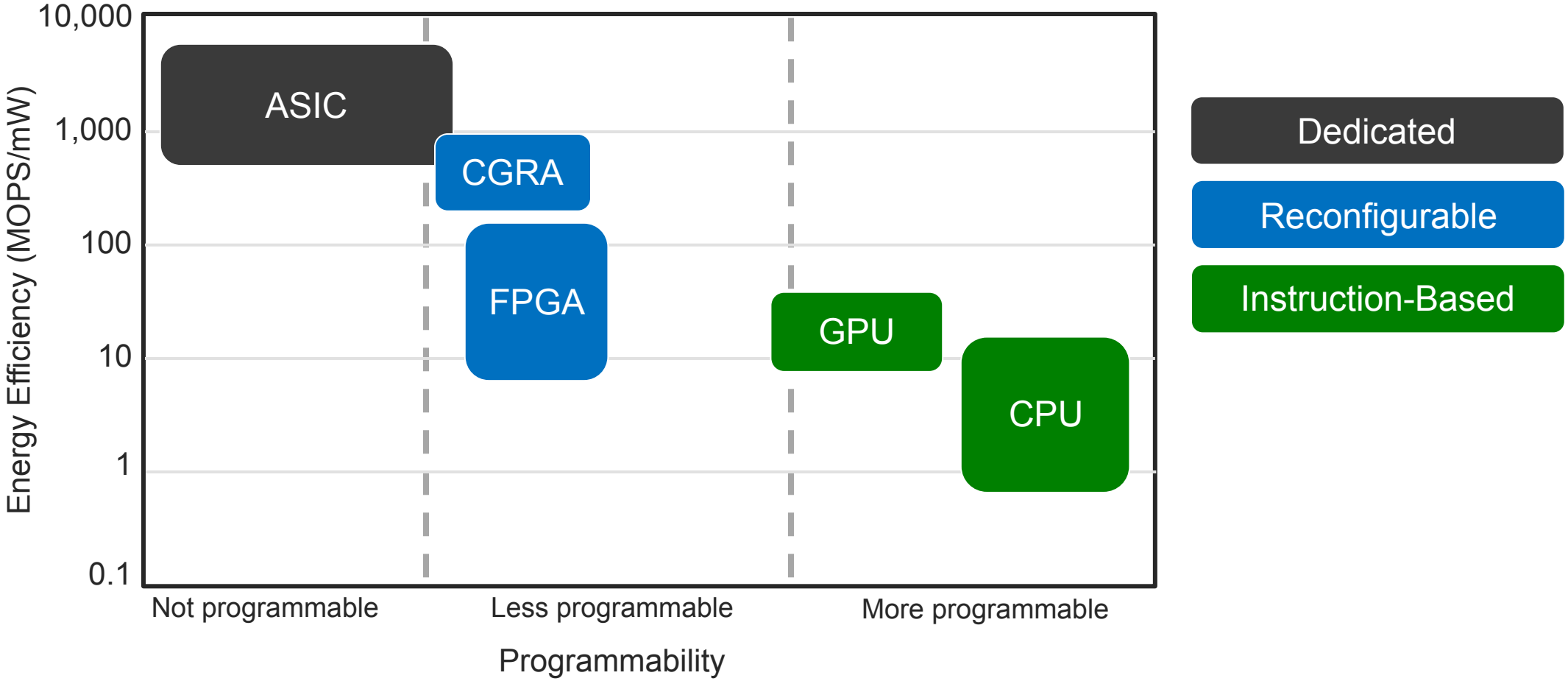
Spatial: A Language and Compiler for Application Accelerators

Raghu Prabhakar

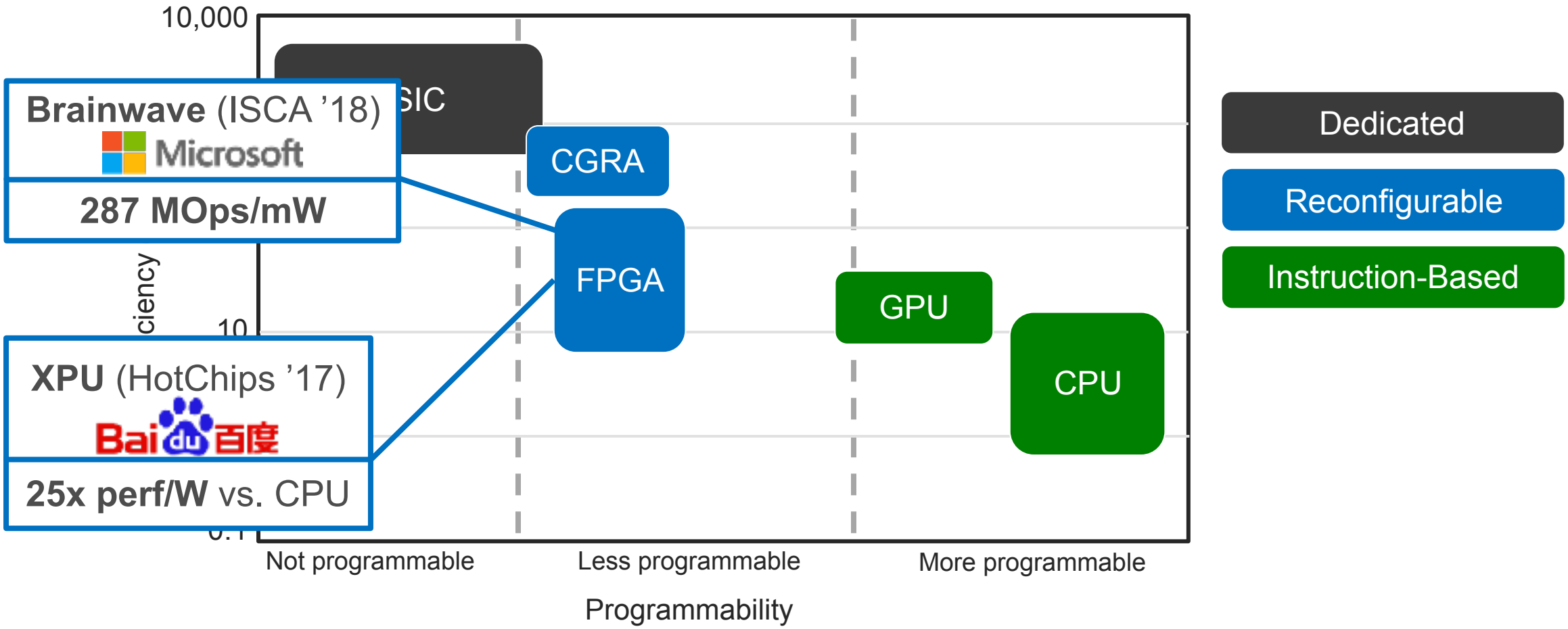
Stanford University / SambaNova Systems

TVM Conference
Dec 13, 2018

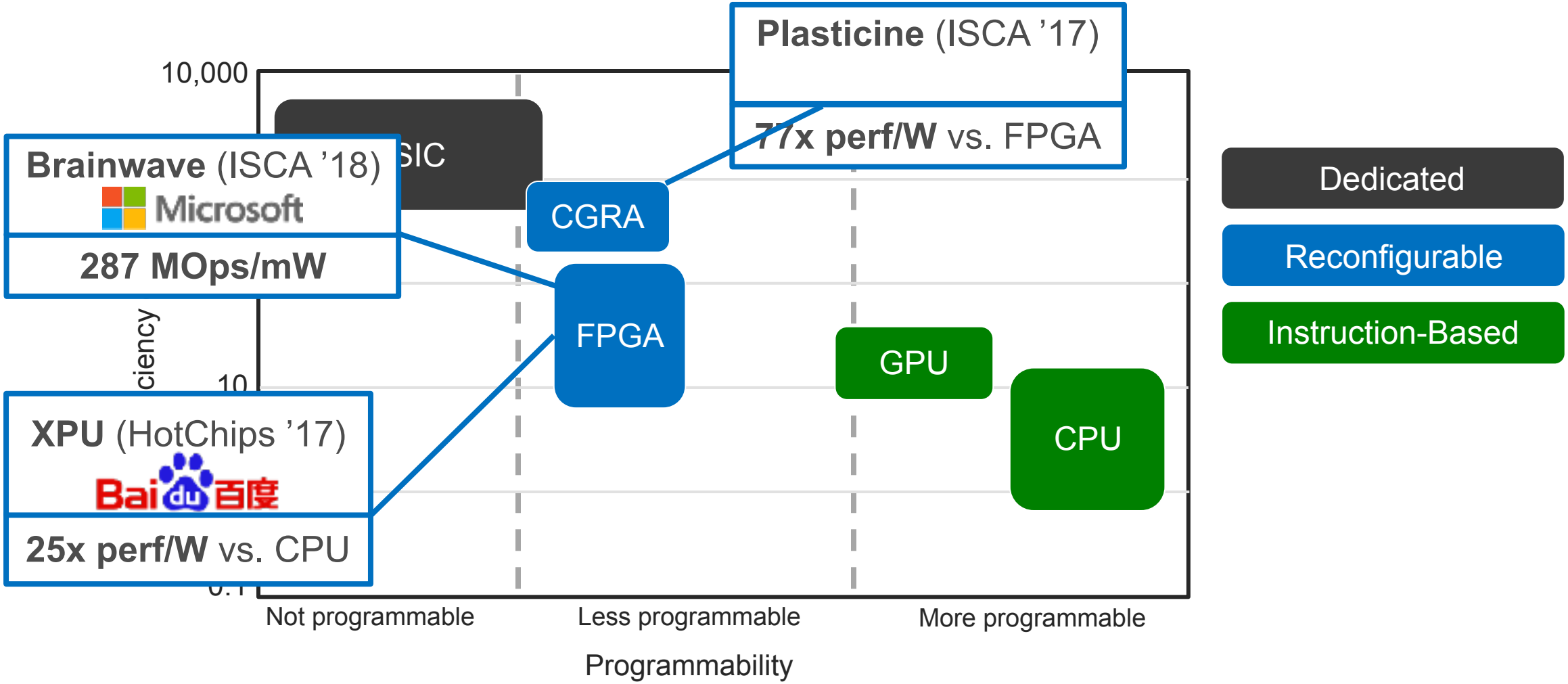
The Future Is (Probably) Reconfigurable



The Future Is (Probably) Reconfigurable



The Future Is (Probably) Reconfigurable

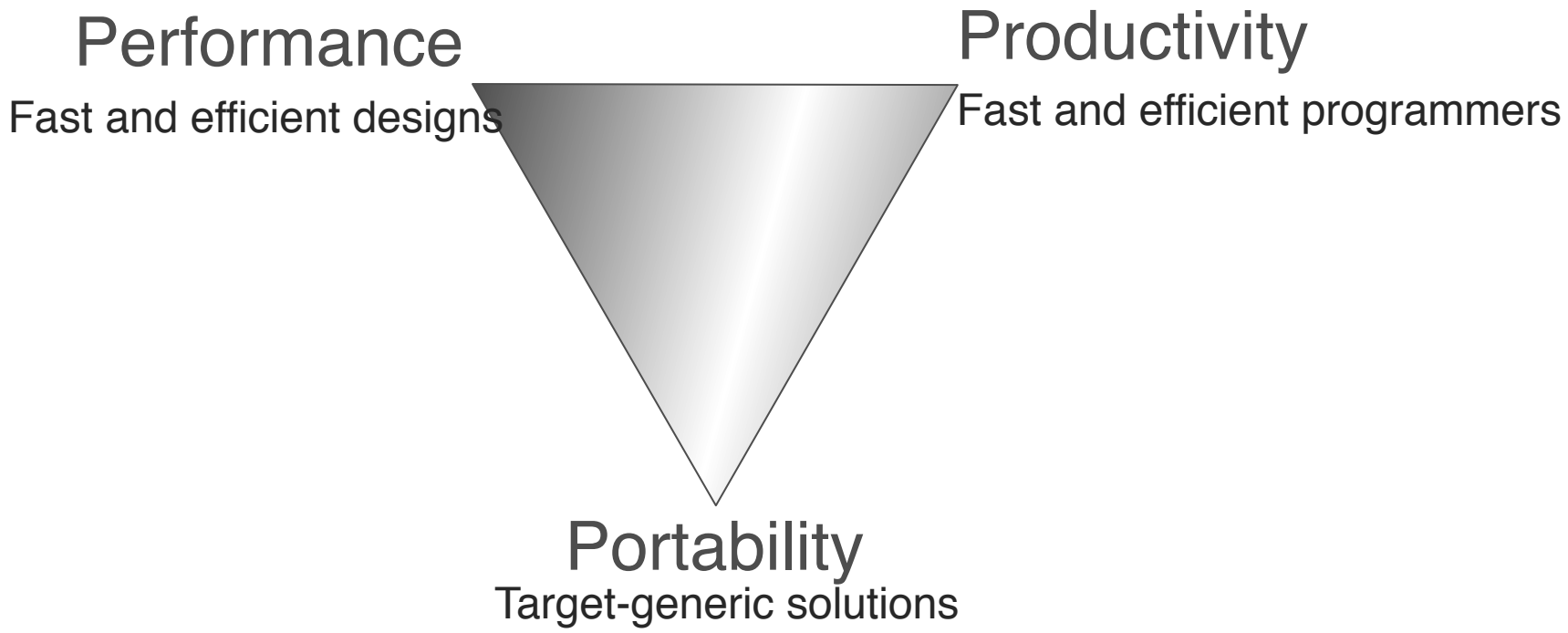


Key Question

How can we more productively target **reconfigurable architectures** like FPGAs?

Key Question

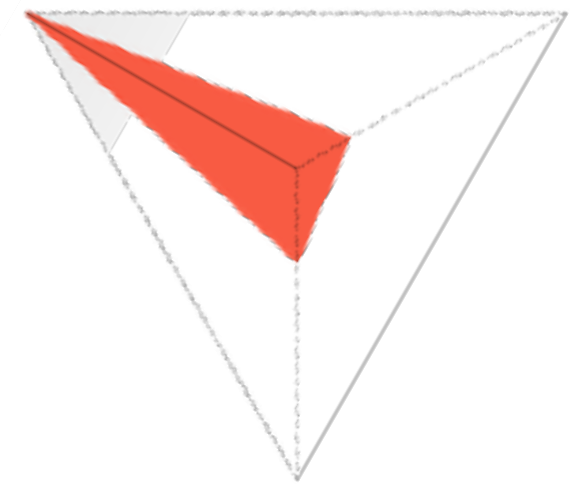
How can we more productively target **reconfigurable architectures** like FPGAs?



HDLs

Hardware Description Languages (HDLs)

e.g. Verilog, VHDL, Chisel, Bluespec



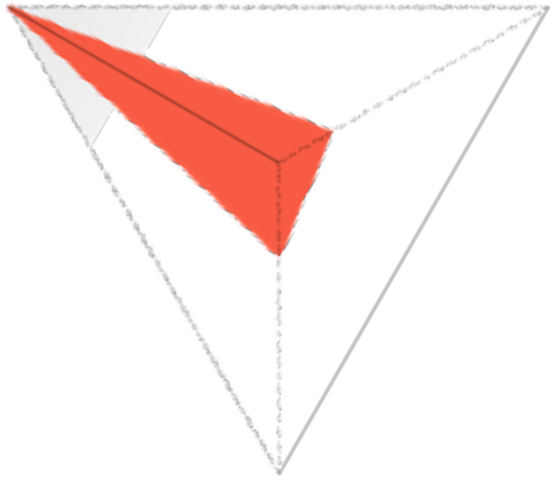
HDLs

Hardware Description Languages (HDLs)

e.g. Verilog, VHDL, Chisel, Bluespec

Performance

✓ Arbitrary RTL



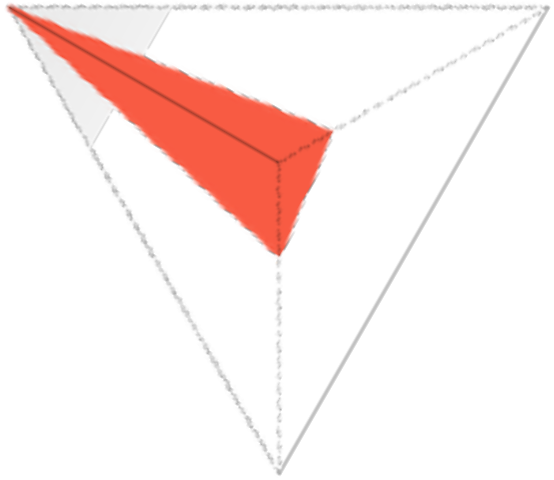
HDLs

Hardware Description Languages (HDLs)

e.g. Verilog, VHDL, Chisel, Bluespec

Performance

✓ Arbitrary RTL



Portability

✘ Significant target-specific code

HDLs

Hardware Description Languages (HDLs)

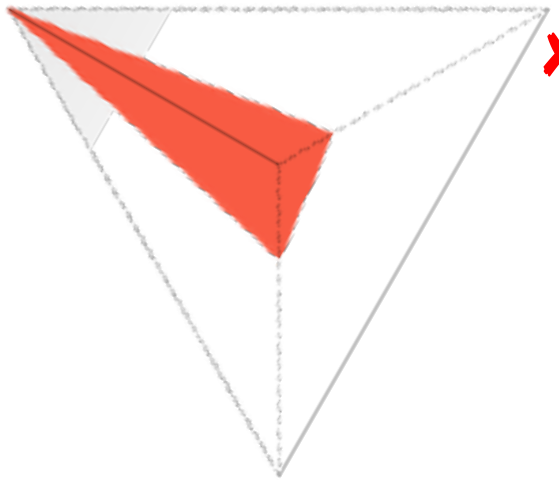
e.g. Verilog, VHDL, Chisel, Bluespec

Performance

✓ Arbitrary RTL

Productivity

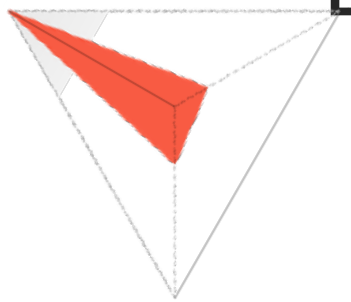
✘ No high-level abstractions



Portability

✘ Significant target-specific code

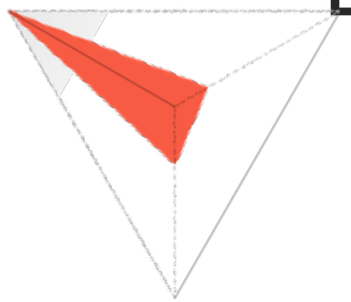
C + Pragmas



Existing High Level Synthesis (C + Pragmas)

e.g. Vivado HLS, SDAccel, Altera OpenCL

C + Pragmas



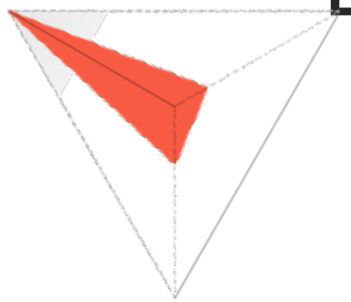
Existing High Level Synthesis (C + Pragmas)

e.g. Vivado HLS, SDAccel, Altera OpenCL

Performance

- ✘ No memory hierarchy
- ✘ No arbitrary pipelining

C + Pragmas



Existing High Level Synthesis (C + Pragmas)

e.g. Vivado HLS, SDAccel, Altera OpenCL

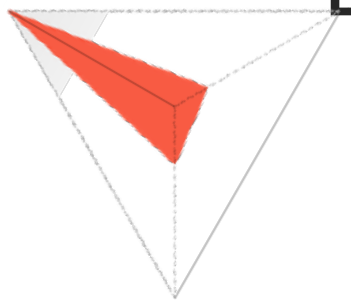
Performance

- ✘ No memory hierarchy
- ✘ No arbitrary pipelining

Portability

- ✓ Portable for single vendor

C + Pragmas

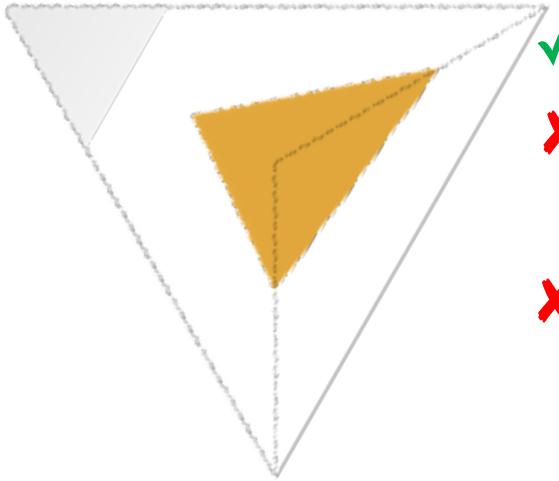


Existing High Level Synthesis (C + Pragmas)

e.g. Vivado HLS, SDAccel, Altera OpenCL

Performance

- ✘ No memory hierarchy
- ✘ No arbitrary pipelining



Productivity

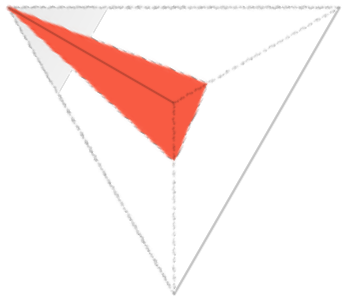
- ✓ Nested loops
- ✘ Ad-hoc mix of software/hardware
- ✘ Difficult to optimize

Portability

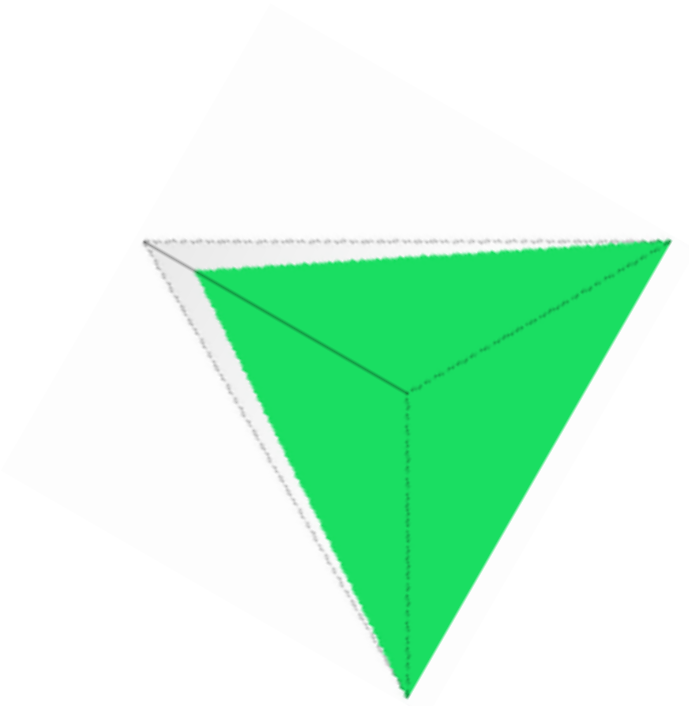
- ✓ Portable for single vendor

Rethinking HLS

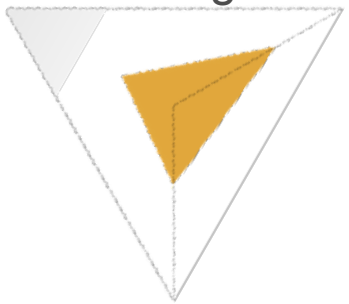
HDLs



Improved HLS

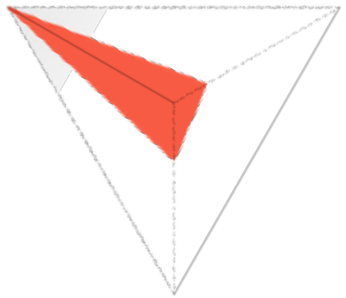


C + Pragmas



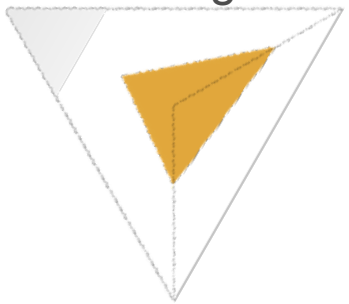
Rethinking HLS

HDLs



Improved HLS

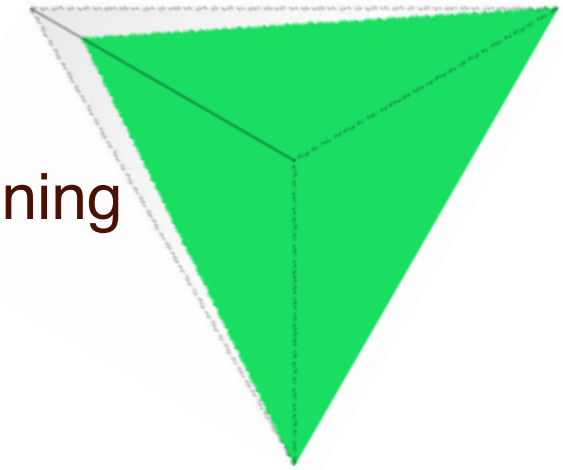
C + Pragmas



Performance

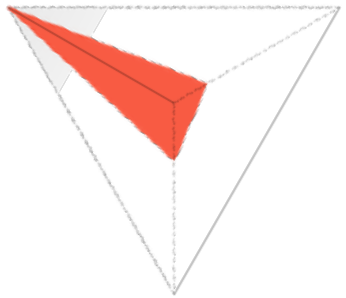
✓ Memory hierarchy

✓ Arbitrary pipelining



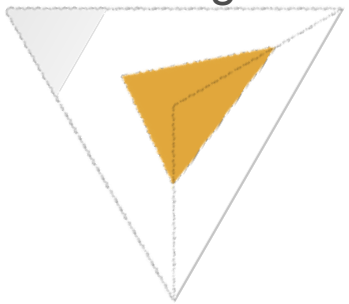
Rethinking HLS

HDLs



Improved HLS

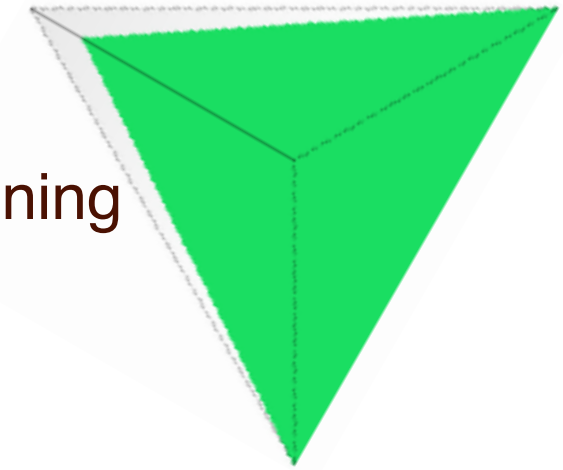
C + Pragmas



Performance

✓ Memory hierarchy

✓ Arbitrary pipelining

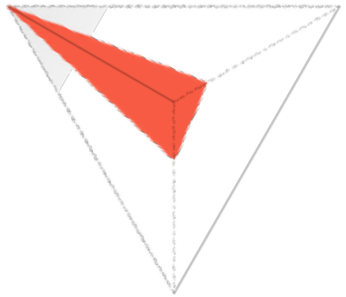


Portability

✓ Target-generic source across reconfigurable architectures

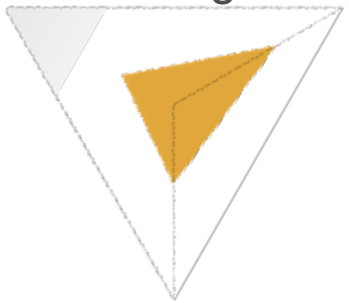
Rethinking HLS

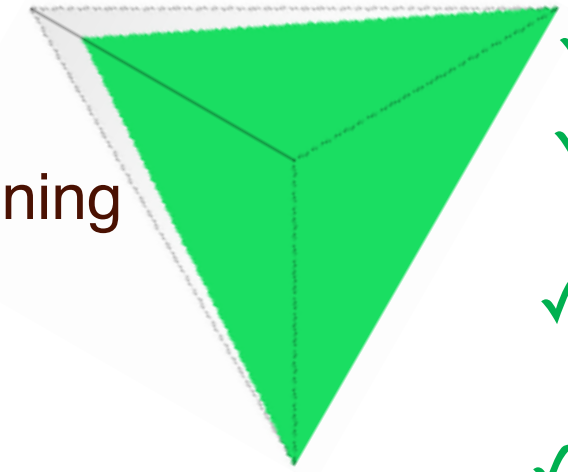
HDLs



Improved HLS

C + Pragmas





Performance

- ✓ Memory hierarchy
- ✓ Arbitrary pipelining

Productivity

- ✓ Nested loops
- ✓ Automatic memory banking/buffering
- ✓ Implicit design parameters (unrolling, banking, etc.)
- ✓ Automated design tuning

Portability

- ✓ Target-generic source across reconfigurable architectures

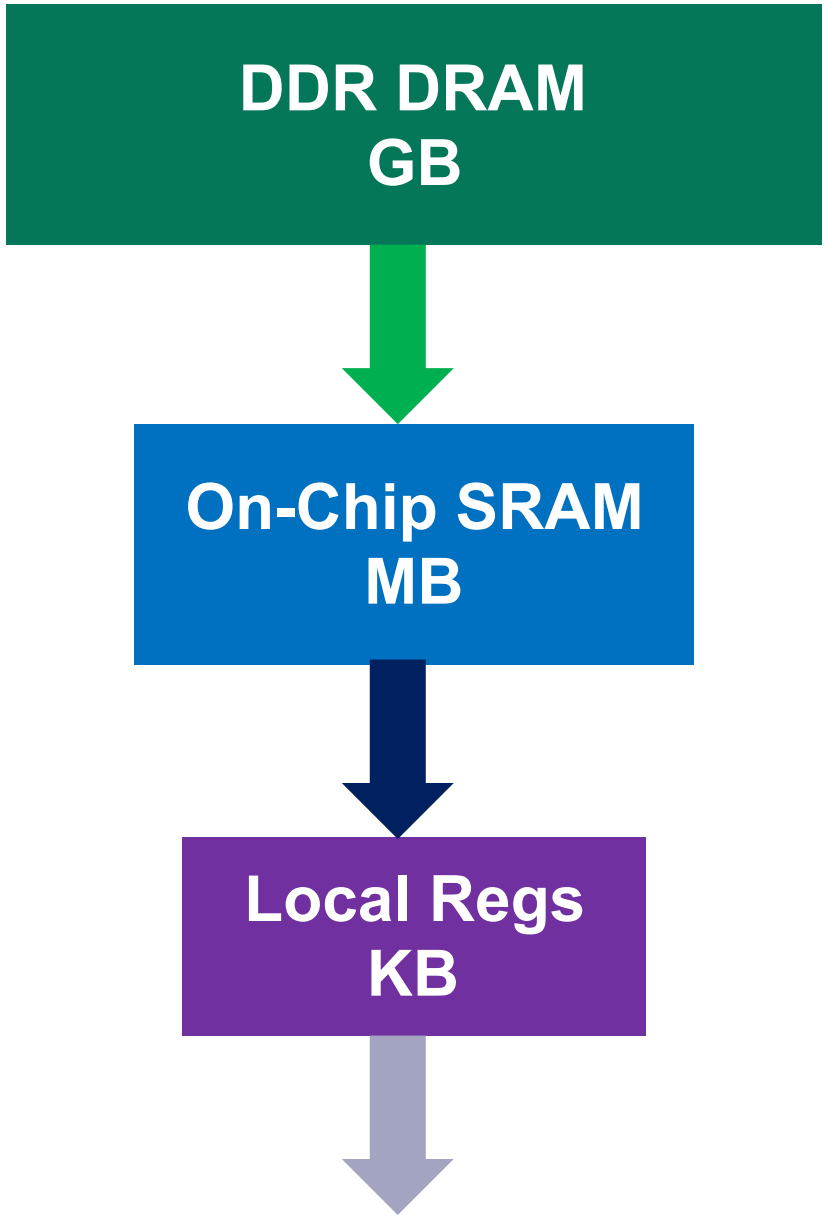
Introducing Spatial

- Programming language to simplify configurable accelerator design
  - Constructs to express:
    - Hierarchical parallel and pipelined data paths
    - explicit memory hierarchies
  - Simple APIs to manage CPU Accelerator communication
- Open source: https://spatial-lang.org/

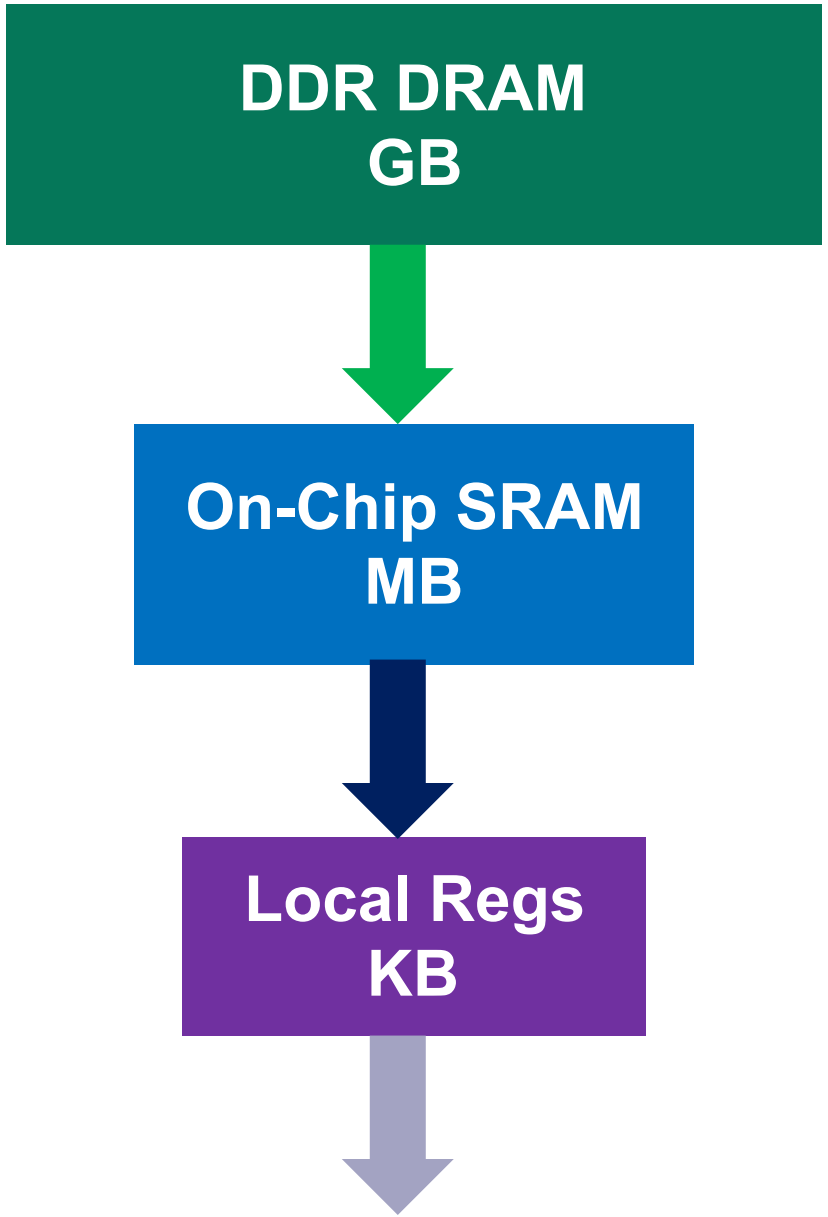
- Allows programmers to focus on "interesting stuff"
  - Designed for performance oriented programmers
  - More intuitive than CUDA: dataflow instead of threads

David Koeplinger et al, *"Spatial: A Language And Compiler For Application Accelerators", PLDI 2018*

Spatial: Memory Hierarchy

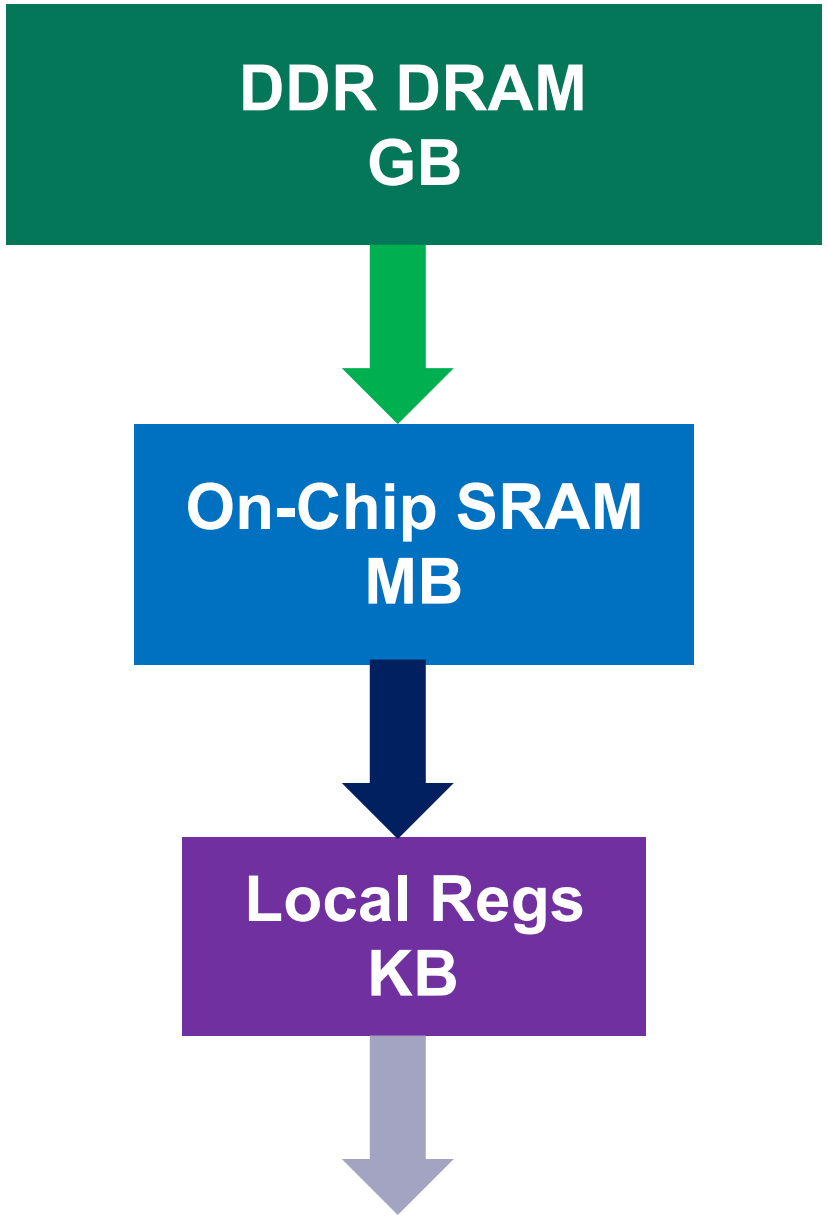


Spatial: Memory Hierarchy




```
val image  = DRAM[UInt8]
(H,W)
```

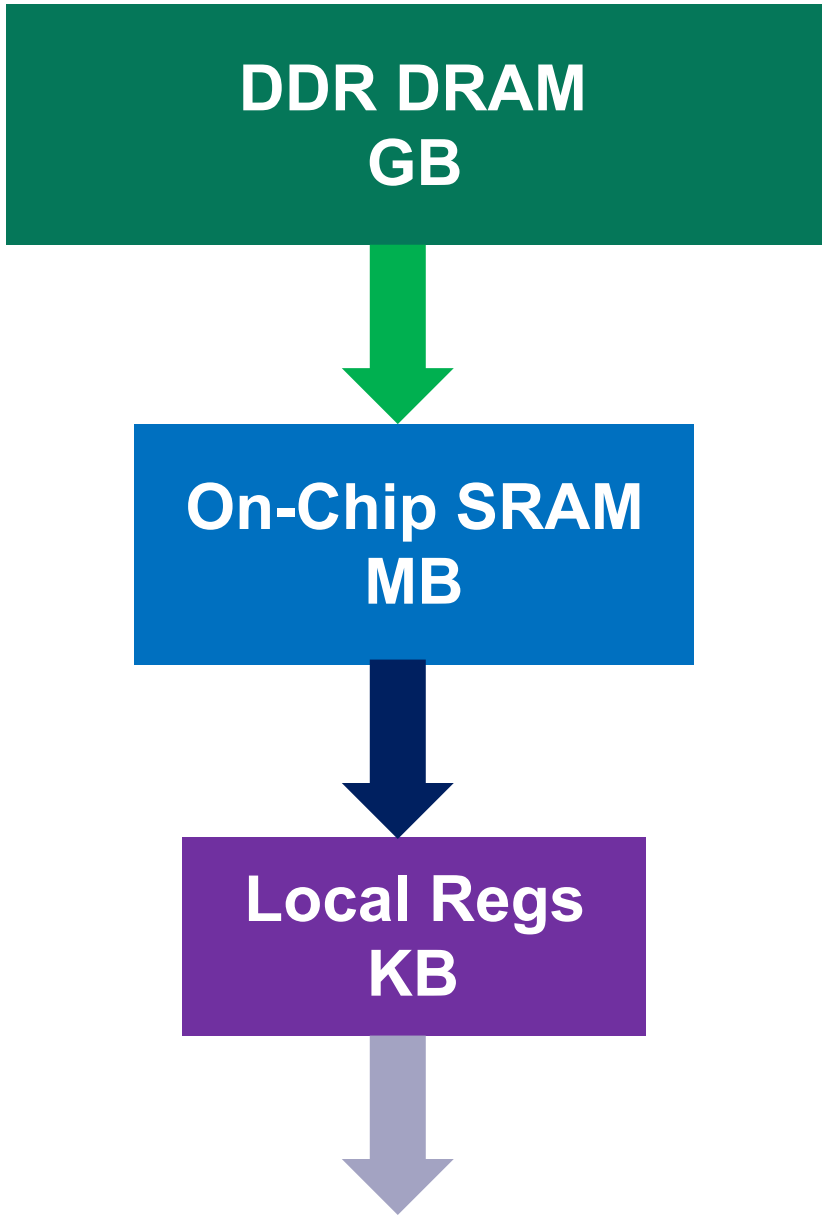
Spatial: Memory Hierarchy




```
val image  = DRAM[UInt8]
(H,W)
```

```
val buffer = SRAM[UInt8](C)
val fifo   = FIFO[Float](D)
val lbuf   = LineBuffer[Int](R,C)
```

Spatial: Memory Hierarchy

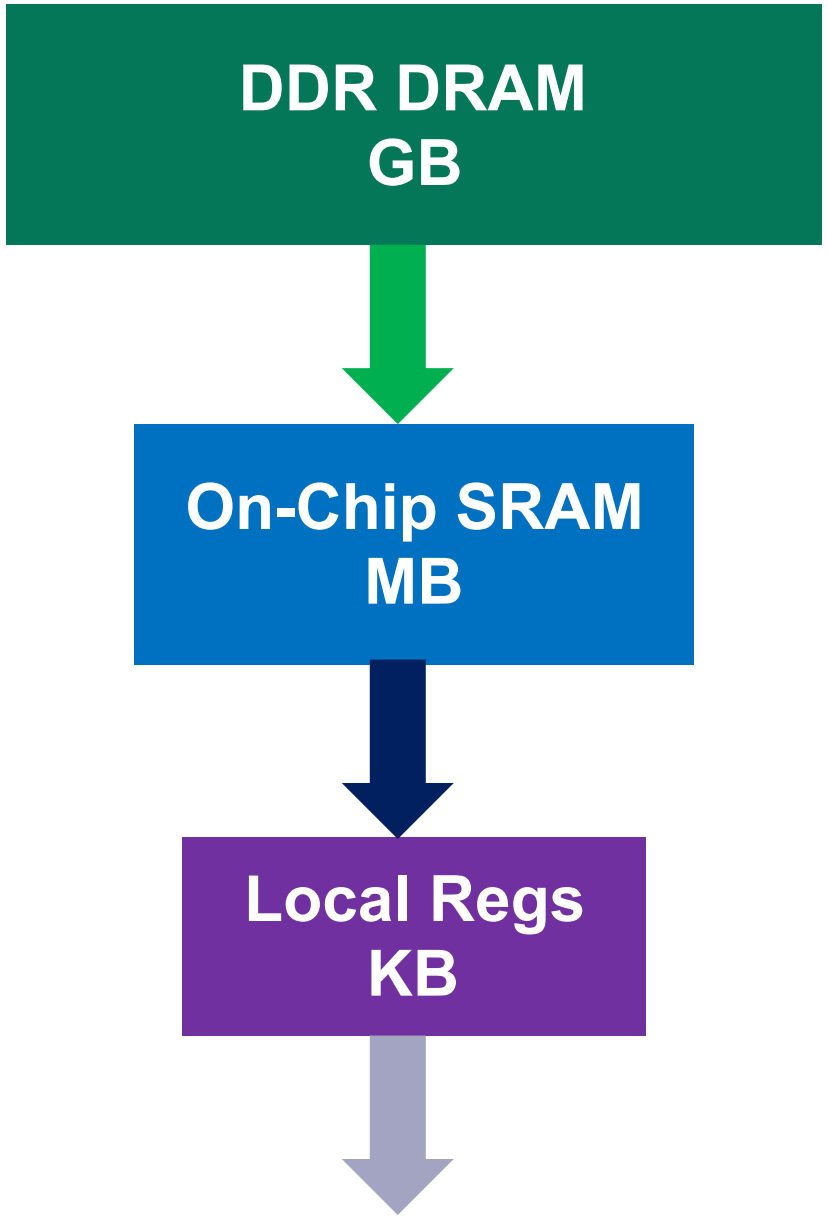



```
val image  = DRAM[UInt8]
(H,W)

buffer load   image(i, j::j+C) // dense
buffer gather image(a)         // sparse

val buffer = SRAM[UInt8](C)
val fifo   = FIFO[Float](D)
val lbuf   = LineBuffer[Int](R,C)
```

Spatial: Memory Hierarchy




```
val image  = DRAM[UInt8]
(H,W)

buffer load   image(i, j::j+C) // dense
buffer gather image(a)         // sparse

val buffer = SRAM[UInt8](C)
val fifo   = FIFO[Float](D)
val lbuf   = LineBuffer[Int](R,C)

val accum  = Reg[Double]
val pixels = RegFile[UInt8](R,C)
```

Spatial: Control And Design Parameters

Spatial: Control And Design Parameters

Implicit/Explicit parallelization factors
(optional, but can be explicitly declared)

```
val P = 16 (1 → 32)
Reduce(0)(N by 1 par P){i =>
  data(i)
}{(a,b) => a + b}
```

Spatial: Control And Design Parameters

Implicit/Explicit parallelization factors
(optional, but can be explicitly declared)

```
val P = 16 (1 → 32)
Reduce(0)(N by 1 par P){i =>
  data(i)
}{(a,b) => a + b}
```

Implicit/Explicit control schemes
(also optional, but can be used to override compiler)

```
Stream.Foreach(0 until N){i =>
  …
}
```

Spatial: Control And Design Parameters

Implicit/Explicit parallelization factors
(optional, but can be explicitly declared)

```
val P = 16 (1 → 32)
Reduce(0)(N by 1 par P){i =>
  data(i)
}{(a,b) => a + b}
```

Implicit/Explicit control schemes
(also optional, but can be used to override compiler)

```
Stream.Foreach(0 until N){i =>
  …
}
```

Explicit size parameters for loop step size and buffer sizes
(informs compiler it can tune this value)

```
val B = 64 (64 → 1024)
val buffer = SRAM[Float](B)
Foreach(N by B){i =>
  …
}
```

Spatial: Control And Design Parameters

Implicit/Explicit parallelization factors
(optional, but can be explicitly declared)

```
val P = 16 (1 → 32)
Reduce(0)(N by 1 par P){i =>
  data(i)
}{(a,b) => a + b}
```

Implicit/Explicit control schemes
(also optional, but can be used to override compiler)

```
Stream.Foreach(0 until N){i =>
  …
}
```

Explicit size parameters for loop step size and buffer sizes
(informs compiler it can tune this value)

```
val B = 64 (64 → 1024)
val buffer = SRAM[Float](B)
Foreach(N by B){i =>
  …
}
```

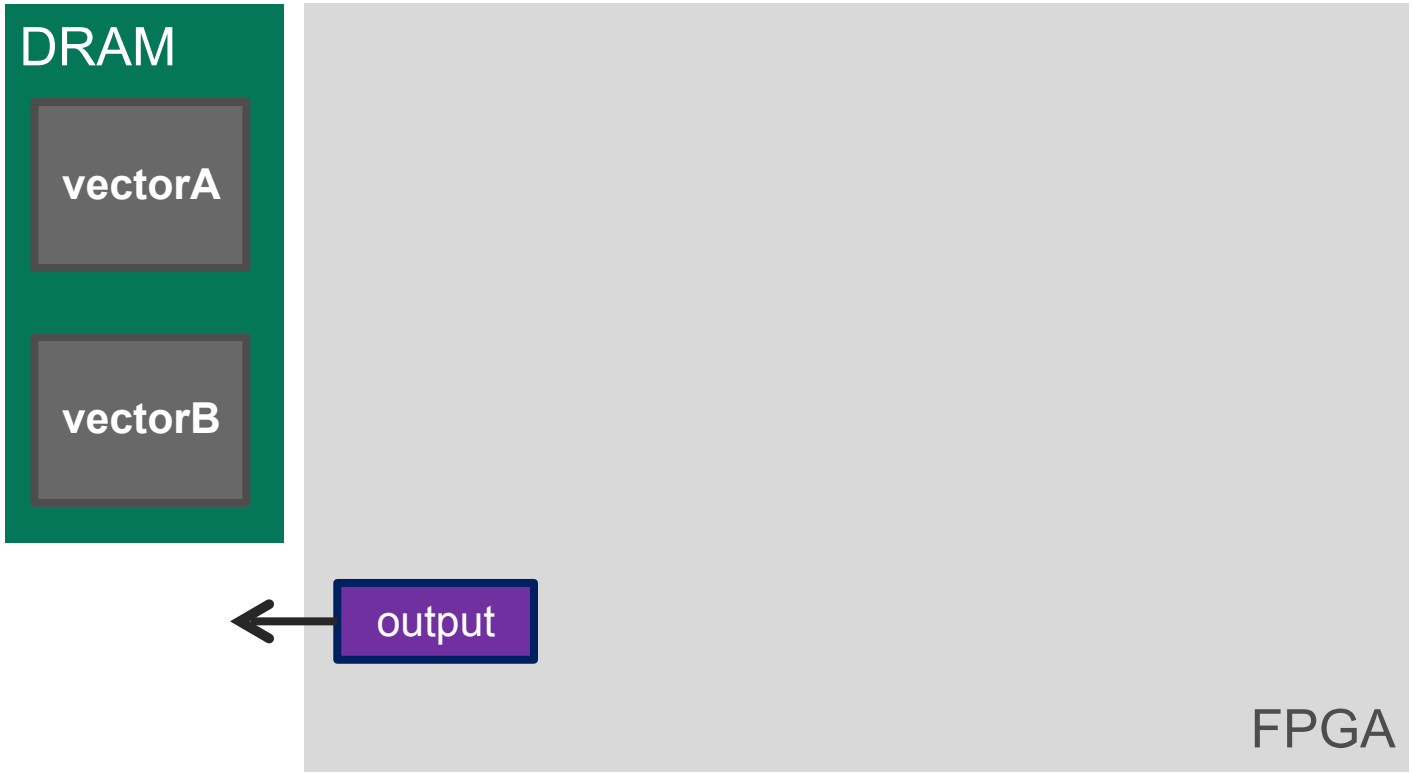
Implicit memory banking and buffering schemes for parallelized access

```
Foreach(64 par 16){i =>
  buffer(i) // Parallel read
}
```

Dot Product in Spatial

```
val output  = ArgOut[Float]
val vectorA = DRAM[Float](N)
val vectorB = DRAM[Float](N)
```

Off-chip memory declarations

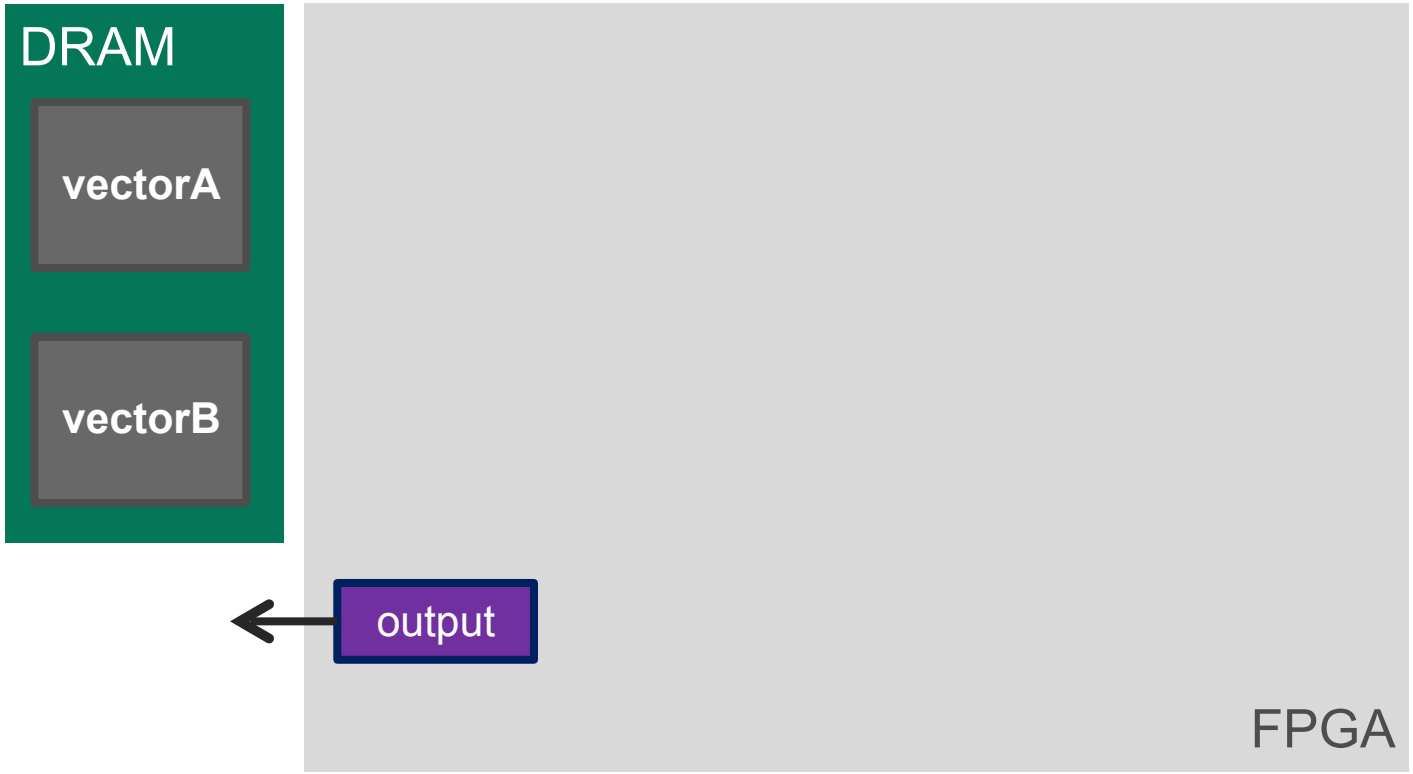


Dot Product in Spatial

```
val output  = ArgOut[Float]
val vectorA = DRAM[Float](N)
val vectorB = DRAM[Float](N)

Accel {
```

Explicit work division in IR

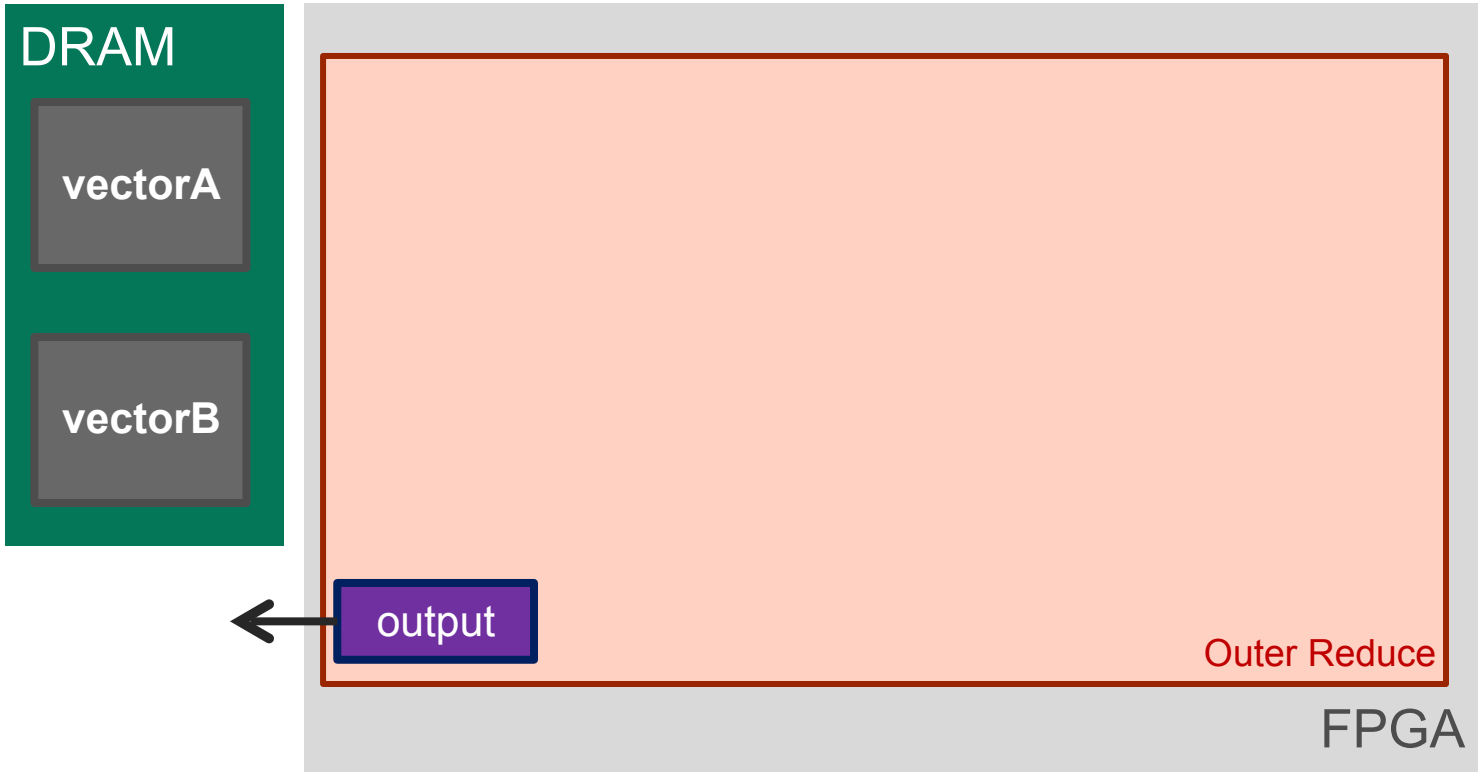


Dot Product in Spatial

```
val output  = ArgOut[Float]
val vectorA = DRAM[Float](N)
val vectorB = DRAM[Float](N)

Accel {
  Reduce(output)(N by B){ i =>
```

Tiled reduction (outer)



Dot Product in Spatial

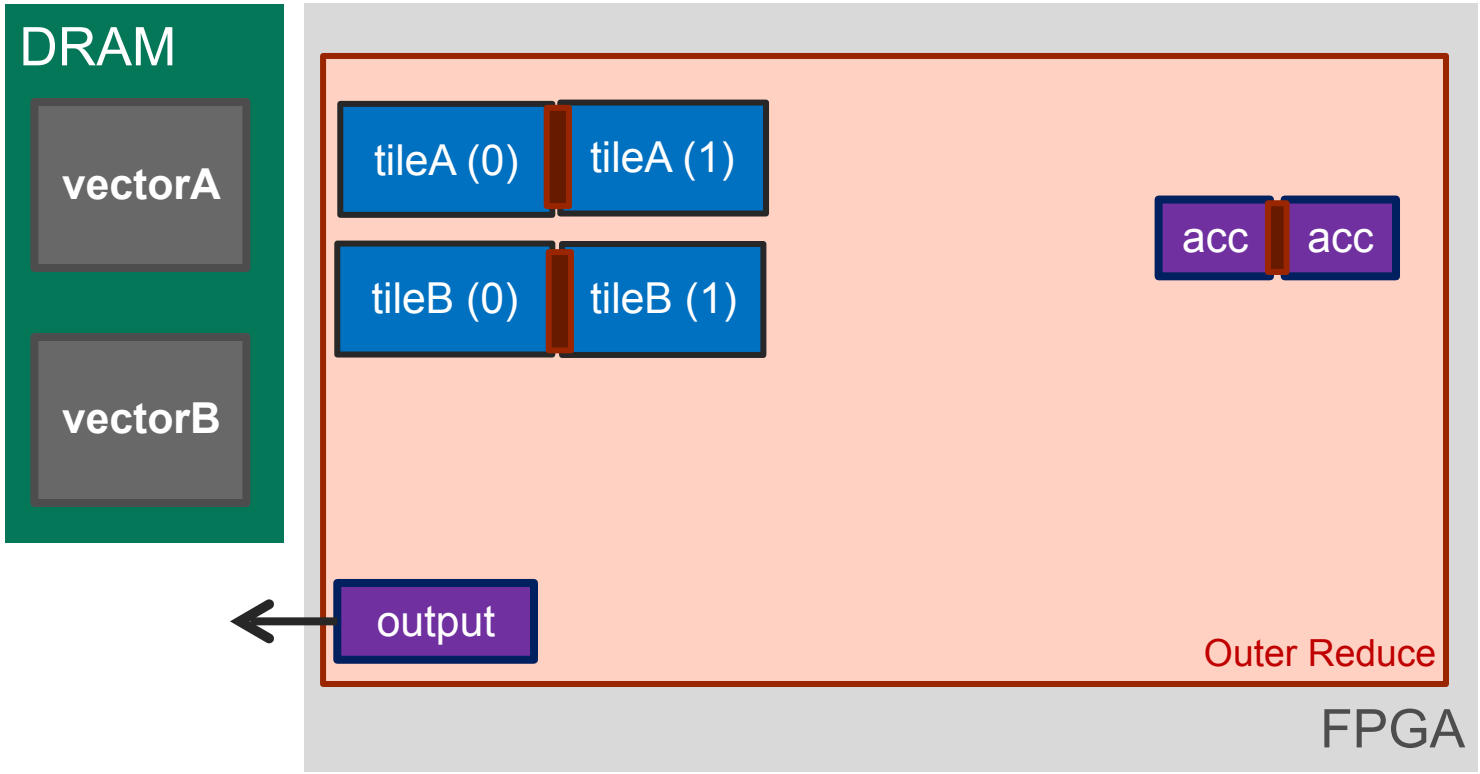
```
val output  = ArgOut[Float]
val vectorA = DRAM[Float](N)
val vectorB = DRAM[Float](N)

Accel {
  Reduce(output)(N by B){ i =>
    val tileA = SRAM[Float](B)
    val tileB = SRAM[Float](B)
    val acc   = Reg[Float]



}
```

On-chip memory declarations



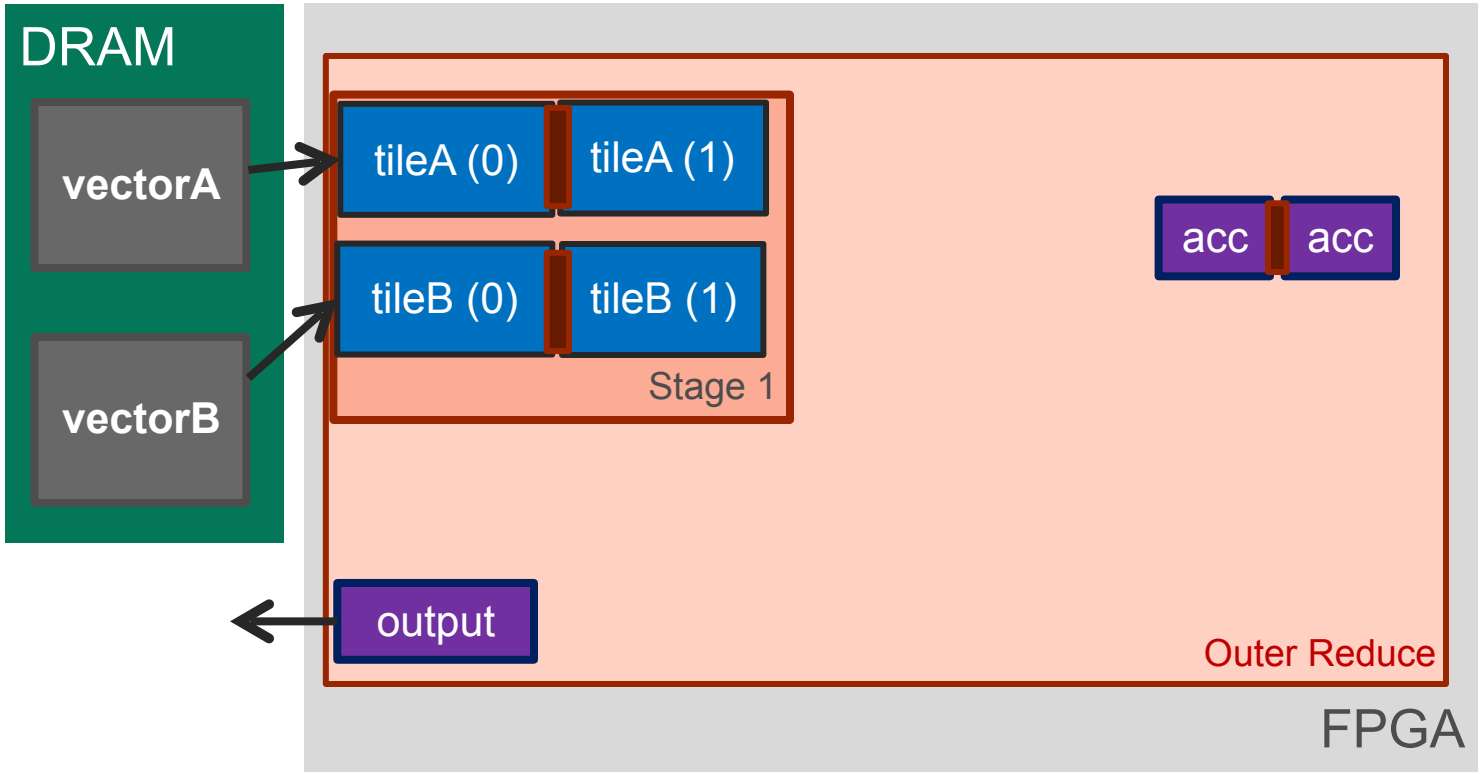
Dot Product in Spatial

```
val output  = ArgOut[Float]
val vectorA = DRAM[Float](N)
val vectorB = DRAM[Float](N)

Accel {
  Reduce(output)(N by B){ i =>
    val tileA = SRAM[Float](B)
    val tileB = SRAM[Float](B)
    val acc   = Reg[Float]

    tileA load vectorA(i ::
i+B)
    tileB load vectorB(i ::
i+B)
```

DRAM → SRAM transfers
(also have **store**, **scatter**, and **gather**)



Dot Product in Spatial

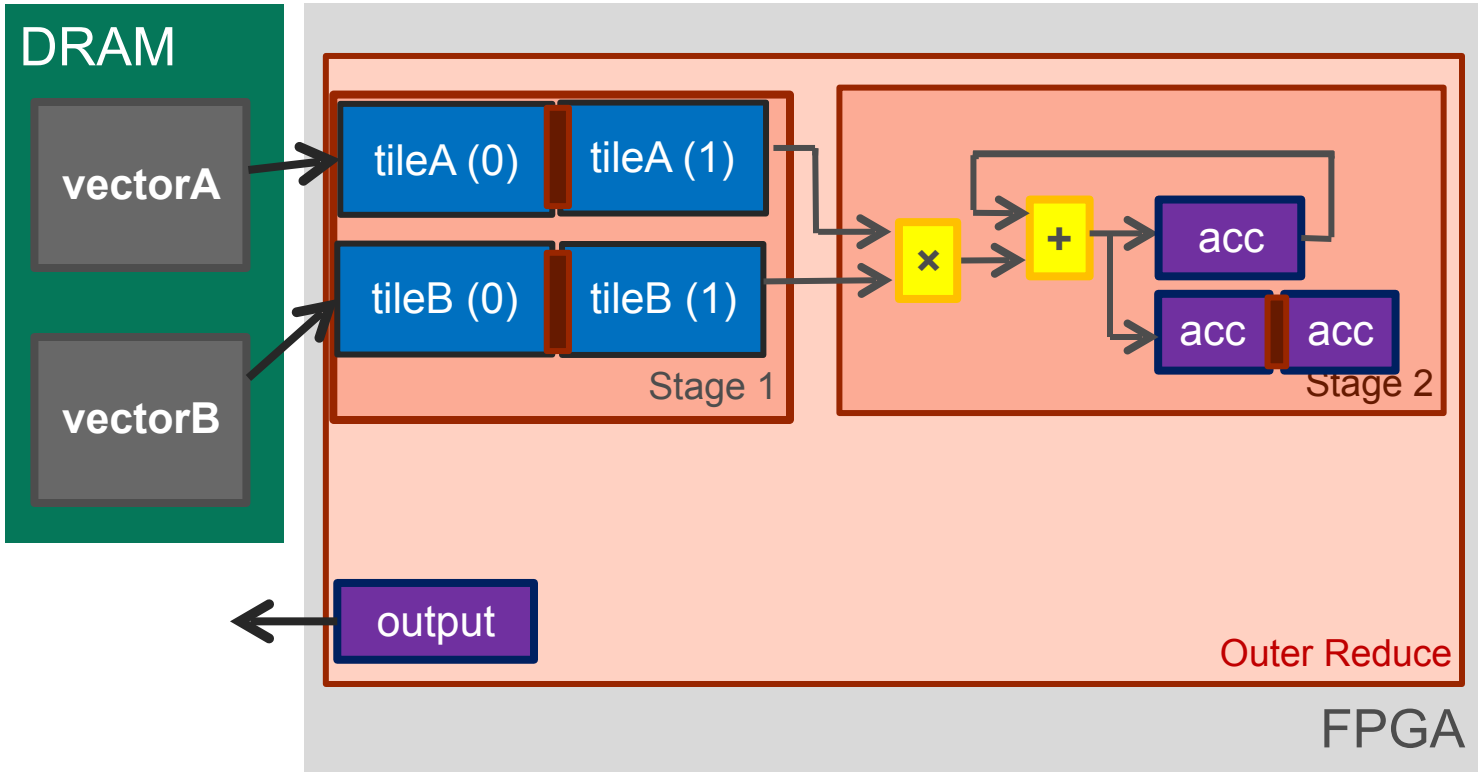
```
val output  = ArgOut[Float]
val vectorA = DRAM[Float](N)
val vectorB = DRAM[Float](N)

Accel {
  Reduce(output)(N by B){ i =>
    val tileA = SRAM[Float](B)
    val tileB = SRAM[Float](B)
    val acc   = Reg[Float]

    tileA load vectorA(i ::
i+B)
    tileB load vectorB(i ::
i+B)

    Reduce(acc)(B by 1){ j =>
      tileA(j) * tileB(j)
    }{a, b => a + b}
```

Tiled reduction (pipelined)



Dot Product in Spatial

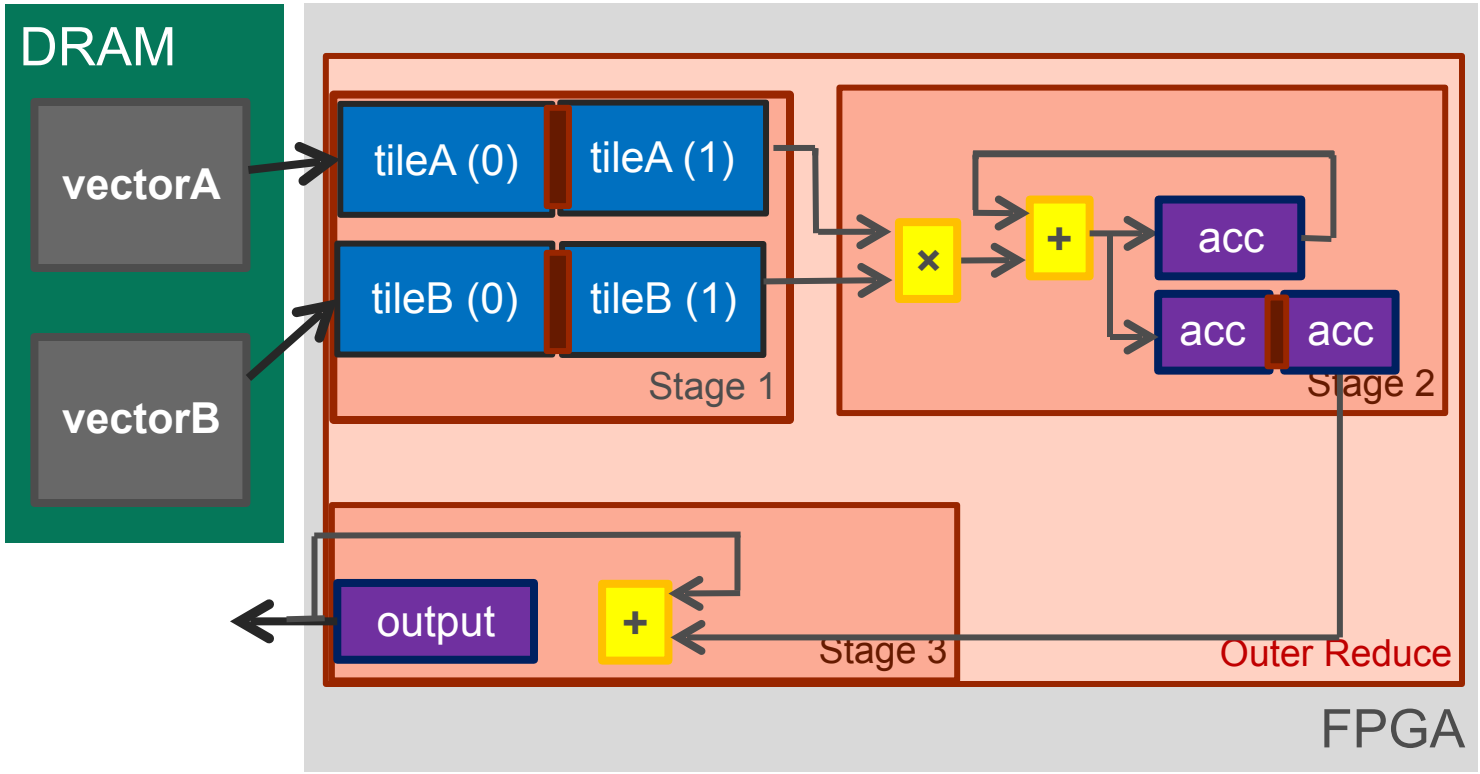
```
val output  = ArgOut[Float]
val vectorA = DRAM[Float](N)
val vectorB = DRAM[Float](N)

Accel {
  Reduce(output)(N by B){ i =>
    val tileA = SRAM[Float](B)
    val tileB = SRAM[Float](B)
    val acc   = Reg[Float]

    tileA load vectorA(i ::
i+B)
    tileB load vectorB(i ::
i+B)

    Reduce(acc)(B by 1){ j =>
      tileA(j) * tileB(j)
    }{a, b => a + b}
  }{a, b => a + b}
```

Outer reduce function



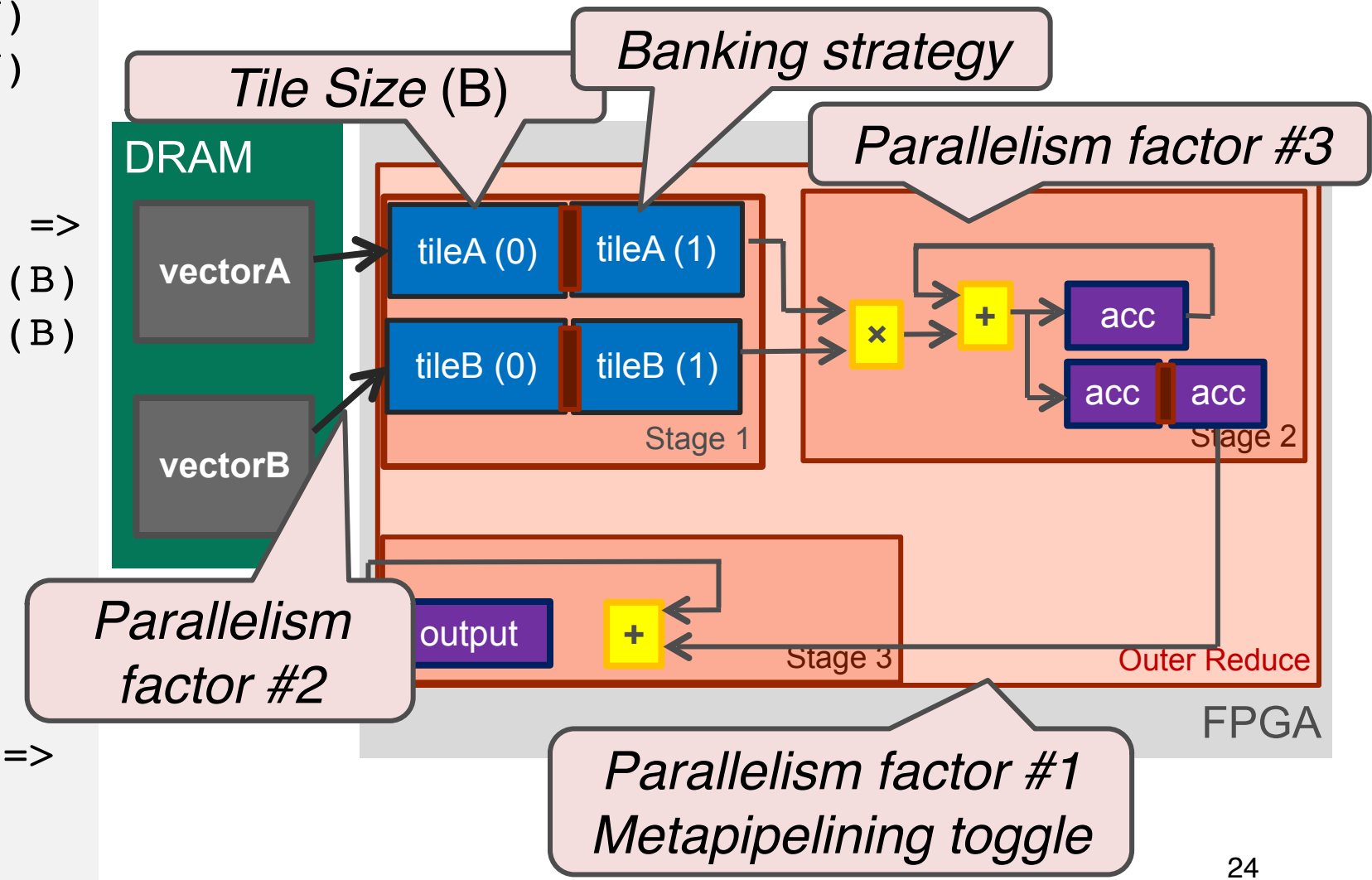
Dot Product in Spatial

```
val output  = ArgOut[Float]
val vectorA = DRAM[Float](N)
val vectorB = DRAM[Float](N)

Accel {
  Reduce(output)(N by B){ i =>
    val tileA = SRAM[Float](B)
    val tileB = SRAM[Float](B)
    val acc   = Reg[Float]

    tileA load vectorA(i ::
i+B)
    tileB load vectorB(i ::
i+B)

    Reduce(acc)(B by 1){ j =>
      tileA(j) * tileB(j)
    }{a, b => a + b}
  }{a, b => a + b}
```



Dot Product in Spatial

```
val output  = ArgOut[Float]
val vectorA = DRAM[Float](N)
val vectorB = DRAM[Float](N)

Accel {
  Reduce(output)(N by B){ i =>
    val tileA = SRAM[Float](B)
    val tileB = SRAM[Float](B)
    val acc   = Reg[Float]

    tileA load vectorA(i ::
i+B)
    tileB load vectorB(i ::
i+B)

    Reduce(acc)(B by 1){ j =>
      tileA(j) * tileB(j)
    }{a, b => a + b}
  }{a, b => a + b}
```

Dot Product in Spatial

Spatial Program *Design Parameters*

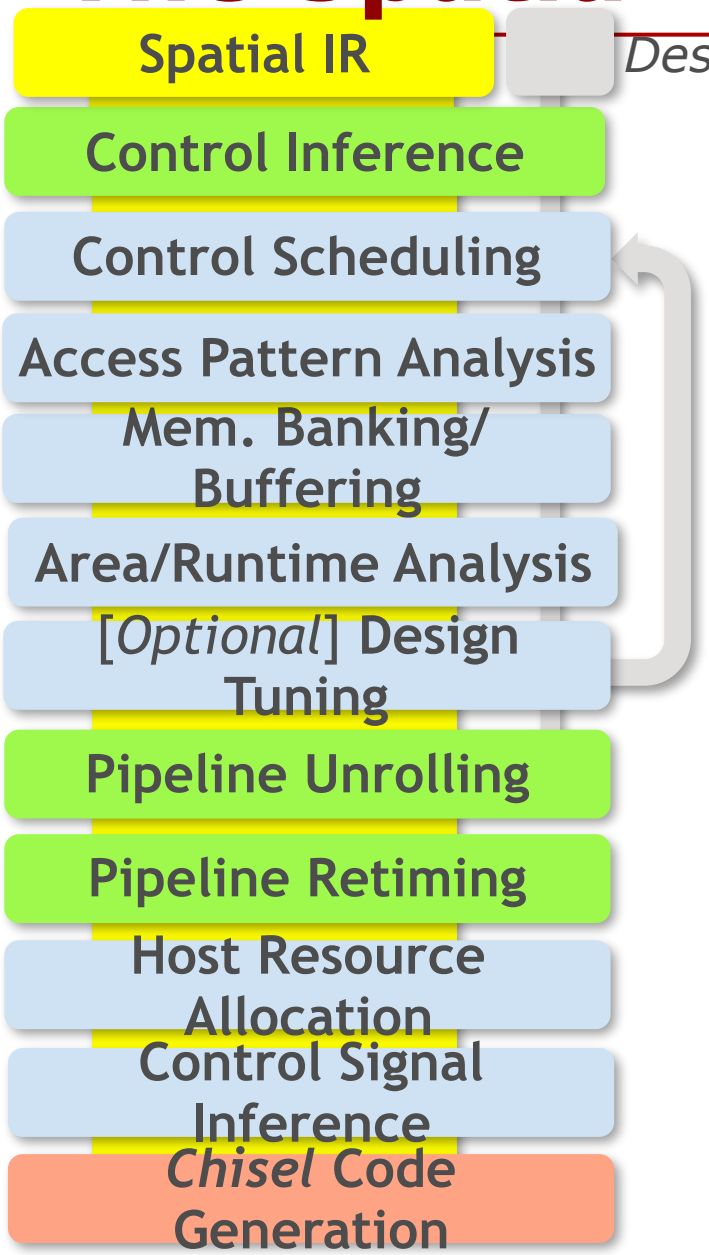
The Spatial Compiler

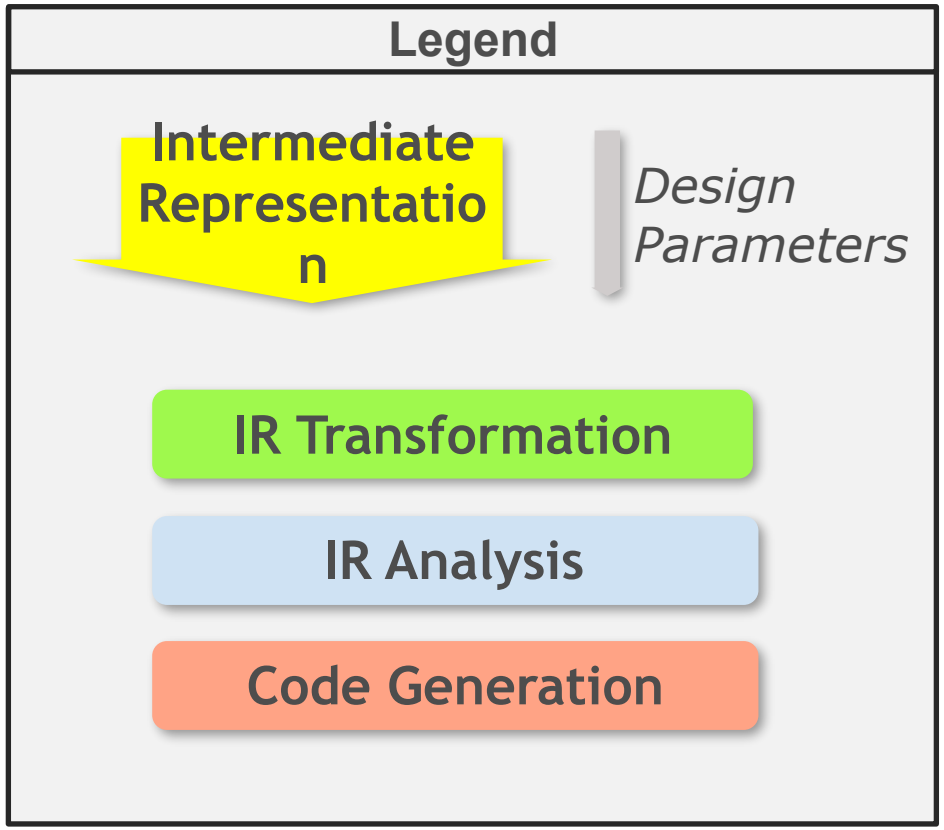
Spatial Program

The Spatial Compiler

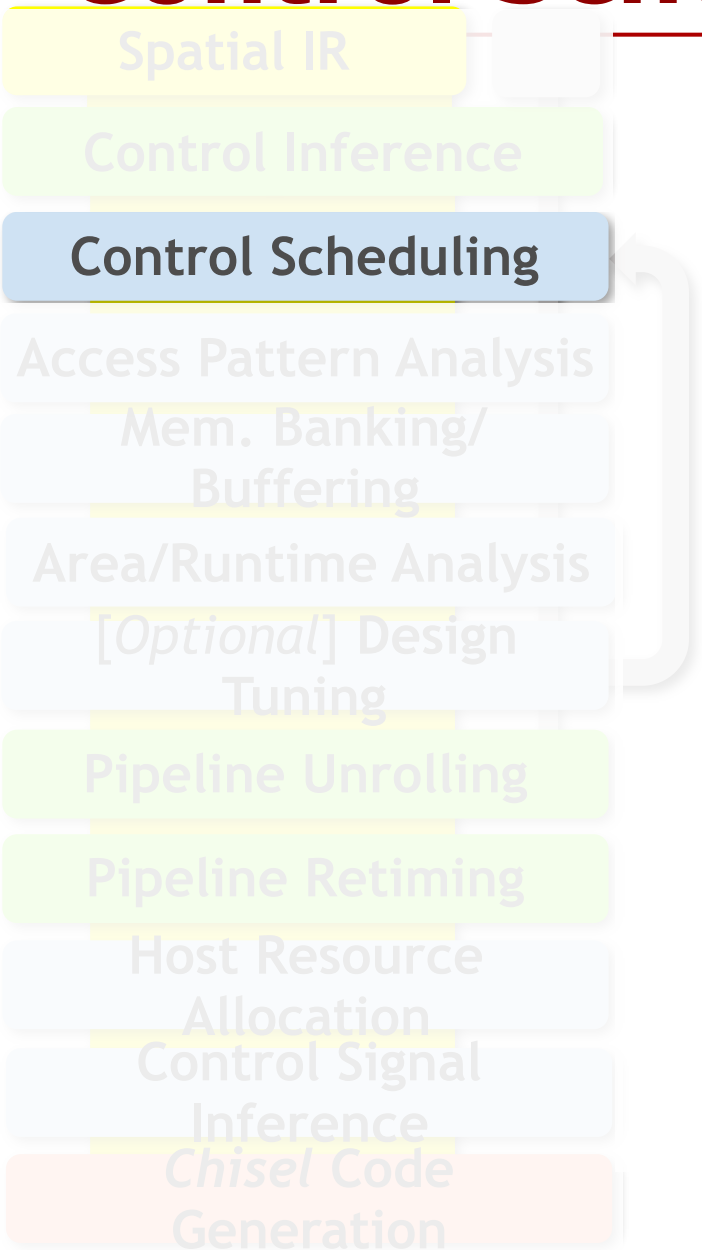
Spatial IR

The Spatial Compiler





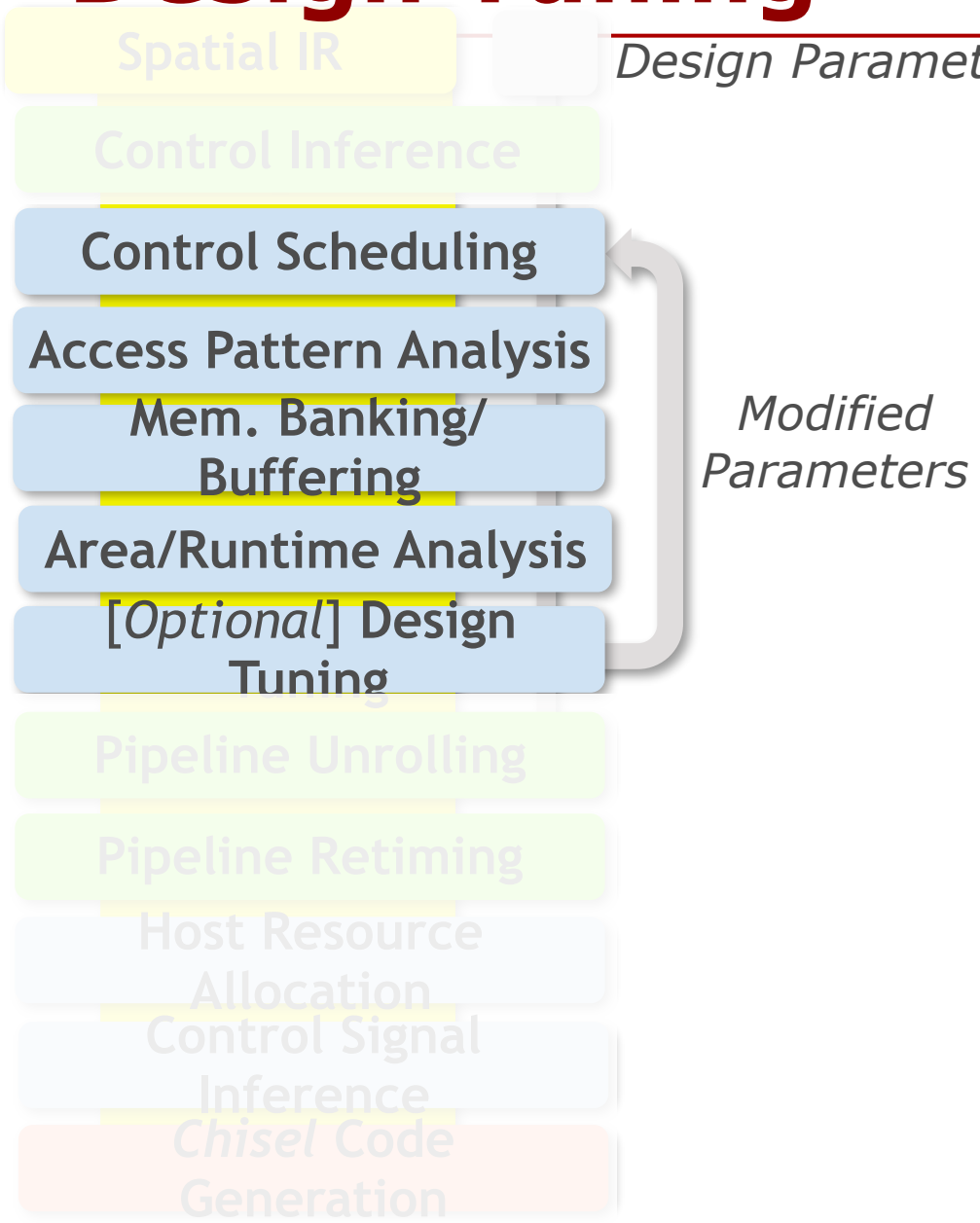
Control Scheduling



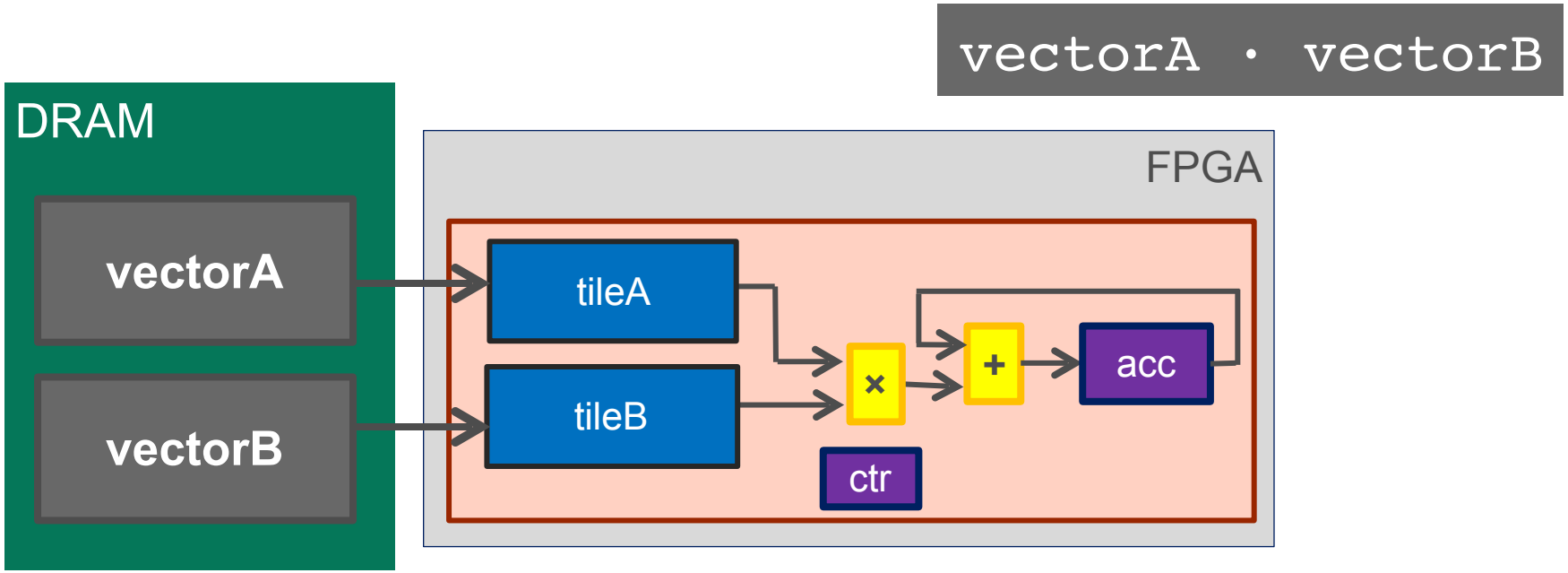
- Creates loop pipeline schedules
  - Detects data dependencies across loop intervals
  - Calculate initiation interval of pipelines
  - Set maximum depth of buffers
- Supports **arbitrarily nested** pipelines

(Commercial HLS tools don't support this)

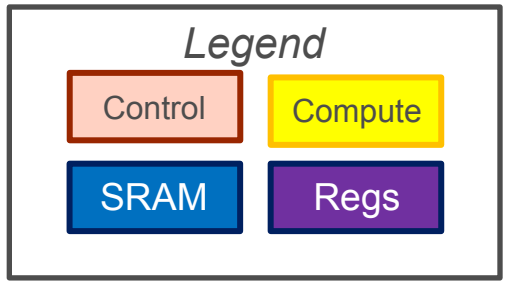
Design Tuning



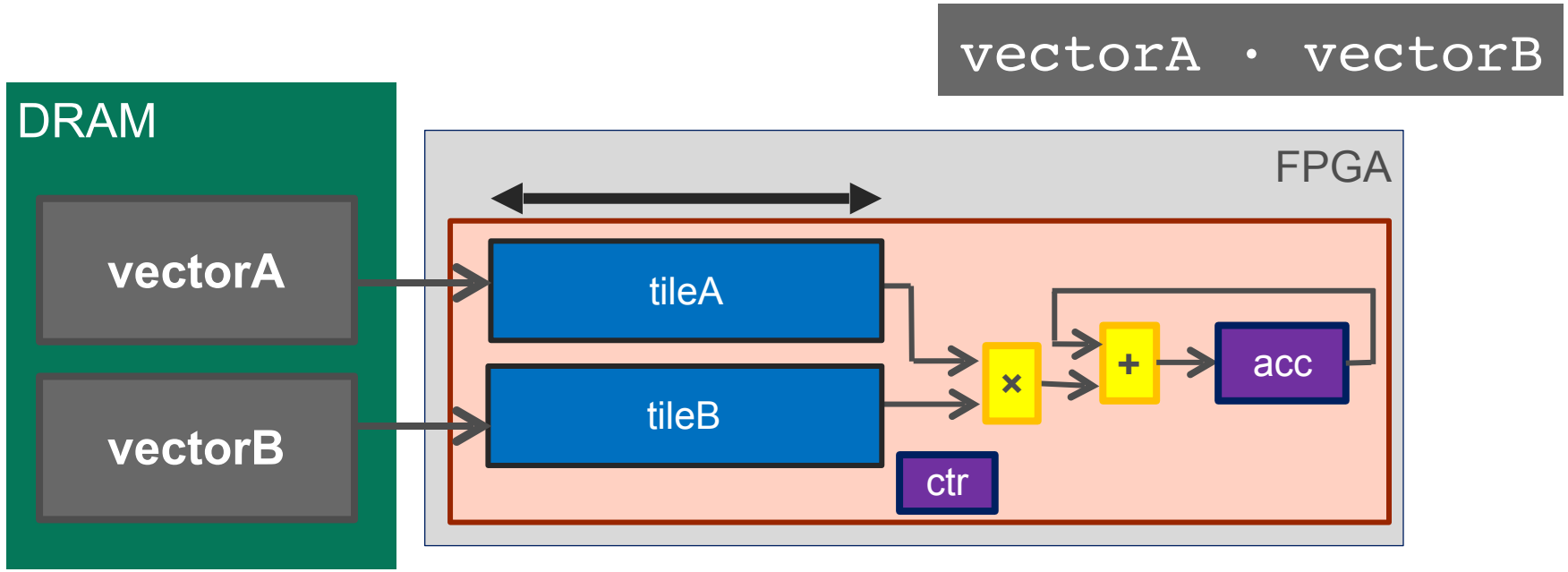
Design Space Parameters Example



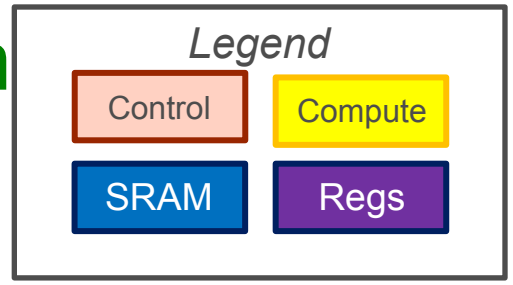
Small and simple, but slow!



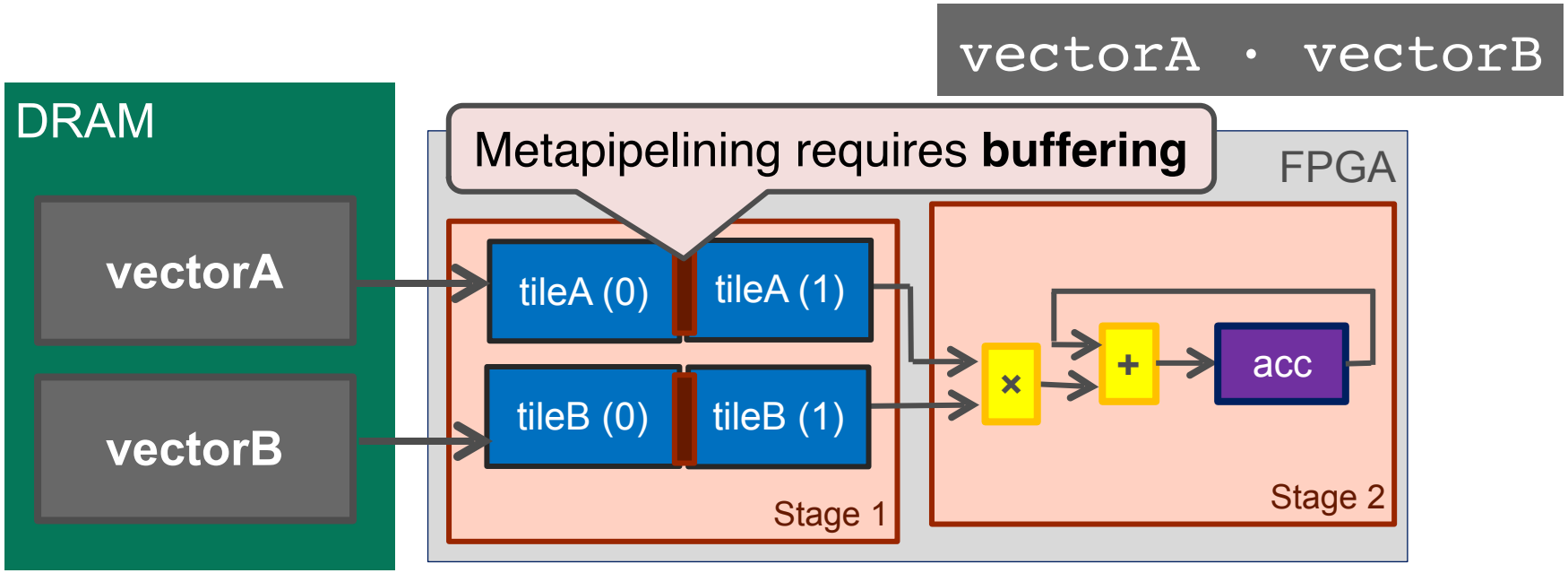
Important Parameters: Buffer Sizes



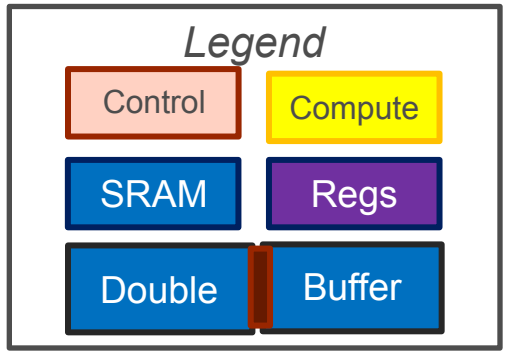
- Increases length of DRAM accesses ↓ Runtime
- Increases exploited locality ↓ Runtim
- Increases local memory sizes ↑ Area



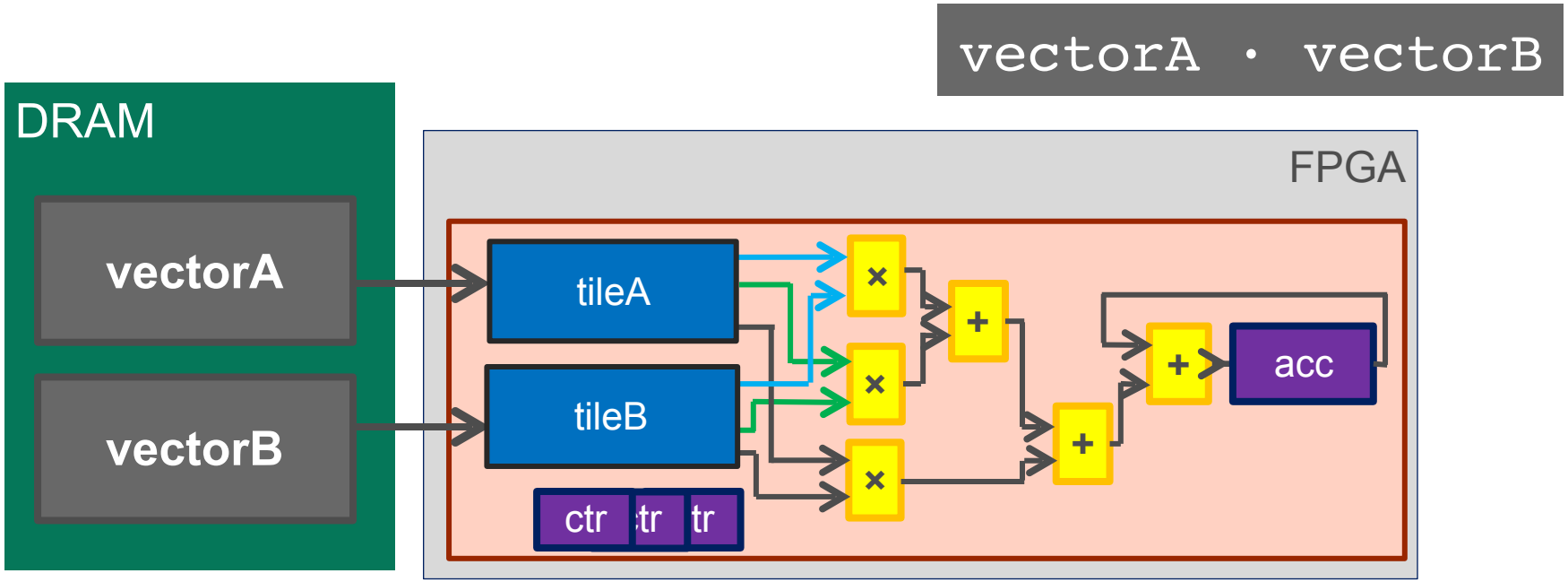
Important Parameters: Pipelining



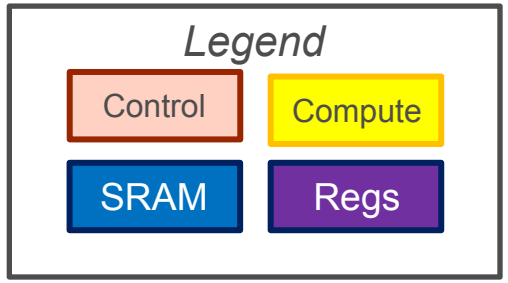
- Overlaps memory and comput ↓ Runtime
- Increases local memory sizes ↑ Area
- Adds synchronization logic ↑ Area



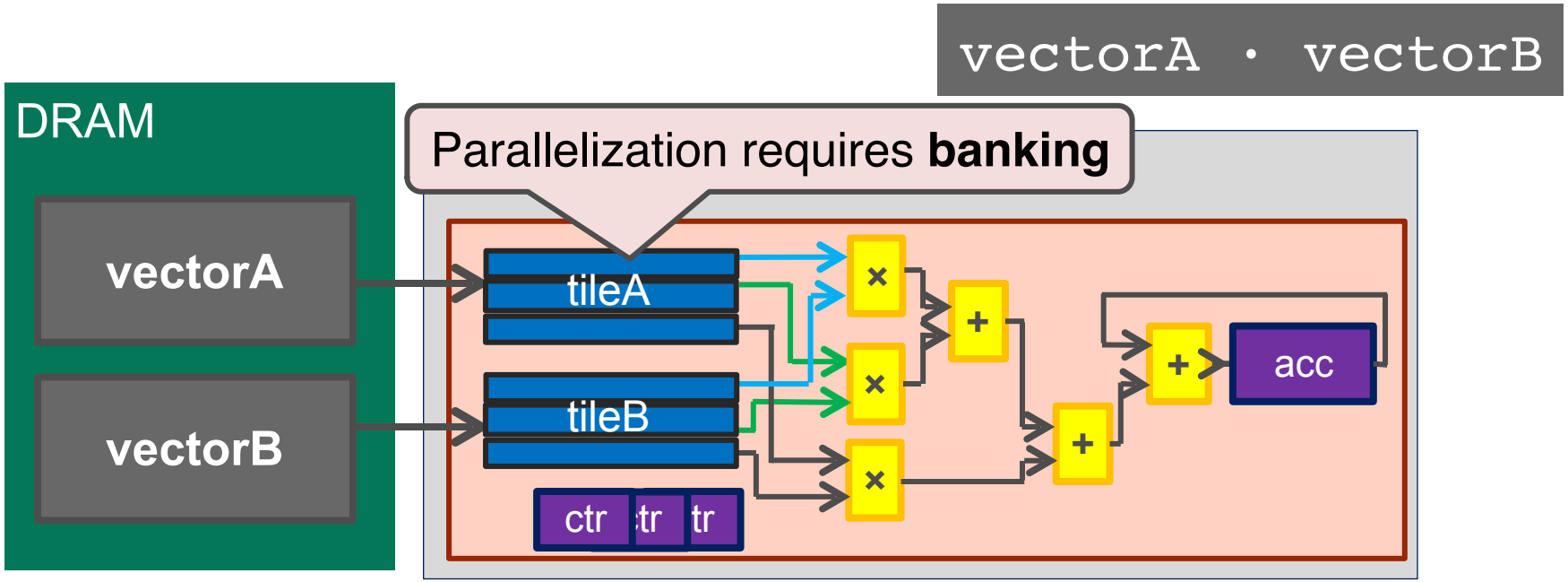
Important Parameters: Parallelization



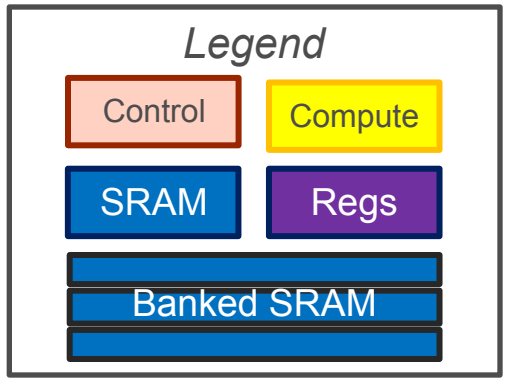
- Improves element throughput ↓ Runtime
- Duplicates compute resources ↑ Area



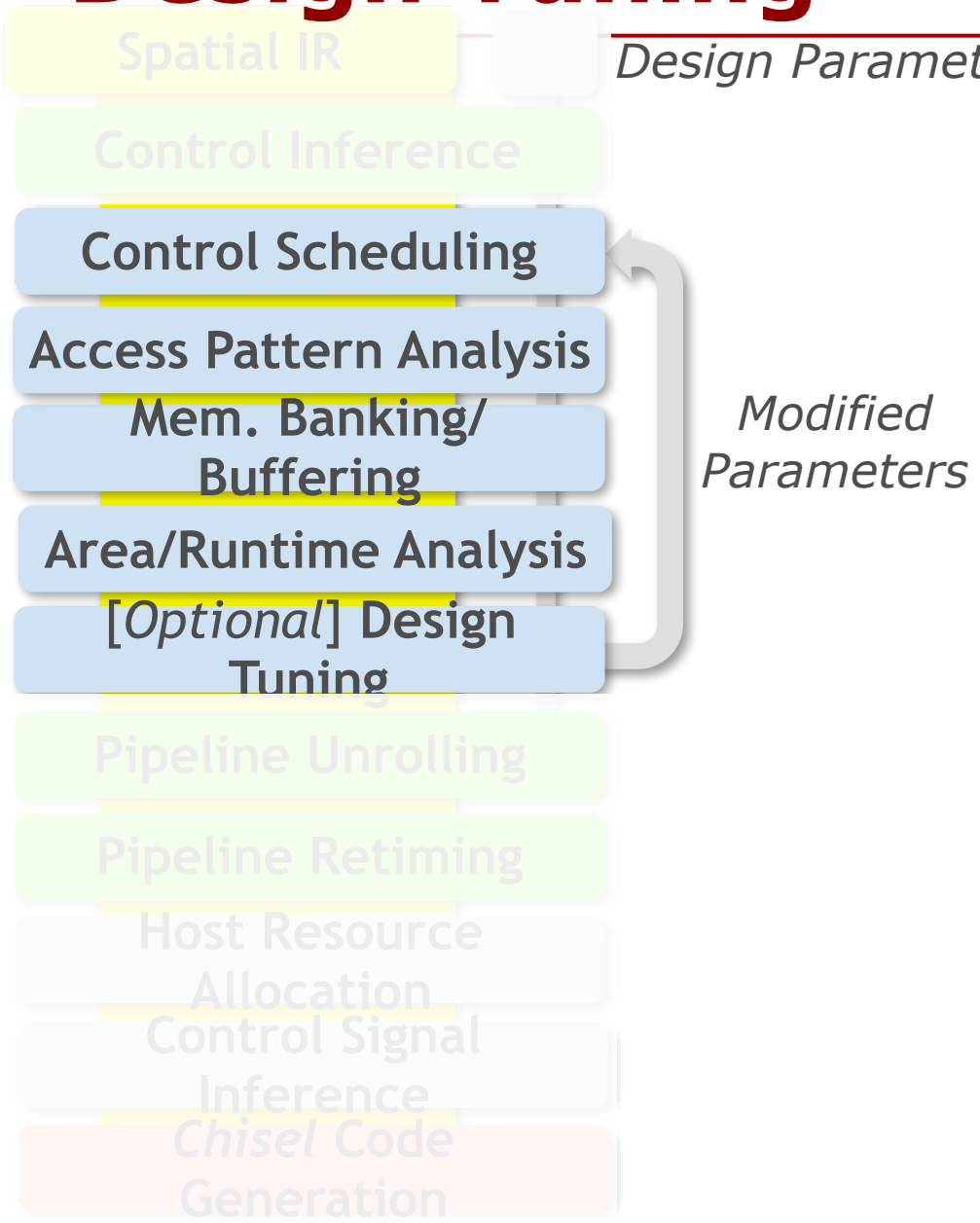
Important Parameters: Memory Banking



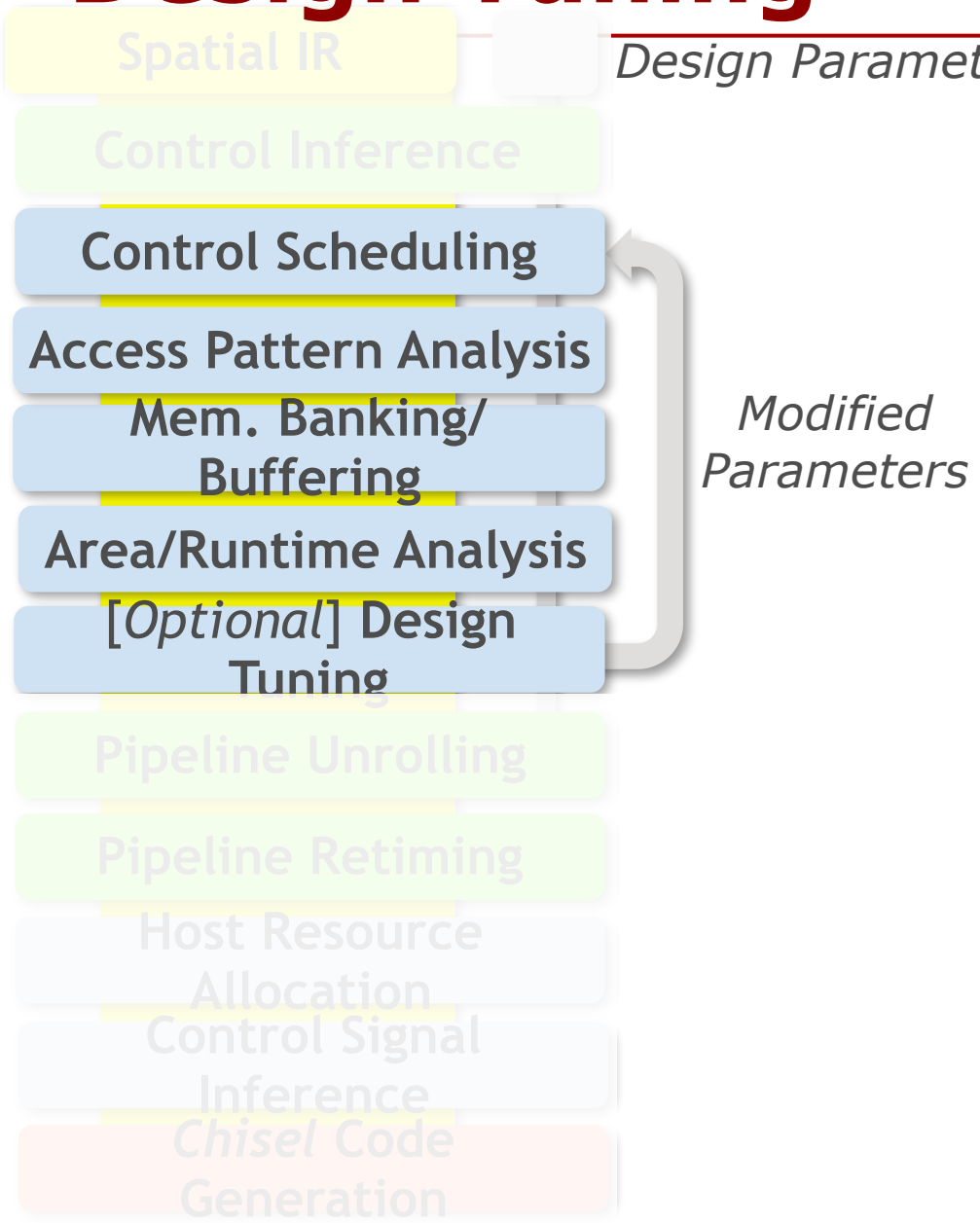
- Improves memory bandwidth ⬇ Runtime
- May duplicate memory resource ⬆ Area



Design Tuning



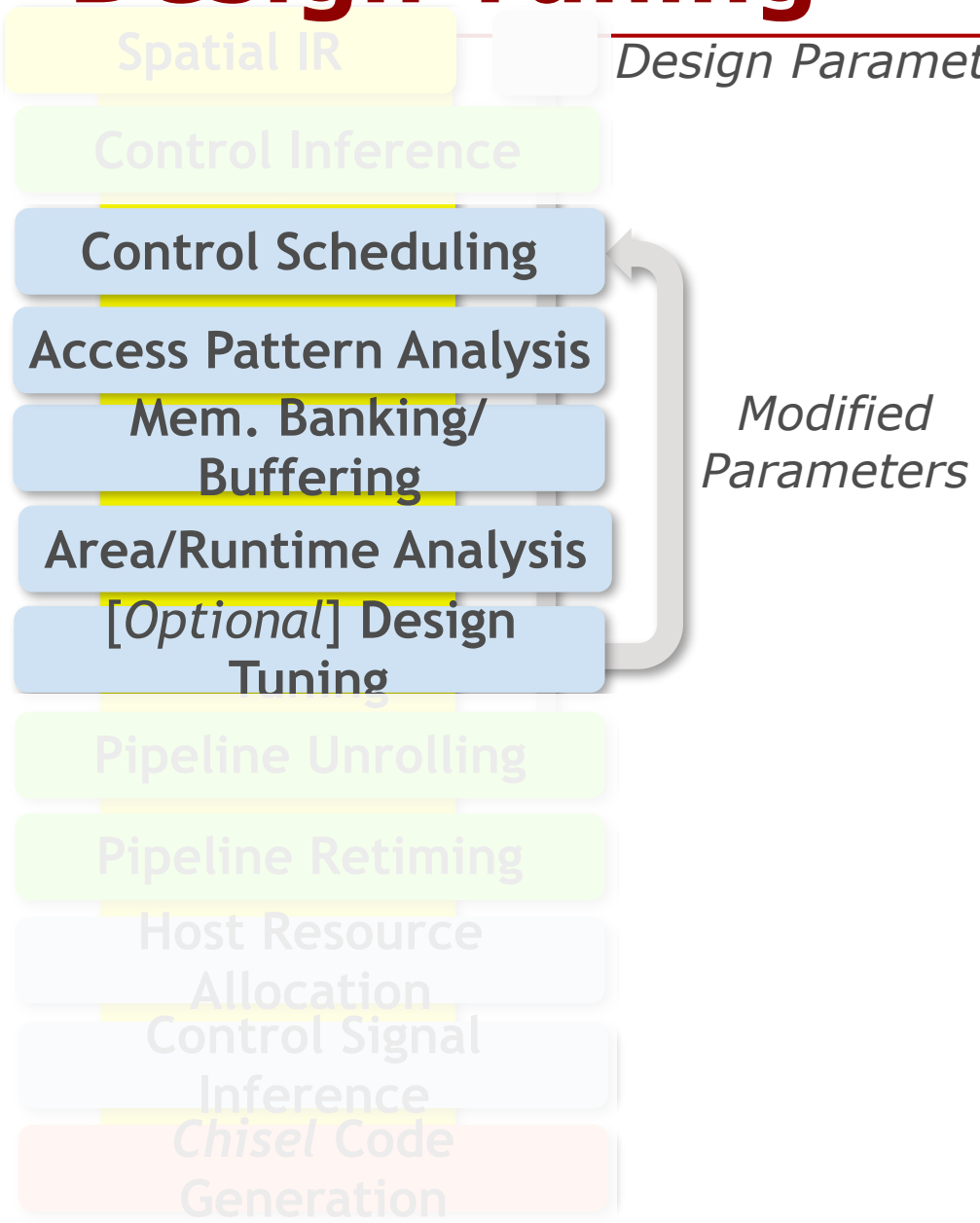
Design Tuning



Original tuning methods:

- Pre-prune space using simple heuristics
- Randomly sample ~100,000 design points
- **Model** area/runtime of each point

Design Tuning



Original tuning methods:

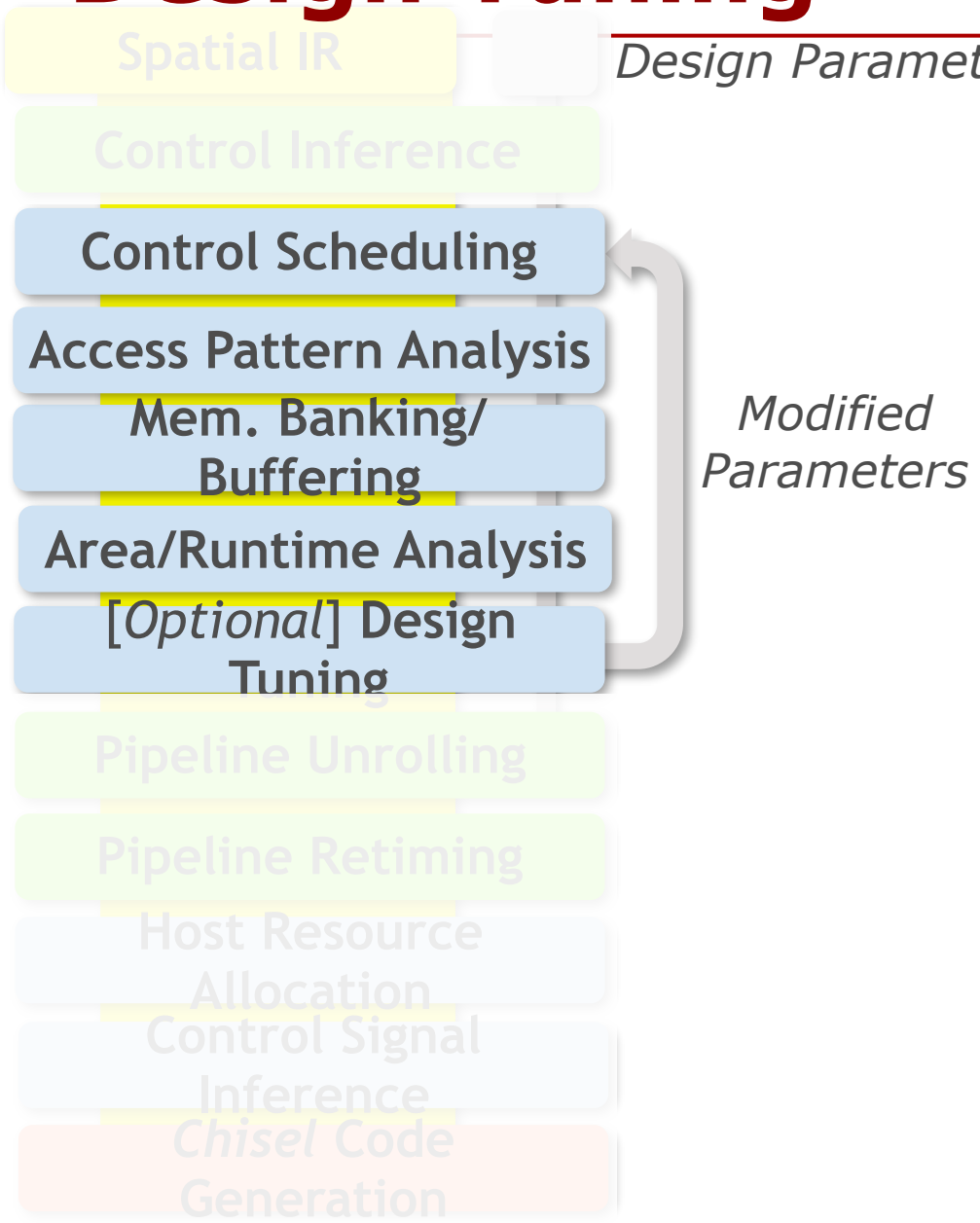
- Pre-prune space using simple heuristics
- Randomly sample ~100,000 design points
- **Model** area/runtime of each point

Proposed tuning method

- Active learning: HyperMapper

(More details in paper)

Design Tuning



Original tuning methods:

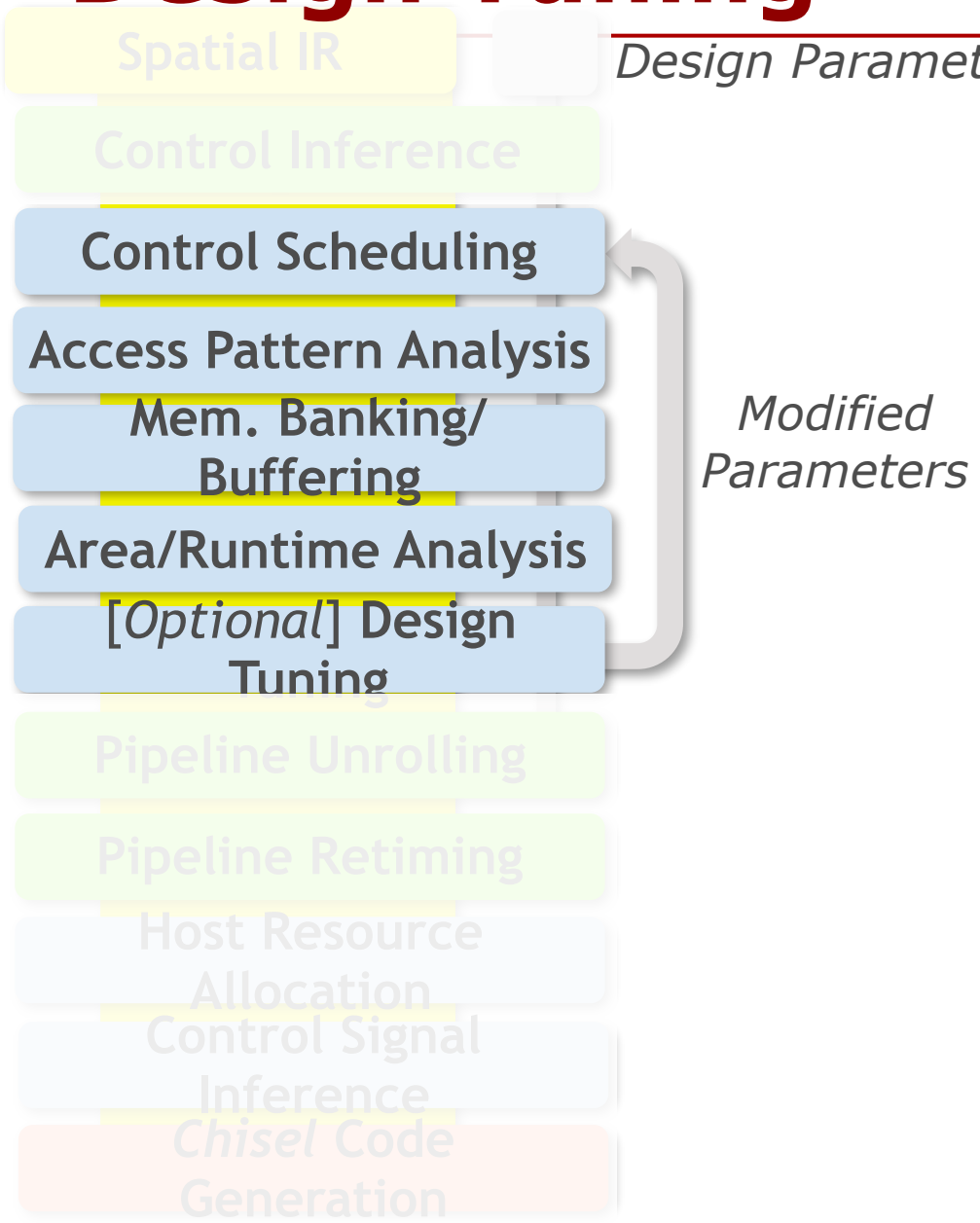
- Pre-prune space using simple heuristics
- Randomly sample ~100,000 design points
- **Model** area/runtime of each point

Proposed tuning method

- Active learning: HyperMapper

(More details in paper)

Design Tuning



Original tuning methods:

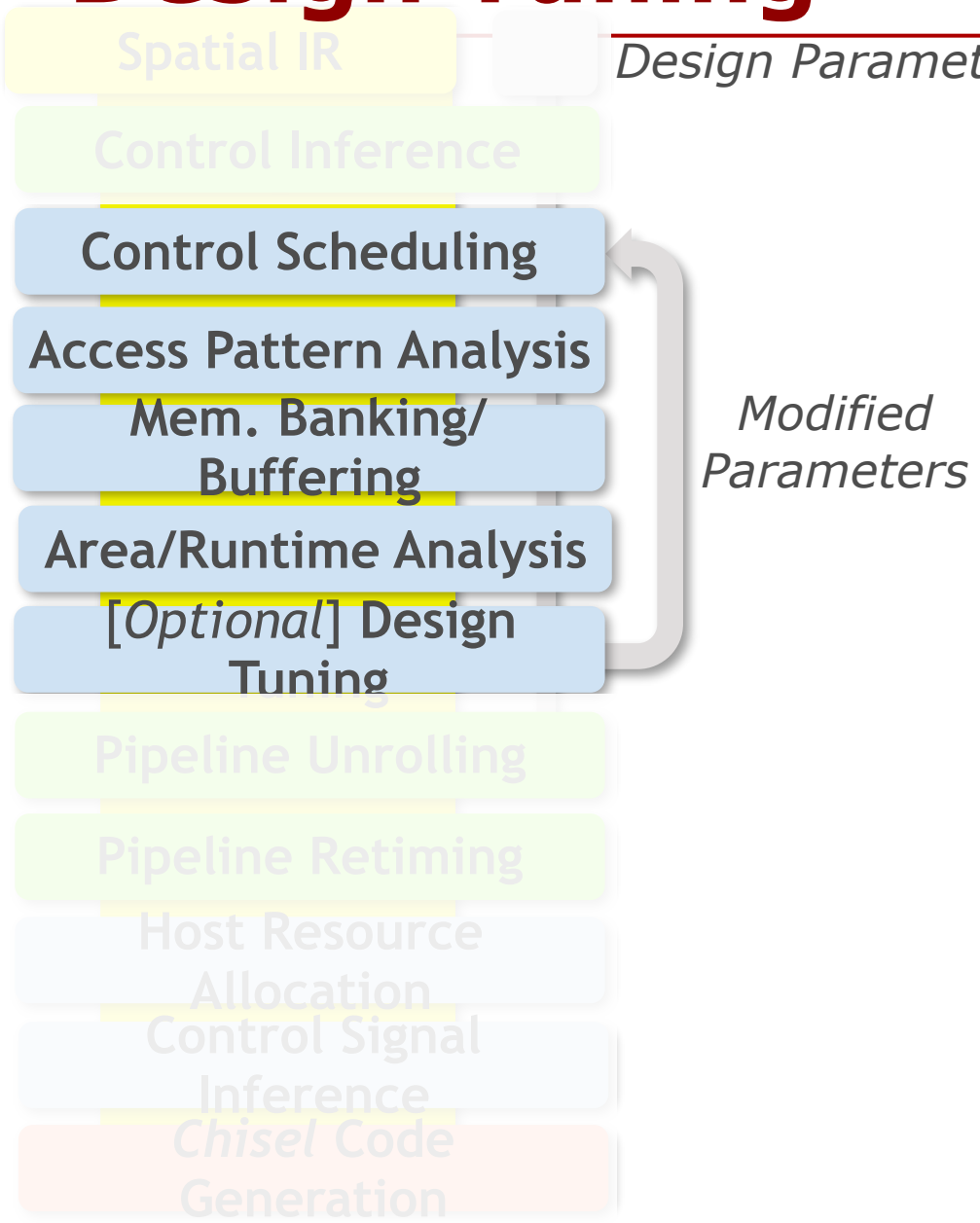
- Pre-prune space using simple heuristics
- Randomly sample ~100,000 design points
- **Model** area/runtime of each point

Proposed tuning method

- Active learning: HyperMapper

(More details in paper)

Design Tuning



Original tuning methods:

- Pre-prune space using simple heuristics
- Randomly sample ~100,000 design points
- **Model** area/runtime of each point

Proposed tuning method

- Active learning: HyperMapper

(More details in paper)

The Spatial Compiler: The Rest

Code generation

- Synthesizable Chisel
- C++ code for host CPU

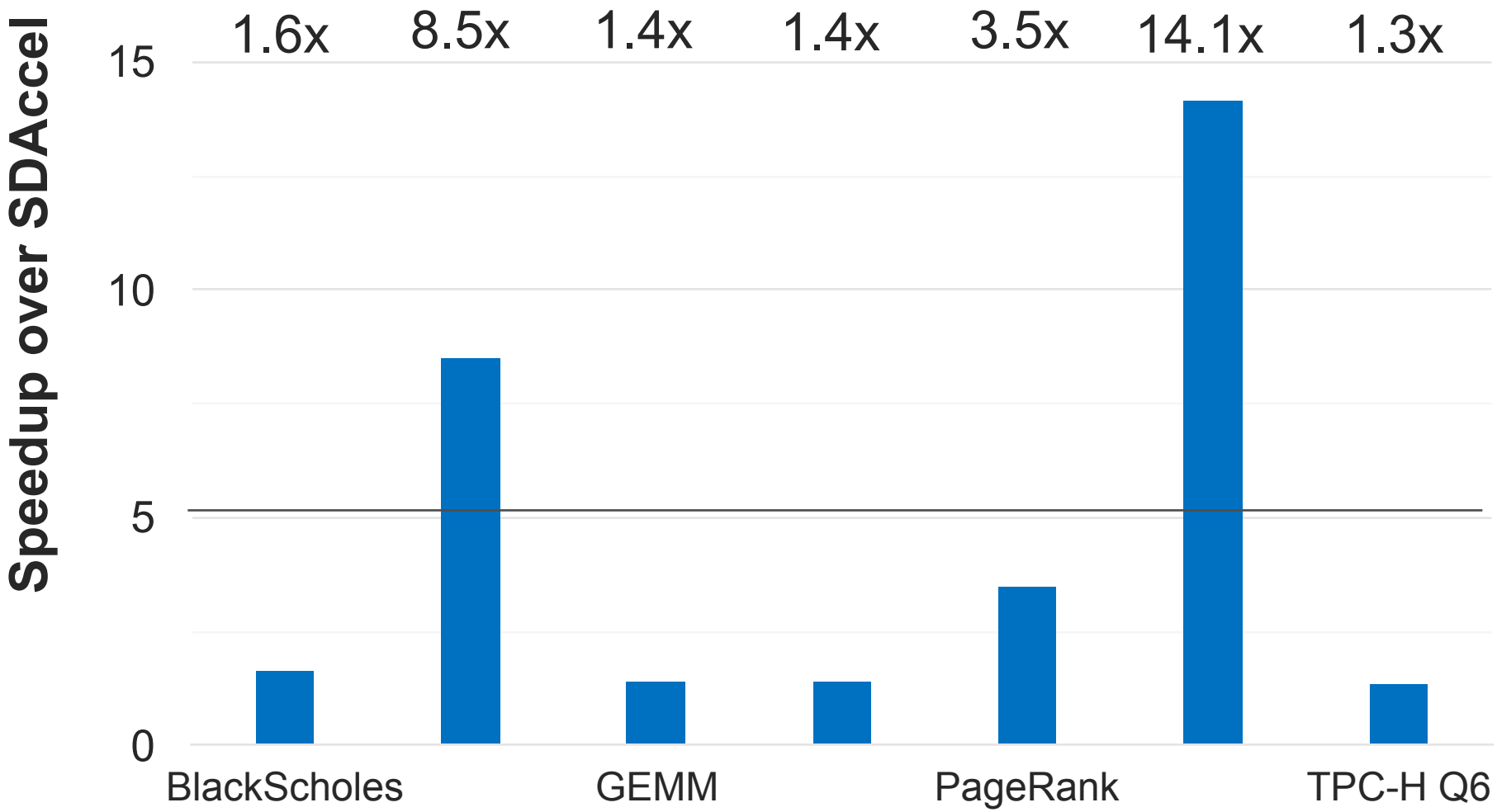
Evaluation: Performance

- FPGA:
  - Amazon EC2 F1 Instance: Xilinx VU9P FPGA
  - Fixed clock rate of 150 MHz
- Applications
  - SDAccel: Hand optimized, tuned implementations
  - Spatial: Hand written, automatically tuned implementations

- Execution time = ***FPGA execution time***

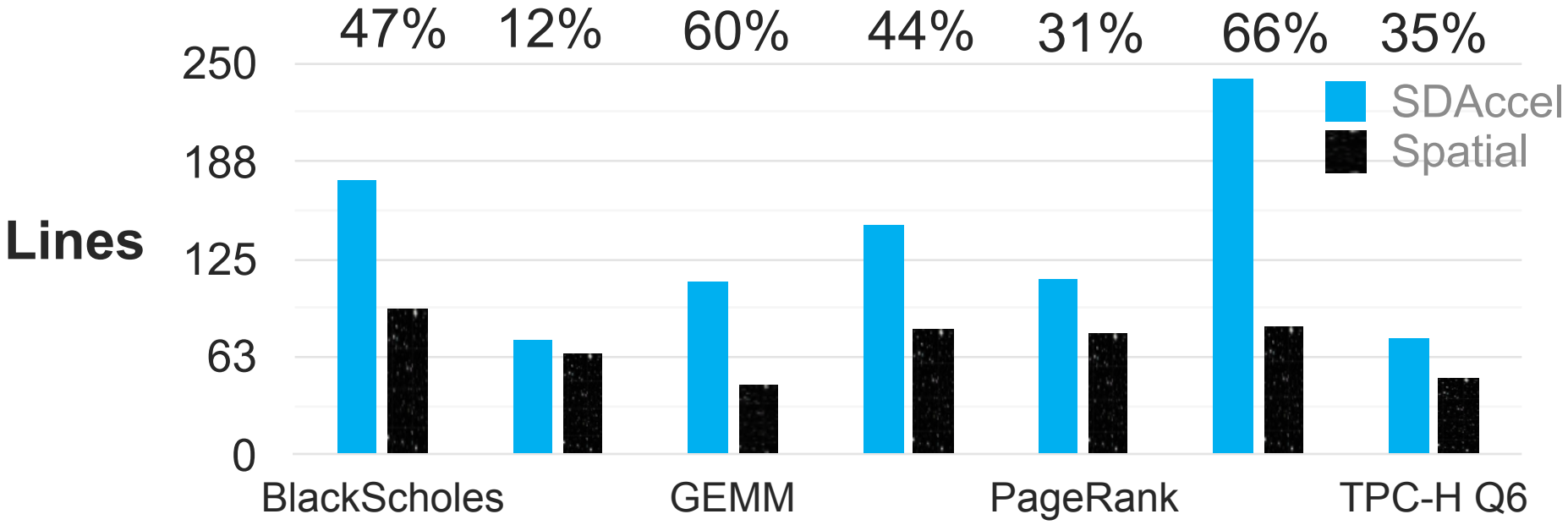
Performance (Spatial vs. SDAccel)

Average **2.9x faster hardware** than SDAccel



Productivity: Lines of Code

Average **42% shorter** programs versus SDAccel

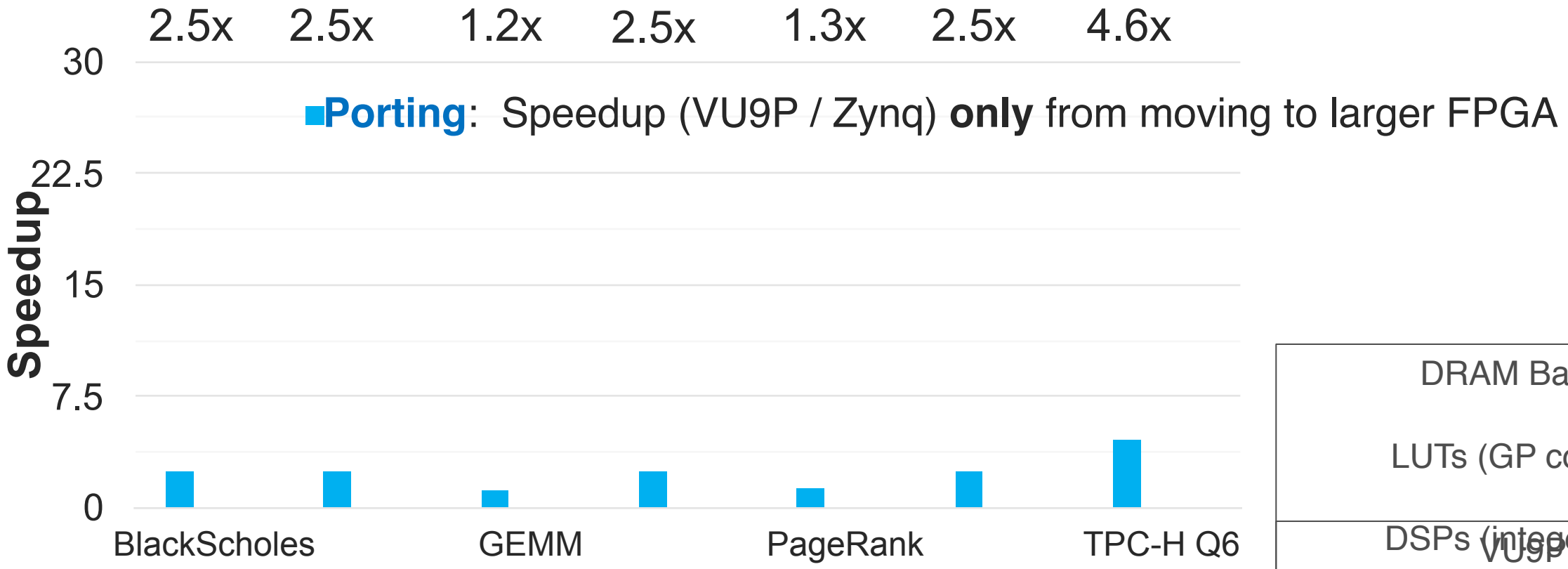


Evaluation: Portability

- FPGA 1
  - Amazon EC2 F1 Instance: Xilinx VU9P FPGA
  - 19.2 GB/s DRAM bandwidth (single channel)
- FPGA 2
  - Xilinx Zynq ZC706
  - 4.3 GB/s
- Applications
  - Spatial: Hand written, automatically tuned implementations
  - Fixed clock rate of 150 MHz

Portability: VU9P vs. Zynq ZC706

Identical Spatial source, multiple targets

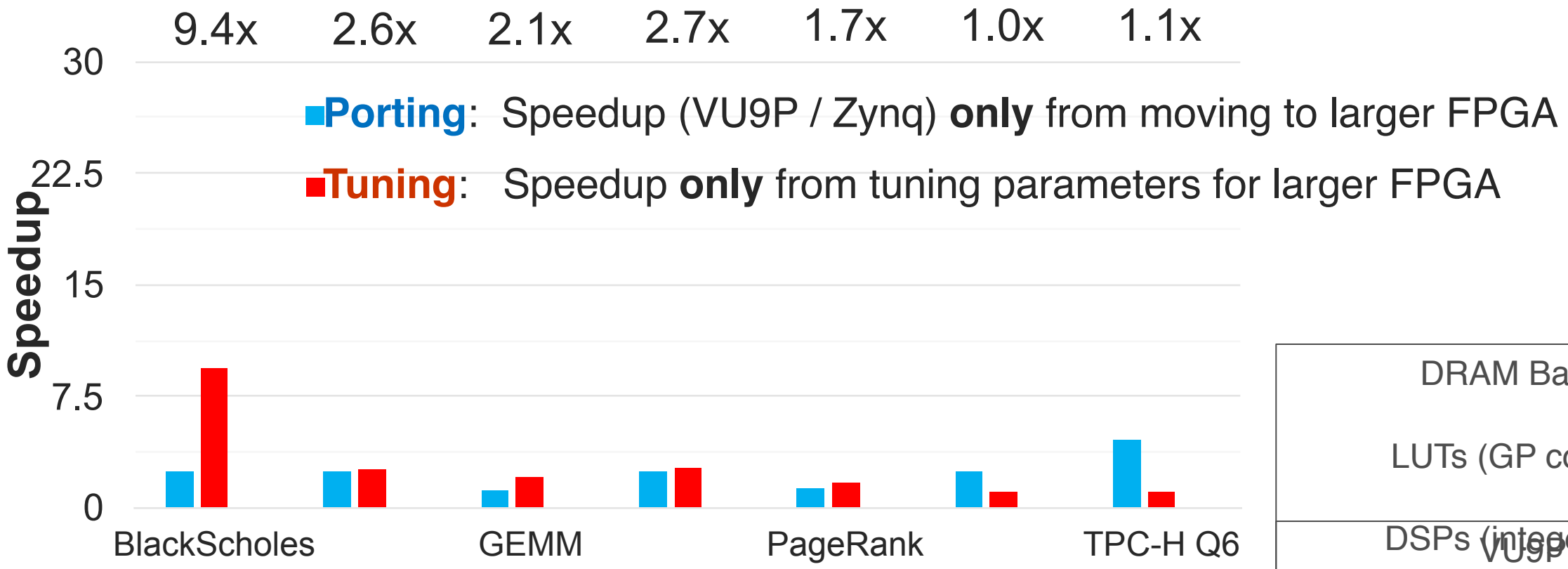


VU9P / ZC706	
DRAM Bandwidth:	**4.5x**
LUTs (GP compute):	**47.3x**
DSPs (integer FMA):	**7.6x**
On-chip memory*:	**4.0x**

* No URAM used on VU9P

Portability: VU9P vs. Zynq ZC706

Identical Spatial source, multiple targets

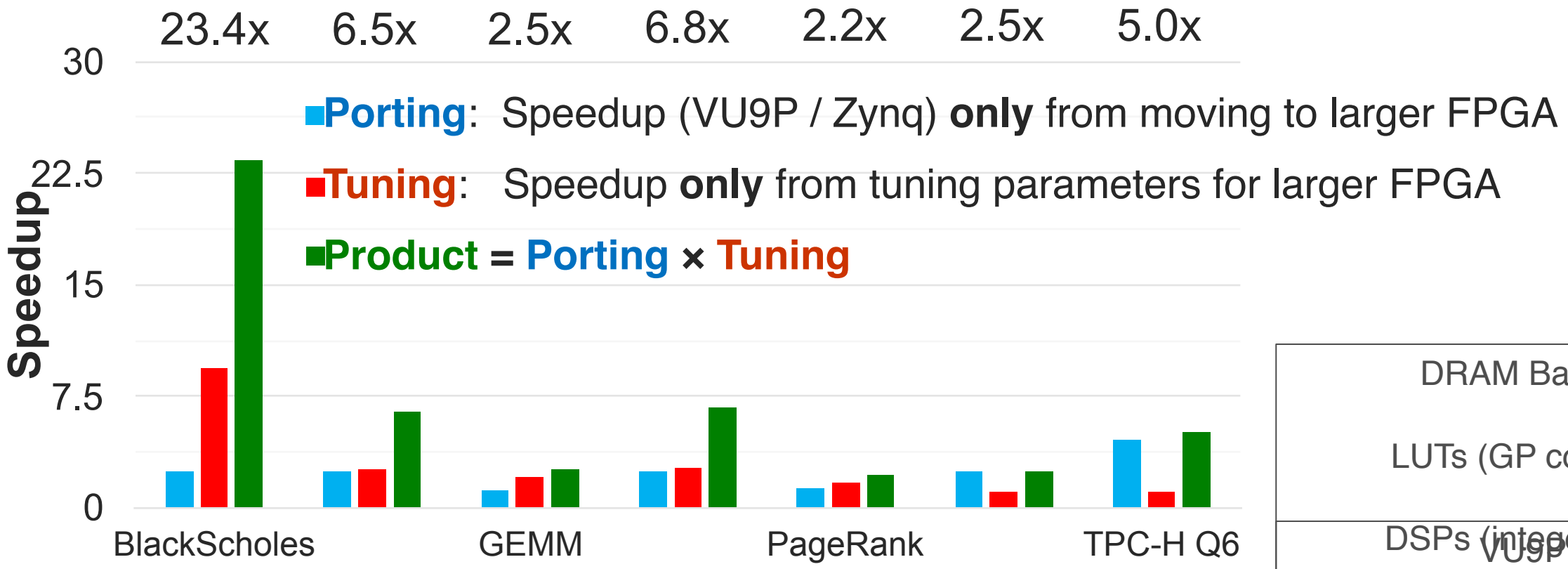


VU9P / ZC706	
DRAM Bandwidth:	**4.5x**
LUTs (GP compute):	**47.3x**
DSPs (integer FMA):	**7.6x**
On-chip memory*:	**4.0x**

* No URAM used on VU9P

Portability: VU9P vs. Zynq ZC706

Identical Spatial source, multiple targets



VU9P / ZC706	
DRAM Bandwidth:	**4.5x**
LUTs (GP compute):	**47.3x**
DSPs (integer FMA):	**7.6x**
On-chip memory*:	**4.0x**

* No URAM used on VU9P

Portability: Plasticine CGRA

Identical Spatial source, multiple targets
Even reconfigurable hardware that isn't an FPGA!

Benchmark	DRAM Bandwidth (%) Load	Store	Resource Utilization (%) PCU	PMU	AG	Speedup vs. VU9P
BlackScholes	77.4	12.9	**73.4**	10.9	20.6	1.6
GDA	24.0	0.2	**95.3**	73.4	38.2	9.8
GEMM	20.5	2.1	**96.8**	64.1	11.7	55.0
K-Means	8.0	0.4	**89.1**	57.8	17.6	6.3
TPC-H Q6	**97.2**	0.0	29.7	37.5	**70.6**	1.6

Prabhakar et al. *Plasticine: A Reconfigurable Architecture For Parallel Patterns* (ISCA '17)

Halide to Spatial

What is Halide?

- DSL for computational photography
- Separation between algorithm (what to compute) and schedule (how to compute)
- Straightforward to express and iterate over various schedules

Algorithm

```
Var x, y;
Func f;
f(x, y) = x + y;
```

Schedule #1

```
f.tile(x,y,xi,yi,8,8);
```

Implementations

What is Halide?

- DSL for computational photography
- Separation between algorithm (what to compute) and schedule (how to compute)
- Straightforward to express and iterate over various schedules

Algorithm

```
Var x, y;
Func f;
f(x, y) = x + y;
```

Schedule #2

```
f.parallel(y);
f.vectorize(x, 8);
```

Implementations

Why use Halide as Front-End to Spatial?

- Separation of concerns
  - High-level transformations: Tiling, Vectorization etc can happen in Halide
  - Lift the hard work of transforming loop nests to Halide
  - Optimized code can be lowered into spatial
- Loop-based IR
  - Easy mapping to Spatial front-end

Halide IR

```
// Algorithm
Var x, y;
Func f;
f(x, y) = x + y;

// Schedule
f.parallel(y);
f.vectorize(x, 8);

f.realize(32, 32);
```

```
produce f {
  let t6 = (f.extent.0 + f.min.0)
  let t7 = (f.min.1*f.stride.1)
  let t8 = max((f.extent.0/8), 0)
  let t3 = (t8 < ((f.extent.0 + 7)/8))
  let t2 = (0 - t7)
  let t5 = (((t6 - t7) - f.min.0) + -8)
  let t4 = (t6 + -8)
  parallel (f.s0.y, f.min.1, f.extent.1) {
    let t10 = ((f.s0.y*f.stride.1) + t2)
    let t9 = (f.min.0 + f.s0.y)
    for (f.s0.x.x, 0, t8) {
      f[ramp(((f.s0.x.x*8) + t10), 1, 8)] = ramp(((f.s0.x.x*8) + t9), 1, 8)
    }
    if (t3) {
      f[ramp(((f.s0.y*f.stride.1) + t5), 1, 8)] = ramp((f.s0.y + t4), 1, 8)
    }
  }
}
```

Example: Halide to Spatial

```
// Algorithm
f(x, y) = x + y;
g(x, y) = (f(x, y) + f(x, y+1))/2;

// Schedule
g.in().spatial();
g.store_in(MemoryType::SRAM)
 .compute_at(g.in(), Var::outermost());
g.tile(x, y, xo, yo, xi, yi, 4, 4);

f.compute_root();
f.in()
 .copy_to_device()
 .store_in(MemoryType::SRAM)
 .compute_at(g, xo);

g.in().copy_to_host();

wrapper.compile_to_spatial(...);
```

Example: Halide to Spatial

```
// Algorithm
f(x, y) = x + y;
g(x, y) = (f(x, y) + f(x, y+1))/2;

// Schedule
g.in().spatial();
g.store_in(MemoryType::SRAM)
 .compute_at(g.in(), Var::outermost());
g.tile(x, y, xo, yo, xi, yi, 4, 4);

f.compute_root();
f.in()
 .copy_to_device()
 .store_in(MemoryType::SRAM)
 .compute_at(g, xo);

g.in().copy_to_host();

wrapper.compile_to_spatial(...);
```

```
val g_wrapper = DRAM[Int](16, 16);
Accel {
  val g = SRAM[Int](16, 16);
  Foreach(0 until 4 by 1) {yo =>
    Foreach(0 until 4 by 1) {xo =>
      val f_wrapper = SRAM[Int](4, 5);
      f_wrapper load f(xo*4::xo*4+4, yo*4::yo*4+5);
      Foreach(0 until 4 by 1) {yi =>
        Foreach(0 until 4 by 1) {xi =>
          g(xo*4+xi, yo*4+yi) =
            (f_wrapper(xi,yi)+f_wrapper(xi,yi+1))/2;
        }
      }
    }
  }
  g_wrapper store g;
}
```

Example: Halide to Spatial

```
// Algorithm
f(x, y) = x + y;
g(x, y) = (f(x, y) + f(x, y+1)

// Schedule
g.in().spatial();
g.store_in(MemoryType::SRAM)
 .compute_at(g.in(), Var::outermost());
g.tile(x, y, xo, yo, xi, yi, 4, 4);

f.compute_root();
f.in()
 .copy_to_device()
 .store_in(MemoryType::SRAM)
 .compute_at(g, xo);

g.in().copy_to_host();

wrapper.compile_to_spatial(...);
```

Compute at Accelerator

```
val g_wrapper = DRAM[Int](16, 16);
Accel {
  val g = SRAM[Int](16, 16);
  Foreach(0 until 4 by 1) {yo =>
    Foreach(0 until 4 by 1) {xo =>
      val f_wrapper = SRAM[Int](4, 5);
      f_wrapper load f(xo*4::xo*4+4, yo*4::yo*4+5);
      Foreach(0 until 4 by 1) {yi =>
        Foreach(0 until 4 by 1) {xi =>
          g(xo*4+xi, yo*4+yi) =
            (f_wrapper(xi,yi)+f_wrapper(xi,yi+1))/2;
        }
      }
    }
  }
  g_wrapper store g;
}
```

Example: Halide to Spatial

```
// Algorithm
f(x, y) = x + y;
g(x, y) = (f(x, y) + f(x, y+1))/2;

// Schedule
g.in().spatial();
g.store_in(MemoryType::SRAM)
 .compute_at(g.in(), Var::outermost());
g.tile(x, y, xo, yo, xi, yi, 4, 4);

f.compute_root();
f.in()
 .copy_to_device()
 .store_in(MemoryType::SRAM)
 .compute_at(g, xo);

g.in().copy_to_host();

wrapper.compile_to_spatial(...);
```

Allocate SRAM to store 'g'

```
val g_wrapper = DRAM[Int](16, 16);
Accel {
  val g = SRAM[Int](16, 16);
  Foreach(0 until 4 by 1) {yo =>
    Foreach(0 until 4 by 1) {xo =>
      val f_wrapper = SRAM[Int](4, 5);
      f_wrapper load f(xo*4::xo*4+4, yo*4::yo*4+5);
      Foreach(0 until 4 by 1) {yi =>
        Foreach(0 until 4 by 1) {xi =>
          g(xo*4+xi, yo*4+yi) =
            (f_wrapper(xi,yi)+f_wrapper(xi,yi+1))/2;
        }
      }
    }
  }
  g_wrapper store g;
}
```

Example: Halide to Spatial

```
// Algorithm
f(x, y) = x + y;
g(x, y) = (f(x, y) + f(x, y+1))/2;

// Schedule
g.in().spatial();
g.store_in(MemoryType::SRAM)
 .compute_at(g.in(), Var::outermost());
g.tile(x, y, xo, yo, xi, yi, 4, 4);

f.compute_root();
f.in()
 .copy_to_device()
 .store_in(MemoryType::SRAM)
 .compute_at(g, xo);

g.in().copy_to_host();

wrapper.compile_to_spatial(...);
```

Tile g

```
val g_wrapper = DRAM[Int](16, 16);
Accel {
  val g = SRAM[Int](16, 16);
  Foreach(0 until 4 by 1) {yo =>
    Foreach(0 until 4 by 1) {xo =>
      val f_wrapper = SRAM[Int](4, 5);
      f_wrapper load f(xo*4::xo*4+4, yo*4::yo*4+5);
      Foreach(0 until 4 by 1) {yi =>
        Foreach(0 until 4 by 1) {xi =>
          g(xo*4+xi, yo*4+yi) =
            (f_wrapper(xi,yi)+f_wrapper(xi,yi+1))/2;
        }
      }
    }
  }
  g_wrapper store g;
}
```

Example: Halide to Spatial

```
// Algorithm
f(x, y) = x + y;
g(x, y) = (f(x, y) + f(x, y+1))/2;

// Schedule
g.in().spatial();
g.store_in(MemoryType::SRAM)
 .compute_at(g.in(), Var::outermost());
g.tile(x, y, xo, yo, xi, yi, 4, 4);

f.compute_root();
f.in()
 .copy_to_device()
 .store_in(MemoryType::SRAM)
 .compute_at(g, xo);

g.in().copy_to_host();

wrapper.compile_to_spatial(...);
```

Load 'f' into the accelerator's memory

```
val g_wrapper = DRAM[Int](16, 16);
Accel {
  val g = SRAM[Int](16, 16);
  Foreach(0 until 4 by 1) {yo =>
    Foreach(0 until 4 by 1) {xo =>
      val f_wrapper = SRAM[Int](4, 5);
      f_wrapper load f(xo*4::xo*4+4, yo*4::yo*4+5);
      Foreach(0 until 4 by 1) {yi =>
        Foreach(0 until 4 by 1) {xi =>
          g(xo*4+xi, yo*4+yi) =
            (f_wrapper(xi,yi)+f_wrapper(xi,yi+1))/2;
        }
      }
    }
  }
  g_wrapper store g;
}
```

Example: Halide to Spatial

```
// Algorithm
f(x, y) = x + y;
g(x, y) = (f(x, y) + f(x, y+1))/2;

// Schedule
g.in().spatial();
g.store_in(MemoryType::SRAM)
 .compute_at(g.in(), Var::out
g.tile(x, y, xo, yo, xi, yi, 4, 4);

f.compute_root();
f.in()
 .copy_to_device()
 .store_in(MemoryType::SRAM)
 .compute_at(g, xo);

g.in().copy_to_host();

wrapper.compile_to_spatial(...);
```

Do the load at loop level ‘xo’ and store in SRAM

```
val g_wrapper = DRAM[Int](16, 16);
Accel {
  val g = SRAM[Int](16, 16);
  Foreach(0 until 4 by 1) {yo =>
    Foreach(0 until 4 by 1) {xo =>
      val f_wrapper = SRAM[Int](4, 5);
      f_wrapper load f(xo*4::xo*4+4, yo*4::yo*4+5);
      Foreach(0 until 4 by 1) {yi =>
        Foreach(0 until 4 by 1) {xi =>
          g(xo*4+xi, yo*4+yi) =
            (f_wrapper(xi,yi)+f_wrapper(xi,yi+1))/2;
        }
      }
    }
  }
  g_wrapper store g;
}
```

Example: Halide to Spatial

```
// Algorithm
f(x, y) = x + y;
g(x, y) = (f(x, y) + f(x, y+1))/2;

// Schedule
g.in().spatial();
g.store_in(MemoryType::SRAM)
 .compute_at(g.in(), Var::outermost());
g.tile(x, y, xo, yo, xi, yi, 4, 4);

f.compute_root();
f.in()
 .copy_to_device()
 .store_in(MemoryType::SRAM)
 .compute_at(g, xo);

g.in().copy_to_host();

wrapper.compile_to_spatial(...);
```

Store 'g' back into host's DRAM

```
val g_wrapper = DRAM[Int](16, 16);
Accel {
  val g = SRAM[Int](16, 16);
  Foreach(0 until 4 by 1) {yo =>
    Foreach(0 until 4 by 1) {xo =>
      val f_wrapper = SRAM[Int](4, 5);
      f_wrapper load f(xo*4::xo*4+4, yo*4::yo*4+5);
      Foreach(0 until 4 by 1) {yi =>
        Foreach(0 until 4 by 1) {xi =>
          g(xo*4+xi, yo*4+yi) =
            (f_wrapper(xi,yi)+f_wrapper(xi,yi+1))/2;
        }
      }
    }
  }
  g_wrapper store g;
}
```

Conclusion

Conclusion

- **Reconfigurable architectures** are becoming key for performance / energy efficiency

Conclusion

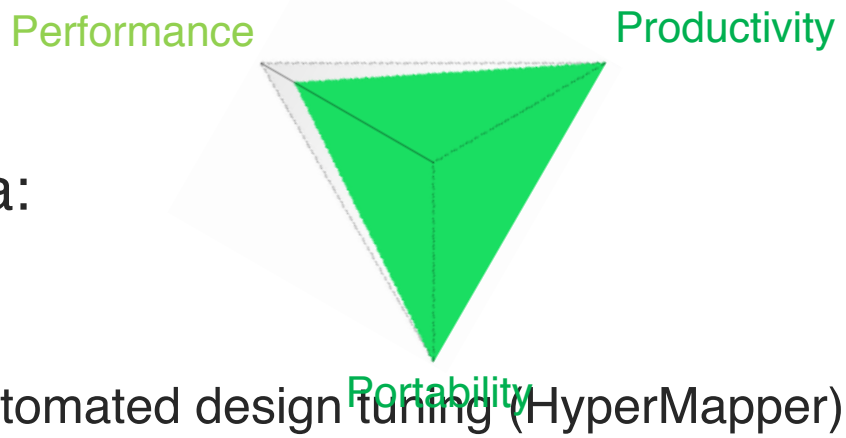
- **Reconfigurable architectures** are becoming key for performance / energy efficiency
- Current programming solutions for reconfigurables are still inadequate

Conclusion

- **Reconfigurable architectures** are becoming key for performance / energy efficiency
- Current programming solutions for reconfigurables are still inadequate
- Need to rethink outside of the C box for high level synthesis:
  - **Memory hierarchy for optimization**
  - **Design parameters for tuning**
  - **Arbitrarily nestable pipelines**

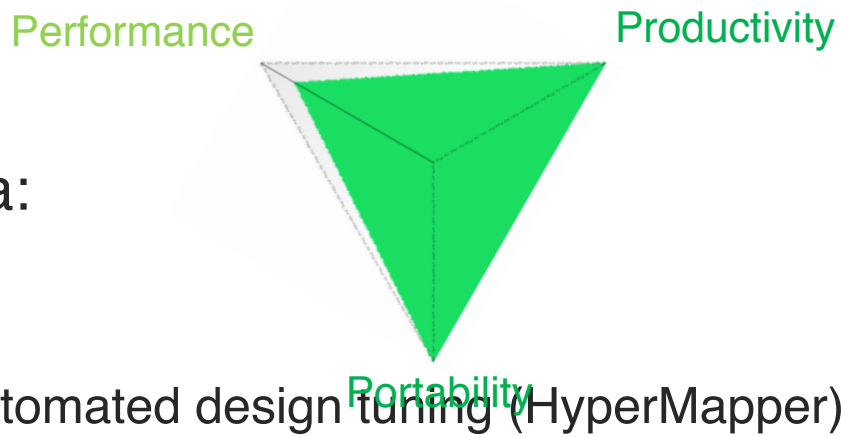
Conclusion

- **Reconfigurable architectures** are becoming key for performance / energy efficiency
- Current programming solutions for reconfigurables are still inadequate
- Need to rethink outside of the C box for high level synthesis:
  - **Memory hierarchy for optimization**
  - **Design parameters for tuning**
  - **Arbitrarily nestable pipelines**
- **Spatial** prototypes these language and compiler criteria:
  - Average **speedup of 2.9x versus SDAccel** on VU9P
  - Average **42% less code than SDAccel**
  - Achieves transparent portability through internal support for automated design tuning (HyperMapper)



Conclusion

- **Reconfigurable architectures** are becoming key for performance / energy efficiency
- Current programming solutions for reconfigurables are still inadequate
- Need to rethink outside of the C box for high level synthesis:
  - **Memory hierarchy for optimization**
  - **Design parameters for tuning**
  - **Arbitrarily nestable pipelines**
- **Spatial** prototypes these language and compiler criteria:
  - Average **speedup of 2.9x versus SDAccel** on VU9P
  - Average **42% less code than SDAccel**
  - Achieves transparent portability through internal support for automated design tuning (HyperMapper)



Spatial is open source: https://spatial-lang.org/

Backup Slides

The Team

David
Koeplinger

Matt
Feldman

Raghu
Prabhakar

Yaqi
Zhang

Stefan
Hadjis

Ruben
Fiszel

Tian
Zhao

Ardavan
Pedram

Luigi
Nardi

Christos
Kozyrakis

Kunle
Olukotun

Custom ASICs

Custom ASICs

Good for widely used, **fixed** specifications (like compression)

Custom ASICs

Good for widely used, **fixed** specifications (like compression)

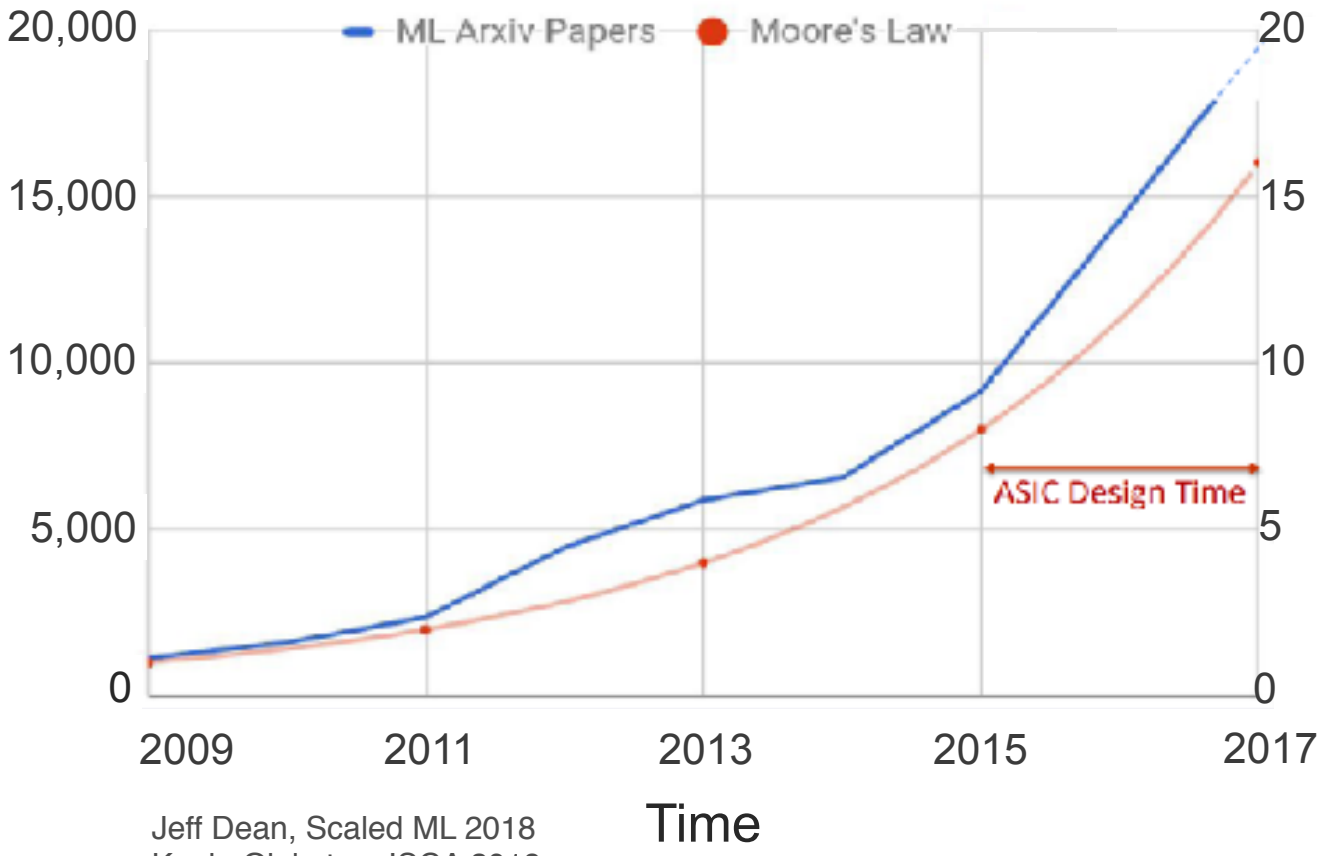
Expensive with **long design turnaround** for developing fields like ML

Custom ASICs

Good for widely used, **fixed** specifications (like compression)

Expensive with **long design turnaround** for developing fields like ML

ML *Arxiv* Papers



Relative # of Papers / Year Since 2009

Jeff Dean, Scaled ML 2018
Kunle Olukotun, ISCA 2018

C + Pragmas Example

Add 512 integers originating from accelerator DRAM

```
void sum(int* mem) {

    mem[512] = 0;

    for(int i=0; i < 512; i++) {
        mem[512] += mem[i];
    }

}
```

C + Pragmas Example

Add 512 integers originating from accelerator DRAM

```
void sum(int* mem) {

    mem[512] = 0;

    for(int i=0; i < 512; i++) {
        mem[512] += mem[i];
    }

}
```

Commercial HLS Tool

Runtime: 27,236 clock cycles

(100x too long!)

C + Pragmas Example

Add 512 integers originating from external DRAM

```
#define CHUNKSIZE (sizeof(MPort)/sizeof(int))
#define LOOPCOUNT (512/CHUNKSIZE)

void sum(MPort* mem) {
    MPort buff[LOOPCOUNT];
    memcpy(buff, mem, LOOPCOUNT);

    int sum = 0;
    for(int i=1; i<LOOPCOUNT; i++) {
        #pragma PIPELINE
        for(int j=0; j<CHUNKSIZE; j++) {
            #pragma UNROLL
            sum += (int)
(buff[i]>>j*sizeof(int)*8);
        }
    }
    mem[512] = sum;
}
```

Runtime: 302 clock cycles

C + Pragmas Example

Add 512 integers originating from external DRAM

```
#define CHUNKSIZE (sizeof(MPort)/sizeof(int))
#define LOOPCOUNT (512/CHUNKSIZE)

void sum(MPort* mem) {
    MPort buff[LOOPCOUNT];
    memcpy(buff, mem, LOOPCOUNT);

    int sum = 0;
    for(int i=1; i<LOOPCOUNT; i++) {
        #pragma PIPELINE
        for(int j=0; j<CHUNKSIZE; j++) {
            #pragma UNROLL
            sum += (int)
(buff[i]>>j*sizeof(int)*8);
        }
    }
    mem[512] = sum;
}
```

Width of DRAM controller interface

Burst Access

Use local variable

Loop Restructuring

Special compiler directives

Bit shifting to extract individual elements

Runtime: 302 clock cycles

Hardware Design Considerations

Hardware Design Considerations

1. Finite physical compute and memory resources

Hardware Design Considerations

1. Finite physical compute and memory resources
2. Requires aggressive pipelining for performance
   - Maximize useful execution time of compute resources

Hardware Design Considerations

1. Finite physical compute and memory resources
2. Requires aggressive pipelining for performance
   - Maximize useful execution time of compute resources
3. Disjoint memory space
   - No hardware managed memory hierarchy

Hardware Design Considerations

1. Finite physical compute and memory resources
2. Requires aggressive pipelining for performance
   - Maximize useful execution time of compute resources
3. Disjoint memory space
   - No hardware managed memory hierarchy
4. Huge design parameter spaces
   - Parameters are interdependent, change runtime by orders of magnitude

Hardware Design Considerations

1. Finite physical compute and memory resources
2. Requires aggressive pipelining for performance
   - Maximize useful execution time of compute resources
3. Disjoint memory space
   - No hardware managed memory hierarchy
4. Huge design parameter spaces
   - Parameters are interdependent, change runtime by orders of magnitude
5. Others… pipeline timing, clocking, etc.

Local Memory Analysis Example

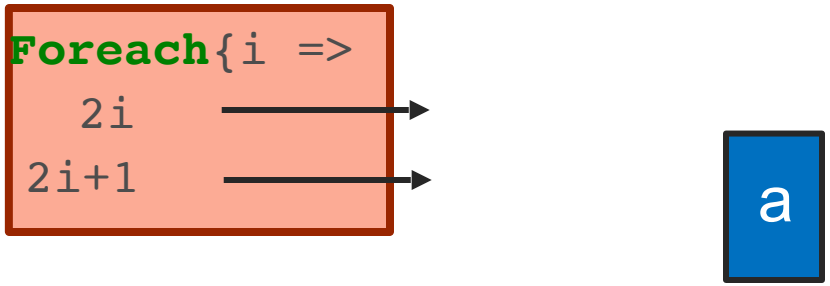
```
Foreach(N by 1){ r =>
  val a = SRAM[Float](D)
  val b = SRAM[Float](D)
  val c = SRAM[Float](D)
  Foreach(D par 2){i =>
    a(i) = …
  }
  Reduce(sum)(D par 2){j =>
    a(b(j))
  }{(a,b) => a + b}
  Foreach(D par 2){k =>
    c(k) = a(k) * sum
  }
}
```

Local Memory Analysis Example

```
Foreach(N by 1){ r =>
  val a = SRAM[Float](D)
  val b = SRAM[Float](D)
  val c = SRAM[Float](D)
  Foreach(D par 2){i =>
    a(i) = …
  }
  Reduce(sum)(D par 2){j =>
    a(b(j))
  }{(a,b) => a + b}
  Foreach(D par 2){k =>
    c(k) = a(k) * sum
  }
}
```

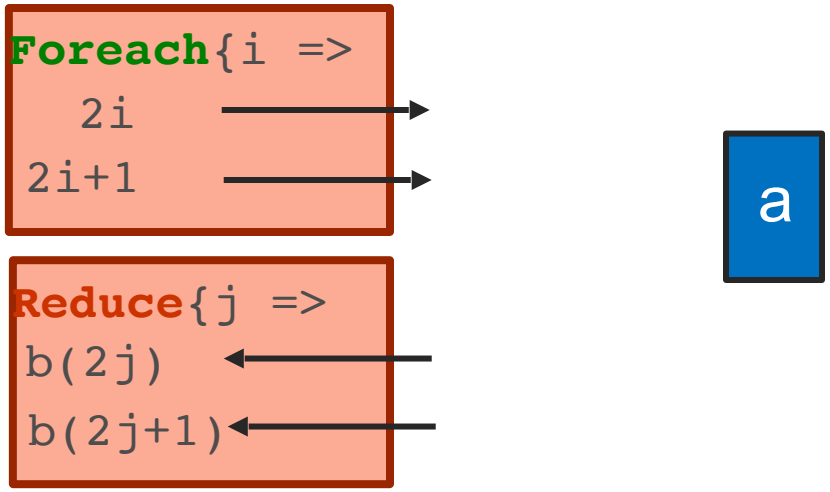
Local Memory Analysis Example

```
Foreach(N by 1){ r =>
  val a = SRAM[Float](D)
  val b = SRAM[Float](D)
  val c = SRAM[Float](D)
  Foreach(D par 2){i =>
    a(i) = …
  }
  Reduce(sum)(D par 2){j =>
    a(b(j))
  }{(a,b) => a + b}
  Foreach(D par 2){k =>
    c(k) = a(k) * sum
  }
}
```



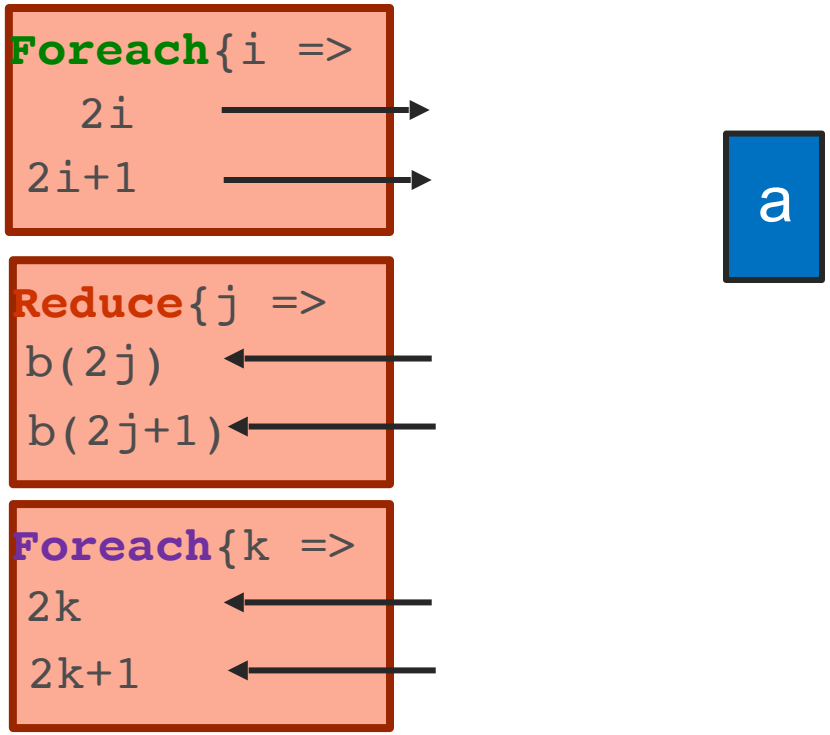
Local Memory Analysis Example

```
Foreach(N by 1){ r =>
  val a = SRAM[Float](D)
  val b = SRAM[Float](D)
  val c = SRAM[Float](D)
  Foreach(D par 2){i =>
    a(i) = …
  }
  Reduce(sum)(D par 2){j =>
    a(b(j))
  }{(a,b) => a + b}
  Foreach(D par 2){k =>
    c(k) = a(k) * sum
  }
}
```



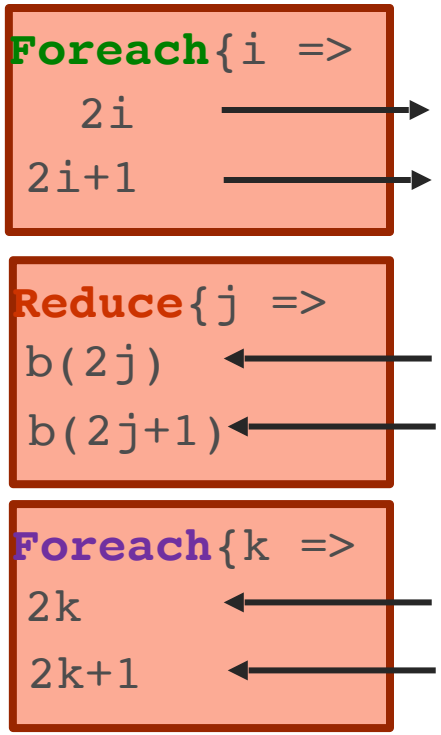
Local Memory Analysis Example

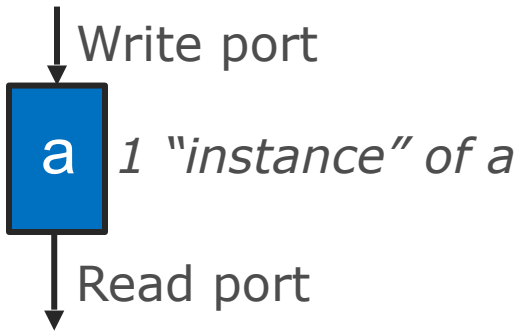
```
Foreach(N by 1){ r =>
  val a = SRAM[Float](D)
  val b = SRAM[Float](D)
  val c = SRAM[Float](D)
  Foreach(D par 2){i =>
    a(i) = …
  }
  Reduce(sum)(D par 2){j =>
    a(b(j))
  }{(a,b) => a + b}
  Foreach(D par 2){k =>
    c(k) = a(k) * sum
  }
}
```



Local Memory Analysis Example

```
Foreach(N by 1){ r =>
  val a = SRAM[Float](D)
  val b = SRAM[Float](D)
  val c = SRAM[Float](D)
  Foreach(D par 2){i =>
    a(i) = …
  }
  Reduce(sum)(D par 2){j =>
    a(b(j))
  }{(a,b) => a + b}
  Foreach(D par 2){k =>
    c(k) = a(k) * sum
  }
}
```





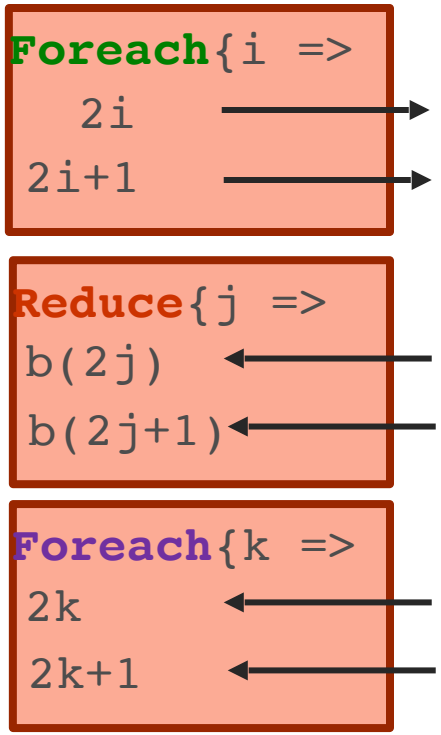
Local Memory Analysis Example

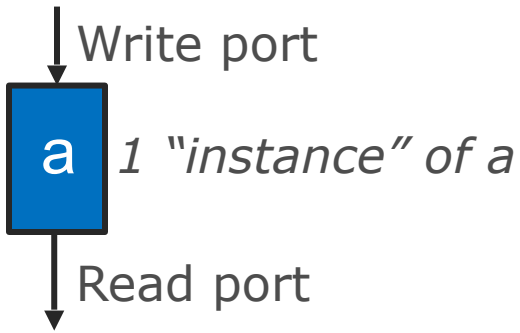
```
Foreach(N by 1){ r =>
  val a = SRAM[Float](D)
  val b = SRAM[Float](D)
  val c = SRAM[Float](D)
  Foreach(D par 2){i =>
    a(i) = …
  }
  Reduce(sum)(D par 2){j =>
    a(b(j))
  }{(a,b) => a + b}
  Foreach(D par 2){k =>
    c(k) = a(k) * sum
  }
}
```

Step 1: For each read:

Find the **banking** and **buffering** for that read

and all writes that may be visible to that





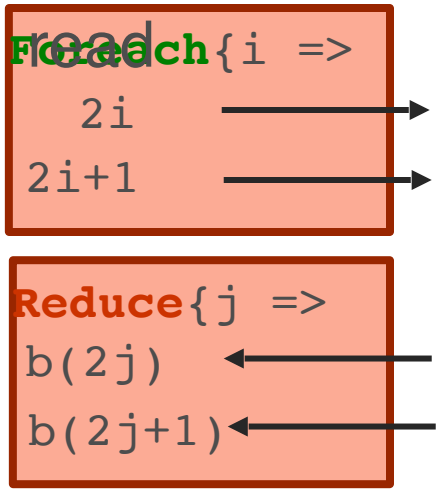
Local Memory Analysis Example (Cont.)

```
Foreach(N by 1){ r =>
  val a = SRAM[Float](D)
  val b = SRAM[Float](D)
  val c = SRAM[Float](D)
  Foreach(D par 2){i =>
    a(i) = …
  }
  Reduce(sum)(D par 2){j =>
    a(b(j))
  }{(a,b) => a + b}
  Foreach(D par 2){k =>
    c(k) = a(k) * sum
  }
}
```

Step 1: For each read:

Find the **banking** and **buffering** for that read

and all writes that may be visible to that read



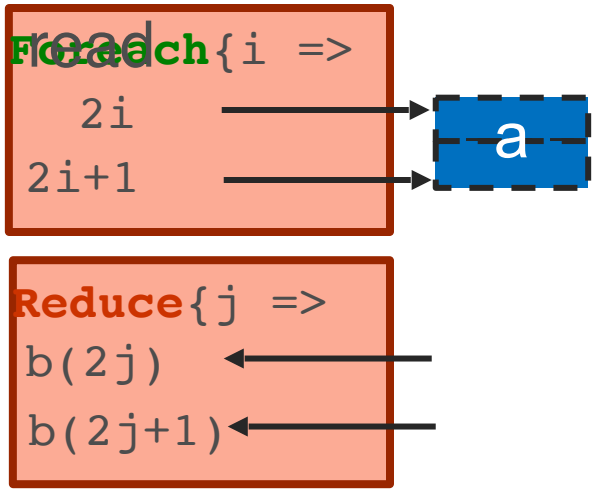
Local Memory Analysis Example (Cont.)

```
Foreach(N by 1){ r =>
  val a = SRAM[Float](D)
  val b = SRAM[Float](D)
  val c = SRAM[Float](D)
  Foreach(D par 2){i =>
    a(i) = …
  }
  Reduce(sum)(D par 2){j =>
    a(b(j))
  }{(a,b) => a + b}
  Foreach(D par 2){k =>
    c(k) = a(k) * sum
  }
}
```

Step 1: For each read:

Find the **banking** and **buffering** for that read

and all writes that may be visible to that read



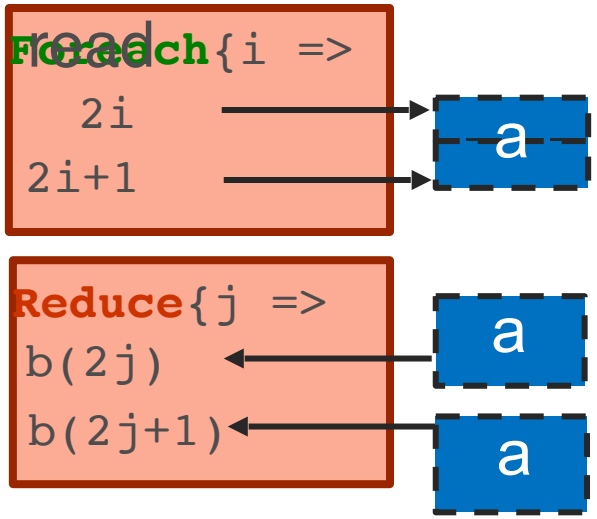
Local Memory Analysis Example (Cont.)

```
Foreach(N by 1){ r =>
  val a = SRAM[Float](D)
  val b = SRAM[Float](D)
  val c = SRAM[Float](D)
  Foreach(D par 2){i =>
    a(i) = …
  }
  Reduce(sum)(D par 2){j =>
    a(b(j))
  }{(a,b) => a + b}
  Foreach(D par 2){k =>
    c(k) = a(k) * sum
  }
}
```

Step 1: For each read:

Find the **banking** and **buffering** for that read

and all writes that may be visible to that read



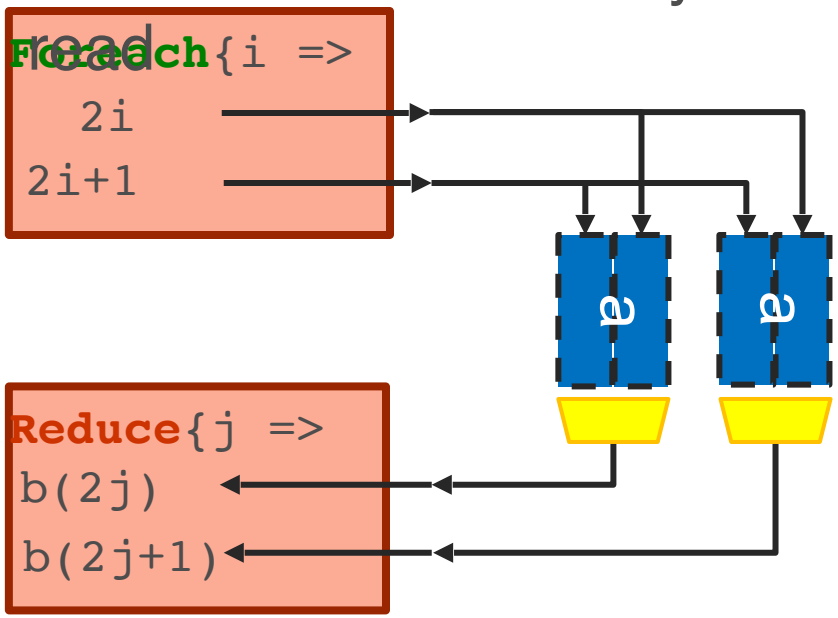
Local Memory Analysis Example (Cont.)

```
Foreach(N by 1){ r =>
  val a = SRAM[Float](D)
  val b = SRAM[Float](D)
  val c = SRAM[Float](D)
  Foreach(D par 2){i =>
    a(i) = …
  }
  Reduce(sum)(D par 2){j =>
    a(b(j))
  }{(a,b) => a + b}
  Foreach(D par 2){k =>
    c(k) = a(k) * sum
  }
}
```

Step 1: For each read:

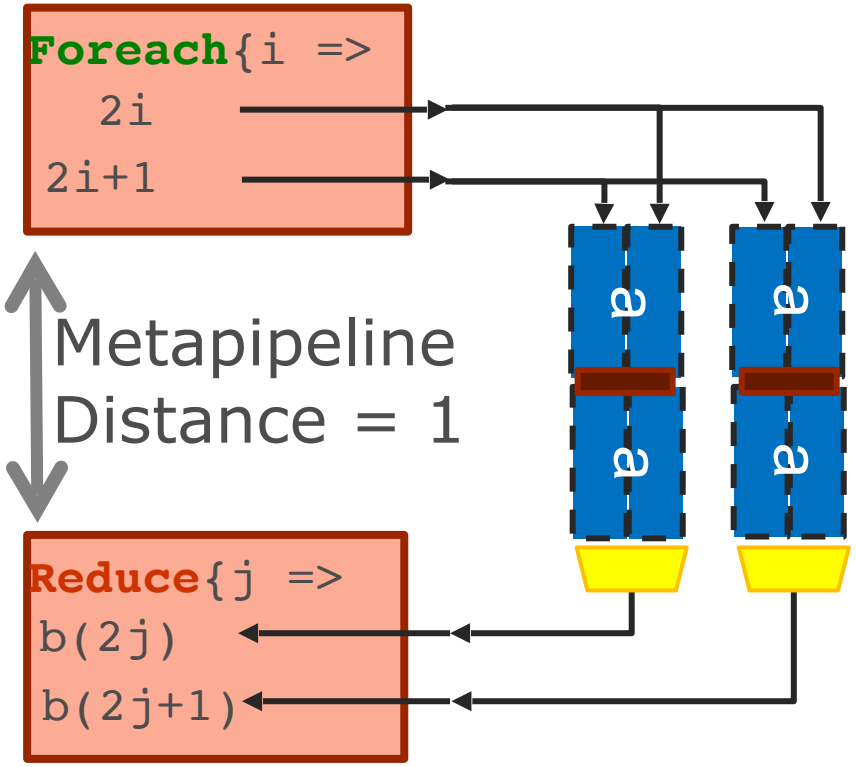
Find the **banking** and **buffering** for that read

and all writes that may be visible to that read



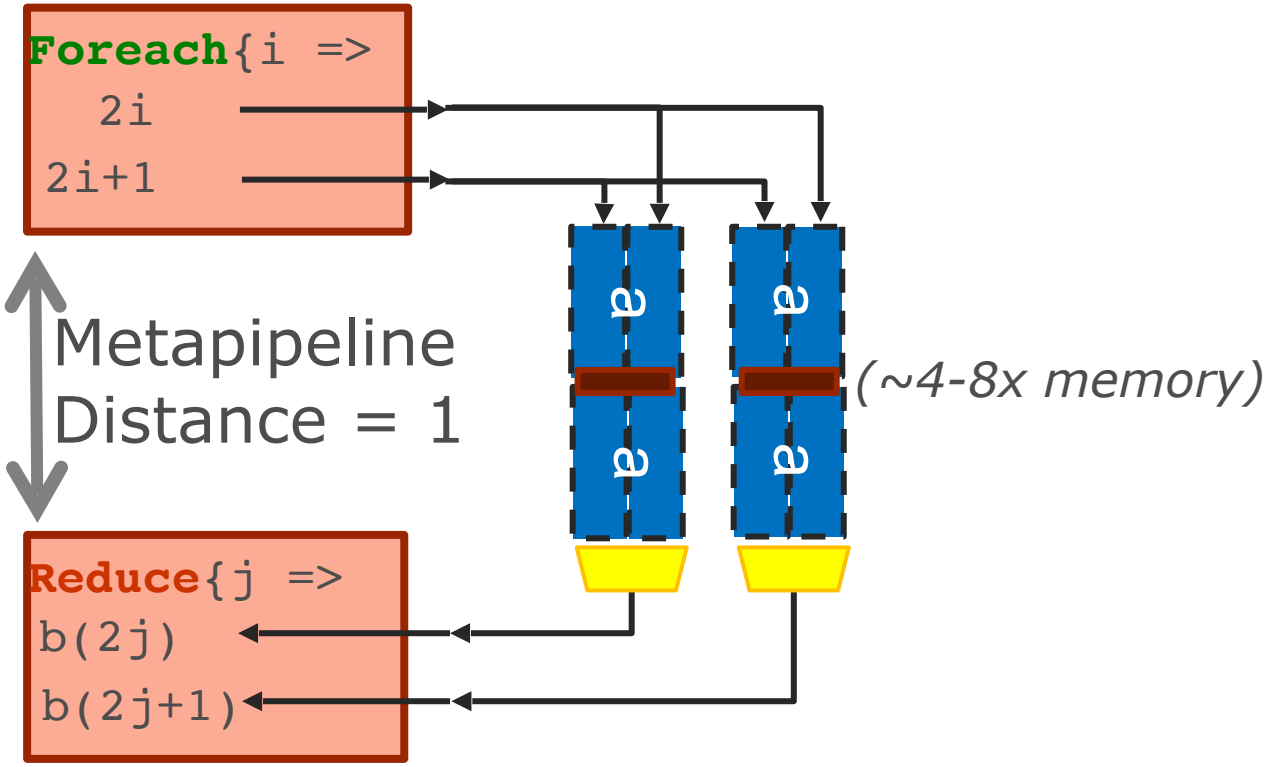
Local Memory Analysis Example (Cont.)

```
Foreach(N by 1){ r =>
  val a = SRAM[Float](D)
  val b = SRAM[Float](D)
  val c = SRAM[Float](D)
  Foreach(D par 2){i =>
    a(i) = …
  }
  Reduce(sum)(D par 2){j =>
    a(b(j))
  }{(a,b) => a + b}
  Foreach(D par 2){k =>
    c(k) = a(k) * sum
  }
}
```



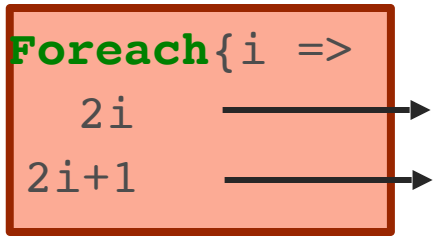
Local Memory Analysis Example (Cont.)

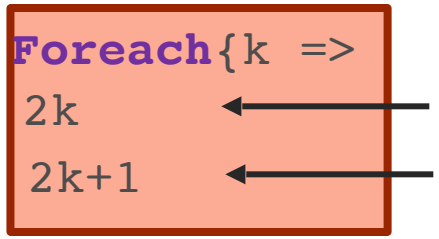
```
Foreach(N by 1){ r =>
  val a = SRAM[Float](D)
  val b = SRAM[Float](D)
  val c = SRAM[Float](D)
  Foreach(D par 2){i =>
    a(i) = …
  }
  Reduce(sum)(D par 2){j =>
    a(b(j))
  }{(a,b) => a + b}
  Foreach(D par 2){k =>
    c(k) = a(k) * sum
  }
}
```



Local Memory Analysis Example (Cont.)

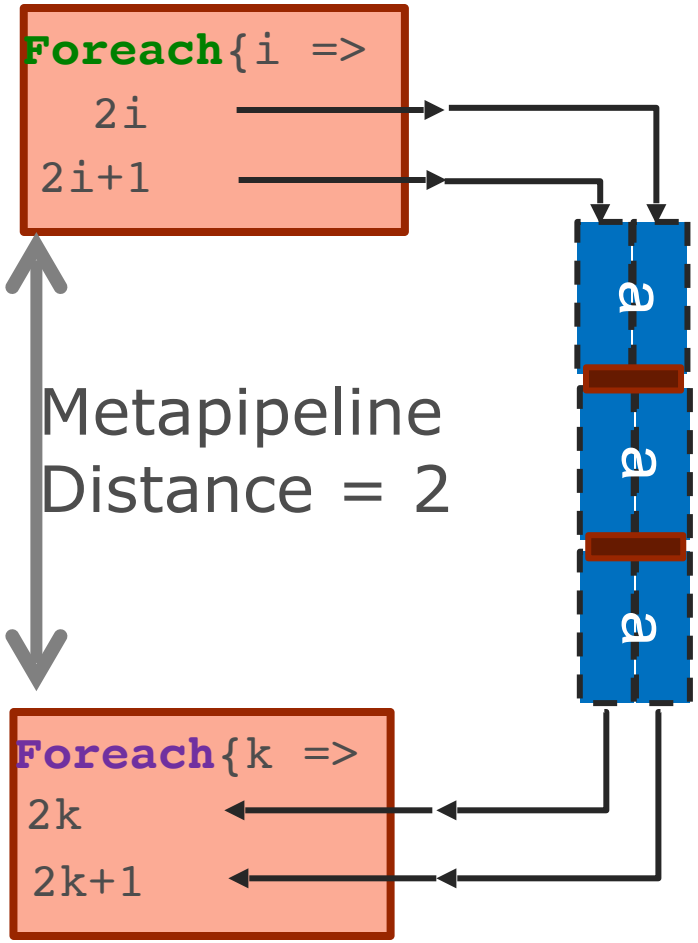
```
Foreach(N by 1){ r =>
  val a = SRAM[Float](D)
  val b = SRAM[Float](D)
  val c = SRAM[Float](D)
  Foreach(D par 2){i =>
    a(i) = …
  }
  Reduce(sum)(D par 2){j =>
    a(b(j))
  }{(a,b) => a + b}
  Foreach(D par 2){k =>
    c(k) = a(k) * sum
  }
}
```





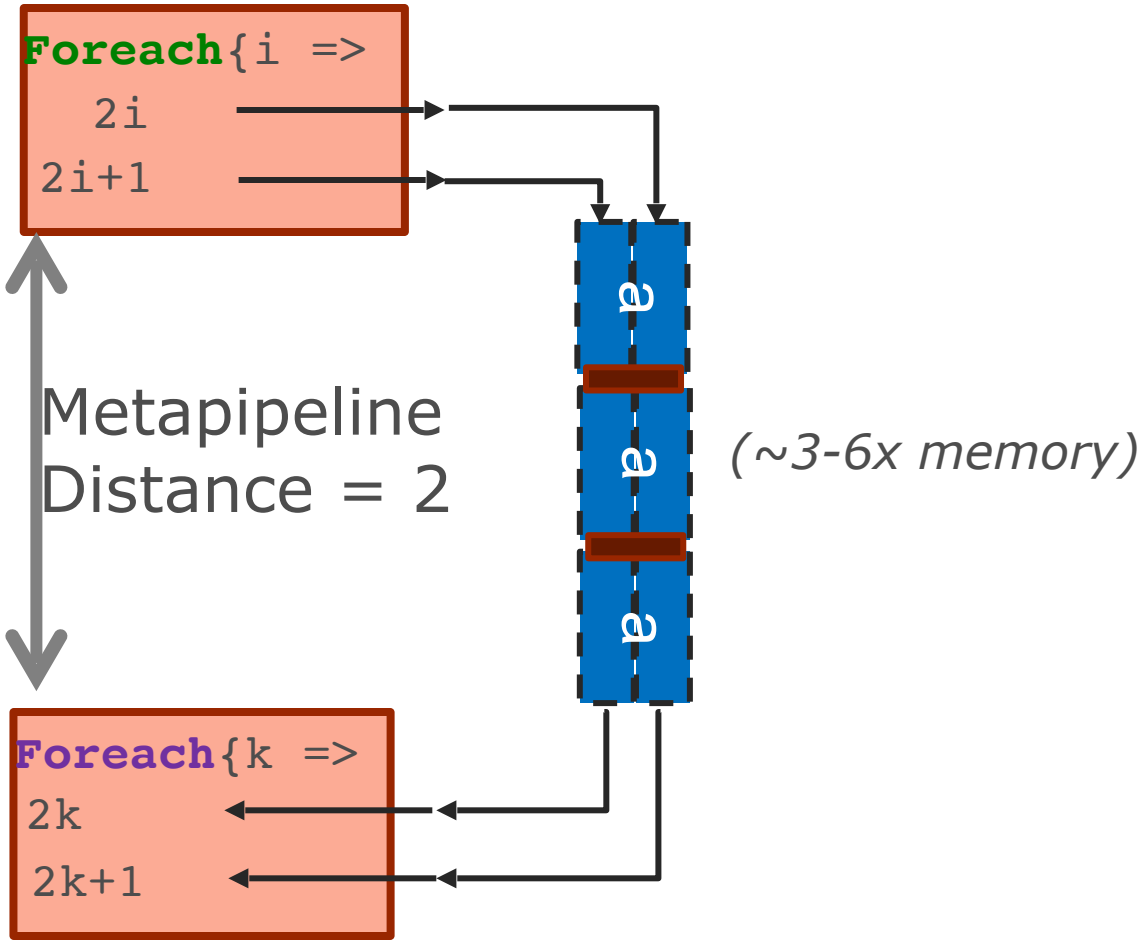
Local Memory Analysis Example (Cont.)

```
Foreach(N by 1){ r =>
  val a = SRAM[Float](D)
  val b = SRAM[Float](D)
  val c = SRAM[Float](D)
  Foreach(D par 2){i =>
    a(i) = …
  }
  Reduce(sum)(D par 2){j =>
    a(b(j))
  }{(a,b) => a + b}
  Foreach(D par 2){k =>
    c(k) = a(k) * sum
  }
}
```



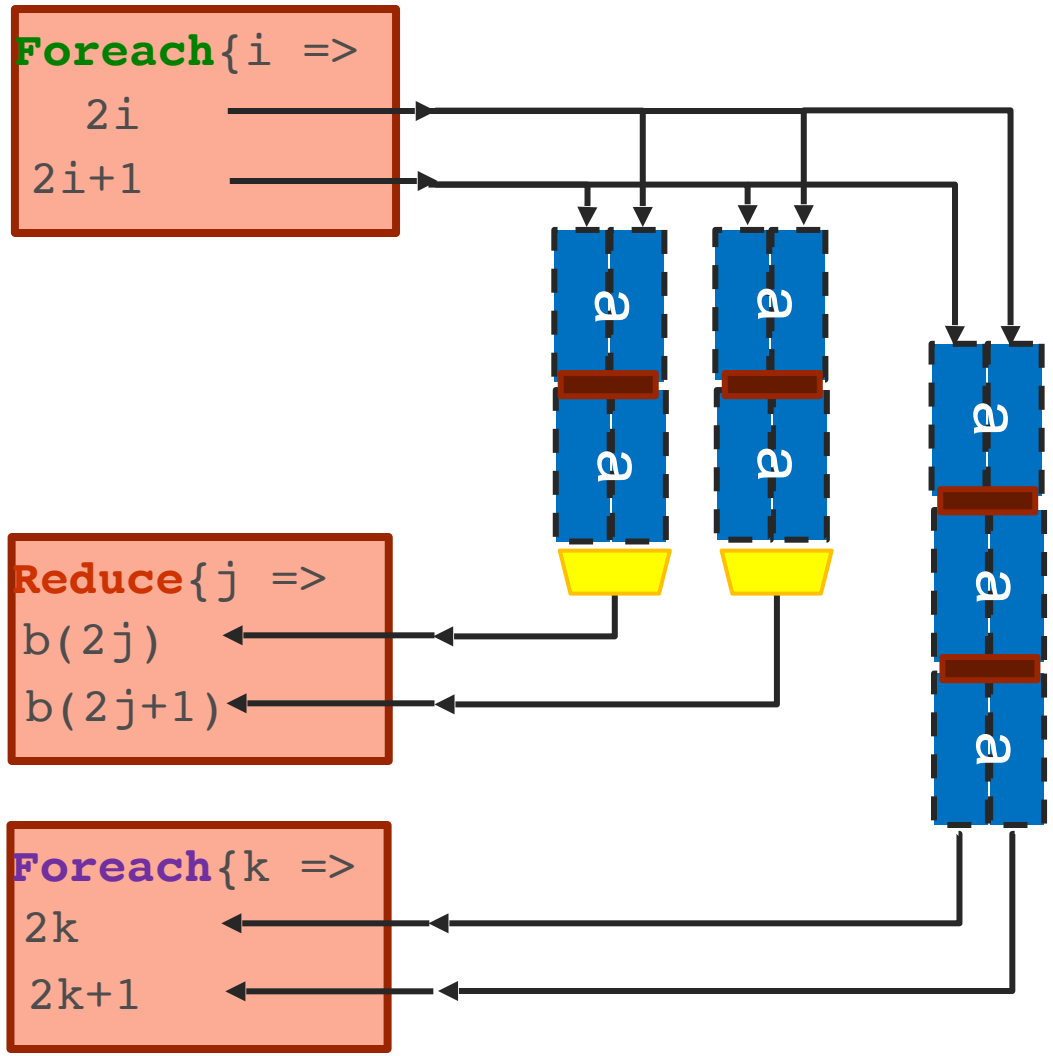
Local Memory Analysis Example (Cont.)

```
Foreach(N by 1){ r =>
  val a = SRAM[Float](D)
  val b = SRAM[Float](D)
  val c = SRAM[Float](D)
  Foreach(D par 2){i =>
    a(i) = …
  }
  Reduce(sum)(D par 2){j =>
    a(b(j))
  }{(a,b) => a + b}
  Foreach(D par 2){k =>
    c(k) = a(k) * sum
  }
}
```



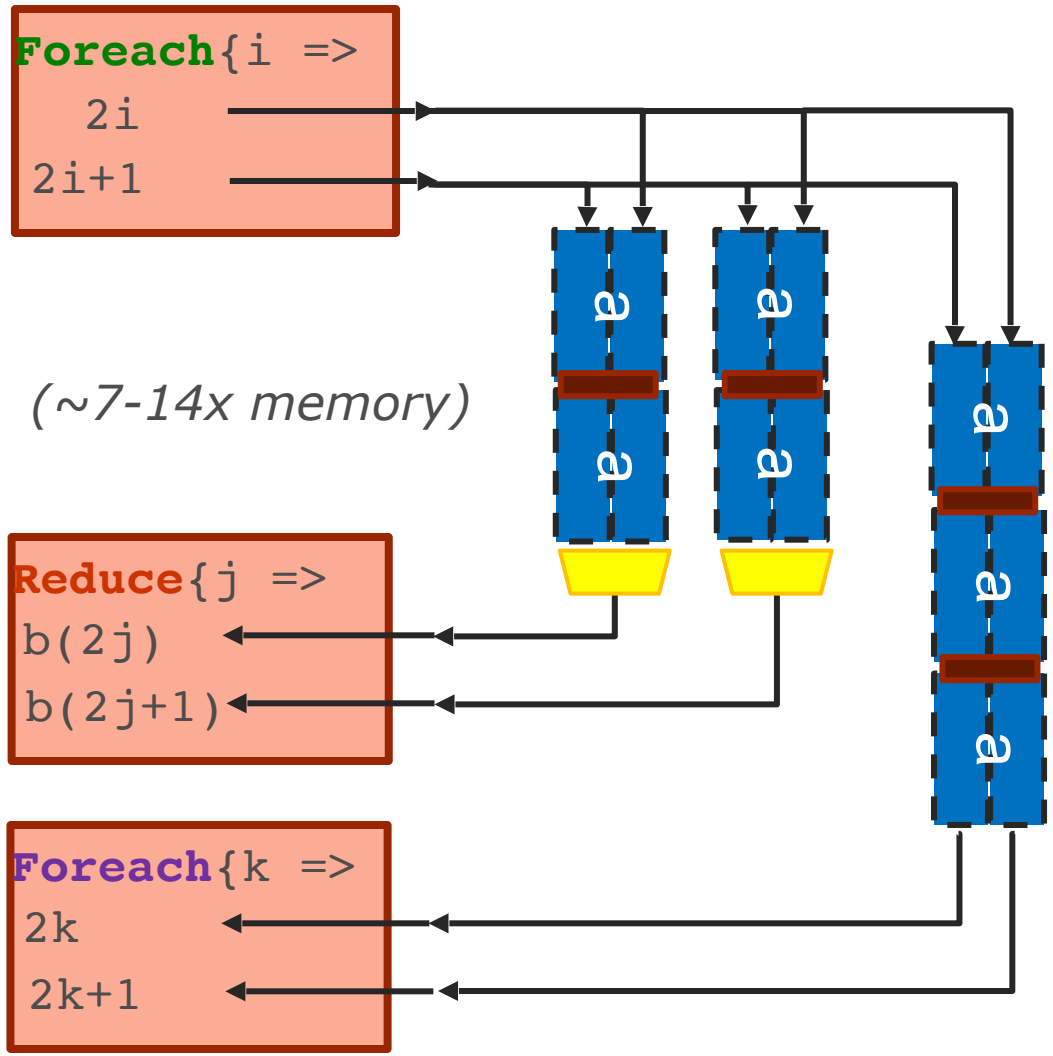
Local Memory Analysis Example (Cont.)

```
Foreach(N by 1){ r =>
  val a = SRAM[Float](D)
  val b = SRAM[Float](D)
  val c = SRAM[Float](D)
  Foreach(D par 2){i =>
    a(i) = …
  }
  Reduce(sum)(D par 2){j =>
    a(b(j))
  }{(a,b) => a + b}
  Foreach(D par 2){k =>
    c(k) = a(k) * sum
  }
}
```



Local Memory Analysis Example (Cont.)

```
Foreach(N by 1){ r =>
  val a = SRAM[Float](D)
  val b = SRAM[Float](D)
  val c = SRAM[Float](D)
  Foreach(D par 2){i =>
    a(i) = …
  }
  Reduce(sum)(D par 2){j =>
    a(b(j))
  }{(a,b) => a + b}
  Foreach(D par 2){k =>
    c(k) = a(k) * sum
  }
}
```



Local Memory Analysis Example (Cont.)

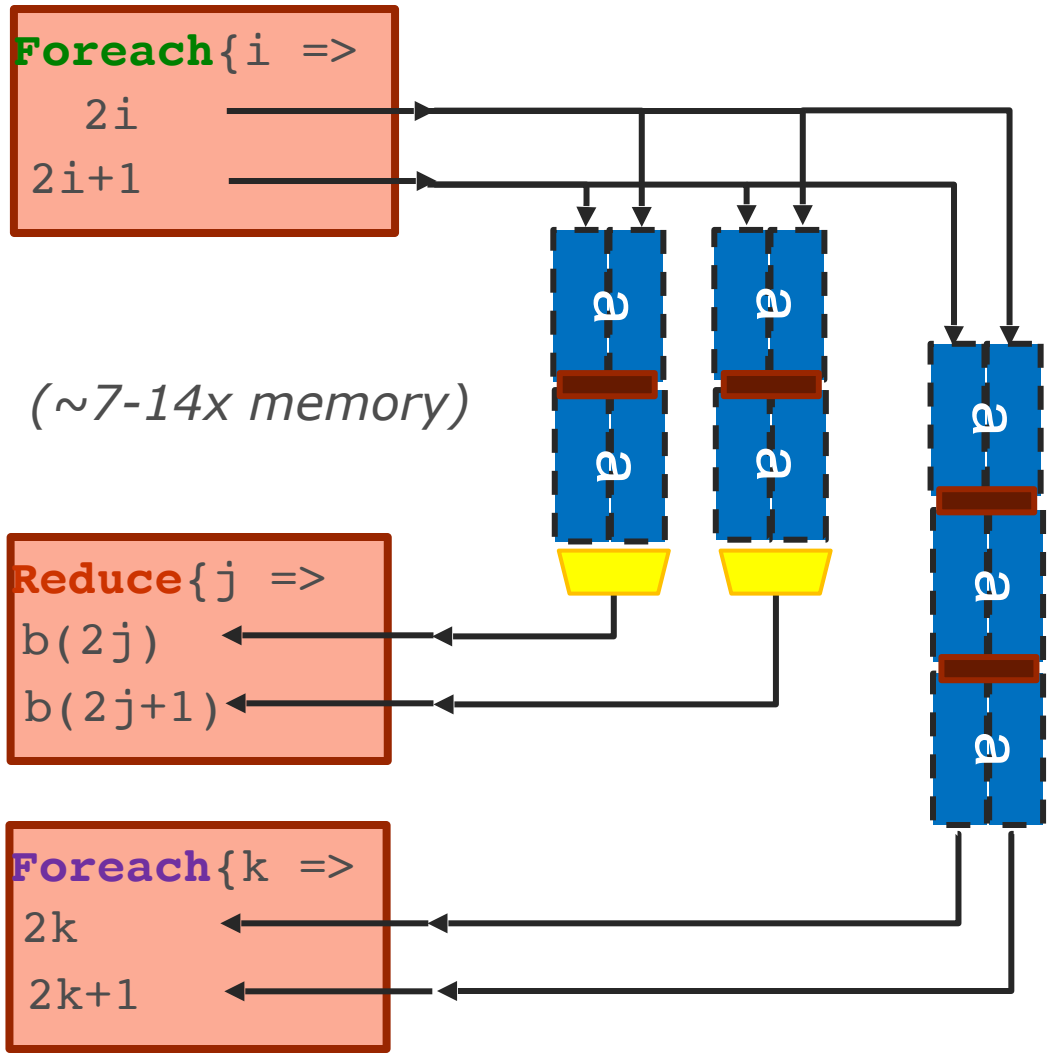
```
Foreach(N by 1){ r =>
  val a = SRAM[Float](D)
  val b = SRAM[Float](D)
```

Step 2: Greedily combine (merge) instances

- Don't combine if there are port conflicts
- Don't combine if the cost of merging is greater than sum of unmerged

****Recompute banking for merged instances!**

```
  }
}
```



Local Memory Analysis

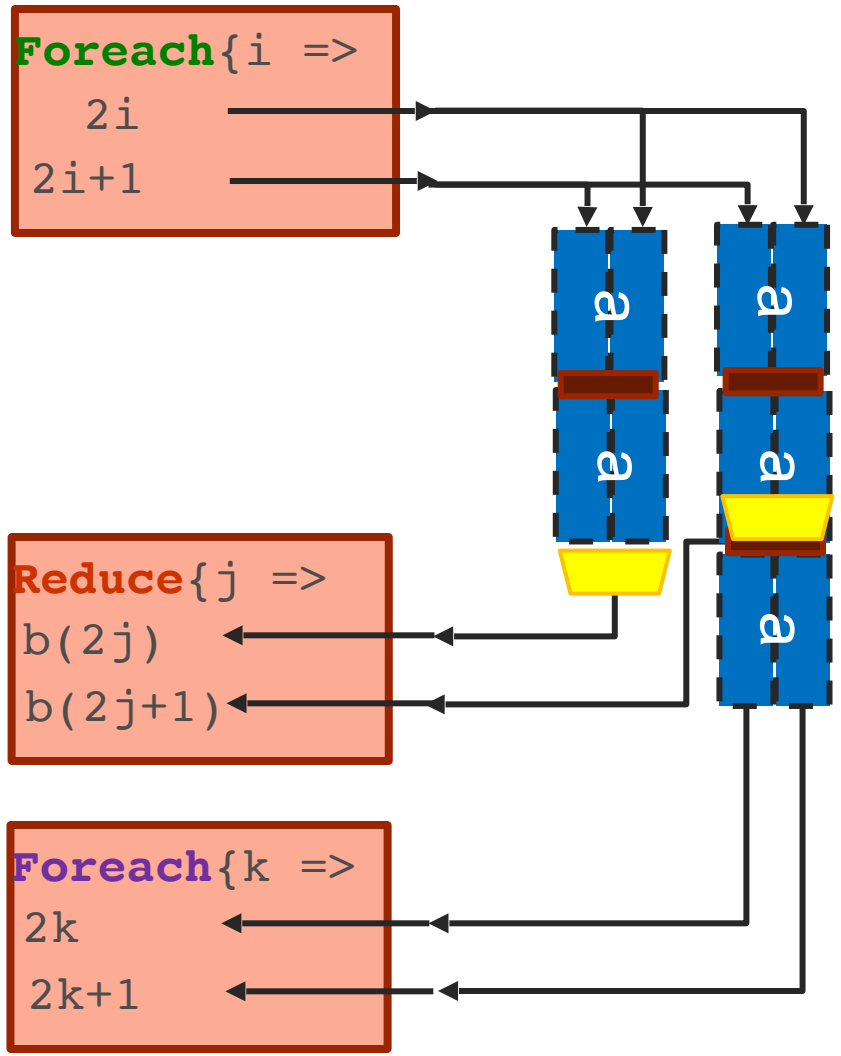
```
Foreach(N by 1){ r =>
  val a = SRAM[Float](D)
  val b = SRAM[Float](D)
```

Step 2: Greedily combine (merge) instances

- Don't combine if there are bank conflicts
- Don't combine if the cost of merging is greater than sum of unmerged

****Recompute banking for merged instances!**

```
  }
}
```



Local Memory Analysis

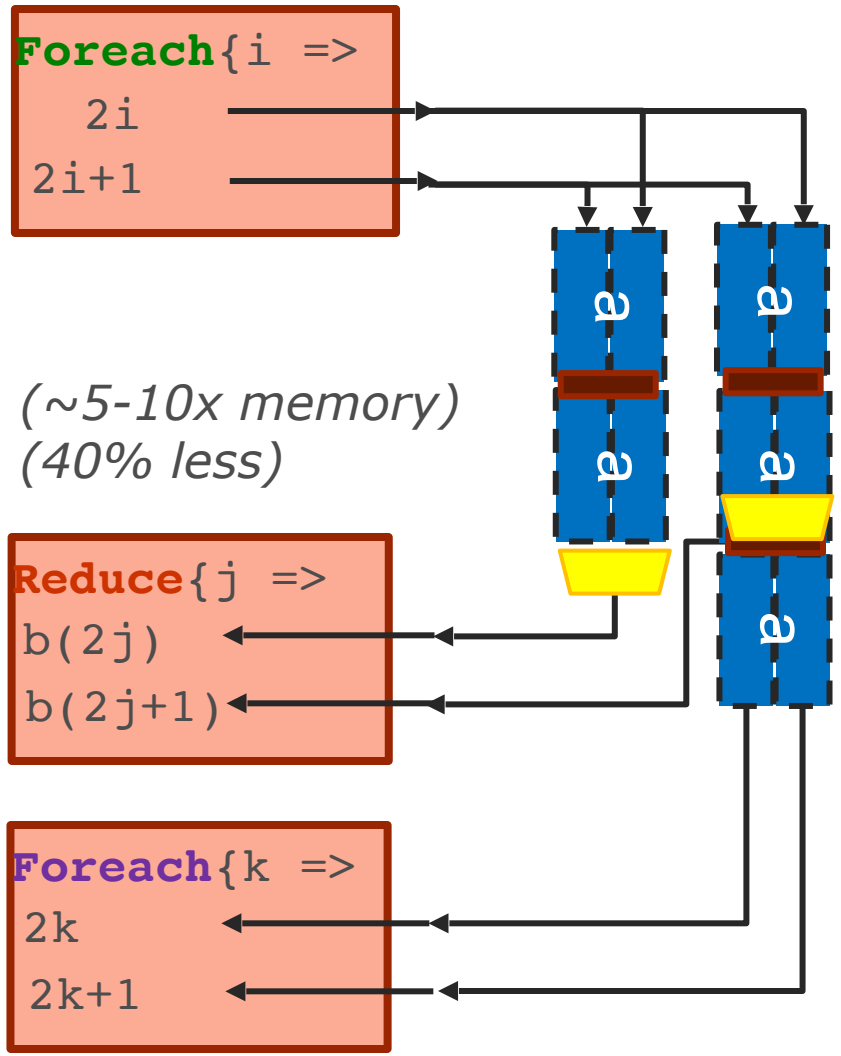
```
Foreach(N by 1){ r =>
  val a = SRAM[Float](D)
  val b = SRAM[Float](D)
```

Step 2: Greedily combine (merge) instances

- Don't combine if there are bank conflicts
- Don't combine if the cost of merging is greater than sum of unmerged

****Recompute banking for merged instances!**

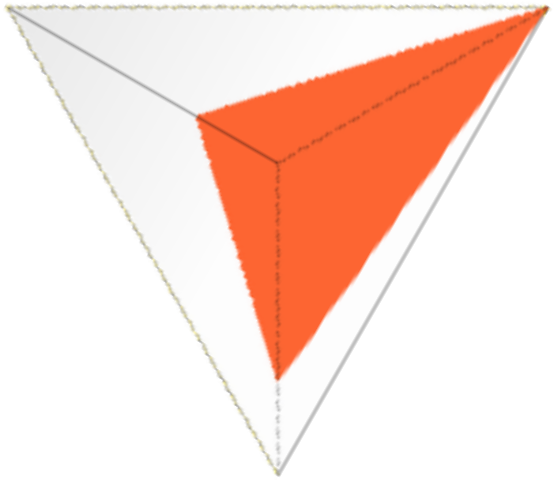
```
  }
}
```



Kernel-Based Approach

Manually implement each DSL operation;

use a simple compiler to stitch them together

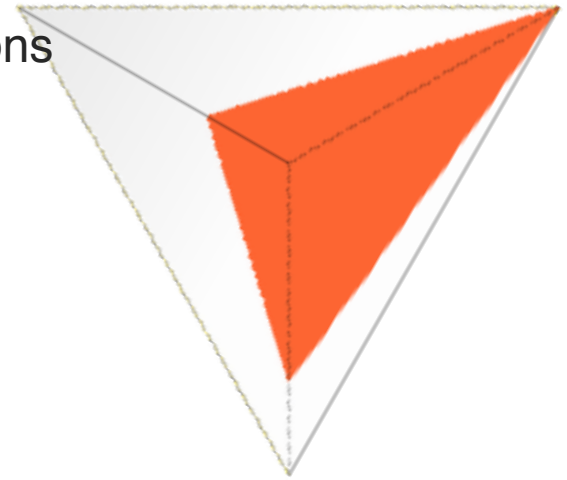


Kernel-Based Approach

Manually implement each DSL operation;
use a simple compiler to stitch them together

Performance

Misses cross-kernel optimizations
Excessive memory transfers
Excessive buffering



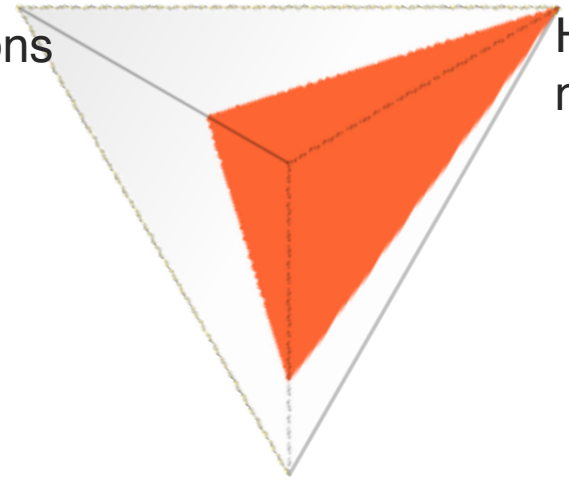
Kernel-Based Approach

Manually implement each DSL operation;

use a simple compiler to stitch them together

Performance

Misses cross-kernel optimizations
Excessive memory transfers
Excessive buffering



Productivity

High level specification
no hardware design knowledge required

Kernel-Based Approach

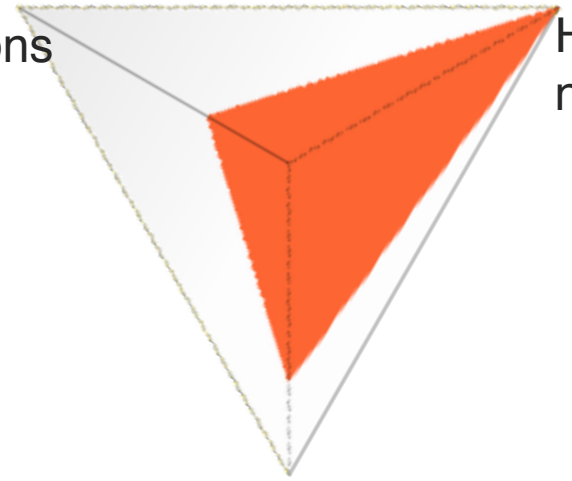
Manually implement each DSL operation;
use a simple compiler to stitch them together

Performance

Misses cross-kernel optimizations
Excessive memory transfers
Excessive buffering

Productivity

High level specification
no hardware design knowledge required



Portability

Reasonably target-generic if done right

Stochastic Gradient Descent in Spatial

```
type TM = FixPt[TRUE,_9,_23]
type TX = FixPt[TRUE,_9,_7]

val data    = DRAM[TX](N, D)
val y       = DRAM[TM](N)
val weights = DRAM[TM](D)

Accel {
  val yAddr  = Reg[Int](-1)
  val yCache = SRAM[TM](CSIZE)
  val wK     = SRAM[TM](D)

  wK load weights(0::D)

  Sequential.Foreach(E by 1){e =>
    epoch(random[Int](N), …)
    breakpoint()
  }

  weights(0 :: D) store wK
}
```

Stochastic Gradient Descent in Spatial

```
type TM = FixPt[TRUE,_9,_23]
type TX = FixPt[TRUE,_9,_7]

val data    = DRAM[TX](N, D)
val y       = DRAM[TM](N)
val weights = DRAM[TM](D)

Accel {
  val yAddr  = Reg[Int](-1)
  val yCache = SRAM[TM](CSIZE)
  val wK     = SRAM[TM](D)

  wK load weights(0::D)

  Sequential.Foreach(E by 1){e =>
    epoch(random[Int](N), …)
    breakpoint()
  }

  weights(0 :: D) store wK
}
```

Arbitrary precision custom types

Stochastic Gradient Descent in Spatial

```
type TM = FixPt[TRUE,_9,_23]
type TX = FixPt[TRUE,_9,_7]

val data    = DRAM[TX](N, D)
val y       = DRAM[TM](N)
val weights = DRAM[TM](D)

Accel {
  val yAddr  = Reg[Int](-1)
  val yCache = SRAM[TM](CSIZE)
  val wK     = SRAM[TM](D)

  wK load weights(0::D)

  Sequential.Foreach(E by 1){e =>
    epoch(random[Int](N), …)
    breakpoint()
  }

  weights(0 :: D) store wK
}
```

- **Arbitrary precision** custom types
- **Off-chip** memory allocations

Stochastic Gradient Descent in Spatial

```
type TM = FixPt[TRUE,_9,_23]
type TX = FixPt[TRUE,_9,_7]

val data    = DRAM[TX](N, D)
val y       = DRAM[TM](N)
val weights = DRAM[TM](D)

Accel {
  val yAddr  = Reg[Int](-1)
  val yCache = SRAM[TM](CSIZE)
  val wK     = SRAM[TM](D)

  wK load weights(0::D)

  Sequential.Foreach(E by 1){e =>
    epoch(random[Int](N), …)
    breakpoint()
  }

  weights(0 :: D) store wK
}
```

- **Arbitrary precision** custom types
- **Off-chip** memory allocations
- Accelerator scope

Stochastic Gradient Descent in Spatial

```
type TM = FixPt[TRUE,_9,_23]
type TX = FixPt[TRUE,_9,_7]

val data    = DRAM[TX](N, D)
val y       = DRAM[TM](N)
val weights = DRAM[TM](D)

Accel {
  val yAddr  = Reg[Int](-1)
  val yCache = SRAM[TM](CSIZE)
  val wK     = SRAM[TM](D)

  wK load weights(0::D)

  Sequential.Foreach(E by 1){e =>
    epoch(random[Int](N), …)
    breakpoint()
  }

  weights(0 :: D) store wK
}
```

- **Arbitrary precision** custom types
- **Off-chip** memory allocations
- Accelerator scope
- **On-chip** memory allocations

Stochastic Gradient Descent in Spatial

```
type TM = FixPt[TRUE,_9,_23]
type TX = FixPt[TRUE,_9,_7]

val data    = DRAM[TX](N, D)
val y       = DRAM[TM](N)
val weights = DRAM[TM](D)

Accel {
  val yAddr  = Reg[Int](-1)
  val yCache = SRAM[TM](CSIZE)
  val wK     = SRAM[TM](D)

  wK load weights(0::D)

  Sequential.Foreach(E by 1){e =>
    epoch(random[Int](N), …)
    breakpoint()
  }

  weights(0 :: D) store wK
}
```

- **Arbitrary precision** custom types
- **Off-chip** memory allocations
- Accelerator scope
- **On-chip** memory allocations
- **Explicit** memory transfer

Stochastic Gradient Descent in Spatial

```
1  type TM = FixPt[TRUE,_9,_23]
2  type TX = FixPt[TRUE,_9,_7]
3
4  val data    = DRAM[TX](N, D)
5  val y       = DRAM[TM](N)
6  val weights = DRAM[TM](D)
7
8  Accel {
9    val yAddr  = Reg[Int](-1)
10   val yCache = SRAM[TM](CSIZE)
11   val wK     = SRAM[TM](D)
12
13   wK load weights(0::D)
14
15   Sequential.Foreach(E by 1){e =>
16     epoch(random[Int](N), …)
17     breakpoint()
18   }
19
20   weights(0 :: D) store wK
21 }
```

- **Arbitrary precision** custom types
- **Off-chip** memory allocations
- Accelerator scope
- **On-chip** memory allocations
- **Explicit** memory transfer
- Declaration of a sequential loop

Stochastic Gradient Descent in Spatial

```
1  type TM = FixPt[TRUE,_9,_23]
2  type TX = FixPt[TRUE,_9,_7]
3
4  val data    = DRAM[TX](N, D)
5  val y       = DRAM[TM](N)
6  val weights = DRAM[TM](D)
7
8  Accel {
9    val yAddr  = Reg[Int](-1)
10   val yCache = SRAM[TM](CSIZE)
11   val wK     = SRAM[TM](D)
12
13   wK load weights(0::D)
14
15   Sequential.Foreach(E by 1){e =>
16     epoch(random[Int](N), …)
17     breakpoint()
18   }
19
20   weights(0 :: D) store wK
21 }
```

Arbitrary precision custom types

Off-chip memory allocations

Accelerator scope

On-chip memory allocations

Explicit memory transfer

Declaration of a sequential loop

Explicit memory transfer

Stochastic Gradient Descent in Spatial

```
1  type TM = FixPt[TRUE,_9,_23]
2  type TX = FixPt[TRUE,_9,_7]
3
4  val data    = DRAM[TX](N, D)
5  val y       = DRAM[TM](N)
6  val weights = DRAM[TM](D)
7
8  Accel {
9    val yAddr  = Reg[Int](-1)
10   val yCache = SRAM[TM](CSIZE)
11   val wK     = SRAM[TM](D)
12
13   wK load weights(0::D)
14
15   Sequential.Foreach(E by 1){e =>
16     epoch(random[Int](N), …)
17     breakpoint()
18   }
19
20   weights(0 :: D) store wK
21 }
```

- **Arbitrary precision** custom types
- **Off-chip** memory allocations
- Accelerator scope
- **On-chip** memory allocations
- **Explicit** memory transfer
- Declaration of a sequential loop
- Debugging breakpoint
- **Explicit** memory transfer

SGD in Spatial

```
22 def epoch(i: Int, ...): Unit = {
23   val yPt = Reg[TM]
24   if (i >= yAddr & i < yAddr+CSIZE & yAddr != -1) {
25     yPt := yCache(i - yAddr)
26   }
27   else {
28     yAddr := i - (i % CSIZE)
29     yCache load y(yAddr::yAddr + CSIZE)
30     yPt := yCache(i % CSIZE)
31   }
32
33   val x = SRAM[TX](D)
34   x load data(i, 0::D)
35
36   // Compute gradient against wK_t
37   val yHat = Reg[TM]
38   Reduce(yHat)(D by 1){j => wK(j) * x(j).to[TM] }
39 {_+_}
40   val yErr = yHat - yPt
41
42   // Update wK_t with reduced variance update
43   Foreach(D by 1){i =>
44     wK(i) = wK(i) – (A.to[TM] * yErr * x(i).to[TM])
45   }
}
```

SGD in Spatial

```
22 def epoch(i: Int, ...): Unit = {
23   val yPt = Reg[TM]
24   if (i >= yAddr & i < yAddr+CSIZE & yAddr != -1) {
25     yPt := yCache(i - yAddr)
26   }
27   else {
28     yAddr := i - (i % CSIZE)
29     yCache load y(yAddr::yAddr + CSIZE)
30     yPt := yCache(i % CSIZE)
31   }
32
33   val x = SRAM[TX](D)
34   x load data(i, 0::D)
35
36   // Compute gradient against wK_t
37   val yHat = Reg[TM]
38   Reduce(yHat)(D by 1){j => wK(j) * x(j).to[TM] }
39 {_+_}
40   val yErr = yHat - yPt
41
42   // Update wK_t with reduced variance update
43   Foreach(D by 1){i =>
44     wK(i) = wK(i) – (A.to[TM] * yErr * x(i).to[TM])
45   }
}
```

Custom caching for random access on *y*

SGD in Spatial

```
22 def epoch(i: Int, ...): Unit = {
23   val yPt = Reg[TM]
24   if (i >= yAddr & i < yAddr+CSIZE & yAddr != -1) {
25     yPt := yCache(i - yAddr)
26   }
27   else {
28     yAddr := i - (i % CSIZE)
29     yCache load y(yAddr::yAddr + CSIZE)
30     yPt := yCache(i % CSIZE)
31   }
32
33   val x = SRAM[TX](D)
34   x load data(i, 0::D)
35
36   // Compute gradient against wK_t
37   val yHat = Reg[TM]
38   Reduce(yHat)(D by 1){j => wK(j) * x(j).to[TM] }
39 {_+_}
40   val yErr = yHat - yPt
41
42   // Update wK_t with reduced variance update
43   Foreach(D by 1){i =>
44     wK(i) = wK(i) – (A.to[TM] * yErr * x(i).to[TM])
45   }
}
```

Custom caching for random access on *y*

Explicit memory transfer

SGD in Spatial

```
22 def epoch(i: Int, ...): Unit = {
23   val yPt = Reg[TM]
24   if (i >= yAddr & i < yAddr+CSIZE & yAddr != -1) {
25     yPt := yCache(i - yAddr)
26   }
27   else {
28     yAddr := i - (i % CSIZE)
29     yCache load y(yAddr::yAddr + CSIZE)
30     yPt := yCache(i % CSIZE)
31   }
32
33   val x = SRAM[TX](D)
34   x load data(i, 0::D)
35
36   // Compute gradient against wK_t
37   val yHat = Reg[TM]
38   Reduce(yHat)(D by 1){j => wK(j) * x(j).to[TM] }
39 {_+_}
40   val yErr = yHat - yPt
41
42   // Update wK_t with reduced variance update
43   Foreach(D by 1){i =>
44     wK(i) = wK(i) – (A.to[TM] * yErr * x(i).to[TM])
45   }
}
```

Custom caching for random access on *y*

Explicit memory transfer

Gradient computation

SGD in Spatial

```
22 def epoch(i: Int, ...): Unit = {
23   val yPt = Reg[TM]
24   if (i >= yAddr & i < yAddr+CSIZE & yAddr != -1) {
25     yPt := yCache(i - yAddr)
26   }
27   else {
28     yAddr := i - (i % CSIZE)
29     yCache load y(yAddr::yAddr + CSIZE)
30     yPt := yCache(i % CSIZE)
31   }
32
33   val x = SRAM[TX](D)
34   x load data(i, 0::D)
35
36   // Compute gradient against wK_t
37   val yHat = Reg[TM]
38   Reduce(yHat)(D by 1){j => wK(j) * x(j).to[TM] }
39 {_+_}
40   val yErr = yHat - yPt
41
42   // Update wK_t with reduced variance update
43   Foreach(D by 1){i =>
44     wK(i) = wK(i) — (A.to[TM] * yErr * x(i).to[TM])
45   }
}
```

Custom caching for random access on *y*

Explicit memory transfer

Gradient computation

Weight update

SGD in Spatial: Hardware