw_output, xio);
hw_input.in().compute_at(hw_output.in(), xo)
             .store_in(MemoryType::GLB);
hw_input.compute_root().accelerator_input();
```

```
for yo:
  for xo:
    start_xcel()
```



Let’s look at some other applications

Image Processing application:

```
camera_pipeline
```

DNN Application:

```
resnet_output_stationary
```

Let’s try a separable blur

```
// Separable blur algorithm
RDom win_x(0, 3);
RDom win_y(0, 3);
hw_input(x, y) = cast<uint16_t>(input(x, y));

// Do conv in x direction, then y
blur_x(x, y) += (hw_input(x + win_x.x, y)) / 3;
blur_y(x, y) += (blur_x(x, y + win_y.x)) / 3;

// Set the output
hw_output(x, y) = blur_y(x, y);
output(x, y) = cast<uint8_t>( hw_output(x, y) );
```

```
// Scheduling primitives
bound(...)
stream_to_accelerator()
hw_accelerate(...)

tile(...)

update().unroll(...)

compute_root()
compute_at(...)
```

Separable blur schedule

```
// Separable blur algorithm
RDom win_x(0, 3);
RDom win_y(0, 3);
hw_input(x, y) = cast<uint16_t>(input(x, y));

// Do conv in x direction, then y
blur_x(x, y) += (hw_input(x + win_x.x, y)) / 3;
blur_y(x, y) += (blur_x(x, y + win_y.x)) / 3;

// Set the output
hw_output(x, y) = blur_y(x, y);
output(x, y) = cast<uint8_t>( hw_output(x, y) );
```

```
// Schedule
output.bound(x, 0, 62)
      .bound(y, 0, 62);

hw_output.compute_root()
  .tile(x, y, xo, yo, xi, yi, 62, 62)
  .hw_accelerate(xi, xo);

blur_y.compute_at(hw_output, xo);
blur_y.update()
  .unroll(win_y.x);

blur_x.compute_at(hw_output, xo);
blur_x.update()
  .unroll(win_x.x);

hw_input.stream_to_accelerator();
```

Future Directions: warnings on hidden constraints

Gaussian Application

					Unroll Factor			
output width	1	2	3	4	5	6	7	8
48	/	/	x 17	x		x		x
49	/						x	
50	x	/			x			
51	/		x 18					
52	/	/		x				
53	/							
54	x	x 28	x 19			x		
55	/				x			
56	/	/		x 15			x	x
57	/		x 20					
58	x 60	/						
59	/							
60	/	/	/ 21	x 16	x	x		
61	/							
62	x 64	x 32						
63	/		/				x	
64	/	/		x				x
65	/				x			
66	x 68	/	/			x		
67	/							
68	/	/		x				
69	/		x					
70	x 72	x 36			x		x	
71	/							
72	/	/	/	x		x		x

Key	
/	This passes Clockwork generation
x	This passes GLB sim
x 32	This passes GLB sim with a per-bank input width of 32
[blank]	This fails

Two conditions must hold:
output width % unroll == 0
input bank width < 20 || input bank width % 4 == 0

Future Directions: Declarative Hardware Schedule

Can we better match user intent in the hardware schedule?

```
// declarative schedule version
hw_output.hw_accelerate(xo)
      .create_memories({blur, blur_unnormalized})
      .output_rate(8)
      .loopnest(MEM: [(x 56) (y 56)],
                GLB: [(x 1)  (y 1)],
                host:[(x 10) (y 10)]);

hw_input.accelerator_input();
```

Future Directions: Multi-layer DNNs

Inter-layer communication

- Output from one layer becomes an input to the next
- CGRA configuration registers are reconfigured between runs

Halide scheduling

- Perhaps new Halide scheduling is needed
- Need to codegen new collateral for the host processors

Halide Scheduling for CPU

Parallelization: done in a very different way for the CPU

- `parallel()` launches multiple threads
- `vectorize()` uses hardware vector instructions

Temporaries: are much more important and nuanced for the CPU

- `store_at()` specifies at which loop level to create a buffer
- `compute_at()` specifies at which loop to fill the buffer

Future Directions: Scheduling Other Applications

What other applications still need to be scheduled?

- DNN (CONV LAYER, RESNET) - Joey, Taeyoung
- MATMUL - Alex, Kathleen
- NLMEANS - Po-Han, Ritvik
- INTERP - Max, Jake
- LOCAL_LAP - Kalhan, Kavya
- LENSBLUR - Sneha, Jack
- HIST_EQ - Jeff, Chris
- BILATERAL - Jeff
- MAXFILTER - Jeff

Resource Links

Halide-to-Hardware repository:
https://github.com/stanfordaha/Halide-to-Hardware/

Documentation:
https://stanfordaha.github.io/CGRAFlowDoc/halide/writing-schedules.html