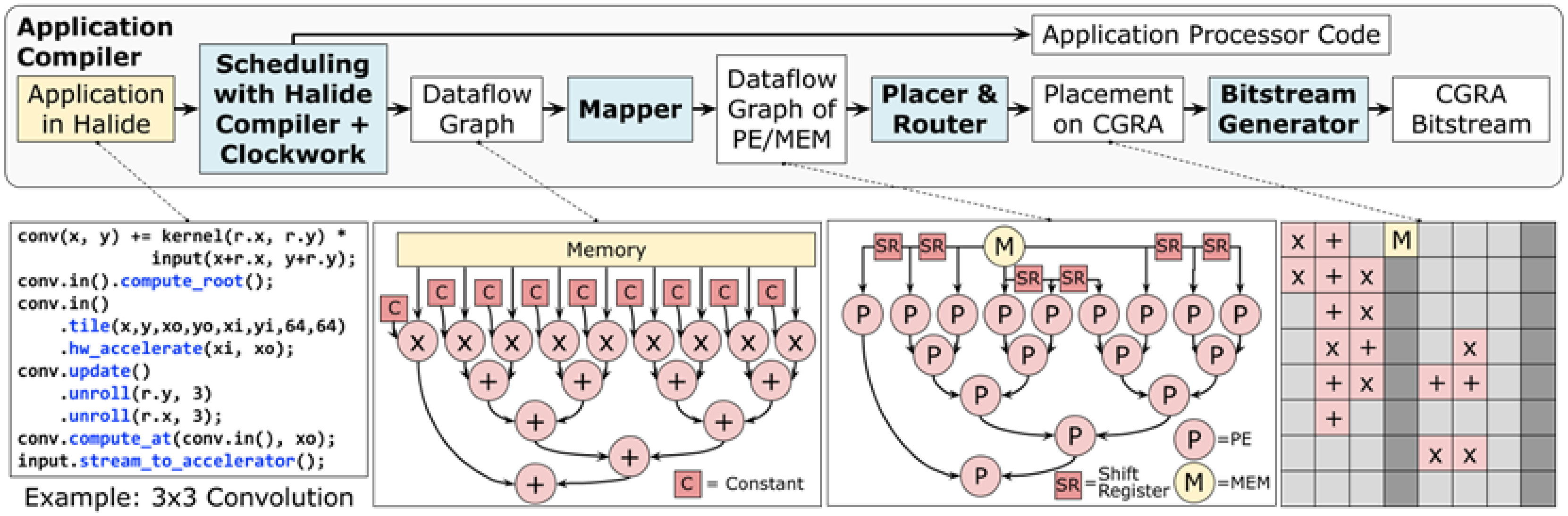
Lake: A Framework for Designing and Automatically Configuring Physical Unified Buffers

Maxwell Strange

Stanford University

Lake – AHA Toolset Context

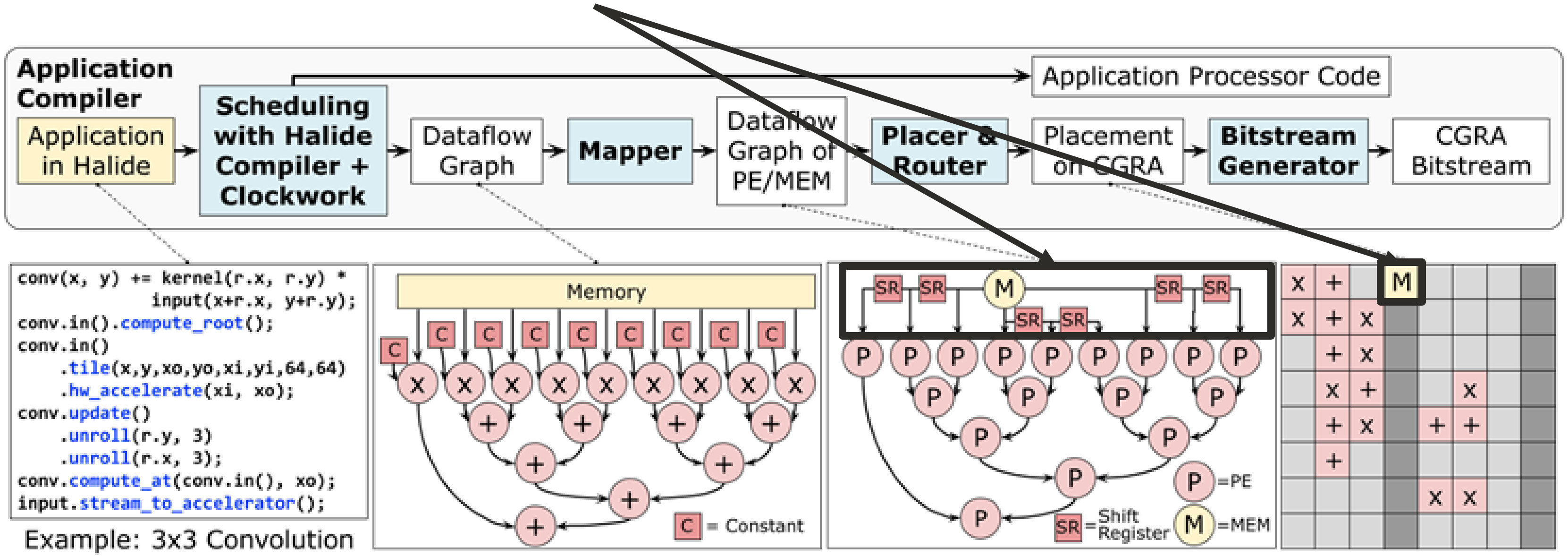
- Lake implements the **Memory (MEM)** portion of the dataflow graph



```
conv(x, y) += kernel(r.x, r.y) *
              input(x+r.x, y+r.y);
conv.in().compute_root();
conv.in()
    .tile(x,y,xo,yo,xi,yi,64,64)
    .hw_accelerate(xi, xo);
conv.update()
    .unroll(r.y, 3)
    .unroll(r.x, 3);
conv.compute_at(conv.in(), xo);
input.stream_to_accelerator();
```

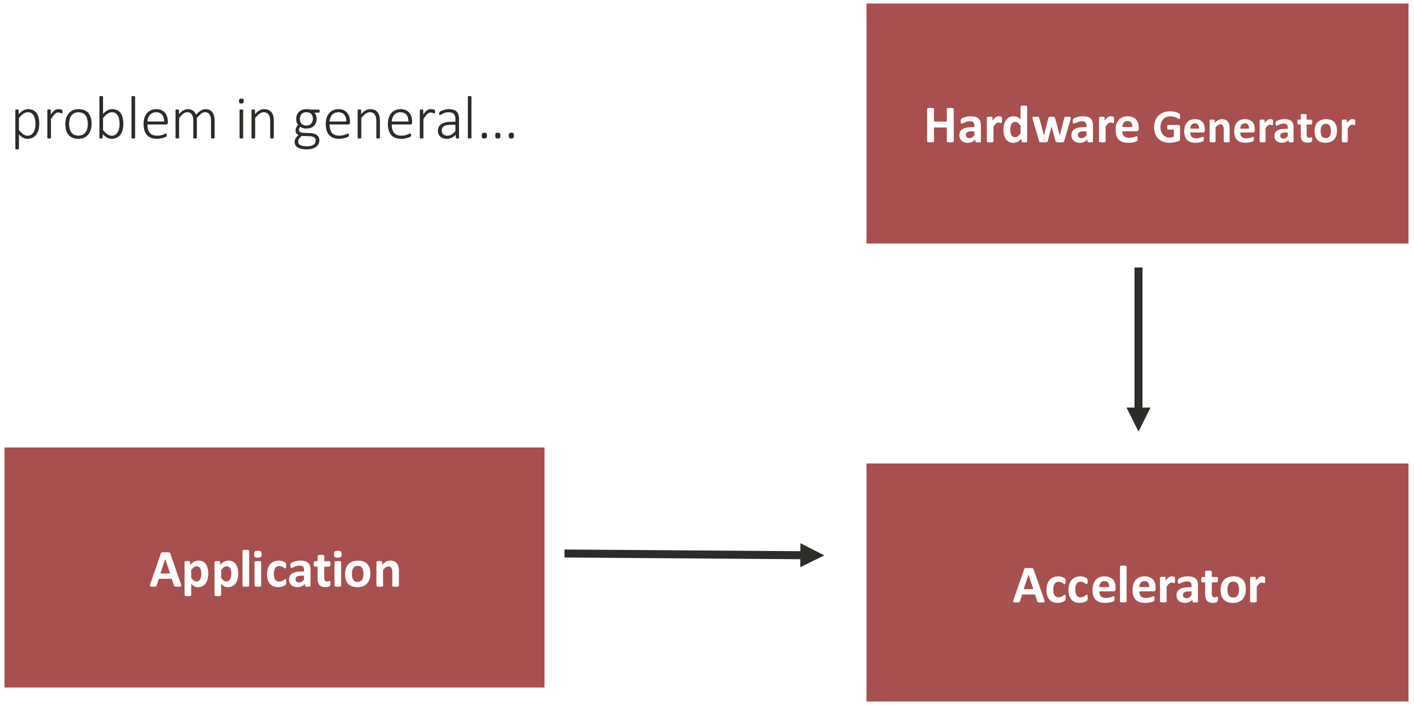
Lake – AHA Toolset Context

- Lake implements the **Memory (MEM)** portion of the dataflow graph

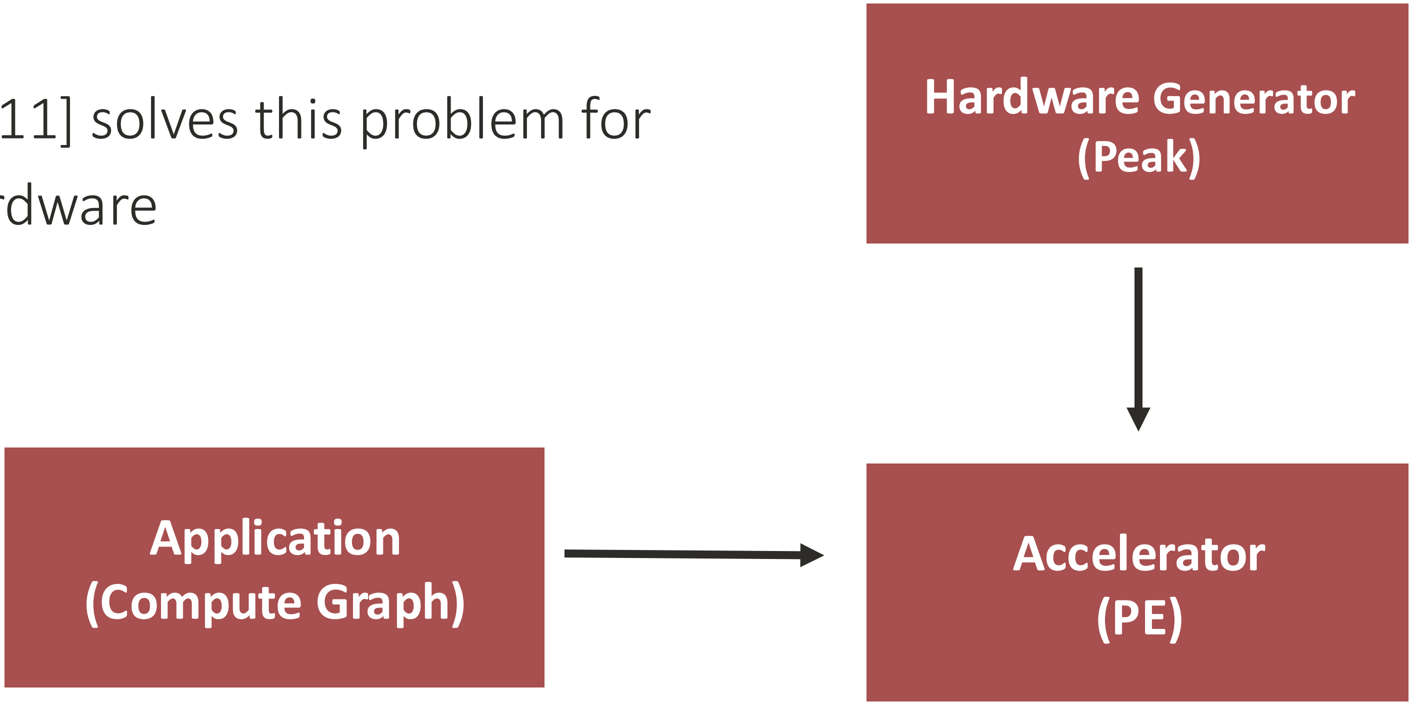


```
conv(x, y) += kernel(r.x, r.y) *
               input(x+r.x, y+r.y);
conv.in().compute_root();
conv.in()
    .tile(x,y,xo,yo,xi,yi,64,64)
    .hw_accelerate(xi, xo);
conv.update()
    .unroll(r.y, 3)
    .unroll(r.x, 3);
conv.compute_at(conv.in(), xo);
input.stream_to_accelerator();
```

Tough problem in general…

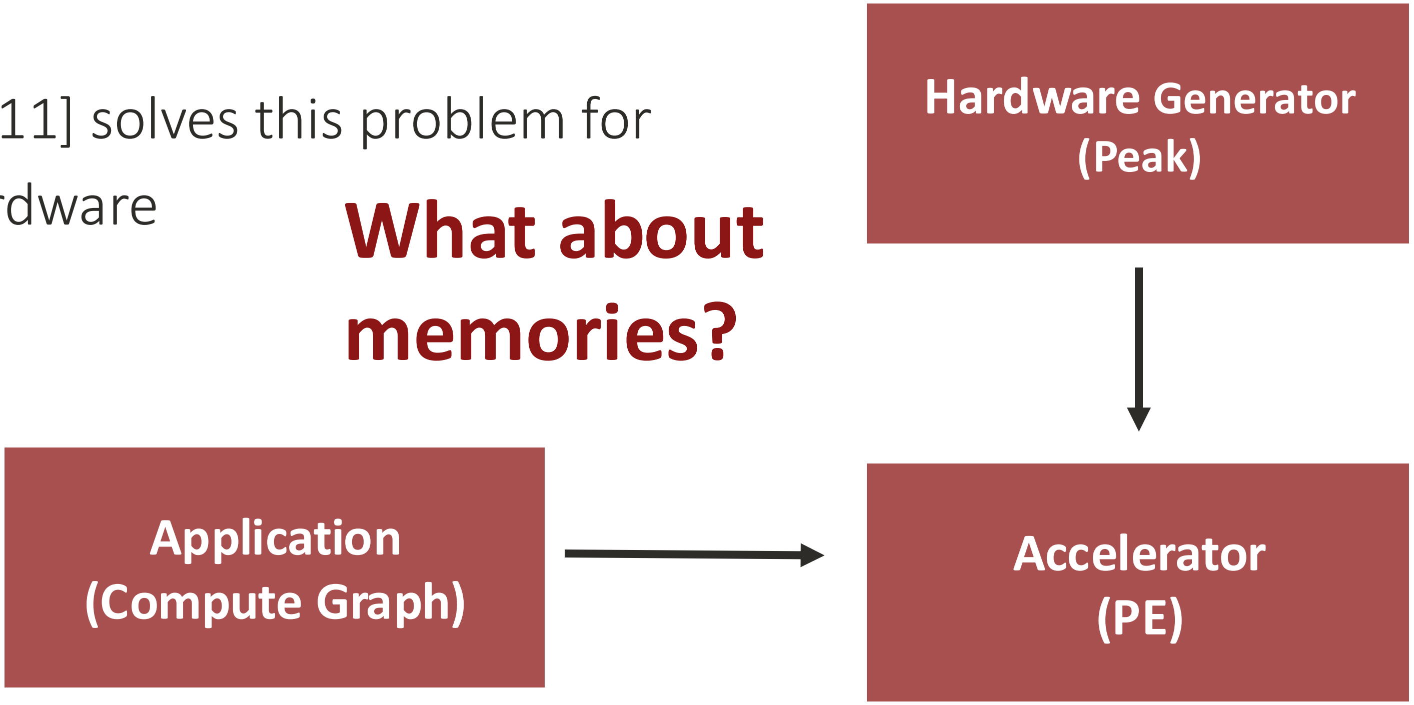


PEak [11] solves this problem for PE hardware



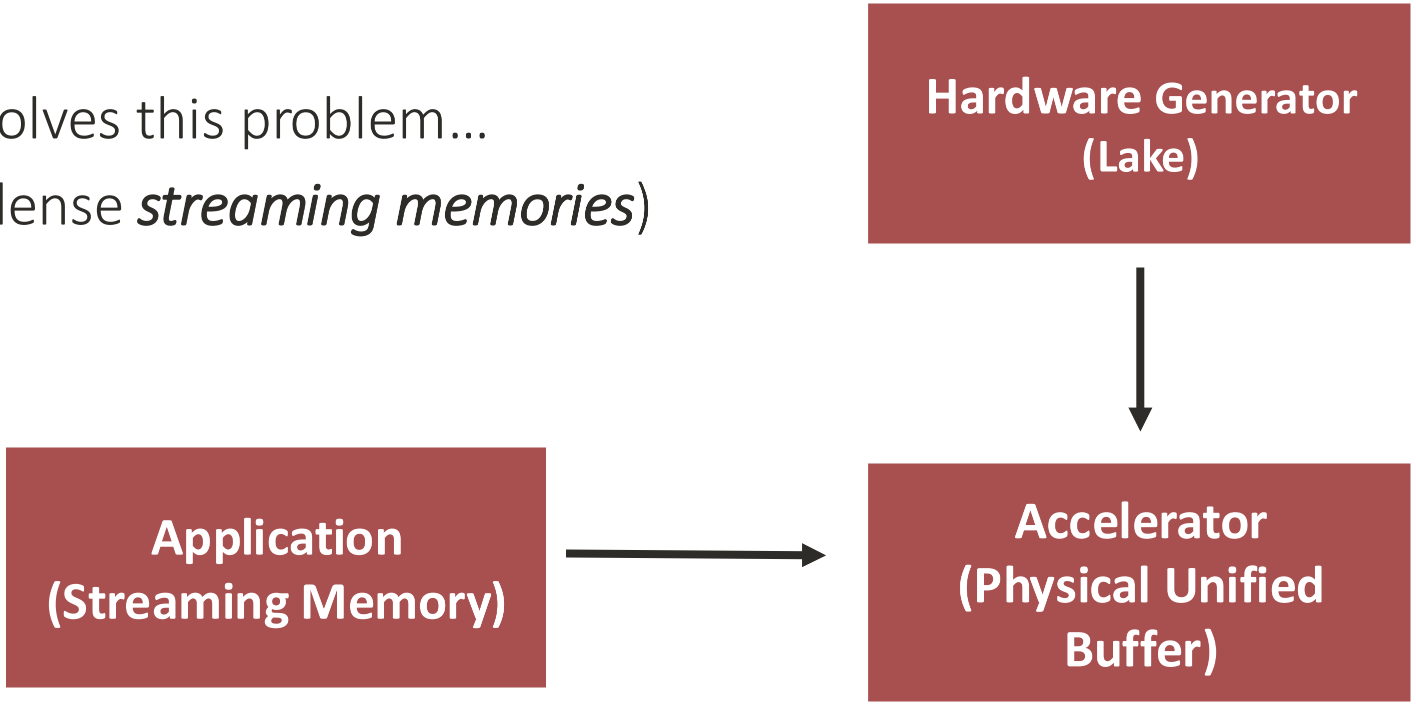
PEak [11] solves this problem for PE hardware

What about memories?



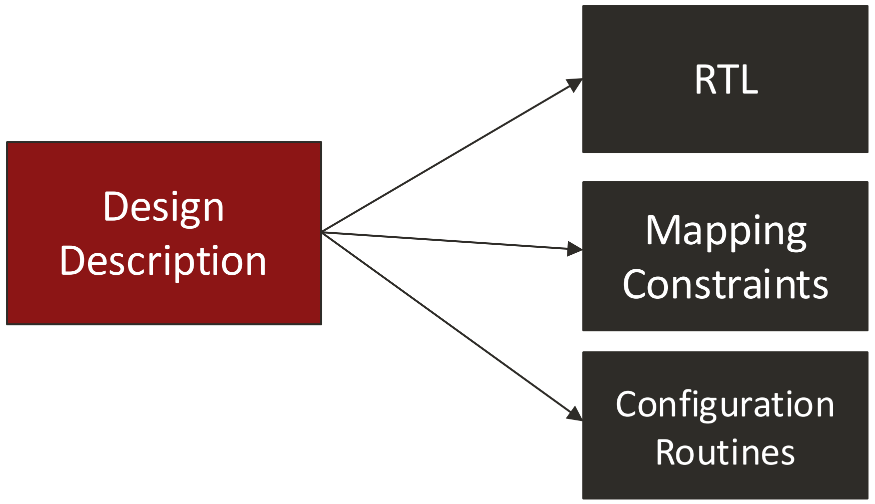
Lake solves this problem…

…for dense ***streaming memories***)


Hardware Generator (Lake)
Application (Streaming Memory)
Accelerator (Physical Unified Buffer)

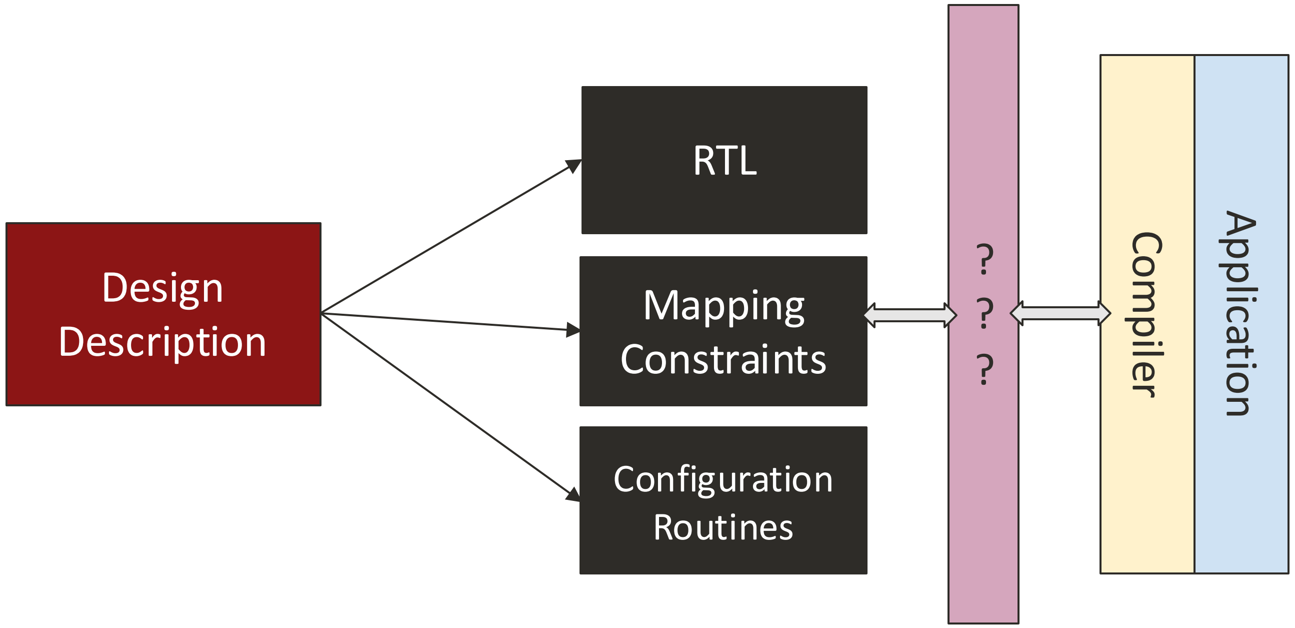
Single Source of Truth

- To enable automatic mapping and design-space exploration, it is crucial to generate mapping collateral from a single source of truth



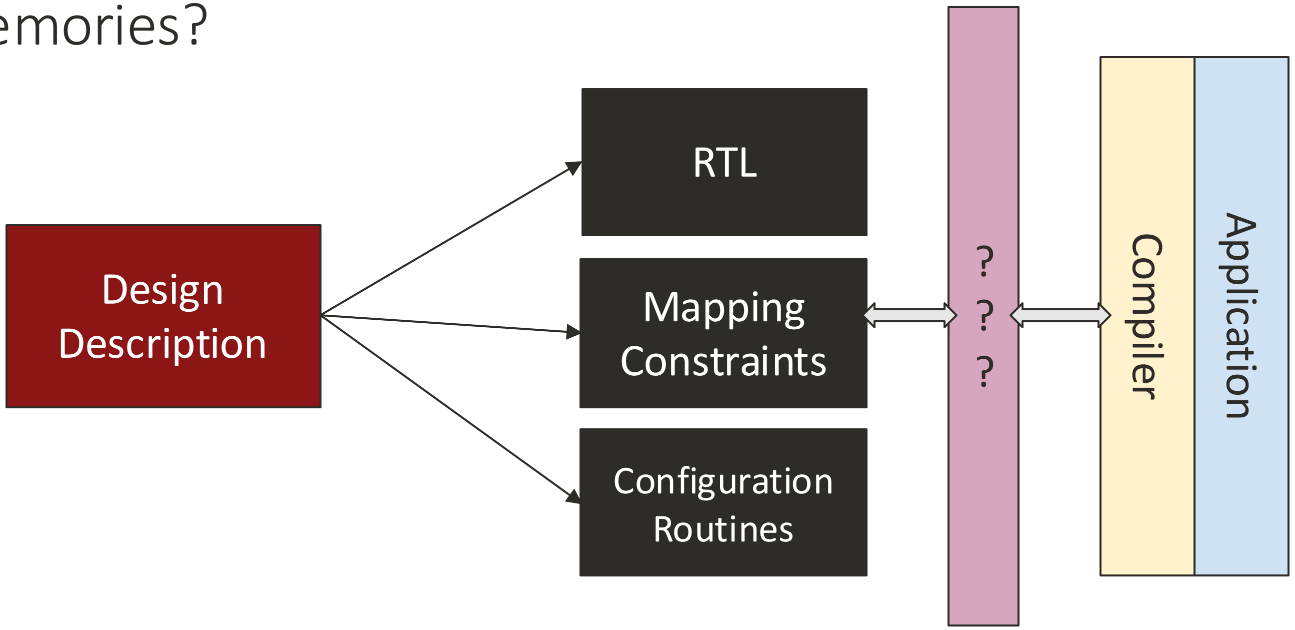
Need the Right Abstraction

- Maintaining compiler compatibility requires a stable HW/SW interface



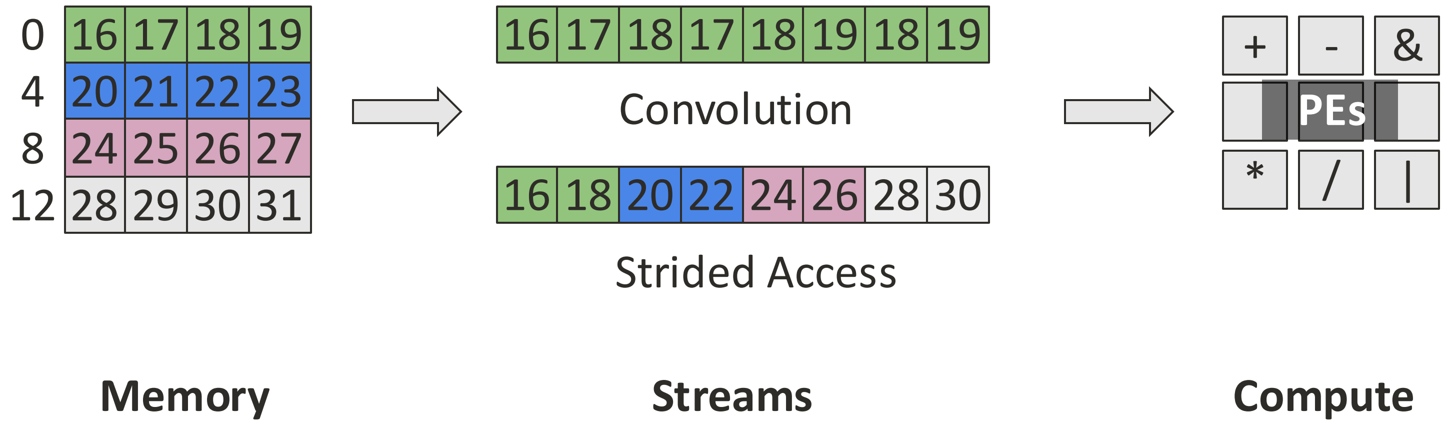
Need the Right Abstraction

- Maintaining compiler compatibility requires a stable HW/SW interface
- PEak used an ISA for PEs – what is analogous to an ISA for streaming memories?



Streaming Memories

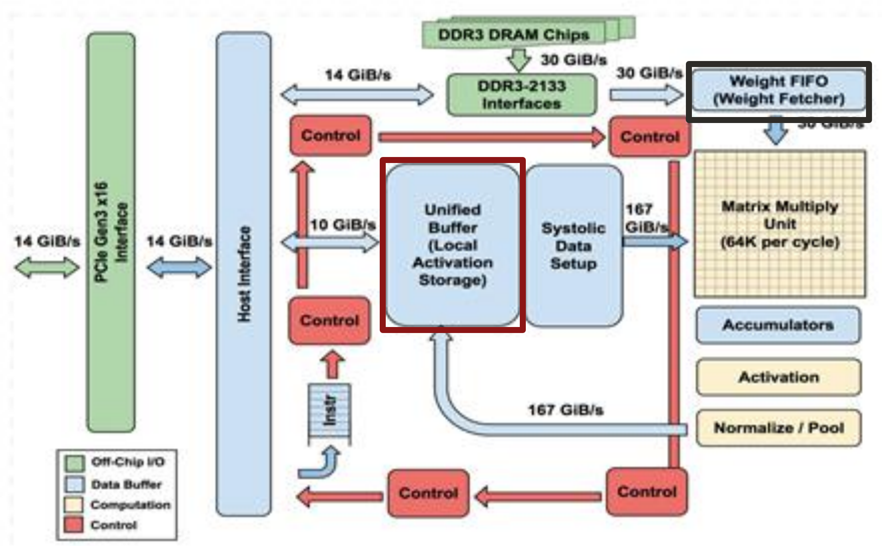
- Streaming memories ingest and emit (re)ordered streams of data to occupy large sets of compute units
  - Streaming memories have no data-dependent flow control



Streaming Memories

TPU [2]

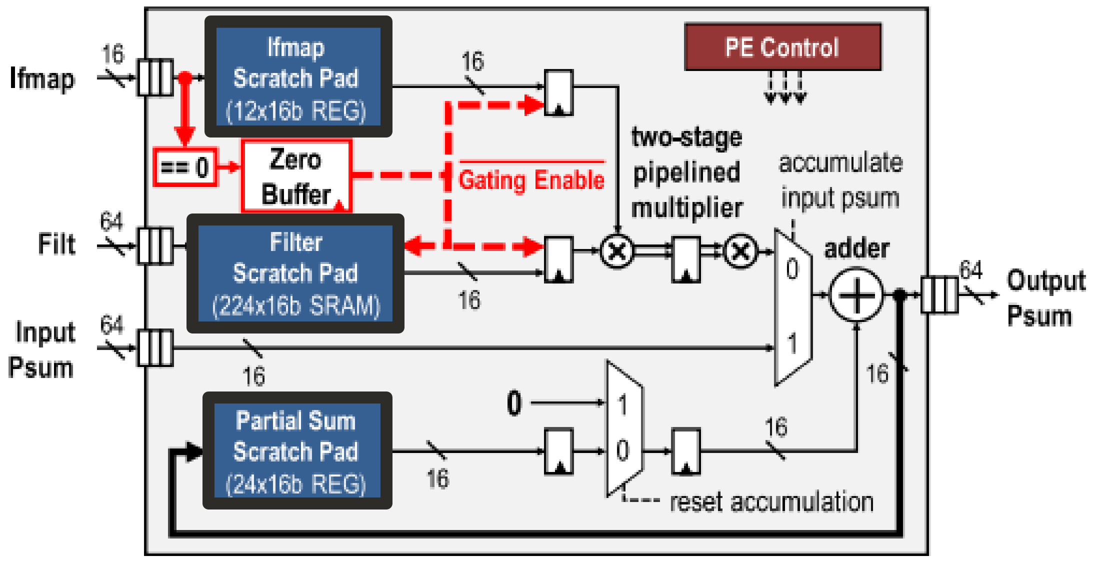
- Push memories used for activation storage, weight FIFO
- Custom compilation pathway (TensorFlow -> TPU Instructions)
- **Aggregate data and push through compute**



Streaming Memories

Eyeriss [5]

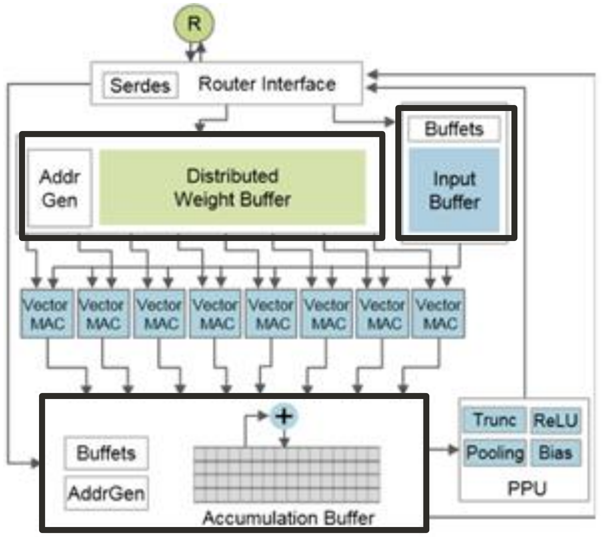
- Push memories for scratchpads
- Manual mapping to CNN parameters
- **Aggregate data and push through compute**



Streaming Memories

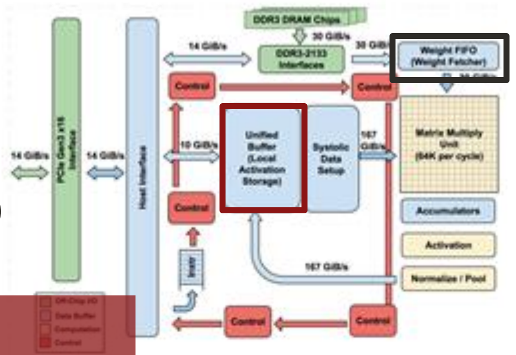
SIMBA PE [4]

- Push memories for inputs, weight buffer, accumulation buffer
- Caffe -> mapper + placer -> configuration binaries
- **Aggregate data and push through compute**



(c) Simba Processing Element

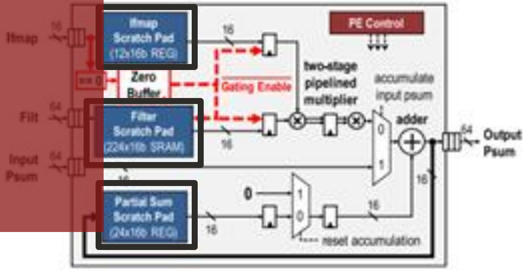
- TPU [2]
  - Push memories used for activation storage, weight FIFO
  - Custom compilation pathway (TensorFlow -> TPU Instructions)



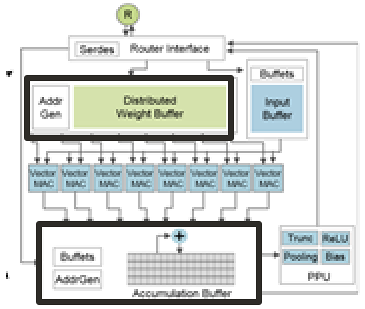
Three streaming memories

Three *different* compilers

- Eyeriss [5]
  - Push memories for scratchpads
  - Manual mapping to CNN parameters


Ifmap Scratch Pad (12x16b REG)
Filter Scratch Pad (224x16b SRAM)
Partial Sum Scratch Pad (24x16b REG)
PE Control
Zero Buffer
Gating Enable
two-stage pipelined multiplier
accumulate input psum
adder
Output Psum
reset accumulation

- SIMBA [4]
  - Push memories for inputs, weight buffer, accum buffer
  - Caffe -> mapper + placer -> configuration binaries



(c) Simba Processing Element

Unified Buffer [3] - Streaming Memory Abstraction

- Describes ***stream*** reorderings in space and time
  - Address sequences (space) to indicate reorderings
  - *Static schedule* sequences (time) to make sure all dependencies are obeyed

```
for(y, 0, 64)
  for(x, 0, 64)
    brighten(x, y) = input(x, y) * 4;

for(y, 0, 63)
  for(x, 0, 64)
    blur(x, y) = (brighten(x, y) +
                  brighten(x, y+1)) / 2;
```

Example streaming memory application

- Easy to extract UB from IP/ML (streaming memory) applications

- Create a **Port** for each access statement in the original application
  - Iteration Domain
  - Address Sequence
  - Schedule Sequence

Unified Buffer [3] - Streaming Memory Abstraction

- Describes ***stream*** reorderings in space and time
  - Address sequences (space) to indicate reorderings
  - *Static schedule* sequences (time) to make sure all dependencies are obeyed

```
for(y, 0, 64)
  for(x, 0, 64)
    brighten(x, y) = input(x, y) * 4;

for(y, 0, 63)
  for(x, 0, 64)
    blur(x, y) = (brighten(x, y) +
                  brighten(x, y+1)) / 2;
```

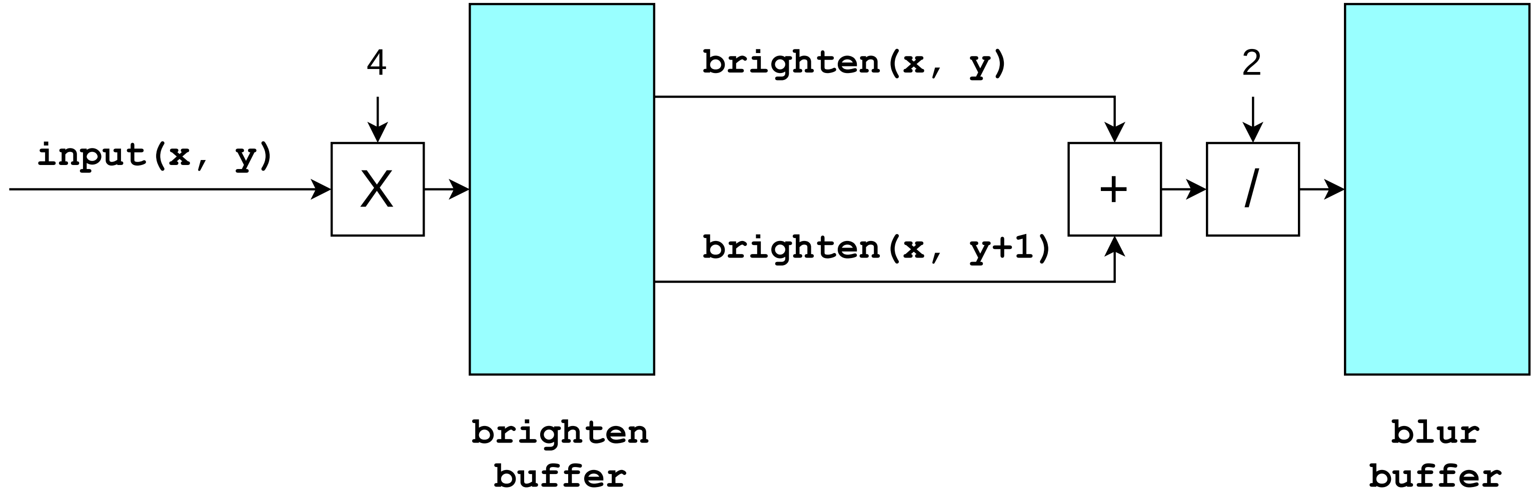
Example streaming memory application

- Easy to extract UB from IP/ML (streaming memory) applications

- Create a **Port** for each access statement in the original application
  - Iteration Domain
  - Address Sequence
  - Schedule Sequence

Kernel Example

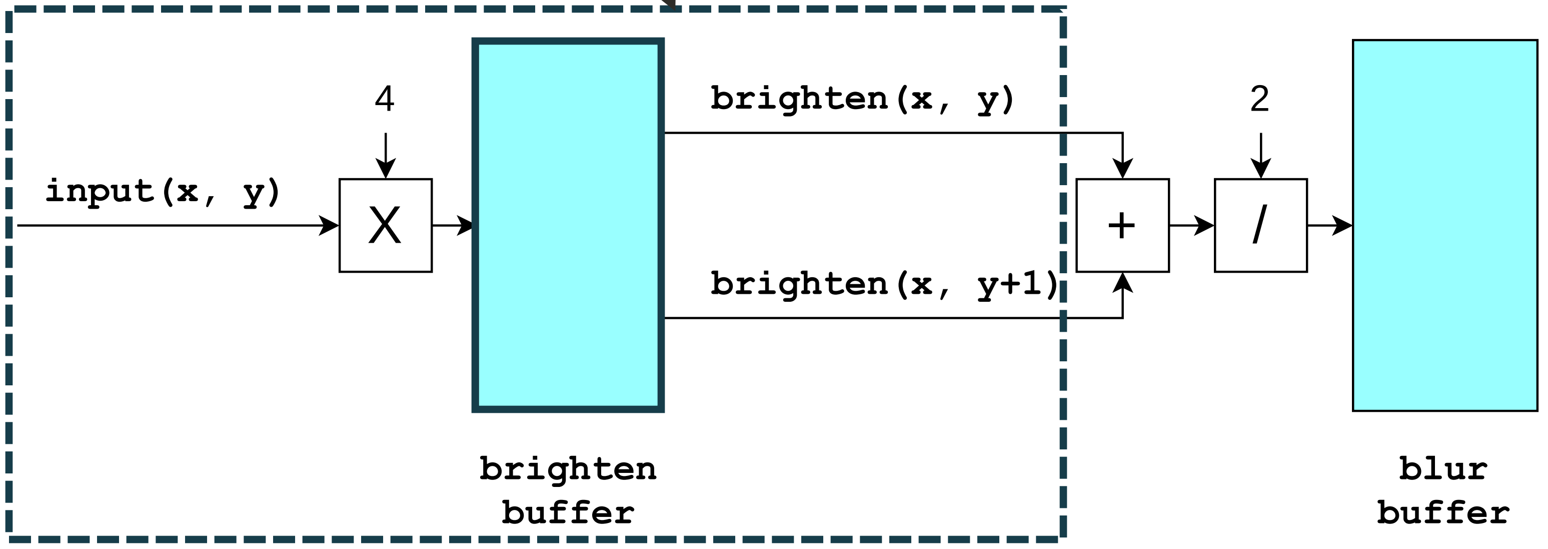
```
for(y, 0, 64)
  for(x, 0, 64)
    brighten(x, y) = input(x, y) * 4;

for(y, 0, 63)
  for(x, 0, 64)
    blur(x, y) = (brighten(x, y) +
                  brighten(x, y+1)) / 2;
```



Kernel Example

```
for(y, 0, 64)
  for(x, 0, 64)
    brighten(x, y) = input(x, y) * 4;

for(y, 0, 63)
  for(x, 0, 64)
    blur(x, y) = (brighten(x, y) +
                  brighten(x, y+1)) / 2;
```

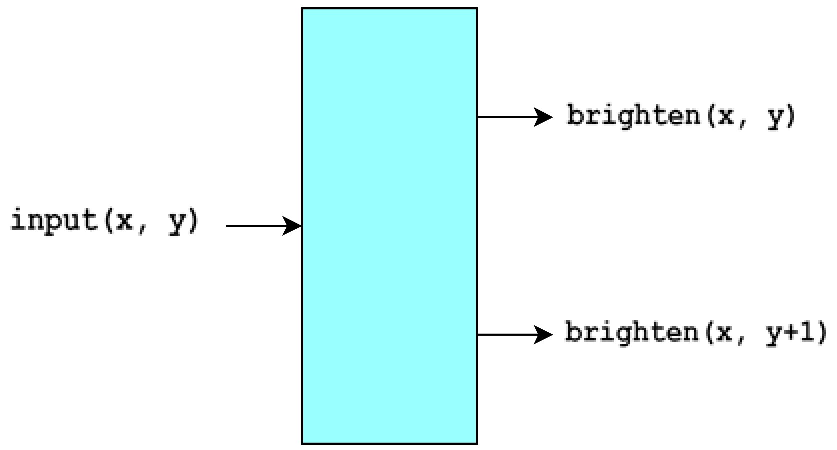
Extract the Unified Buffer for the **brighten** buffer



Kernel Example

One Port for each access statement

```
for(y, 0, 64)
  for(x, 0, 64)
    brighten(x, y) = input(x, y) * 4;

for(y, 0, 63)
  for(x, 0, 64)
    blur(x, y) = (brighten(x, y) +
                  brighten(x, y+1)) / 2;
```

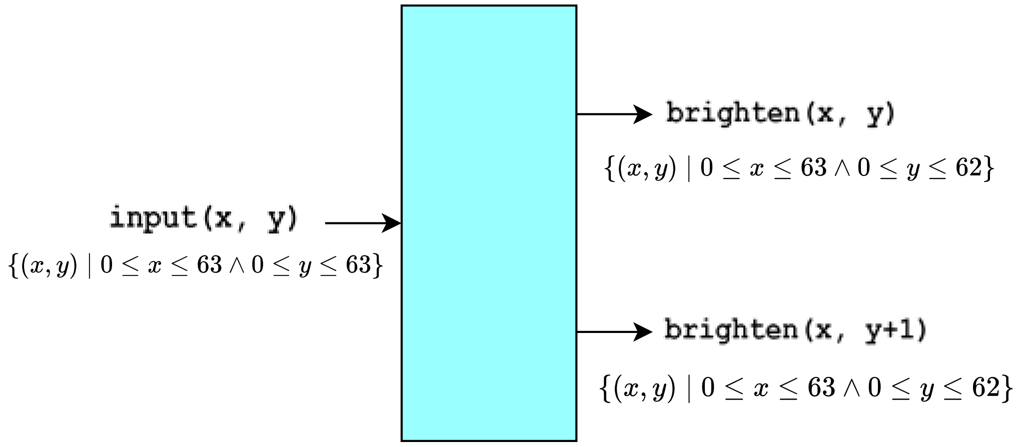


Unified Buffer Abstraction

Kernel Example

Assign each Port an Iteration Domain

```
for(y, 0, 64)
  for(x, 0, 64)
    brighten(x, y) = input(x, y) * 4;

for(y, 0, 63)
  for(x, 0, 64)
    blur(x, y) = (brighten(x, y) +
                  brighten(x, y+1)) / 2;
```


input(x, y)
$\{(x, y) \mid 0 \le x \le 63 \land 0 \le y \le 63\}$
brighten(x, y)
$\{(x, y) \mid 0 \le x \le 63 \land 0 \le y \le 62\}$
brighten(x, y+1)
$\{(x, y) \mid 0 \le x \le 63 \land 0 \le y \le 62\}$

Unified Buffer Abstraction

Kernel Example

Assign each Port an affine map from Iteration Domain to Address

```
for(y, 0, 64)
  for(x, 0, 64)
    brighten(x, y) = input(x, y) * 4;

for(y, 0, 63)
  for(x, 0, 64)
    blur(x, y) = (brighten(x, y) +
                  brighten(x, y+1)) / 2;
```

```
for(y, 0, 64)
  for(x, 0, 64)
    brighten[64 * y + 1 * x + 0] = input * 4;
```

Kernel Example

Assign each Port an affine map from Iteration Domain to Address

```
for(y, 0, 64)
  for(x, 0, 64)
    brighten(x, y) = input(x, y) * 4;

for(y, 0, 63)
  for(x, 0, 64)
    blur(x, y) = (brighten(x, y) +
                  brighten(x, y+1)) / 2;
```

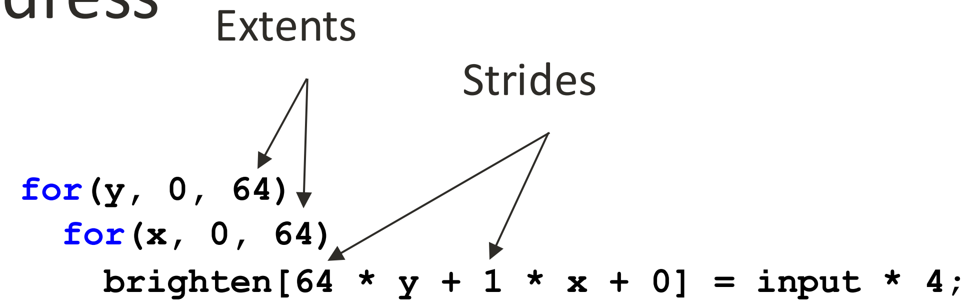
Extents

```
for(y, 0, 64)
  for(x, 0, 64)
    brighten[64 * y + 1 * x + 0] = input * 4;
```

Kernel Example

Assign each Port an affine map from Iteration Domain to Address

```
for(y, 0, 64)
  for(x, 0, 64)
    brighten(x, y) = input(x, y) * 4;

for(y, 0, 63)
  for(x, 0, 64)
    blur(x, y) = (brighten(x, y) +
                  brighten(x, y+1)) / 2;
```

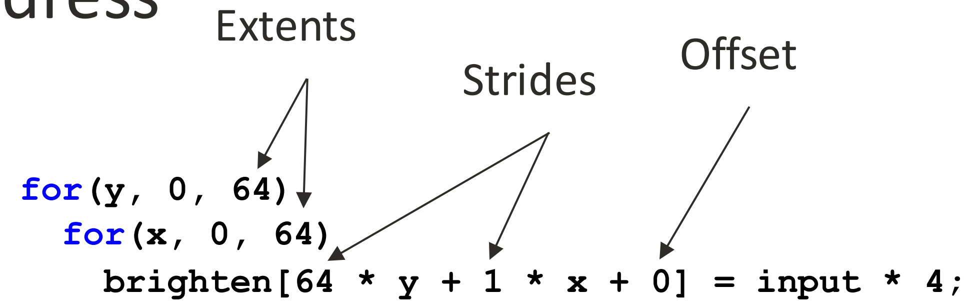


```
for(y, 0, 64)
  for(x, 0, 64)
    brighten[64 * y + 1 * x + 0] = input * 4;
```

Kernel Example

Assign each Port an affine map from Iteration Domain to Address

```
for(y, 0, 64)
  for(x, 0, 64)
    brighten(x, y) = input(x, y) * 4;

for(y, 0, 63)
  for(x, 0, 64)
    blur(x, y) = (brighten(x, y) +
                  brighten(x, y+1)) / 2;
```

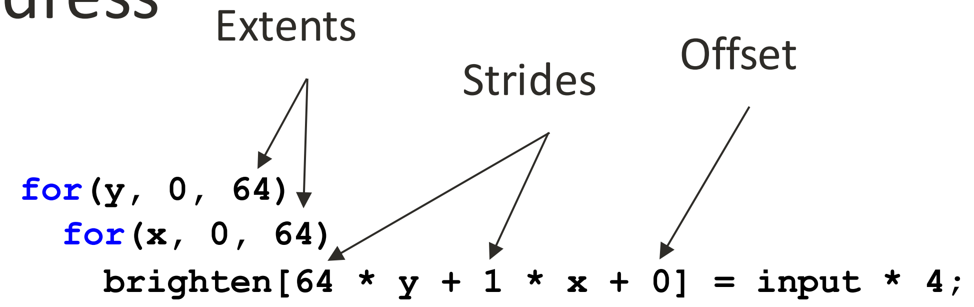


```
for(y, 0, 64)
  for(x, 0, 64)
    brighten[64 * y + 1 * x + 0] = input * 4;
```

Kernel Example

Assign each Port an affine map from Iteration Domain to Address

```
for(y, 0, 64)
  for(x, 0, 64)
    brighten(x, y) = input(x, y) * 4;

for(y, 0, 63)
  for(x, 0, 64)
    blur(x, y) = (brighten(x, y) +
                  brighten(x, y+1)) / 2;
```



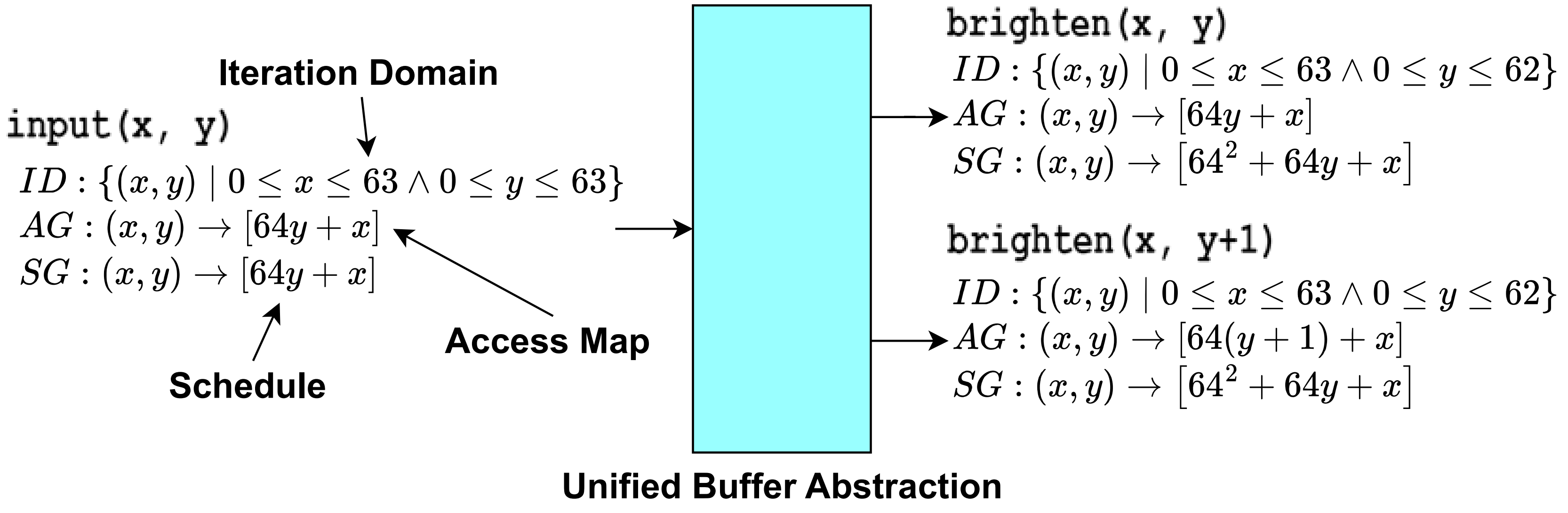
```
for(y, 0, 64)
  for(x, 0, 64)
    brighten[64 * y + 1 * x + 0] = input * 4;
```

Schedule:

As written (without optimizations), all reads occur after all writes

Kernel Example – Extracted UB

```
for(y, 0, 64)
  for(x, 0, 64)
    brighten(x, y) = input(x, y) * 4;

for(y, 0, 63)
  for(x, 0, 64)
    blur(x, y) = (brighten(x, y) +
                  brighten(x, y+1)) / 2;
```



`input(x, y)`

$ID : \{(x, y) \mid 0 \leq x \leq 63 \wedge 0 \leq y \leq 63\}$

$AG : (x, y) \rightarrow [64y + x]$

$SG : (x, y) \rightarrow [64y + x]$

`brighten(x, y)`

$ID : \{(x, y) \mid 0 \leq x \leq 63 \wedge 0 \leq y \leq 62\}$

$AG : (x, y) \rightarrow [64y + x]$

$SG : (x, y) \rightarrow [64^2 + 64y + x]$

`brighten(x, y+1)`

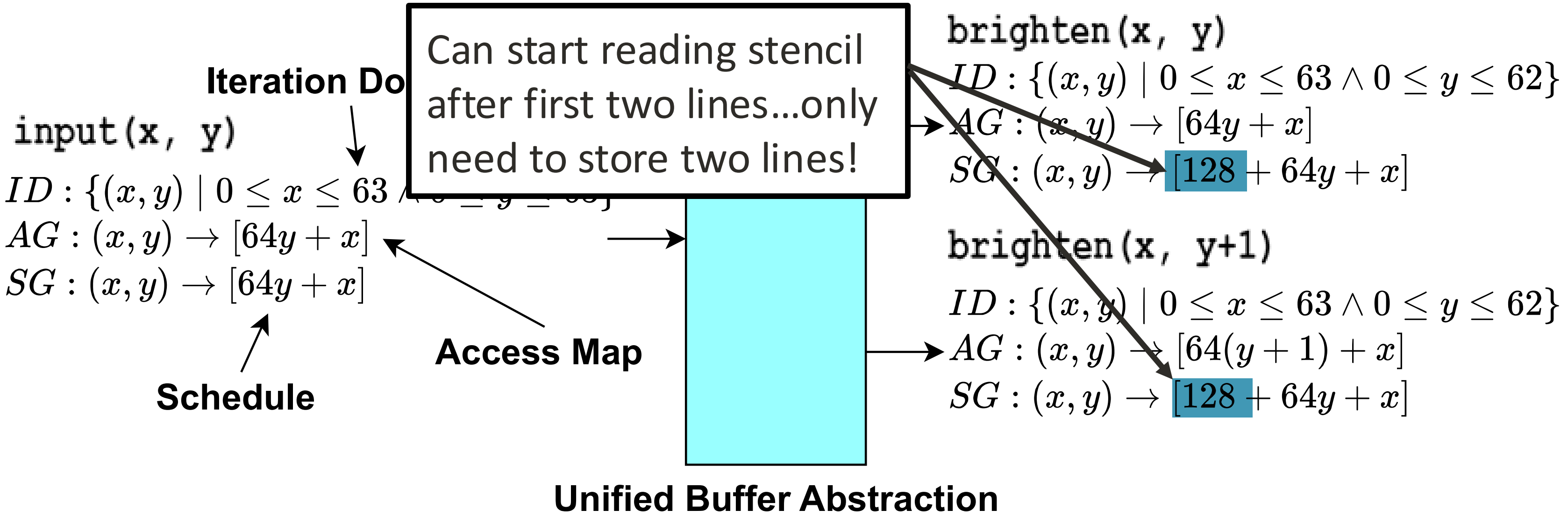
$ID : \{(x, y) \mid 0 \leq x \leq 63 \wedge 0 \leq y \leq 62\}$

$AG : (x, y) \rightarrow [64(y + 1) + x]$

$SG : (x, y) \rightarrow [64^2 + 64y + x]$

Kernel Example – Extracted UB

```
for(y, 0, 64)
  for(x, 0, 64)
    brighten(x, y) = input(x, y) * 4;

for(y, 0, 63)
  for(x, 0, 64)
    blur(x, y) = (brighten(x, y) +
                  brighten(x, y+1)) / 2;
```



Can start reading stencil after first two lines…only need to store two lines!

`input(x, y)`

$ID : \{(x,y) \mid 0 \le x \le 63 \wedge 0 \le y \le 63\}$
$AG : (x,y) \to [64y + x]$
$SG : (x,y) \to [64y + x]$

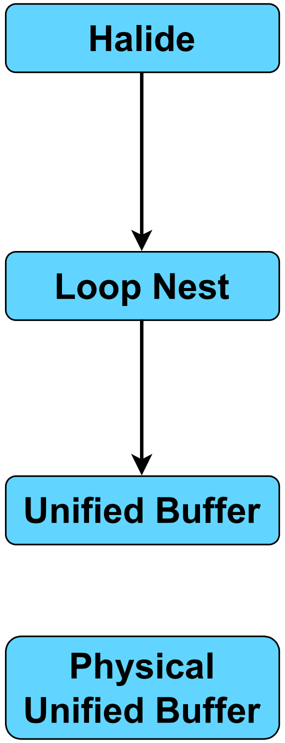
`brighten(x, y)`

$ID : \{(x,y) \mid 0 \le x \le 63 \wedge 0 \le y \le 62\}$
$AG : (x,y) \to [64y + x]$
$SG : (x,y) \to [128 + 64y + x]$

`brighten(x, y+1)`

$ID : \{(x,y) \mid 0 \le x \le 63 \wedge 0 \le y \le 62\}$
$AG : (x,y) \to [64(y+1) + x]$
$SG : (x,y) \to [128 + 64y + x]$

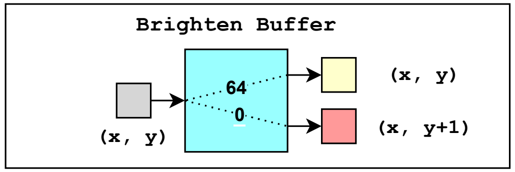
Compiler Flow

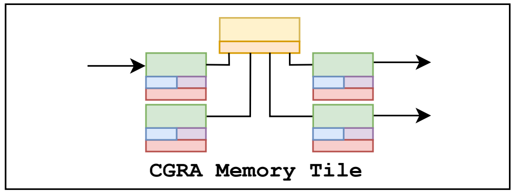


```
brighten(x, y) = input(x, y) * 4;
blur(x, y) = (brighten(x, y) +
              brighten(x, y+1)) / 2;
blur.tile(x, y, xo, yo, xi, yi, 64, 63)
    .hw_accelerate(xi, xo);
brighten.store_at(blur, xo)
        .compute_at(blur, xo);
input.stream_to_accelerator();
```

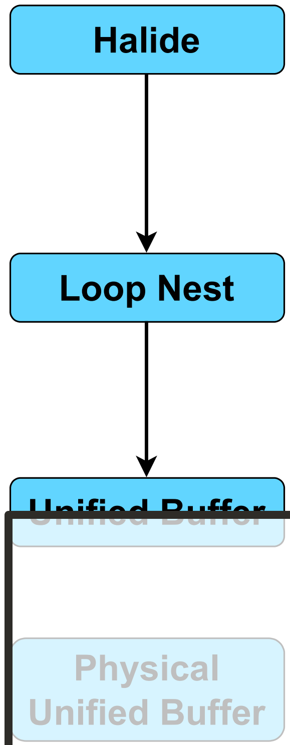
```
for(y, 0, 64)
  for(x, 0, 64)
    brighten(x, y) = input(x, y) * 4;

for(y, 0, 63)
  for(x, 0, 64)
    blur(x, y) = (brighten(x, y) +
                  brighten(x, y+1)) / 2;
```

We know how to extract UBs!



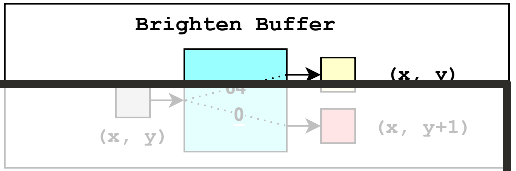


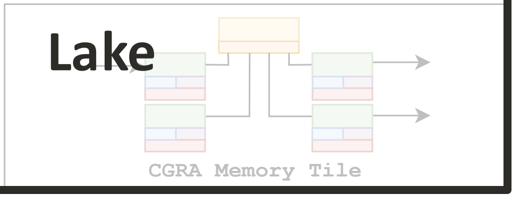
Lake enables Unified Buffer mapping!



```
brighten(x, y) = input(x, y) * 4;
blur(x, y) = (brighten(x, y) +
              brighten(x, y+1)) / 2;
blur.tile(x, y, xo, yo, xi, yi, 64, 63)
    .hw_accelerate(xi, xo);
brighten.store_at(blur, xo)
        .compute_at(blur, xo);
input.stream_to_accelerator();
```

```
for(y, 0, 64)
  for(x, 0, 64)
    brighten(x, y) = input(x, y) * 4;

for(y, 0, 63)
  for(x, 0, 64)
    blur(x, y) = (brighten(x, y) +
                  brighten(x, y+1)) / 2;
```

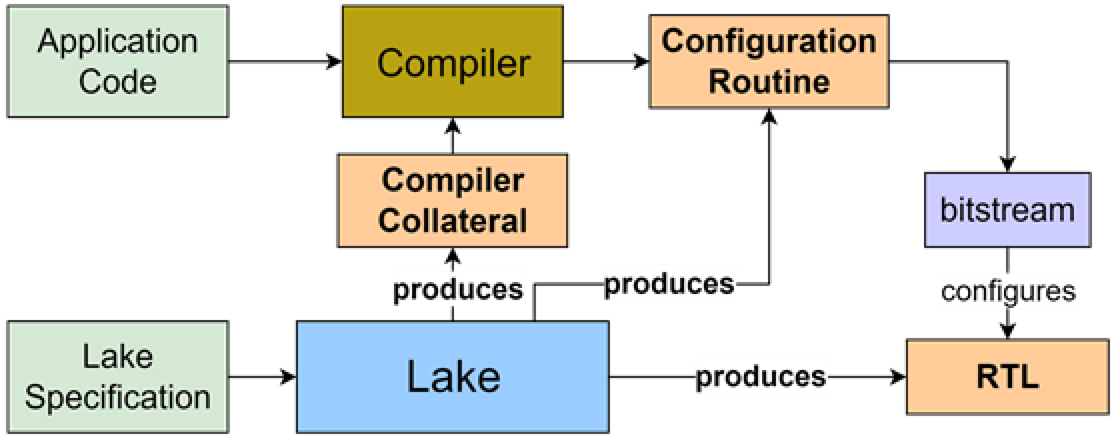




Use the UB abstraction to build RTL and program it

Lake System Overview

- Lake allows users to create physical implementations of Unified Buffers:
  - Hardware Generation (**RTL**)
  - Compiler Targetability (**Compiler Collateral**)
  - Bitstream Generation (**Configuration Routine**)



Lake Requirements

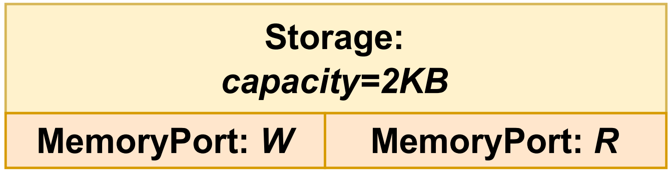
To create physical Unified Buffers (PUB) we need:

1. Type of Memory
   1. Capacity
   2. Memory Width
2. # and types of **Ports**
3. Capabilities of the controllers associated with each **Port**
4. Runtime paradigm (static or ready/valid)

Lake Components - 1

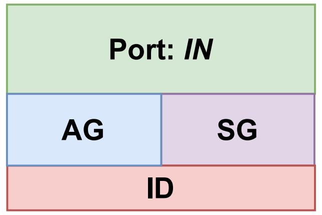
Define the memory system itself

- MemoryPort
  - Used to define physical interfaces to **Storage** element
  - Interface width
  - Type: R, W, R/W
- Storage
  - Actual storage (# bytes)
  - Real implementation will depend on attached **MemoryPorts**



Implied dual-port SRAM

Lake Components - 2



Define the **Ports**

1. Interface width
2. Type: IN, OUT
3. Buffering
   - Inserted when width mismatch with **MemoryPorts**

… and their associated capabilities

- **IterationDomain (ID)**
  1. Number of levels (nest depth)
  2. Maximum loop bounds

- **AddressGenerator (AG)**
  1. Maximum strides
  2. Maximum offset

- **ScheduleGenerator (SG)**
  1. Runtime
     1. Static
     2. Ready/Valid
  2. Maximum strides
  3. Maximum offset

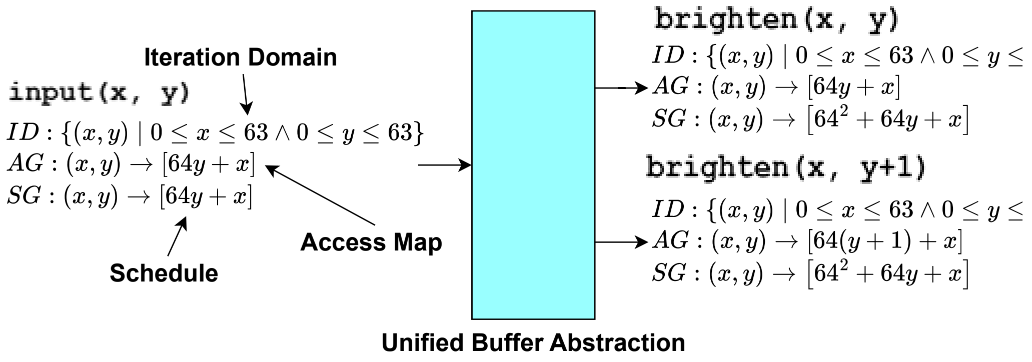
Lake Specification

- Python library
  - Iterate over design space easily
  - Leverage OOP practices like inheritance
- Provide base static and ready/valid implementations
  - Users can write their own by overriding gen_hardware() and gen_bitstream()
  - Class interfaces guarantee proper interconnectivity
- Specify hardware topology through spec.connect()

Spec Example

- Consider applications when building the spec for the hardware
- Previous brighten kernel needed 3 **Ports**
  - Build a 4 **Port** memory design to map 2-4 **Port** applications
- Use wide, single-ported SRAM
  - Almost always use these for efficiency vs multi-ported SRAM

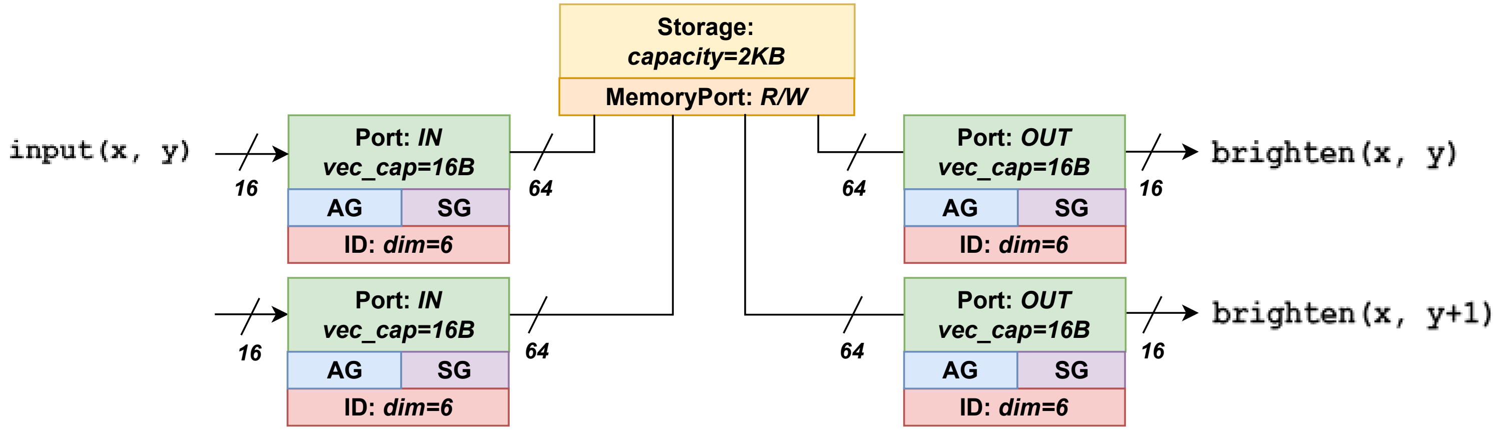
```
for(y, 0, 64)
  for(x, 0, 64)
    brighten(x, y) = input(x, y) * 4;

for(y, 0, 63)
  for(x, 0, 64)
    blur(x, y) = (brighten(x, y) +
                  brighten(x, y+1)) / 2;
```


Iteration Domain
input(x, y)
$ID : \{(x,y) \mid 0 \leq x \leq 63 \wedge 0 \leq y \leq 63\}$
$AG : (x,y) \to [64y + x]$
$SG : (x,y) \to [64y + x]$
Schedule
Access Map
brighten(x, y)
$ID : \{(x,y) \mid 0 \leq x \leq 63 \wedge 0 \leq y \leq$
$AG : (x,y) \to [64y + x]$
$SG : (x,y) \to [64^2 + 64y + x]$
brighten(x, y+1)
$ID : \{(x,y) \mid 0 \leq x \leq 63 \wedge 0 \leq y \leq$
$AG : (x,y) \to [64(y+1) + x]$
$SG : (x,y) \to [64^2 + 64y + x]$
Unified Buffer Abstraction

Spec Example

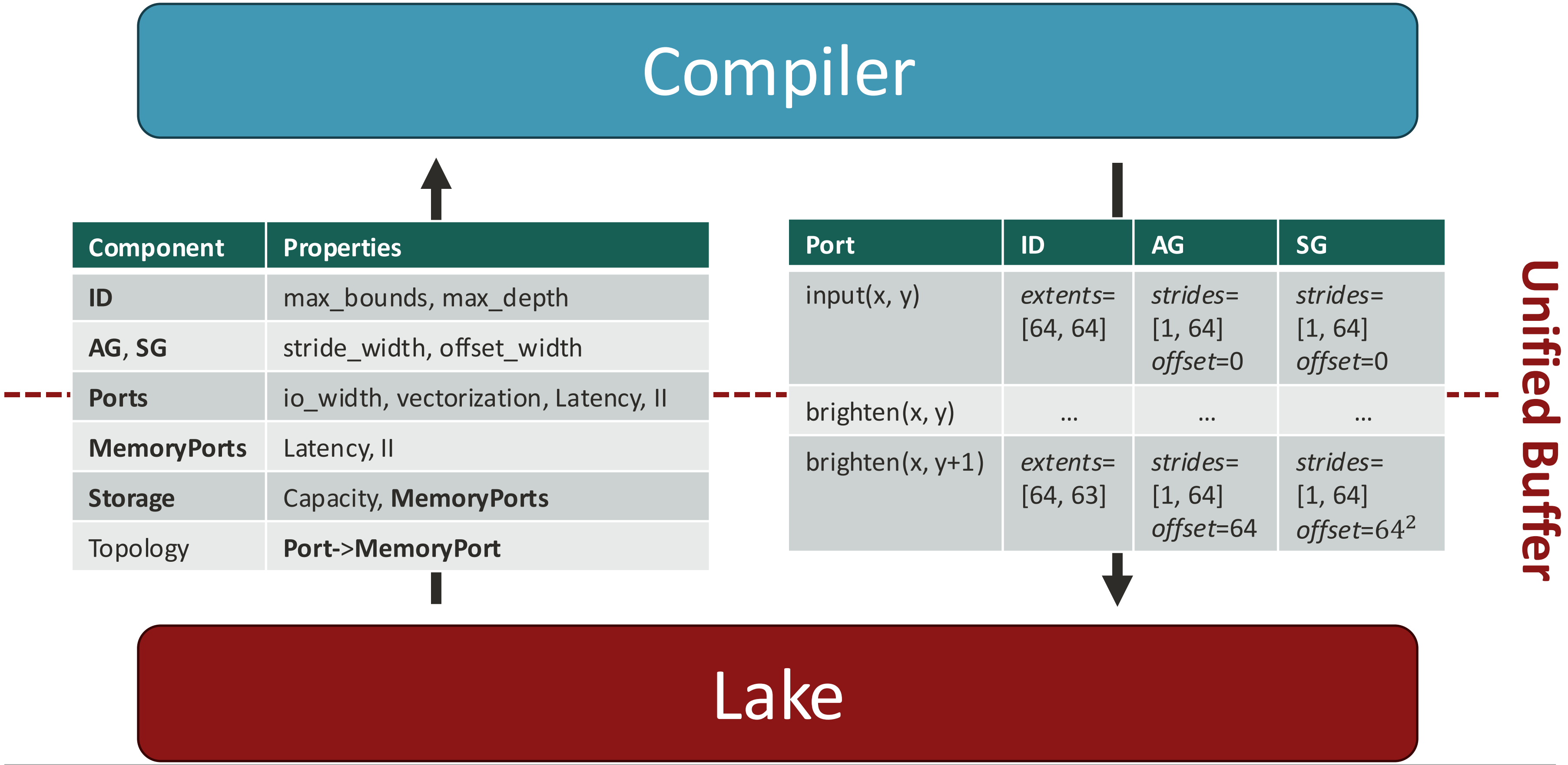
- Consider applications when building the spec for the hardware
- Previous brighten kernel needed 3 **Ports**
  - Build a 4 **Port** memory design to map 2-4 **Port** applications
- Use wide, single-ported SRAM
  - Almost always use these for efficiency vs multi-ported SRAM

```
for(y, 0, 64)
  for(x, 0, 64)
    brighten(x, y) = input(x, y) * 4;

for(y, 0, 63)
  for(x, 0, 64)
    blur(x, y) = (brighten(x, y) +
                  brighten(x, y+1)) / 2;
```



Compiler Collateral

```
for(y, 0, 64)
  for(x, 0, 64)
    brighten(x, y) = input(x, y) * 4;

for(y, 0, 63)
  for(x, 0, 64)
    blur(x, y) = (brighten(x, y) +
                  brighten(x, y+1)) / 2;
```



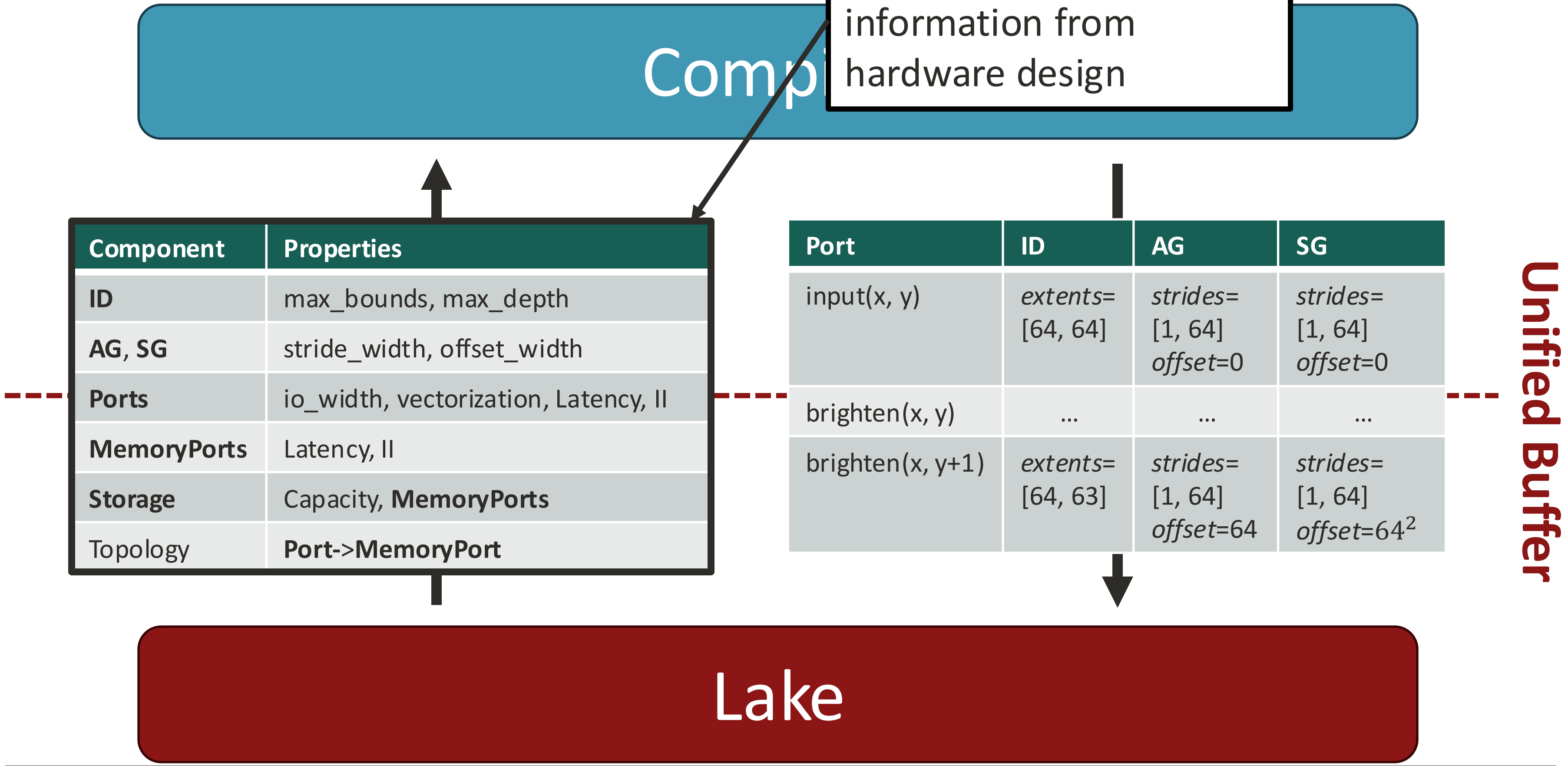
Component	Properties
ID	max_bounds, max_depth
AG, **SG**	stride_width, offset_width
Ports	io_width, vectorization, Latency, II
MemoryPorts	Latency, II
Storage	Capacity, **MemoryPorts**
Topology	**Port->MemoryPort**

Port	ID	AG	SG
input(x, y)	*extents*= [64, 64]	*strides*= [1, 64] *offset*=0	*strides*= [1, 64] *offset*=0
brighten(x, y)	…	…	…
brighten(x, y+1)	*extents*= [64, 63]	*strides*= [1, 64] *offset*=64	*strides*= [1, 64] *offset*=64^2

Compiler Collateral

```
for(y, 0, 64)
  for(x, 0, 64)
    brighten(x, y) = input(x, y) * 4;

for(y, 0, 63)
  for(x, 0, 64)
                                   y) +
                                   y+1)) / 2;
```

Lake extracts UB-relevant information from hardware design

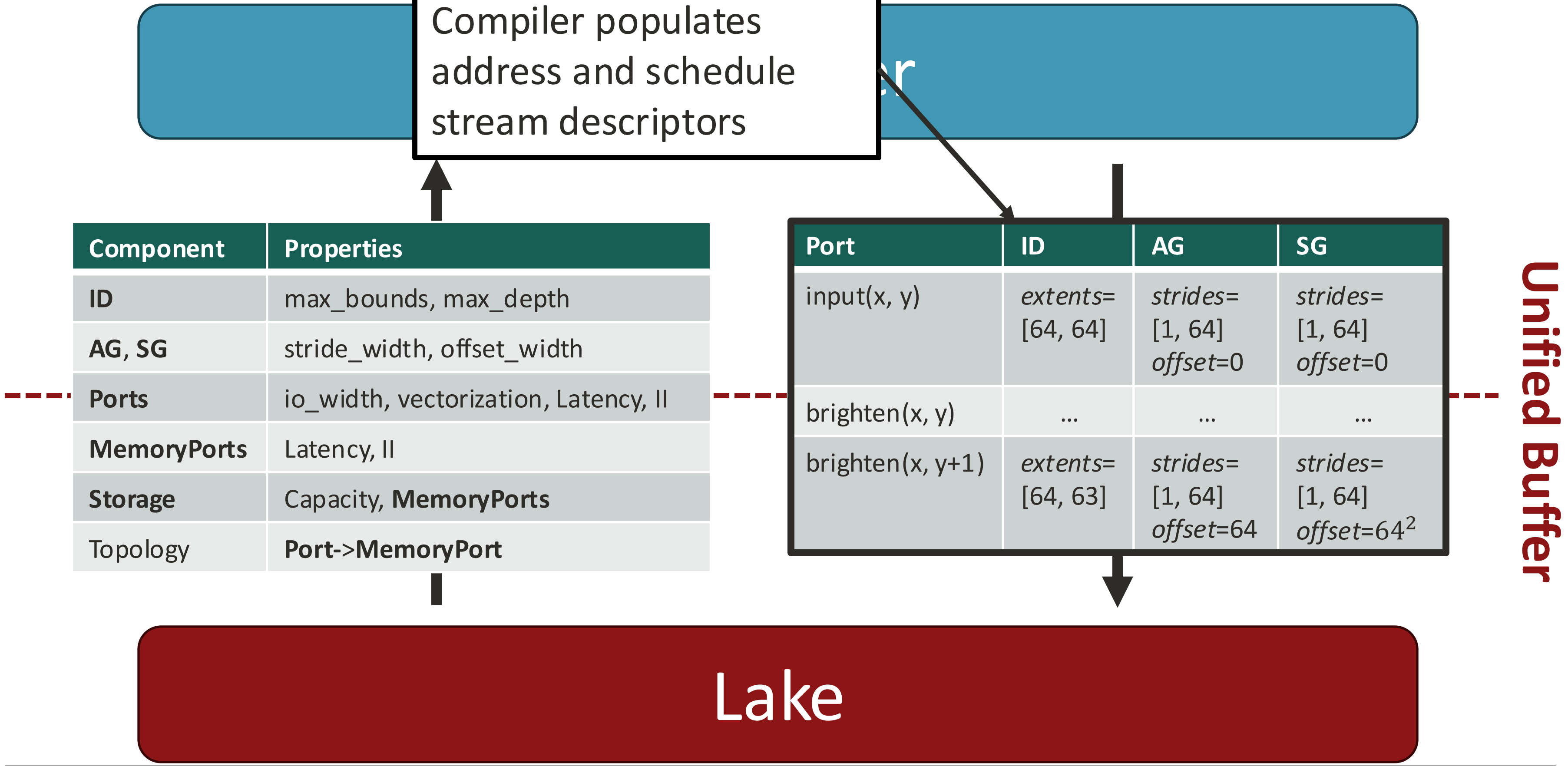


Component	Properties
ID	max_bounds, max_depth
AG, **SG**	stride_width, offset_width
Ports	io_width, vectorization, Latency, II
MemoryPorts	Latency, II
Storage	Capacity, **MemoryPorts**
Topology	**Port->MemoryPort**

Port	ID	AG	SG
input(x, y)	*extents*= [64, 64]	*strides*= [1, 64] *offset*=0	*strides*= [1, 64] *offset*=0
brighten(x, y)	…	…	…
brighten(x, y+1)	*extents*= [64, 63]	*strides*= [1, 64] *offset*=64	*strides*= [1, 64] *offset*=64^2

Compiler Collateral

```
for(y, 0, 64)
  for(x, 0, 64)
    brighten(x, y) = input(x, y) * 4;

for(y, 0, 63)
  for(x, 0, 64)
    blur(x, y) = (brighten(x, y) +
                  brighten(x, y+1)) / 2;
```



Compiler populates address and schedule stream descriptors

Component	Properties
ID	max_bounds, max_depth
AG, **SG**	stride_width, offset_width
Ports	io_width, vectorization, Latency, II
MemoryPorts	Latency, II
Storage	Capacity, **MemoryPorts**
Topology	**Port->MemoryPort**

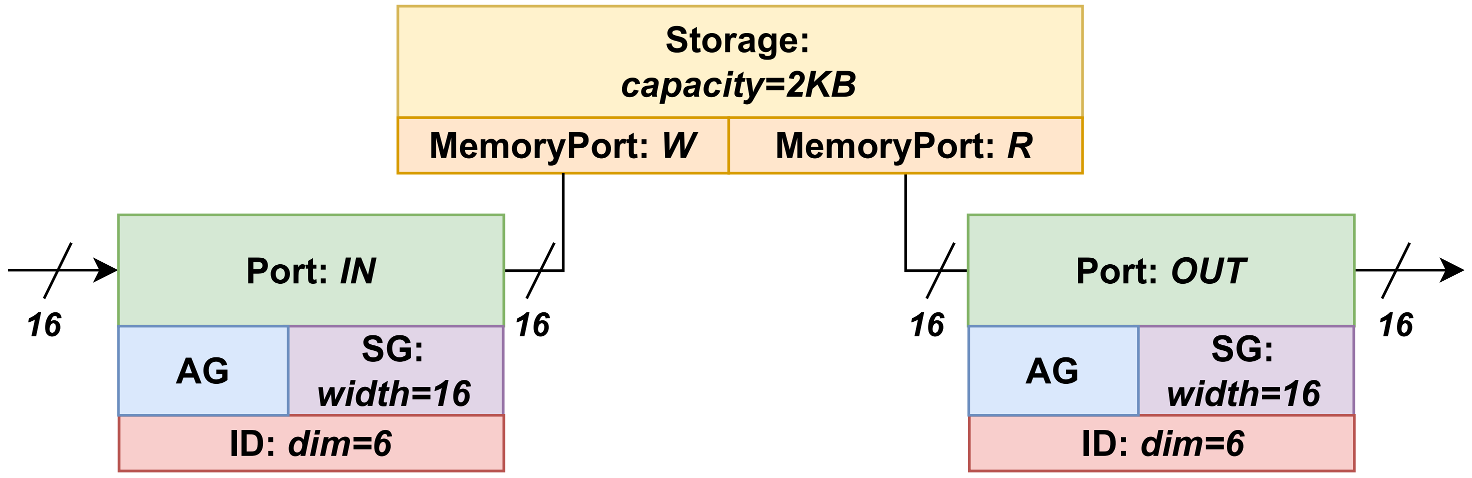
Port	ID	AG	SG
input(x, y)	*extents*= [64, 64]	*strides*= [1, 64] *offset*=0	*strides*= [1, 64] *offset*=0
brighten(x, y)	…	…	…
brighten(x, y+1)	*extents*= [64, 63]	*strides*= [1, 64] *offset*=64	*strides*= [1, 64] *offset*=64^2

Lake Summary

- Lake is a single-source-of-truth generator for ***streaming*** memories
    - RTL
    - Compiler Mapping Constraints
    - Configuration Routines
- Used Lake to build memories in multiple CGRAs
- Choosing the right abstraction (Unified Buffer) can be critical for automation

Demo - Overview

- Create a 1-input, 1-output memory tile with a dual-port SRAM



Demo – File Structure

/aha/lake

- MICRO24_WS
  - simple_dual_port.py
  - demo_driver.py

Use demo_driver.py to generate collateral

Specification already written in simple_dual_port.py

Demo - Specification

- First, we need to write the specification

```
# 1. Initialize the spec...
ls = Spec()
# 2. Define Ports (and register them)
in_port = Port(ext_data_width=data_width,
                            direction=Direction.IN)
out_port = Port(ext_data_width=data_width,
                             direction=Direction.OUT)
ls.register(in_port, out_port)
```

Demo - Specification

```
# 3. Define ID, AG, SG for each Port (and register them)
in_id = IterationDomain()
in_ag = AddressGenerator()
in_sg = ScheduleGenerator()
out_id = IterationDomain()
out_ag = AddressGenerator()
out_sg = ScheduleGenerator()
ls.register(in_id, in_ag, in_sg)
ls.register(out_id, out_ag, out_sg)
```

Demo - Specification

```
# 4. Define the Storage and its MemoryPorts
#    (and register them)

stg = SingleBankStorage(capacity=storage_capacity)

wr_mem_port = MemoryPort(data_width=data_width,
                         mptype=MemoryPortType.W, delay=1)

rd_mem_port = MemoryPort(data_width=data_width,
                         mptype=MemoryPortType.R, delay=1)

ls.register(stg, wr_mem_port, rd_mem_port)
```

Demo - Specification

```
# 5. Connect registered Components (Topology)
# In to In
ls.connect(in_port, in_id)
ls.connect(in_port, in_ag)
ls.connect(in_port, in_sg)
# Out to Out
ls.connect(out_port, out_id)
ls.connect(out_port, out_ag)
ls.connect(out_port, out_sg)
# In and Out to MemoryPorts
ls.connect(in_port, wr_mem_port)
ls.connect(out_port, rd_mem_port)
# MemoryPorts to Storage
ls.connect(wr_mem_port, stg)
ls.connect(rd_mem_port, stg)
```

Demo – Generate Collateral

- Once we've done that, we can generate the Verilog by calling its function

```
simple_dual_port_spec.get_verilog(output_dir=output_dir_verilog)
```

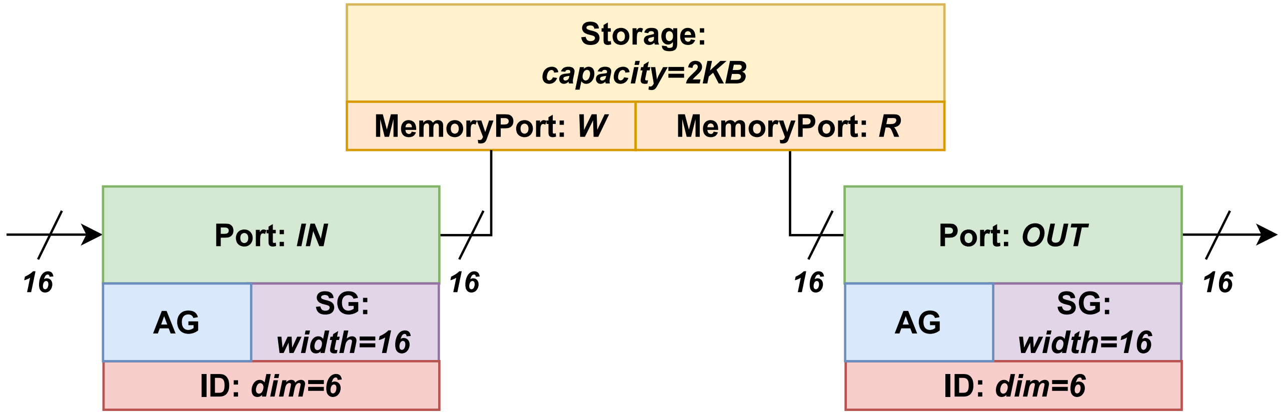
- We can also generate the bitstream for a test – given the schedule from the compiler (handwritten in this case)

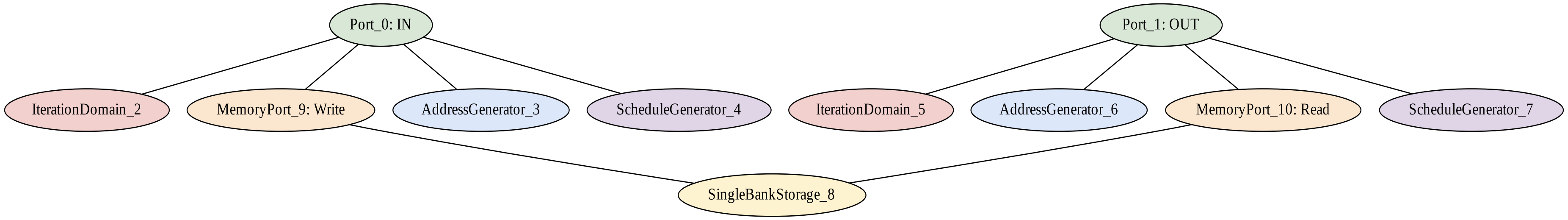
```
bs = simple_dual_port_spec.gen_bitstream(lt)
```

```
cd /aha/lake
python MICRO24_WS/demo_driver.py --outdir LAKE_TEST --visualize
```

- View graph representation in ./LAKE_TEST/simple_dual_port.png

Demo - Visualization



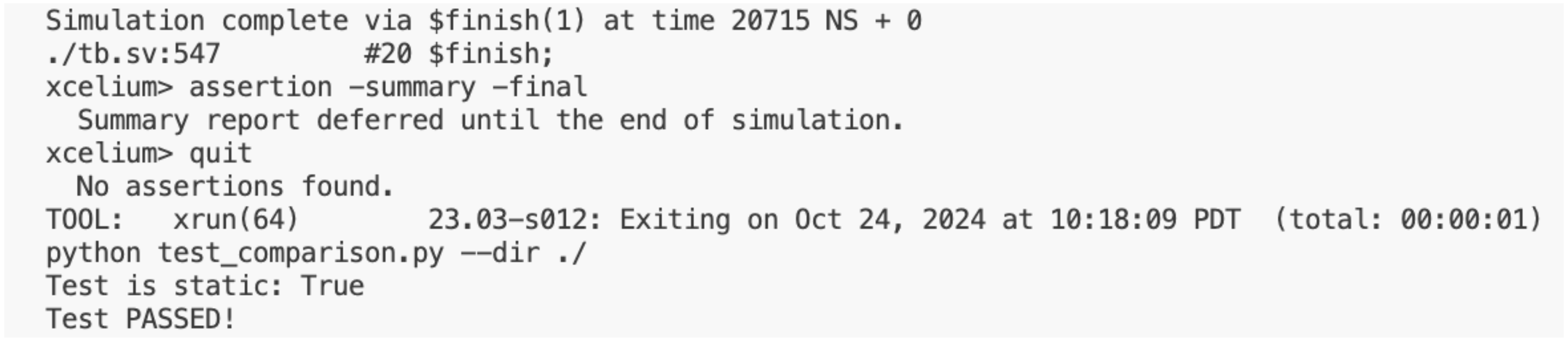


Demo – Run a Test

- Then we can run that test and verify our results!

```
cd LAKE_TEST
# optionally set WAVEFORM
export WAVEFORM=1
make sim
```

- The test passes against the gold schedule



```
Simulation complete via $finish(1) at time 20715 NS + 0
./tb.sv:547         #20 $finish;
xcelium> assertion -summary -final
  Summary report deferred until the end of simulation.
xcelium> quit
  No assertions found.
TOOL:   xrun(64)        23.03-s012: Exiting on Oct 24, 2024 at 10:18:09 PDT  (total: 00:00:01)
python test_comparison.py --dir ./
Test is static: True
Test PASSED!
```

Thank You

References

[1] Alex Carsello, Kathleen Feng, Taeyoung Kong, Kalhan Koul, Qiaoyi Liu, Jackson Melchert, Gedeon Nyengele, Maxwell Strange, Keyi Zhang, Ankita Nayak, Jeff Setter, James Thomas, Kavya Sreedhar, Po-Han Chen, Nikhil Bhagdikar, Zachary Myers, Brandon D'Agostino, Pranil Joshi, Stephen Richardson, Rick Bahr, Christopher Torng, Mark Horowitz, and Priyanka Raina. 2022. Amber: A 367 GOPS, 538 GOPS/W 16nm SoC with a Coarse-Grained Reconfigurable Array for Flexible Acceleration of Dense Linear Algebra. IEEE Symposium on VLSI Technology and Circuits (VLSI) (2022).

[2] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Severn, Chris Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-datacenter performance analysis of a tensor processing unit. International Symposium on Computer Architecture (ISCA) (2017), 1–12.

[3] Qiaoyi Liu, Jeff Setter, Dillon Huff, Maxwell Strange, Kathleen Feng, Mark Horowitz, Priyanka Raina, and Fredrik Kjolstad. 2023. Unified Buffer: Compiling Image Processing and Machine Learning Applications to Push-Memory Accelerators. ACM Transactions on Architecture and Code Optimization (TACO) 20, 2 (2023), 1–26.

[4] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Stephen G. Tell, Yanqing Zhang, William J. Dally, Joel Emer, C. Thomas Gray, Brucek Khailany, and Stephen W. Keckler. 2019. Simba: Scaling Deep-Learning Inference with Multi-Chip-Module-Based Architecture. In Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52). Association for Computing Machinery, New York, NY, USA, 14–27. https://doi.org/10.1145/3352460.3358302

[5] Y. -H. Chen, T. Krishna, J. S. Emer and V. Sze, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," in IEEE Journal of Solid-State Circuits, vol. 52, no. 1, pp. 127-138, Jan. 2017, doi: 10.1109/JSSC.2016.2616357.
keywords: {Shape;Random access memory;Computer architecture;Throughput;Clocks;Neural networks;Hardware;Convolutional neural networks (CNNs);dataflow processing;deep learning;energy-efficient accelerators;spatial architecture},

References

[6] J. Bachrach et al., "Chisel: Constructing hardware in a Scala embedded language," DAC Design Automation Conference 2012, San Francisco, CA, USA, 2012, pp. 1212-1221, doi: 10.1145/2228360.2228584. keywords: {Hardware;Hardware design languages;Generators;Registers;Wires;Vectors;Finite impulse response filter;CAD},

[7] P. Hanrahan, “Magma github.” github.com. https://github.com/phanrahan/magma/

[8] D. Lockhart, G. Zibrat and C. Batten, "PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research," 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture, Cambridge, UK, 2014, pp. 280-292, doi: 10.1109/MICRO.2014.50. keywords: {Object oriented modeling;Computational modeling;Hardware;Hardware design languages;Computer architecture;Computers;Productivity},

[9] M. Gupta, “Google Tensor is a milestone for machine learning.” blog.google. https://blog.google/products/pixel/introducing-google-tensor/ (accessed Oct. 24, 2024)

[10] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Proceedings of the 31st International Conference on Neural Information Processing Systems, NIPS’17, page 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc.

[11] Caleb Donovick, Ross Daly, Jackson Melchert, Lenny Truong, Priyanka Raina, Pat Hanrahan, and Clark Barrett. Peak: A single source of truth for hardware design and verification, 2023.

Backup

Stanford University