Applications for the SoC

Jeff Setter

Rough overview of CGRA application flow

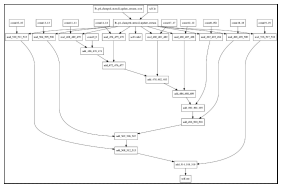
```
Var x, y, xi, yi, xo, yo; Func conv, out;
RDom win(0,3, 0,3);
kernel(x,y) = {{11,12,13},{14,15,16},{17,18,19}};

// algorithm
conv(x, y) += input(x+win.x, y+win.y) *
              kernel(win.x, win.y);
out(x, y) = conv(x, y);

// schedule
conv.update(0).unroll(win.x).unroll(win.y);
out.tile(x,y, xo,yo, xi,yi, 62,62).reorder(xi,yi, xo,yo);
conv.linebuffer();
```

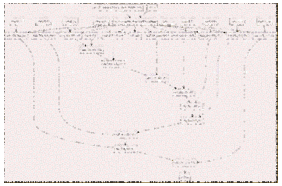
Halide

Compiler



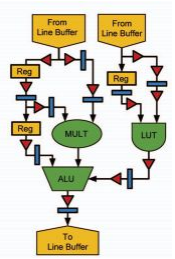
CoreIR

Tech Mapper



Mapped

PnR



CGRA bitfile

Rough overview of CGRA application flow

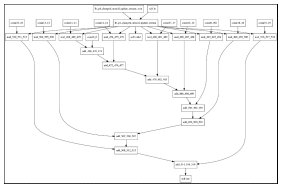
```
Var x, y, xi, yi, xo, yo; Func conv, out;
RDom win(0,3, 0,3);
kernel(x,y) = {{11,12,13},{14,15,16},{17,18,19}};

// algorithm
conv(x, y) += input(x+win.x, y+win.y) *
              kernel(win.x, win.y);
out(x, y) = conv(x, y);

// schedule
conv.update(0).unroll(win.x).unroll(win.y);
out.tile(x,y, xo,yo, xi,yi, 62,62).reorder(xi,yi, xo,yo);
conv.linebuffer();
```

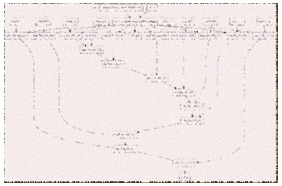
Halide

Compiler



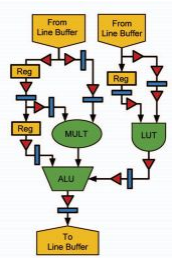
CoreIR

Tech Mapper



Mapped

PnR



CGRA bitfile

[This Talk]

[3] Lake

[2] Automatic Mapper

[1] Garnet

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.

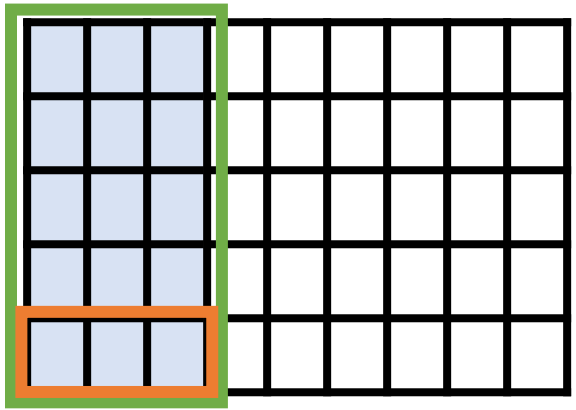
Creates an accelerator from input to output operating on 64x64 image tiles.

Specifies that input should be stored in a line buffer.

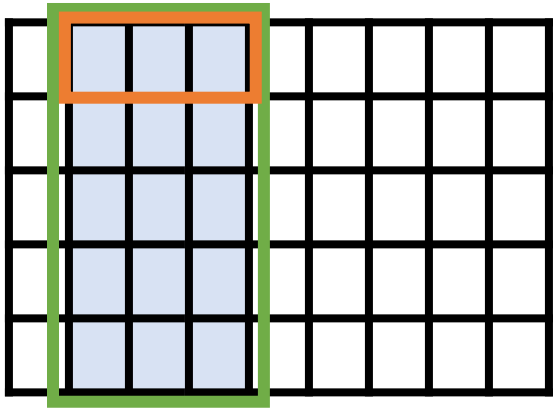
Unrolls the implicit RDom FOR loops; thus 9 multipliers are created.

Unified hardware buffer: combining line buffer and double buffer properties

Block N, cycle 5

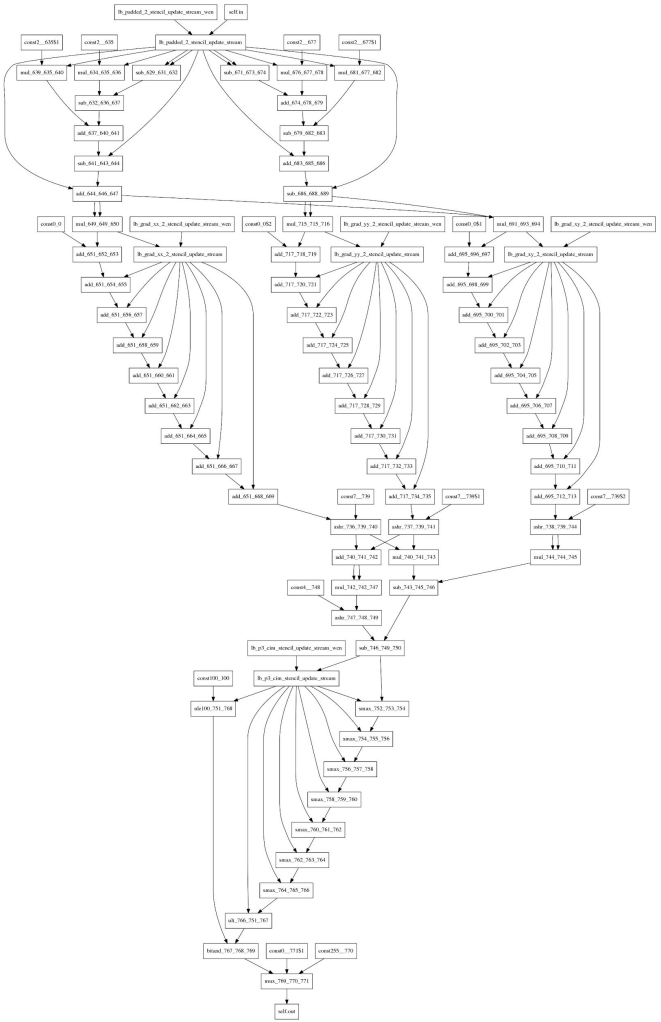


Block N+1, cycle 1



Overlapping output stencils +
multiple cycles to execute each iteration

	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.



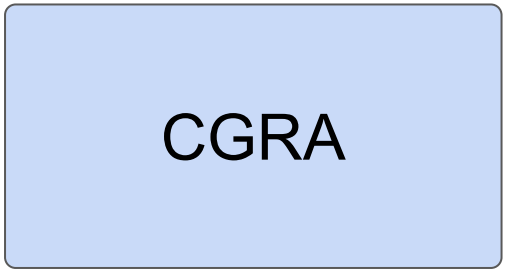
Halide -> CoreIR:
End result is a DAG of computation.

Harris Corner Detector consists of six kernels.

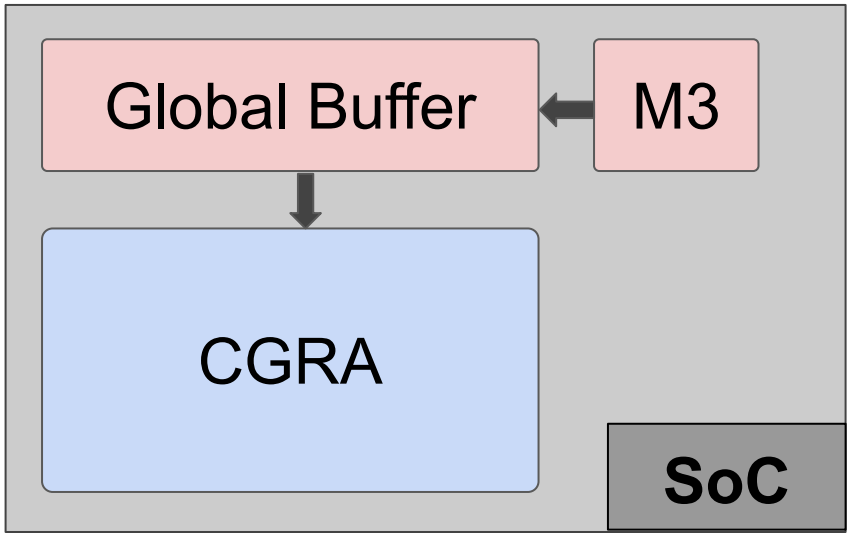
Halide Compiler Status

- Due to limited time, during the tapeout we tested the unified buffer using handcrafted CoreIR examples.
- Compiler progress: extraction of unified buffers in Halide and technology mapping passes are currently being written. Analyses in Halide and mapping are nearly complete, and now working on connecting both together.
- As we finish the automatic mapping, we are starting to think about the next phase in application testing.

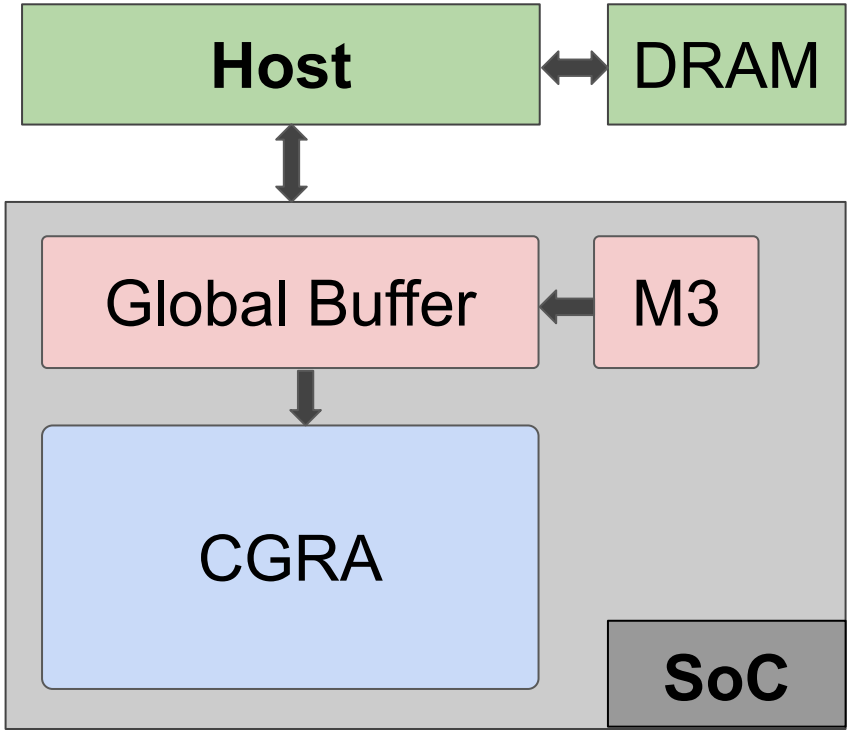
The CGRA is part of a larger system.



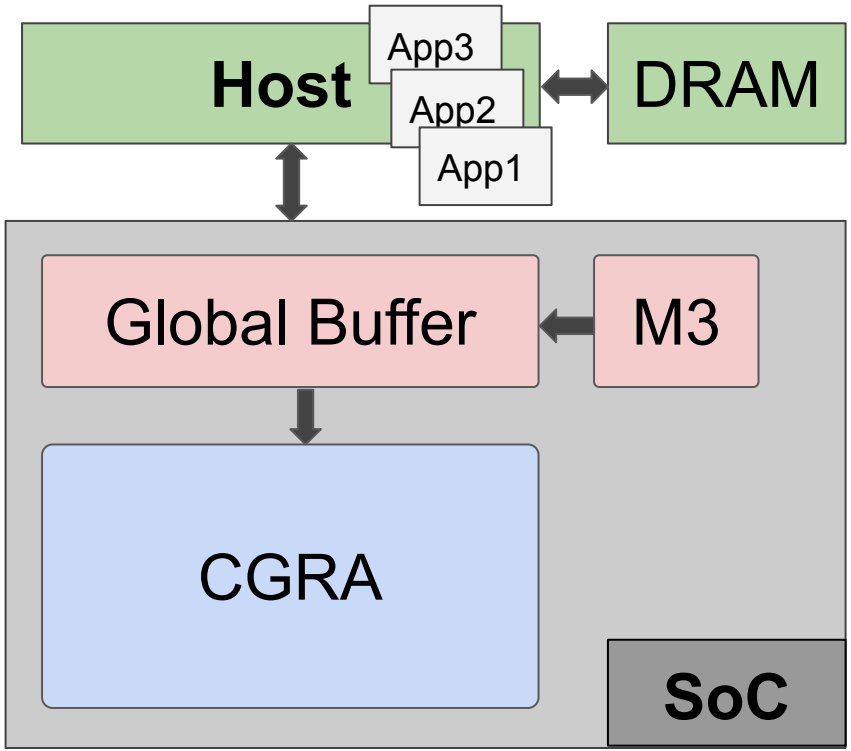
The CGRA is part of a larger system.



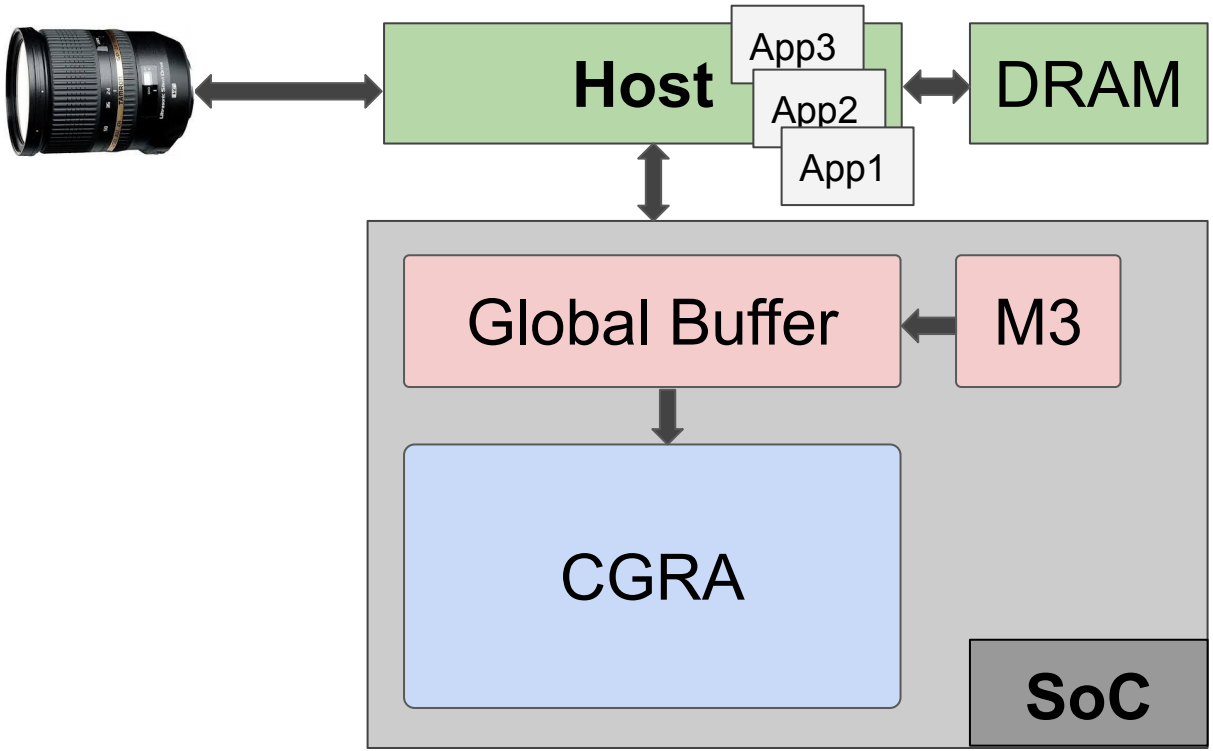
The CGRA is part of a larger system.



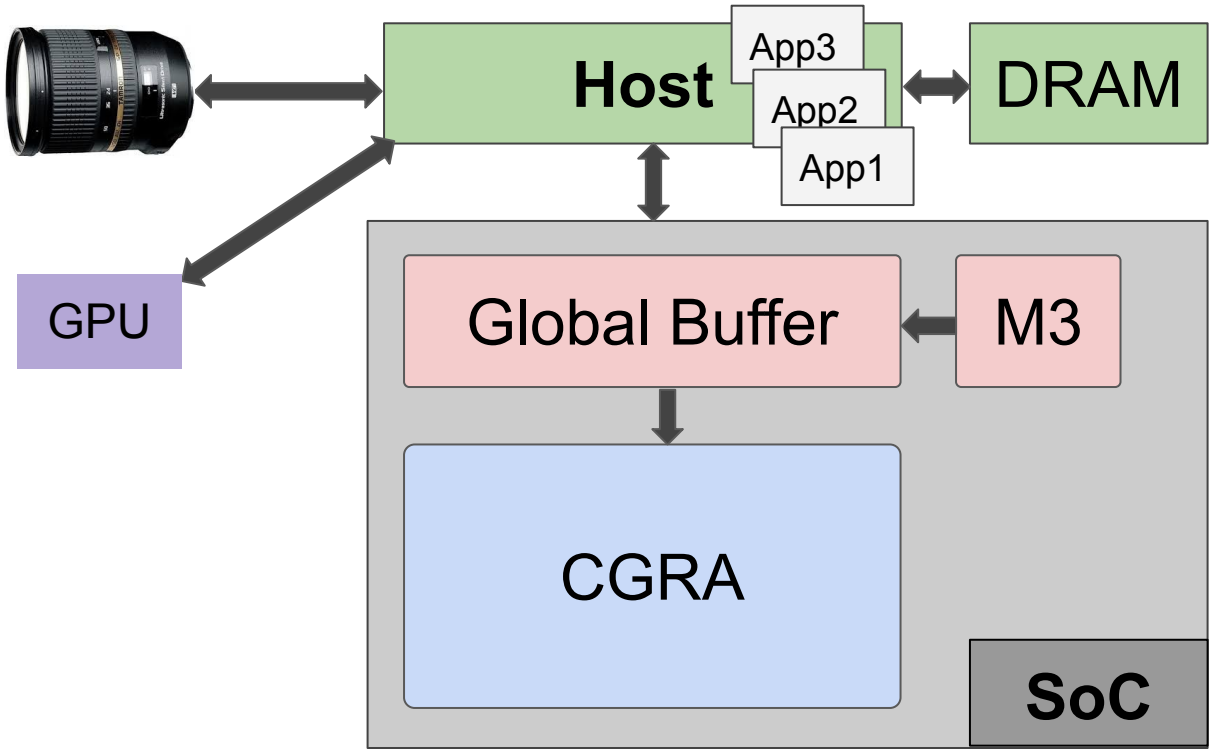
The CGRA is part of a larger system.



The CGRA is part of a larger system.



The CGRA is part of a larger system.



Halide

Three levels in memory hierarchy:

1) host
2) global buffer
3) CGRA memory tiles

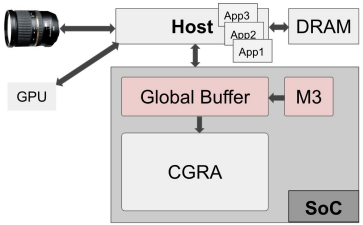
```
/* THE ALGORITHM */
Func hw_input("hw_input");
hw_input(x, y) = cast<uint16_t>(input(x, y));
RDom r(0, 3,   0, 3);
conv(x, y)  += kernel(r.x, r.y) * hw_input(x + r.x, y + r.y);

/* THE SCHEDULE */
// Produce loop levels: host, global buffer, cgra
output.tile(x,y, x_host,y_host, xi,yi, 256-2,256-2);
output.tile(xi,yi, x_gb,y_gb, x_cgra,y_cgra, 64-2,64-2);

// Three buffers: one at host,
//                a copy stage creating the global buffer,
//                another copy stage creating the memory tiles
hw_input.store_root().compute_at(output, x_host);
hw_input.in().store_at(output, x_host).compute_at(output, x_gb);
hw_input.in().in().store_at(output, x_gb).compute_at(output, x_cgra);

// Unroll the computation loops to duplicate hardware
conv.update()
    .unroll(r.x)
    .unroll(r.y);
```

The global buffer feeds the CGRA with data.



- ➢ Execution of application resets the CGRA before feeding valid data into the CGRA.
- ➢ Each CGRA input has its own valid signal.
- ➢ The output data is read, and the global buffer counts the expected number of output signals. This then sends an interrupt to the M3.

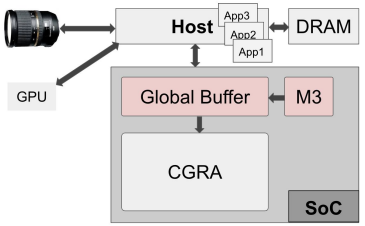
Input signals:

- CGRA output data
- CGRA output data - valid signal

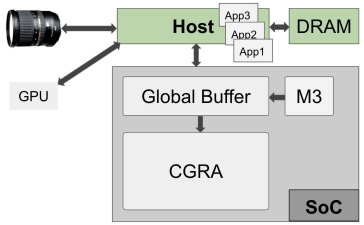
Output signals:

- CGRA input(s) data
- CGRA input(s) data - valid signal(s)
- CGRA reset signal

The global buffer is a unified buffer when address generators are mapped to the M3.



- Global buffer doesn't have full unified buffer functionality
  - Global buffer outputs data only linearly
- Usage of M3 as address generator + global buffer = unified buffer
- Mapping Steps:
  - Global buffer allocation: errors out if global buffer is not large enough
  - IO placement: determine order of buffers from left to right
  - M3 collateral code: for configuration of buffers, IO placement, and app execution



Host code repeats CGRA execution for each tile in the full image.

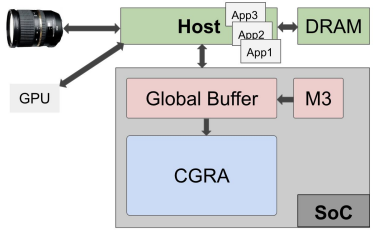
The host code partitions full input and output images in DRAM with tiled image execution on the CGRA.

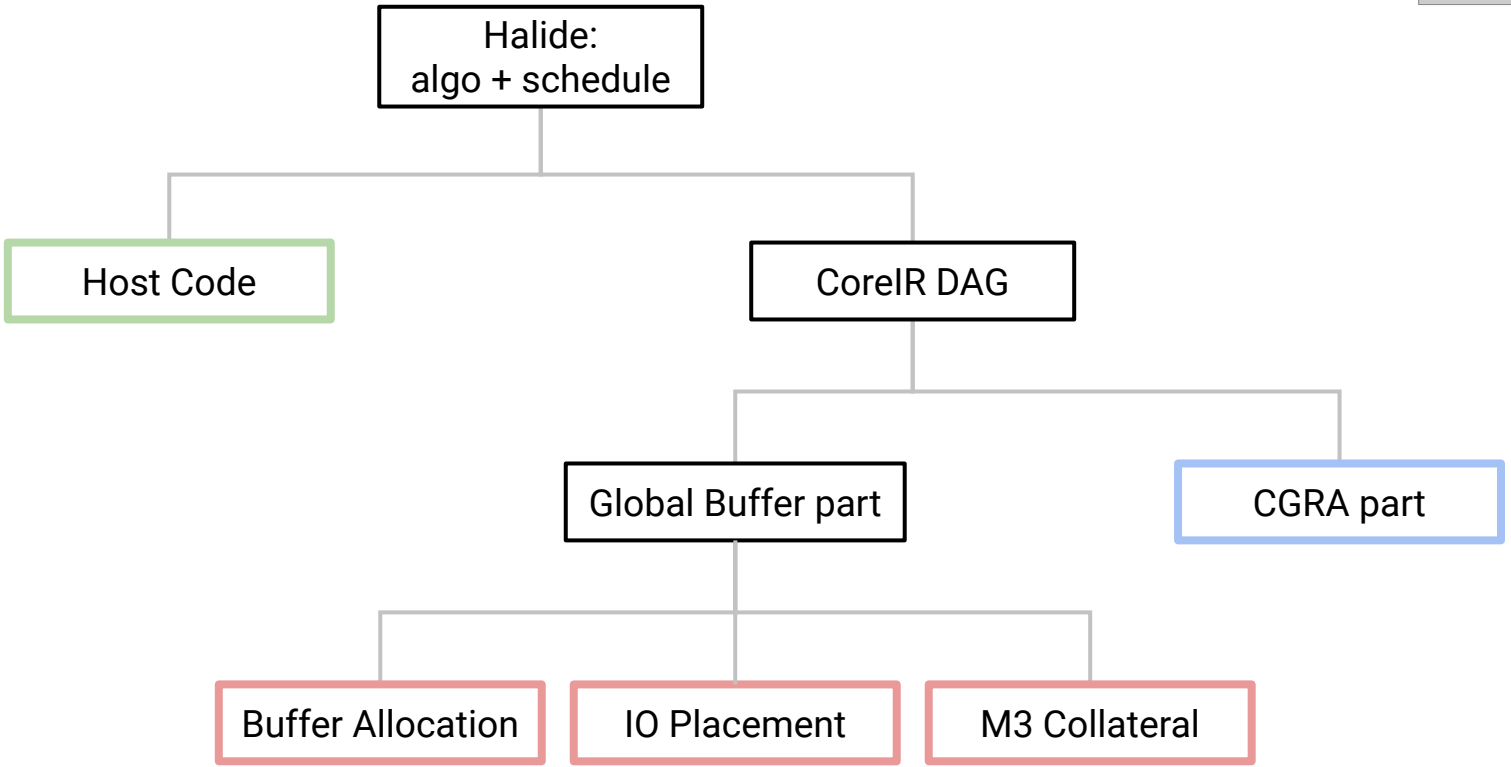
Halide Specification:

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

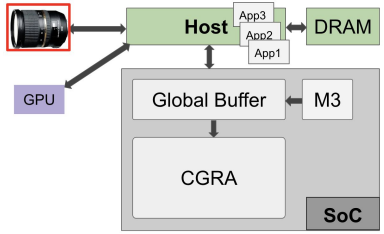
This schedules the CGRA to compute on 256x256 tiles, meaning 8 iterations for a 1024x512 image.

Halide applications compile to the full system.





Beyond Halide: f4graph



Plan is to build a full, working system and then add more features:

- **Halide Scheduling**: Ensure that Halide is scheduled properly for full system.
- **Peripherals**: Use existing code for camera lens and other peripheral connections.
- **Application Processor**: Create application processor runtime for CGRA, including moving data from images captured from the camera lens to the CGRA. These calls should look similar to ones previously used for the FPGA.
- **Drivers**: Create drivers between application processor and CGRA for low-level data movement from DRAM to global buffer using the M3 core.

Conclusion

- For our tapeout, we tested the unified buffer using handcrafted examples. We are working on the Halide compiler to automatically create these examples.

- Unified buffers are used to describe each memory in the hierarchy with unique mappings to the memory tiles and global buffer.

- Our plan is to test the full system (host processor, SoC, and peripherals) using f4graph.