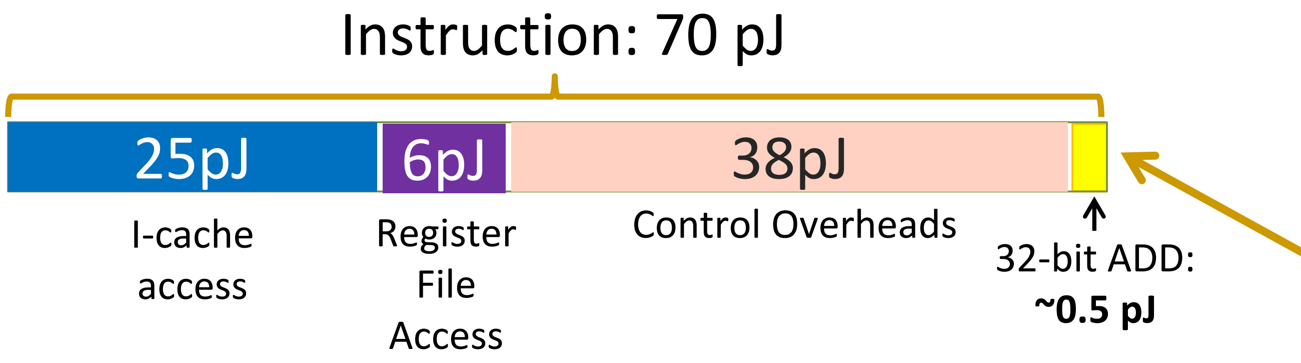
Stanford | ENGINEERING
Electrical Engineering
Computer Science

Spatial: A Language and Compiler for Application Accelerators

David Koeplinger Matthew Feldman Raghu Prabhakar
Yaqi Zhang Stefan Hadjis Ruben Fiszel
Tian Zhao Luigi Nardi Ardavan Pedram
Christos Kozyrakis Kunle Olukotun

PLDI
June 21, 2018

Instructions Add Overheads



Instruction: 70 pJ

25pJ

I-cache access

6pJ

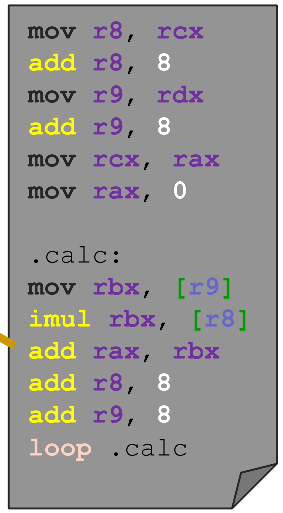
Register File Access

38pJ

Control Overheads

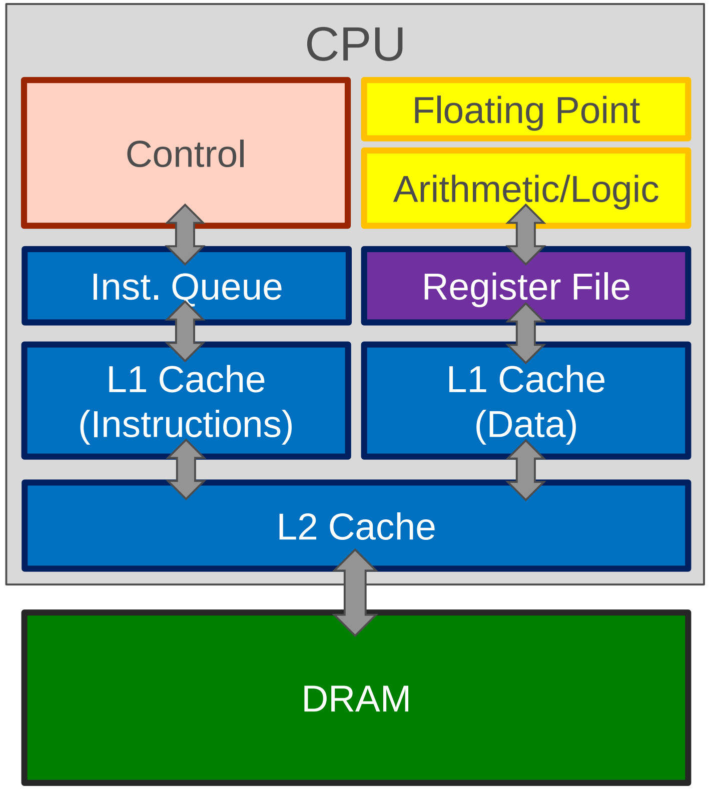
32-bit ADD: **~0.5 pJ**

vectorA · vectorB



```
mov r8, rcx
add r8, 8
mov r9, rdx
add r9, 8
mov rcx, rax
mov rax, 0

.calc:
mov rbx, [r9]
imul rbx, [r8]
add rax, rbx
add r8, 8
add r9, 8
loop .calc
```

Instruction-Based



CPU

Control

Floating Point

Arithmetic/Logic

Inst. Queue

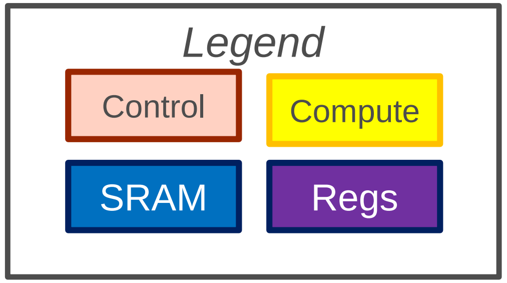
Register File

L1 Cache (Instructions)

L1 Cache (Data)

L2 Cache

DRAM



Legend

Control

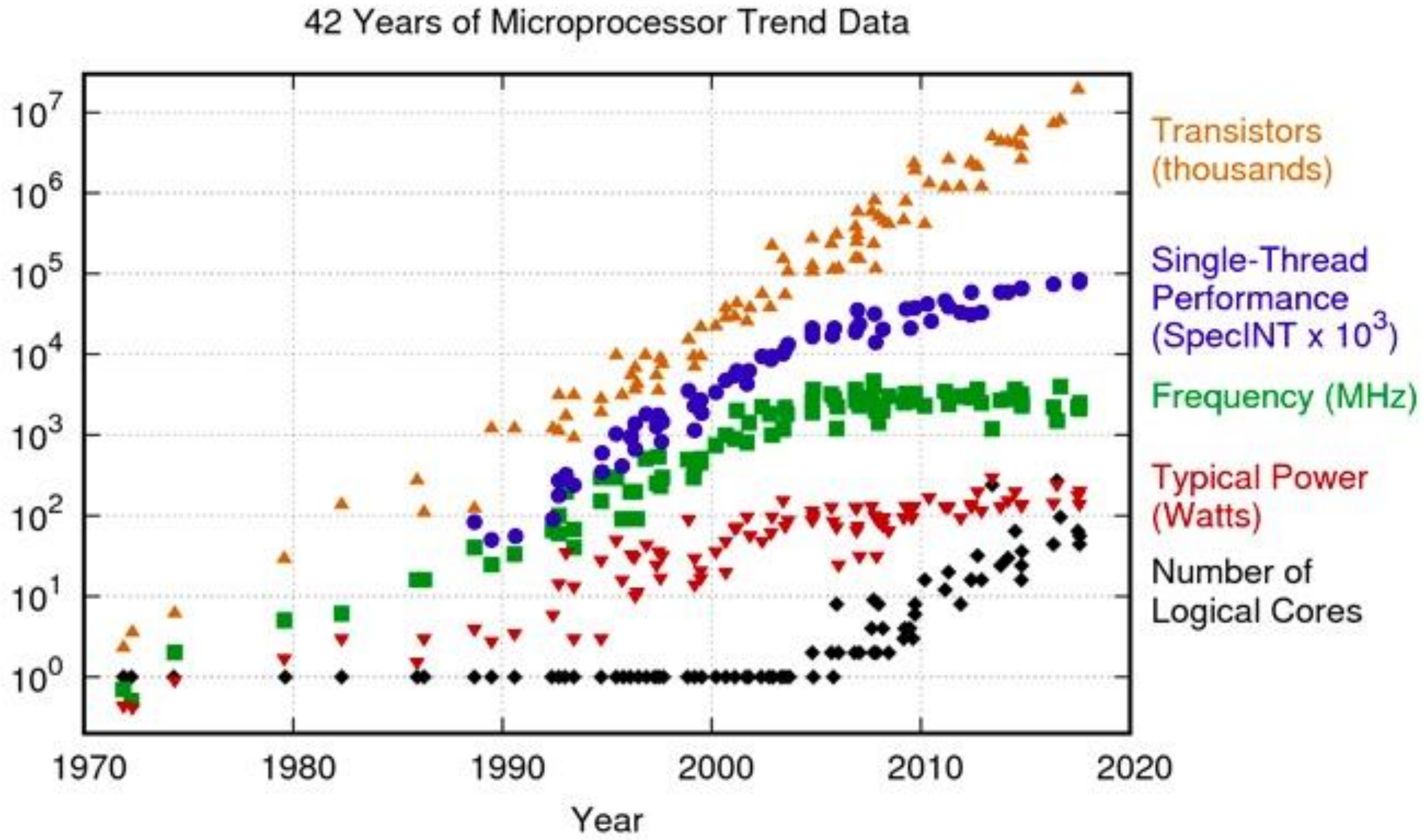
Compute

SRAM

Regs

Mark Horowitz, *Computing's Energy Problem (and what we can do about it)* ISSCC 2014

A Dark Tale: The CPU Power Wall

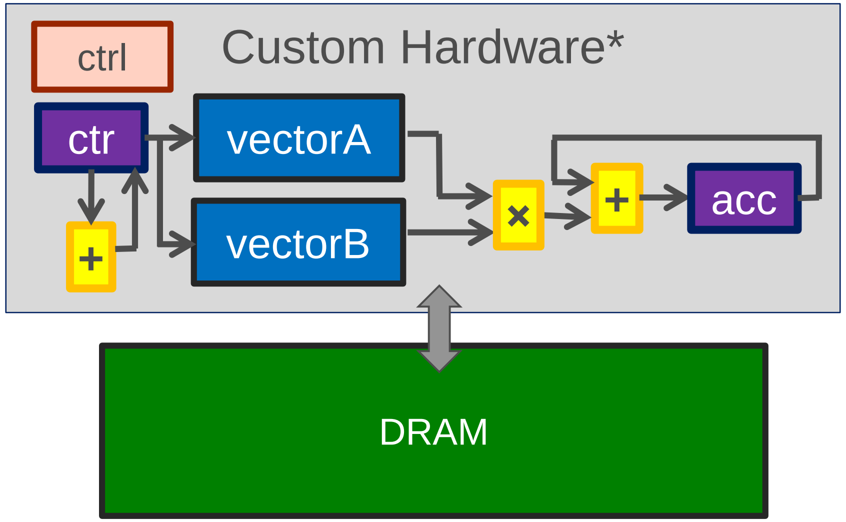


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/

A More Efficient Way

Configuration-Based

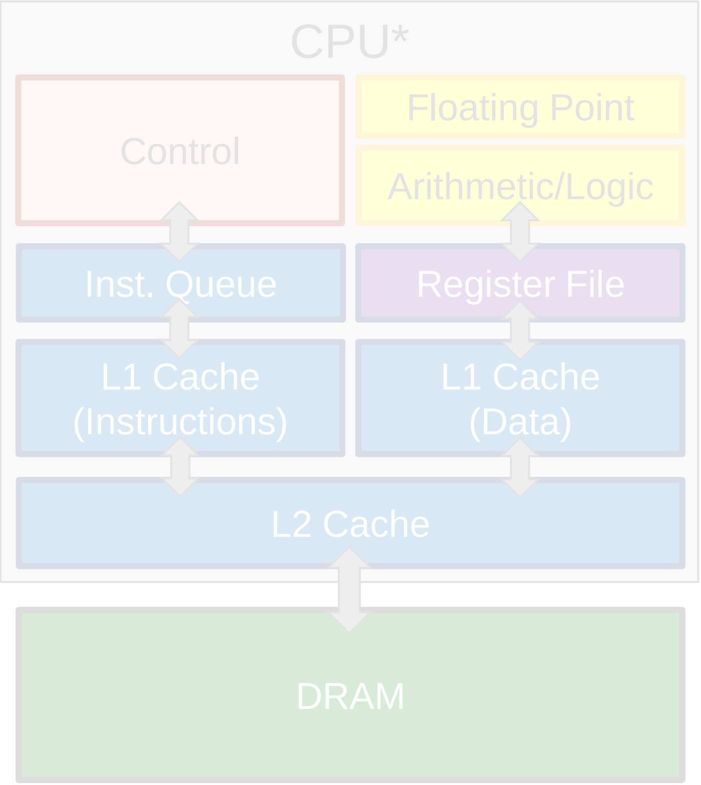


*Also not to scale

vectorA · vectorB

```
mov r8, rcx
add r8, 8
mov r9, rdx
add r8, 8
mov rcx, rax
mov rax, 0

.calc:
mov rbx, [r9]
imul rbx, [r8]
add rax, rbx
add r8, 8
add r9, 8
loop .calc
```

Instruction-Based

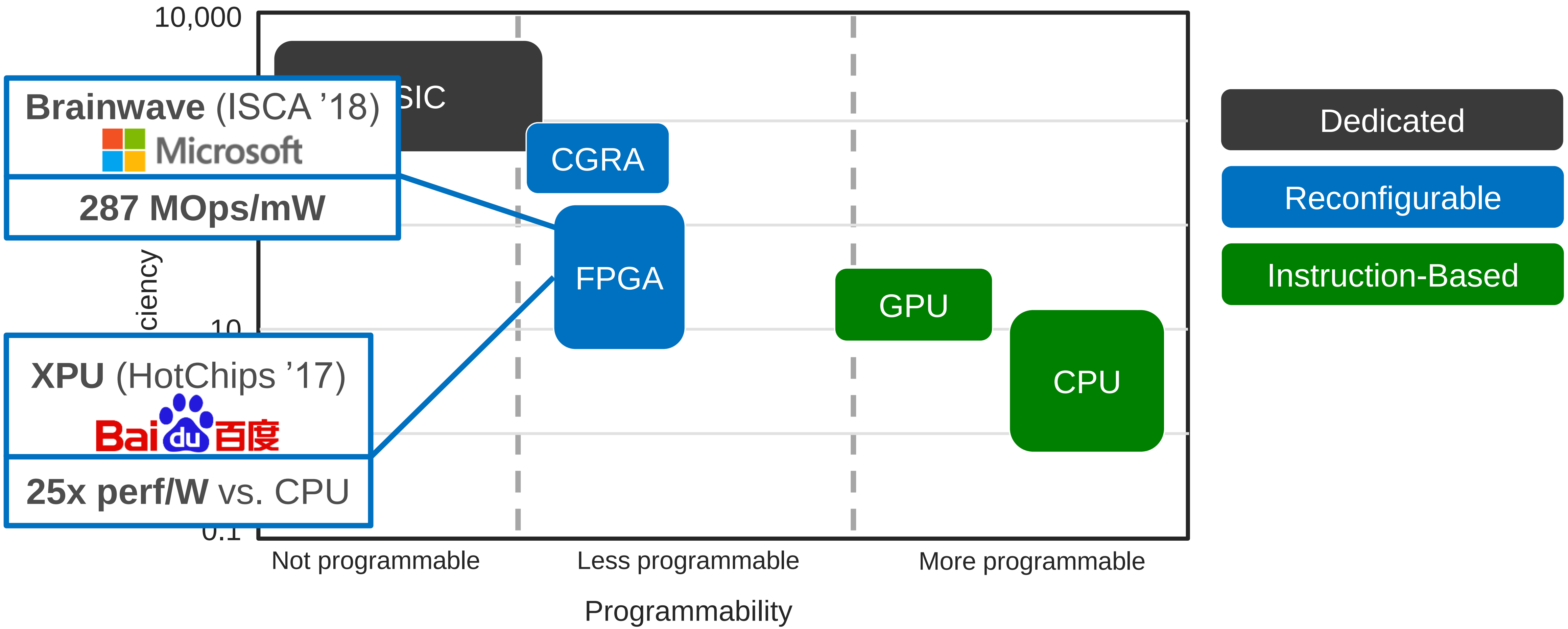


*Not to scale

Legend

🟥 Control
🟨 Compute
🟦 SRAM
🟪 Regs

The Future Is (Probably) Reconfigurable



Brainwave (ISCA ’18)
Microsoft

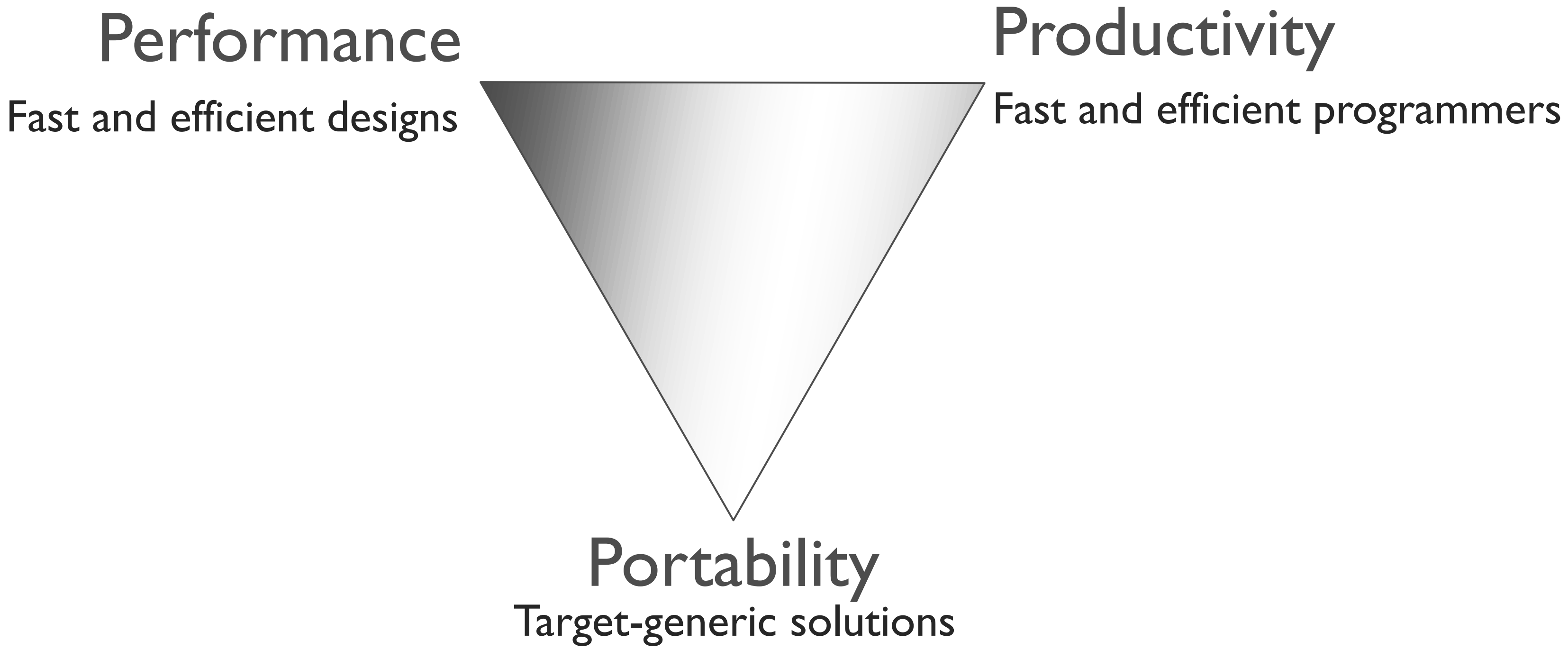
287 MOps/mW

XPU (HotChips ’17)
Baidu 百度

25x perf/W vs. CPU

Key Question

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



Performance

Fast and efficient designs

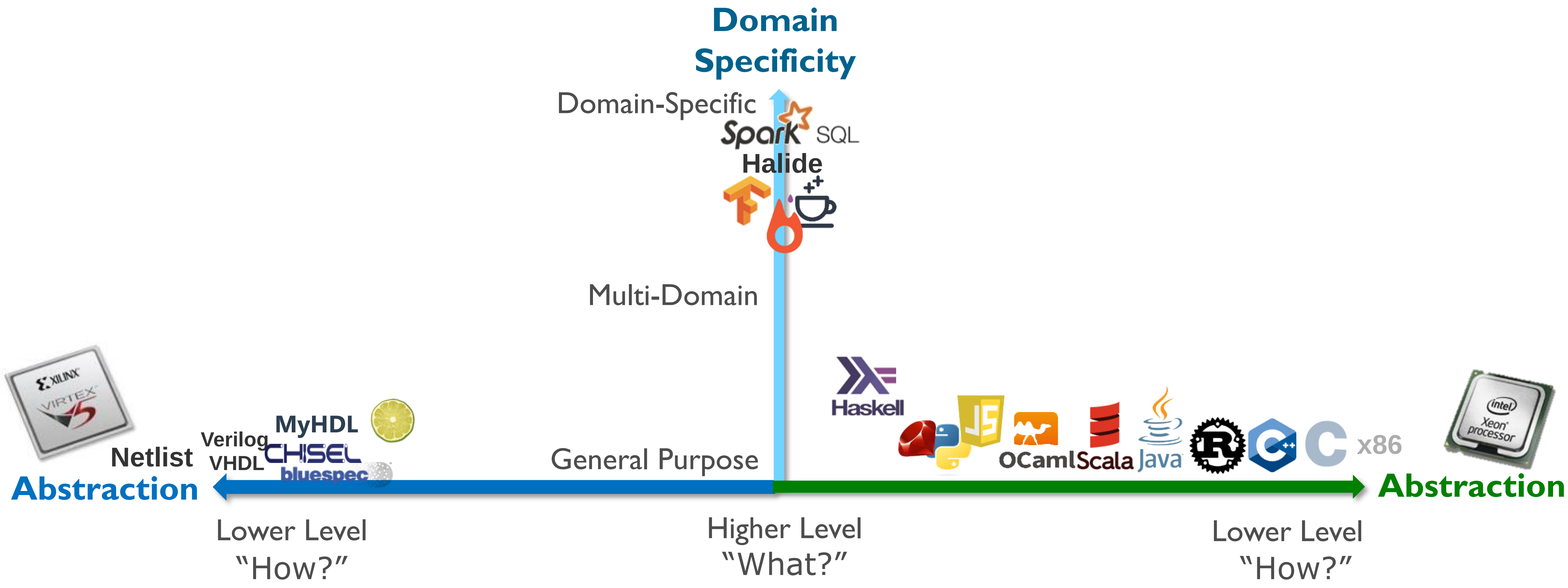
Productivity

Fast and efficient programmers

Portability

Target-generic solutions

Language Taxonomy



Domain Specificity

Domain-Specific

Spark SQL

Halide

Multi-Domain

General Purpose

Netlist

Verilog

VHDL

MyHDL

CHISEL

bluespec

Haskell

OCaml

Scala

Java

x86

Abstraction

Abstraction

Lower Level
"How?"

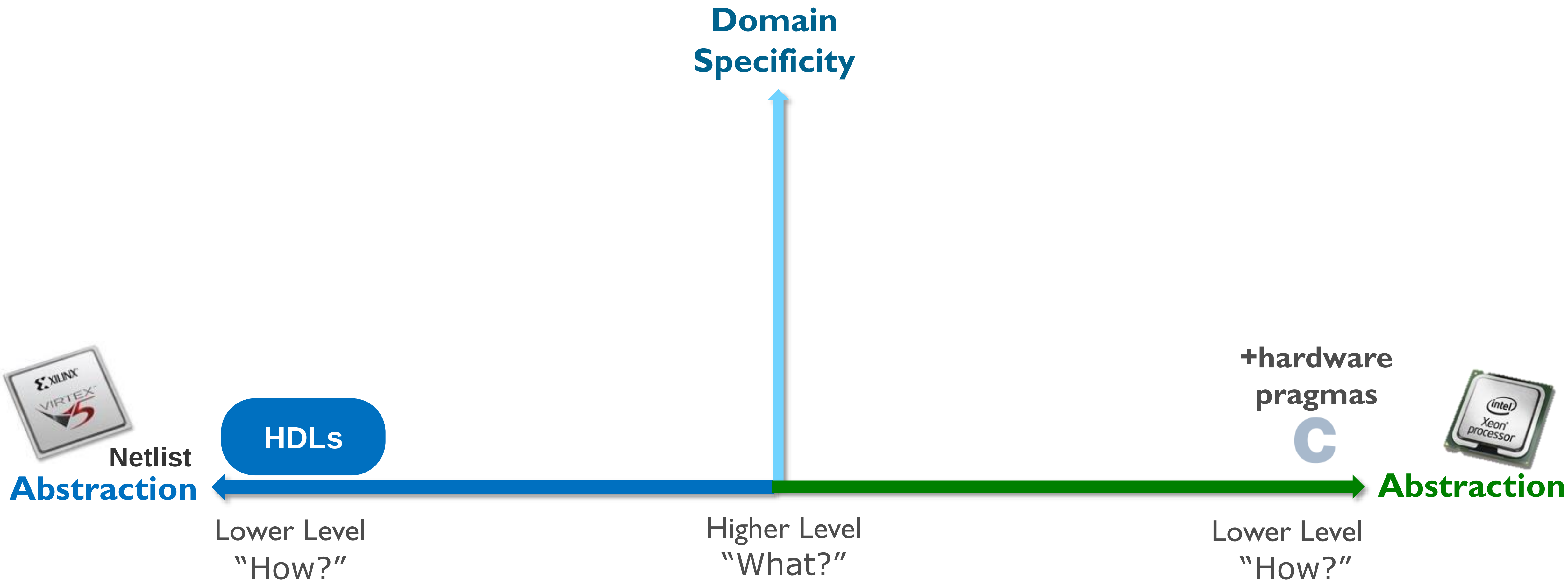
Higher Level
"What?"

Lower Level
"How?"

Reconfigurable Architectures (FPGAs)

Instruction-Based Architectures (CPUs)

Abstracting Hardware Design



Domain Specificity

+hardware pragmas

C

HDLs

Netlist

Abstraction

Abstraction

Lower Level
"How?"

Higher Level
"What?"

Lower Level
"How?"

Reconfigurable Architectures (FPGAs)

Instruction-Based Architectures (CPUs)

HDLs

Hardware Description Languages (HDLs)

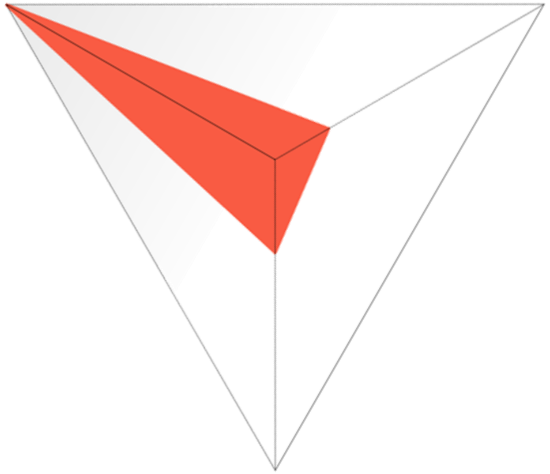
e.g. Verilog, VHDL, Chisel, Bluespec

Performance

✓ Arbitrary RTL

Productivity

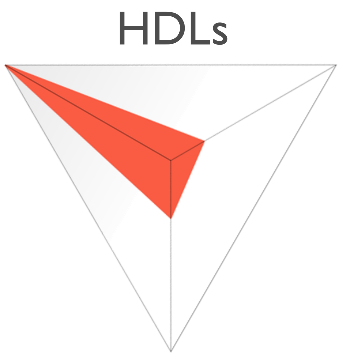
✗ No high-level abstractions



Portability

✗ Significant target-specific code

C + Pragmas



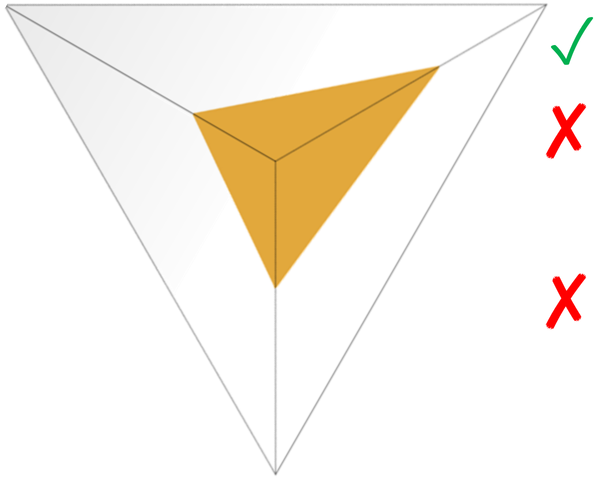
Existing High Level Synthesis (C + Pragmas)

e.g. Vivado HLS, SDAccel, Altera OpenCL

Performance

✗ No memory hierarchy

✗ No arbitrary pipelining



Productivity

✓ Nested loops

✗ Ad-hoc mix of software/hardware

✗ Difficult to optimize

Portability

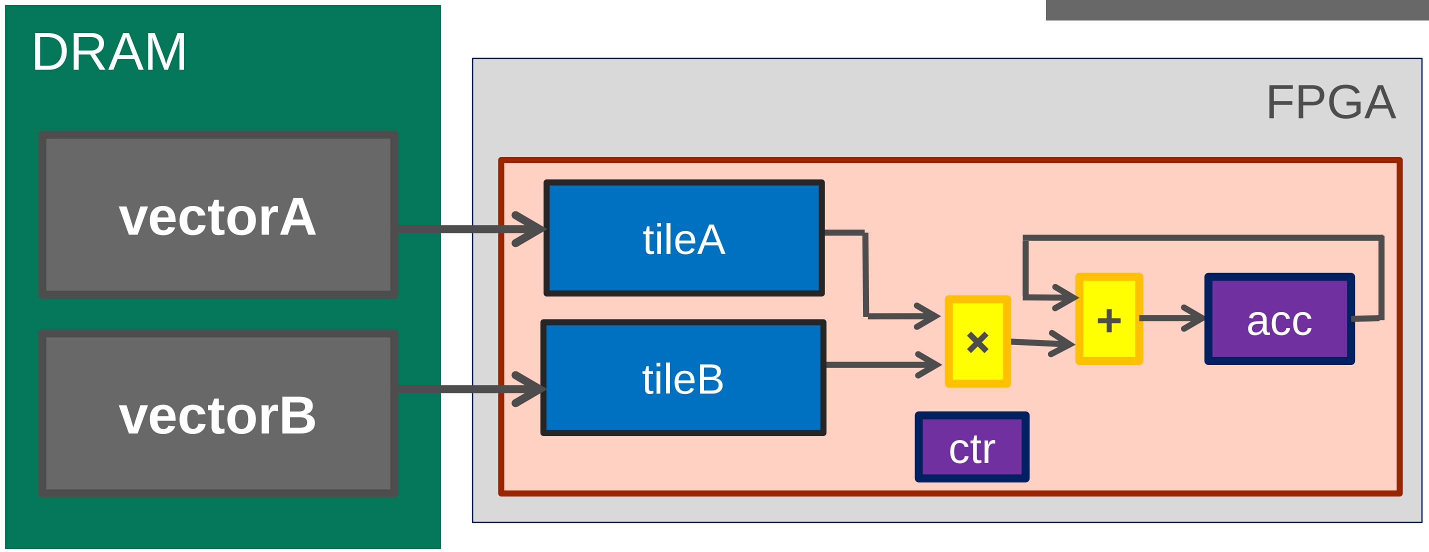
✓ Portable for single vendor

Criteria for Improved HLS

Requirement	C+Pragmas
Represent memory hierarchy explicitly Aids on-chip memory optimization, specialization	✗
Express control as nested loops Enables analysis of access patterns	✓
Support arbitrarily nested pipelining Exploits nested parallelism	✗
Specialize memory transfers Enables customized memory controllers based on access patterns	✗
Capture design parameters Enables automatic design tuning in compiler	✗

Design Space Parameters Example

```
vectorA · vectorB
```



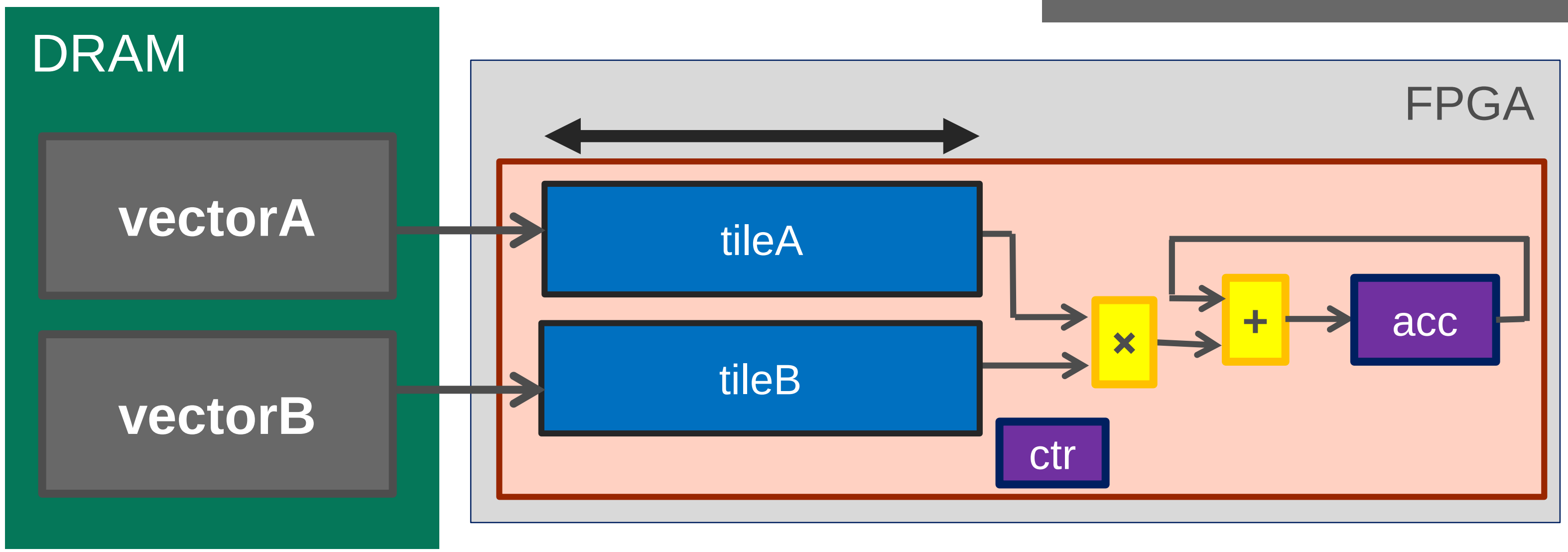
Small and simple, but slow!

Legend

Control	Compute
SRAM	Regs

Important Parameters: Buffer Sizes

`vectorA · vectorB`



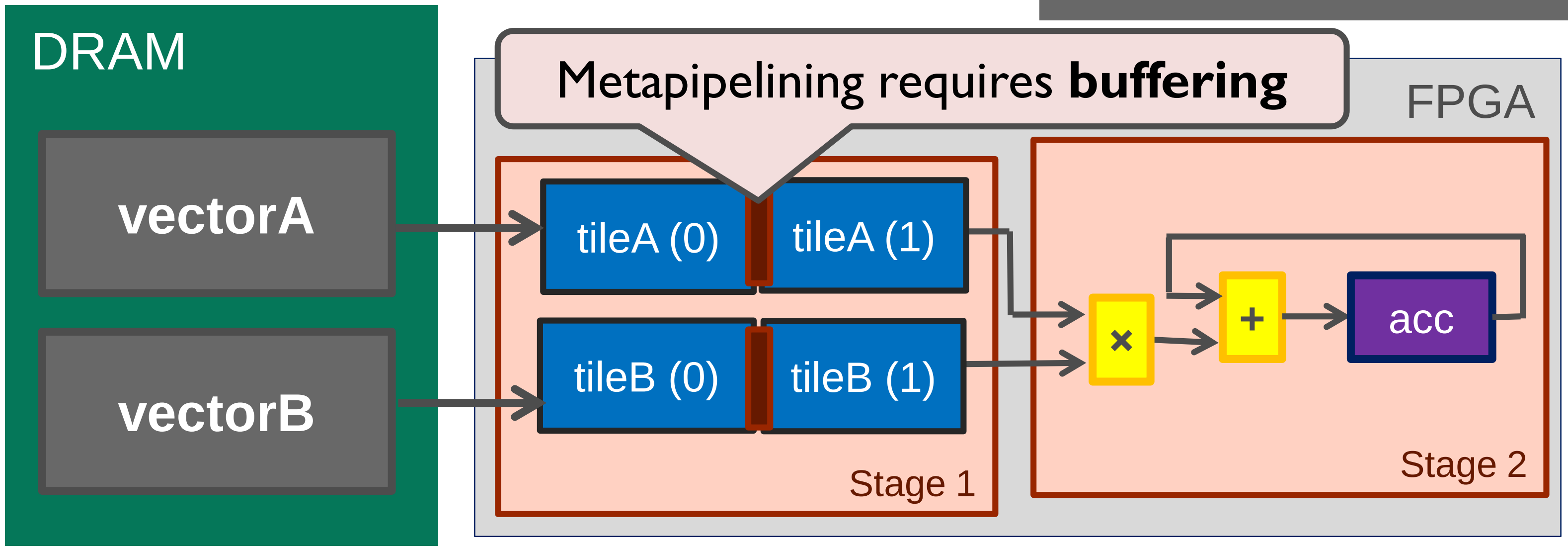
- Increases length of DRAM accesses ⬇ Runtime
- Increases exploited locality ⬇ Runtime
- Increases local memory sizes ⬆ Area

Legend

🟥 Control	🟨 Compute
🟦 SRAM	🟪 Regs

Important Parameters: Pipelining

`vectorA · vectorB`



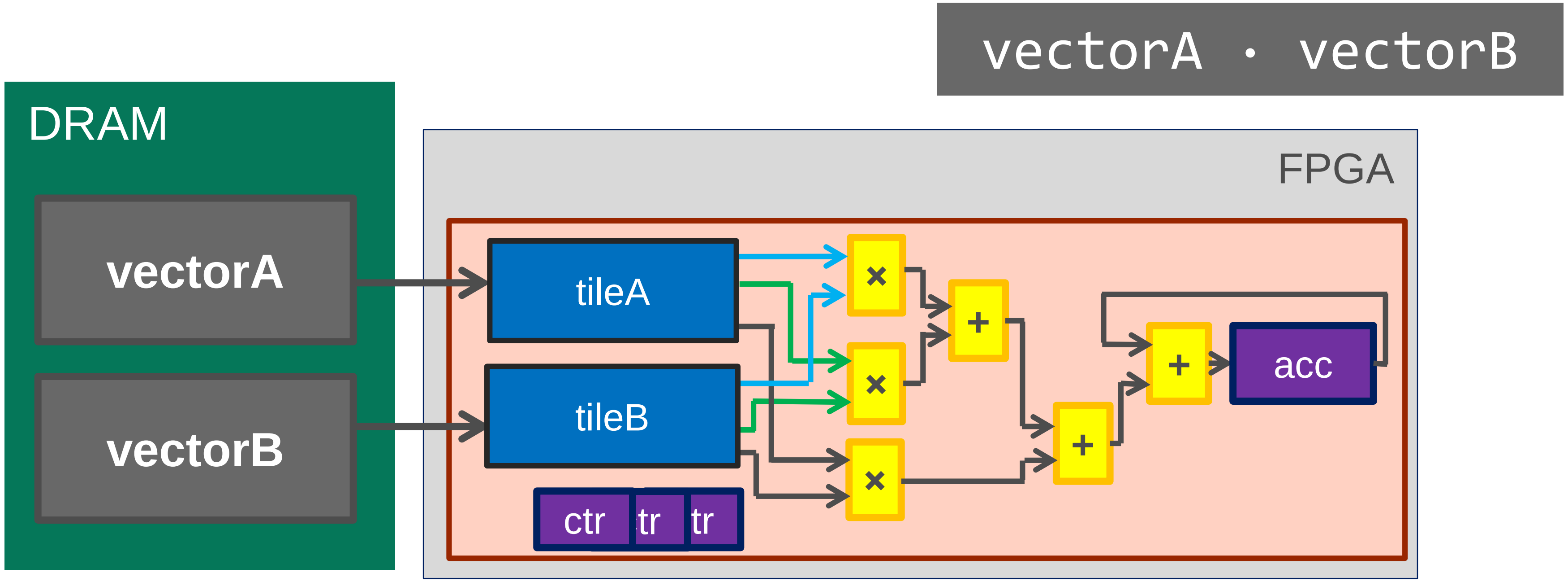
Metapipelining requires **buffering**

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

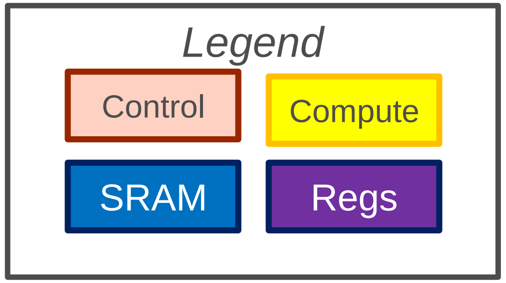
Legend

Control	Compute
SRAM	Regs
Double	Buffer

Important Parameters: Parallelization

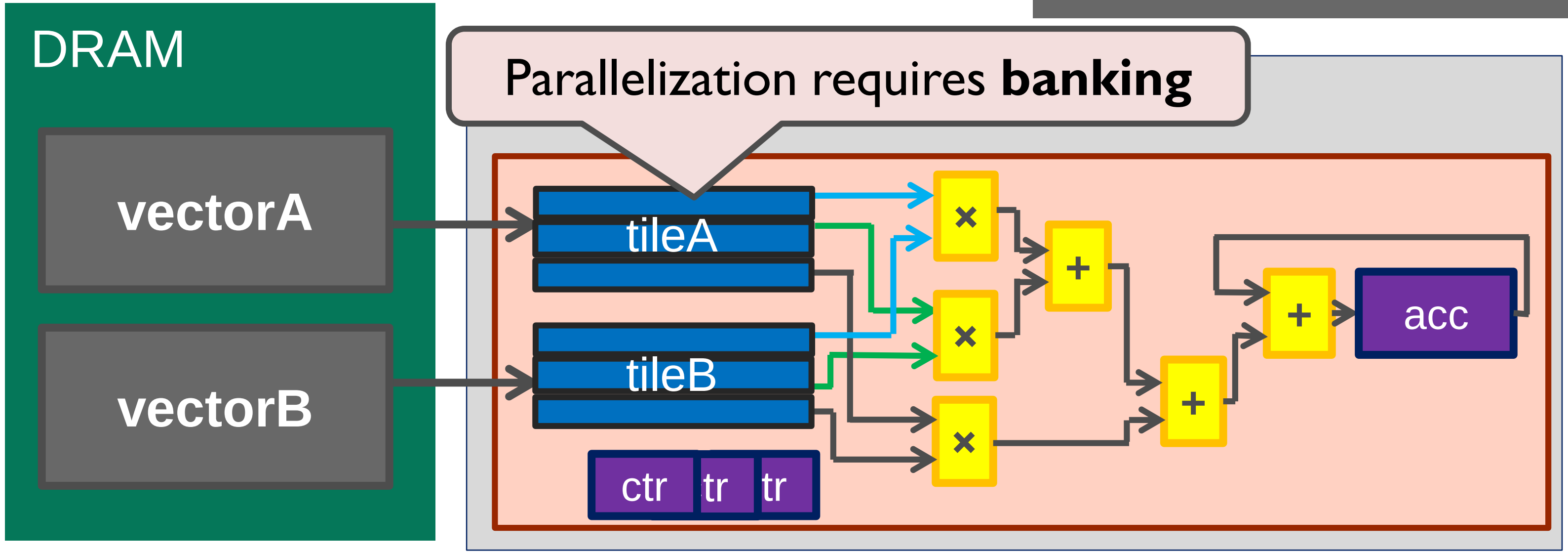


- Improves element throughput ⬇ Runtime
- Duplicates compute resources ⬆ Area

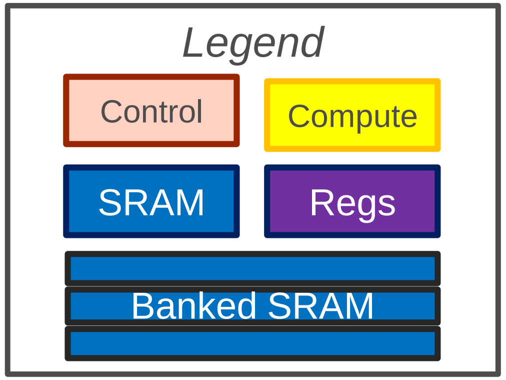


Important Parameters: Memory Banking

vectorA · vectorB



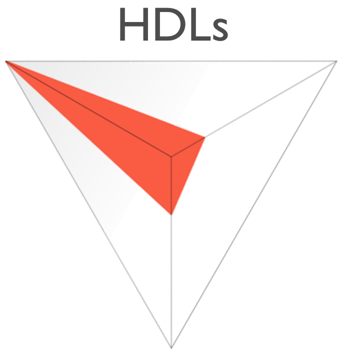
- Improves memory bandwidth ⬇ Runtime
- May duplicate memory resources ⬆ Area



Criteria for Improved HLS

Requirement	C+Pragmas
Represent memory hierarchy explicitly Aids on-chip memory optimization, specialization	✗
Express control as nested loops Enables analysis of access patterns	✓
Support arbitrarily nested pipelining Exploits nested parallelism	✗
Specialize memory transfers Enables customized memory controllers based on access patterns	✗
Capture design parameters Enables automatic design tuning in compiler	✗

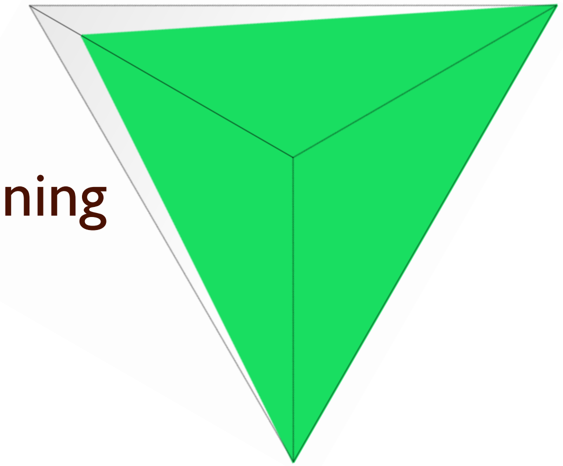
Rethinking HLS



Improved HLS

Performance

- ✓ Memory hierarchy
- ✓ Arbitrary pipelining

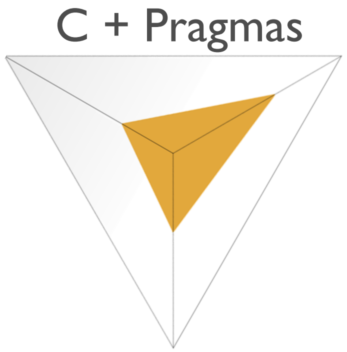


Productivity

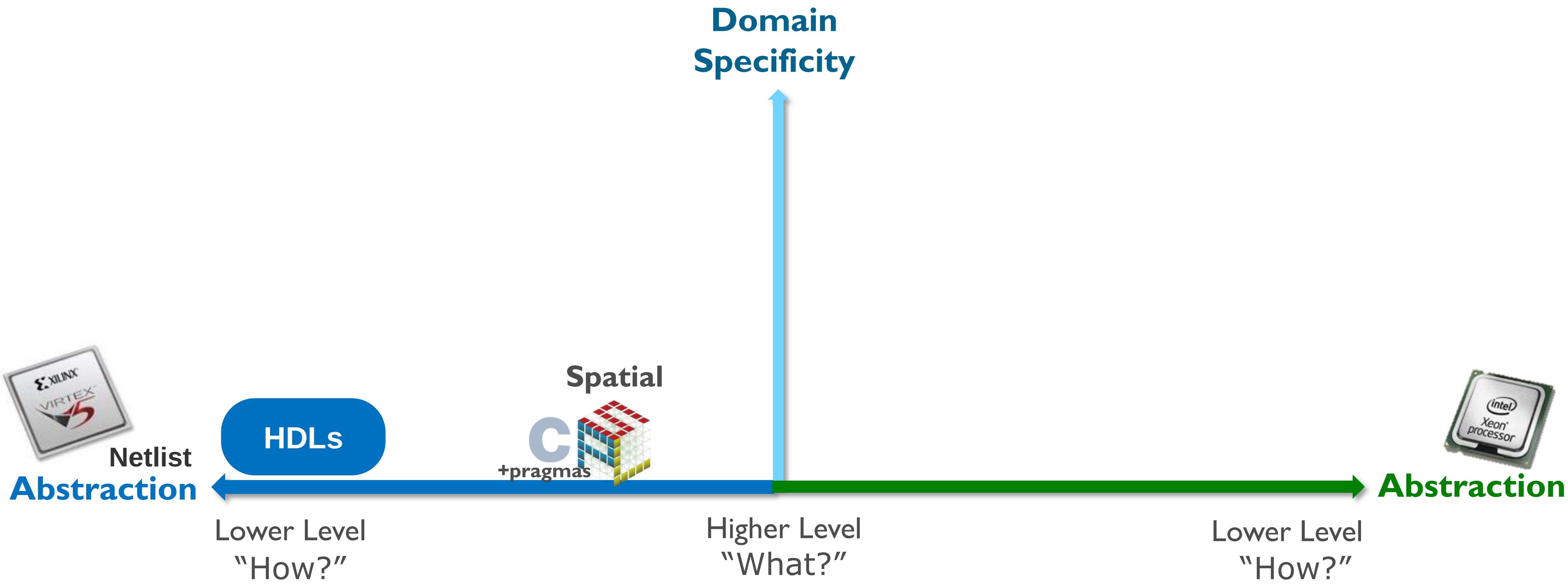
- ✓ Nested loops
- ✓ Automatic memory banking/buffering
- ✓ Implicit design parameters (unrolling, banking, etc.)
- ✓ Automated design tuning

Portability

- ✓ Target-generic source across reconfigurable architectures



Abstracting Hardware Design



Domain Specificity

Spatial

HDLs

C +pragmas

Netlist

Abstraction

Abstraction

Lower Level
"How?"

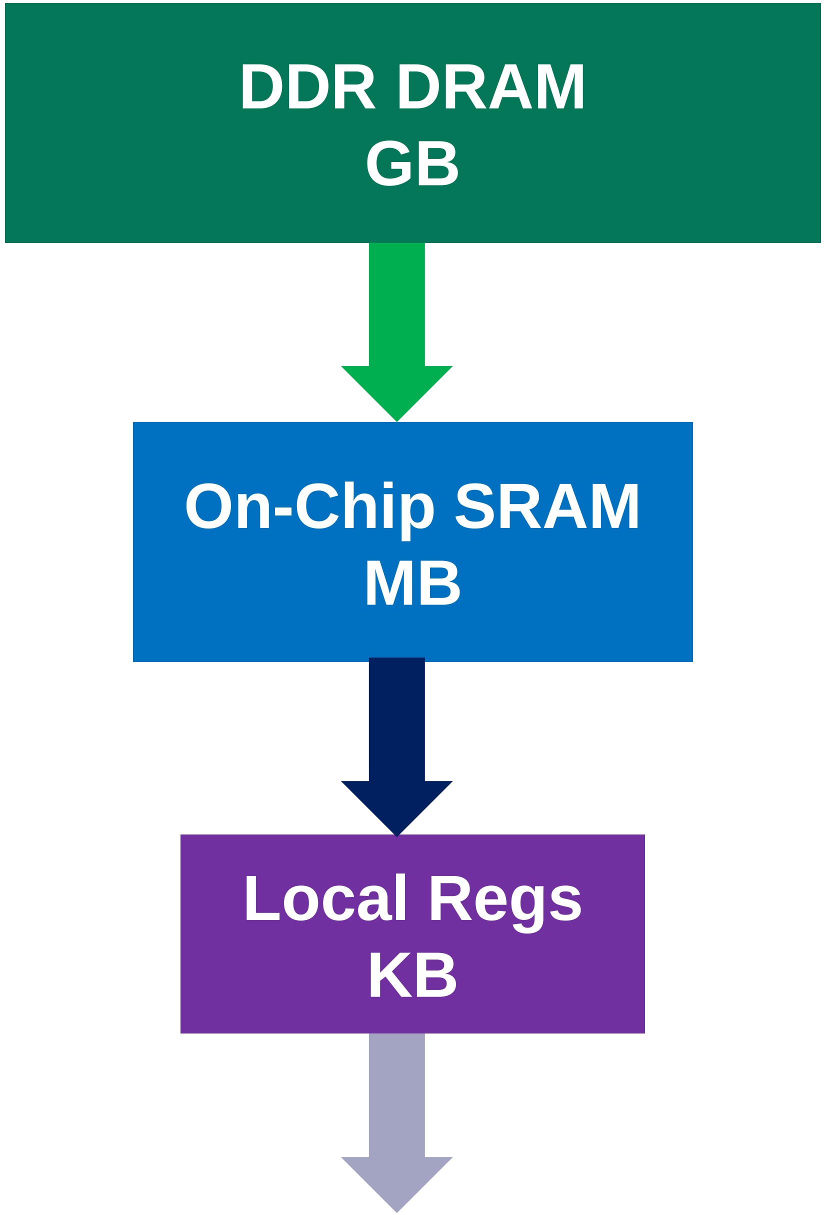
Higher Level
"What?"

Lower Level
"How?"

Reconfigurable Architectures (FPGAs)

Instruction-Based Architectures (CPUs)

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

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 =>
  …
}
```

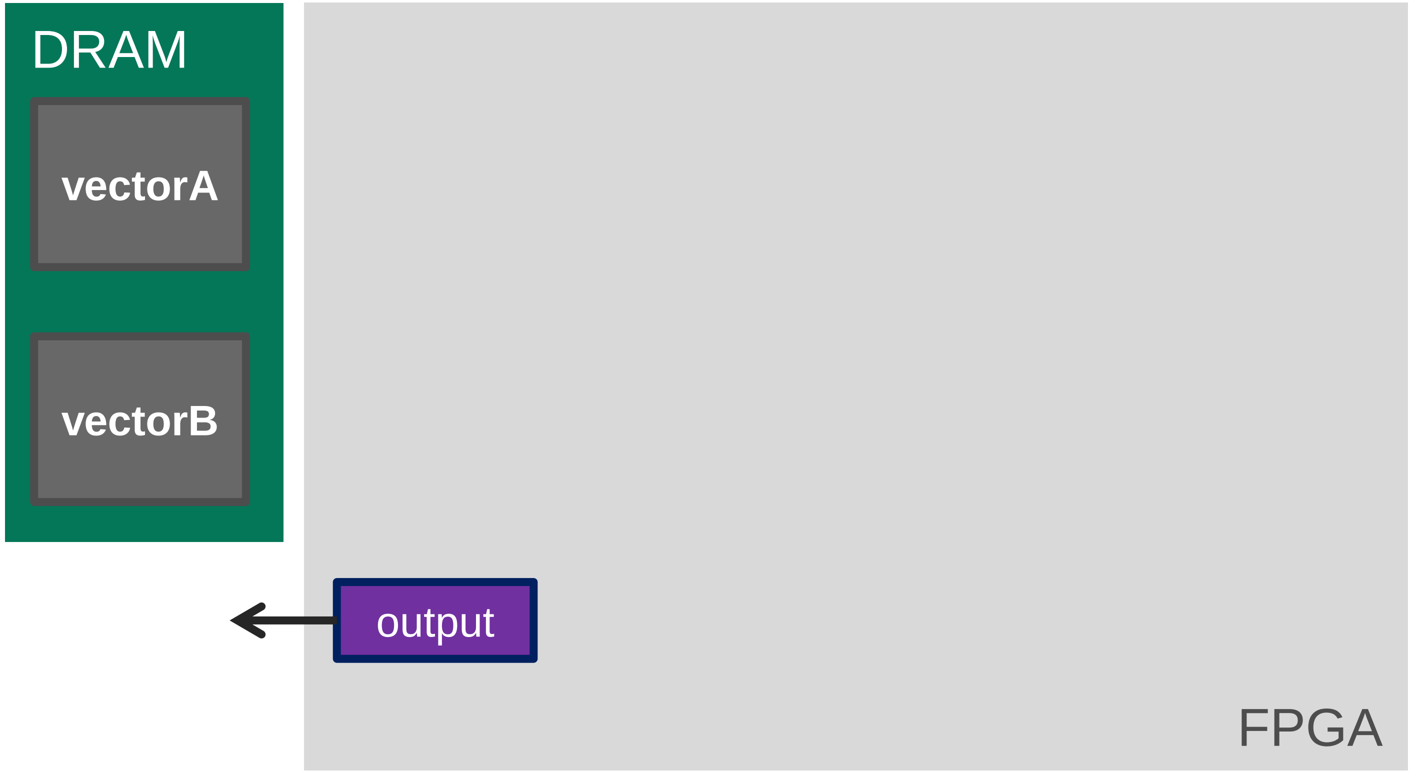
Implicit memory banking and buffering schemes for parallelized access

```
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)
```

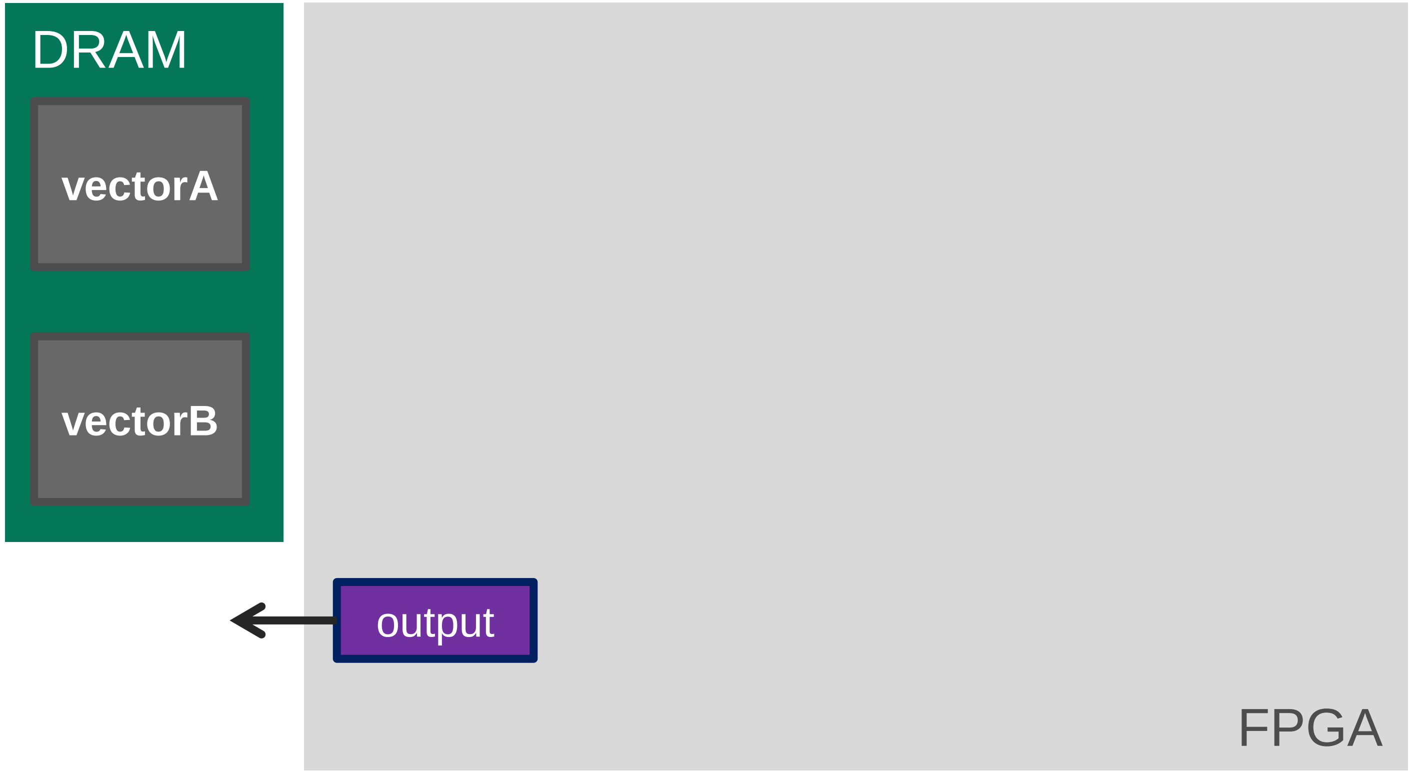
Off-chip memory declarations



Dot Product in Spatial

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

Accel {



}
```

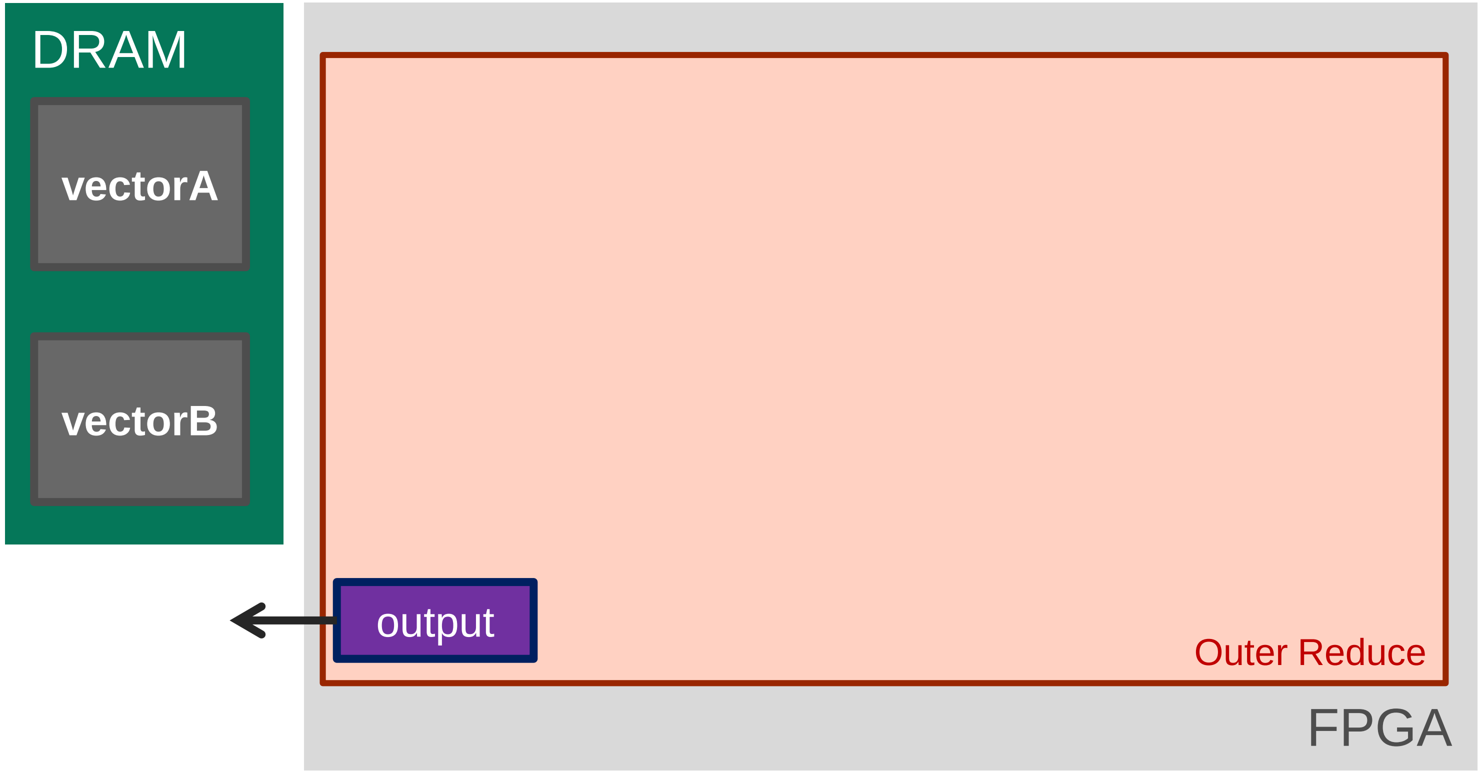
Explicit work division in IR



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 =>







}
```

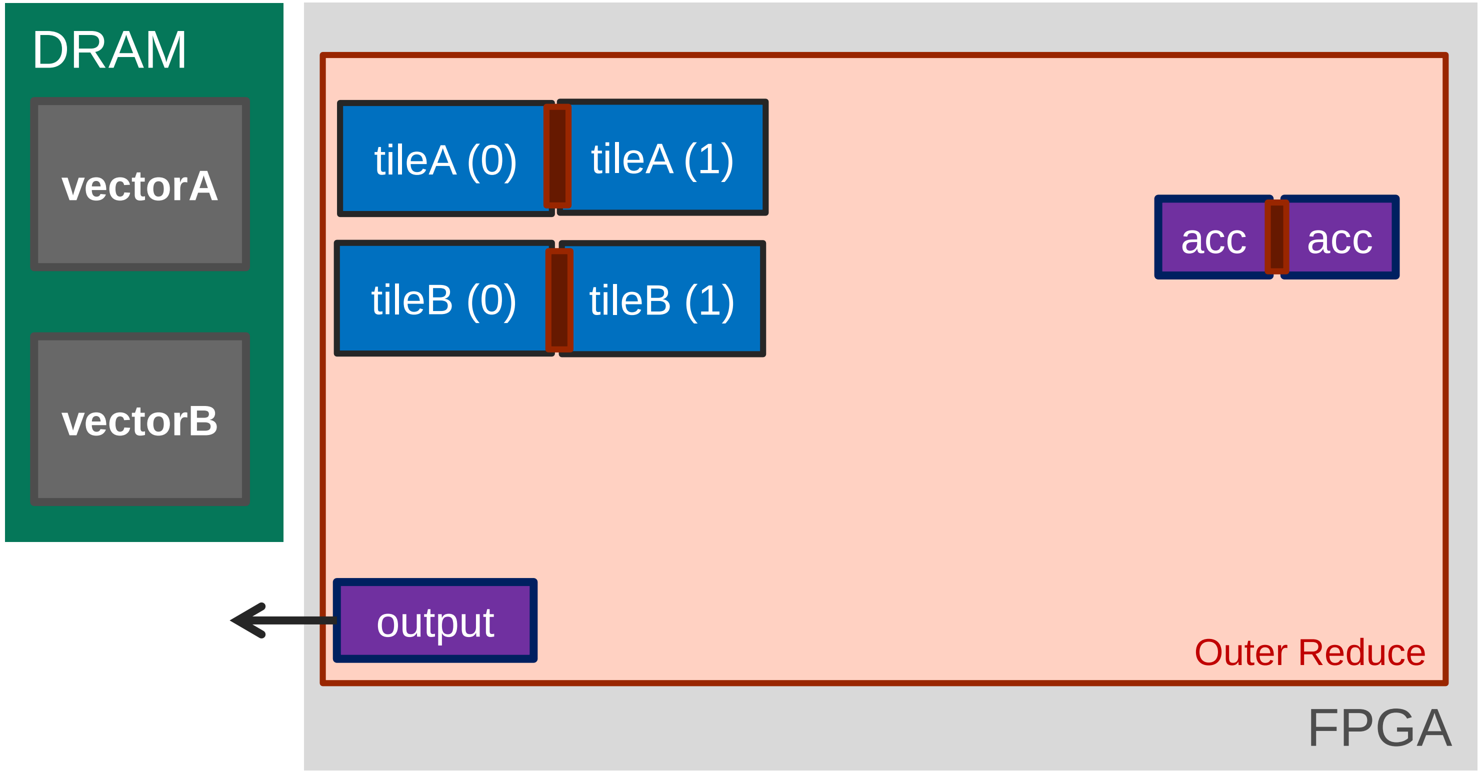
Tiled reduction (outer)



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]









}
```

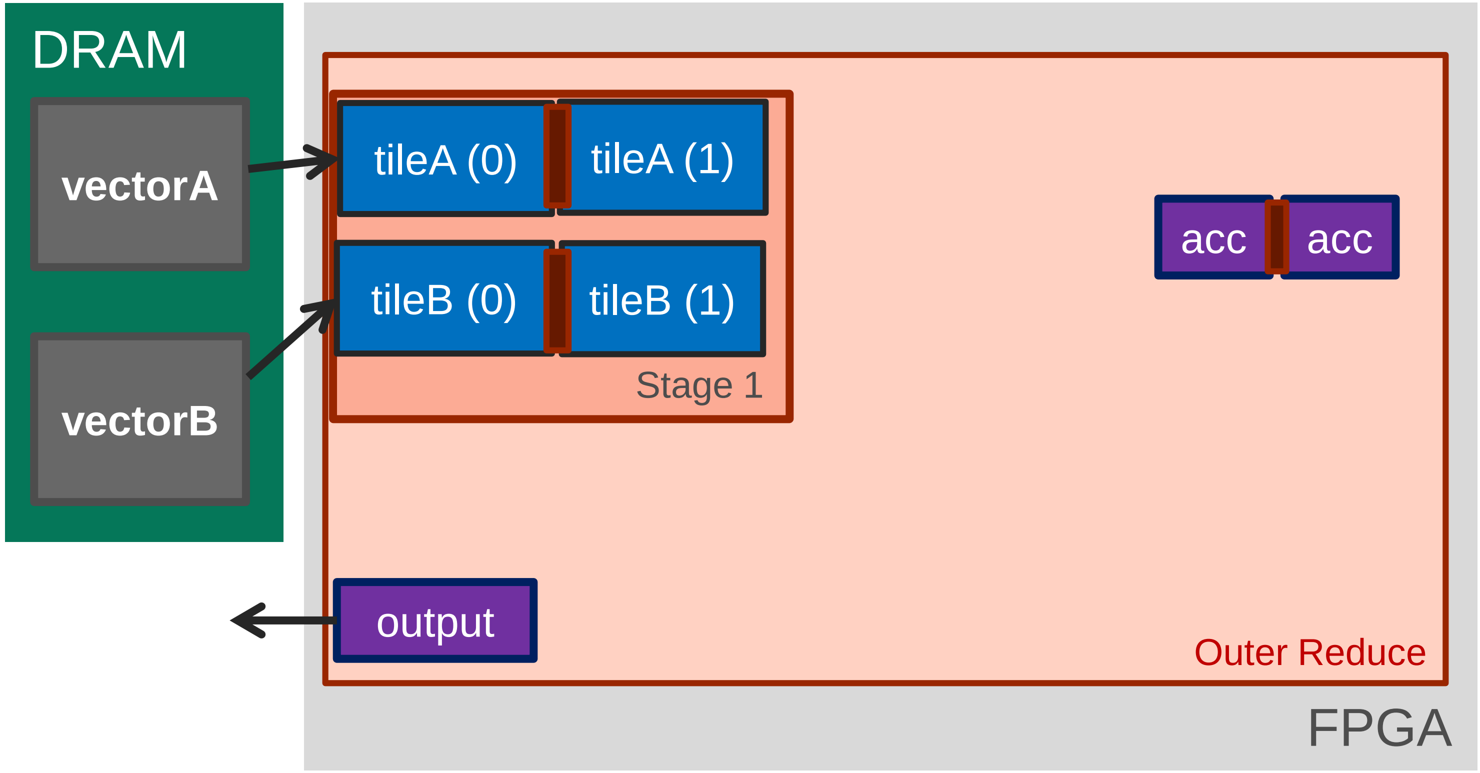
On-chip memory declarations



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)





}
```

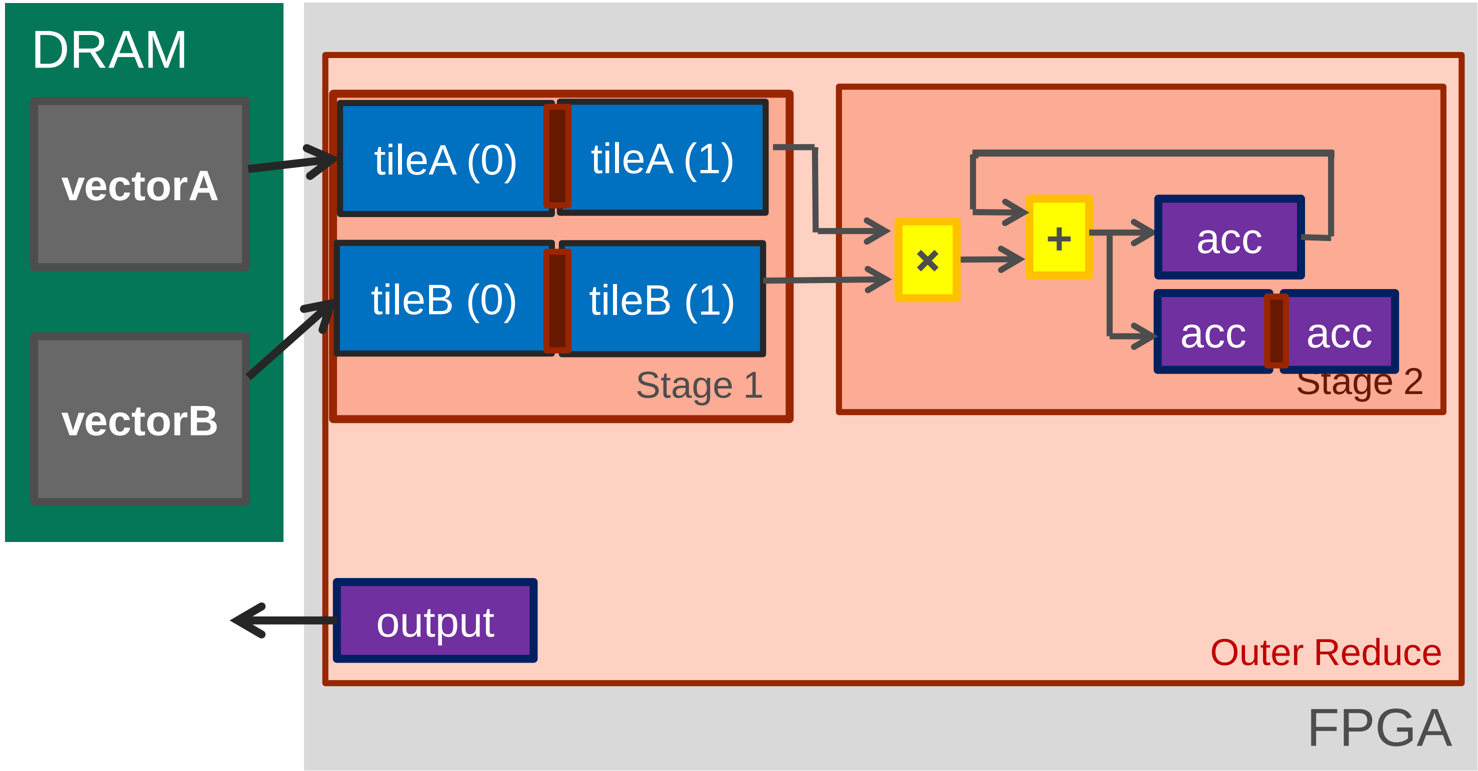
DRAM → SRAM transfers
(also have **store**, **scatter**, and **gather**)



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}

}
```

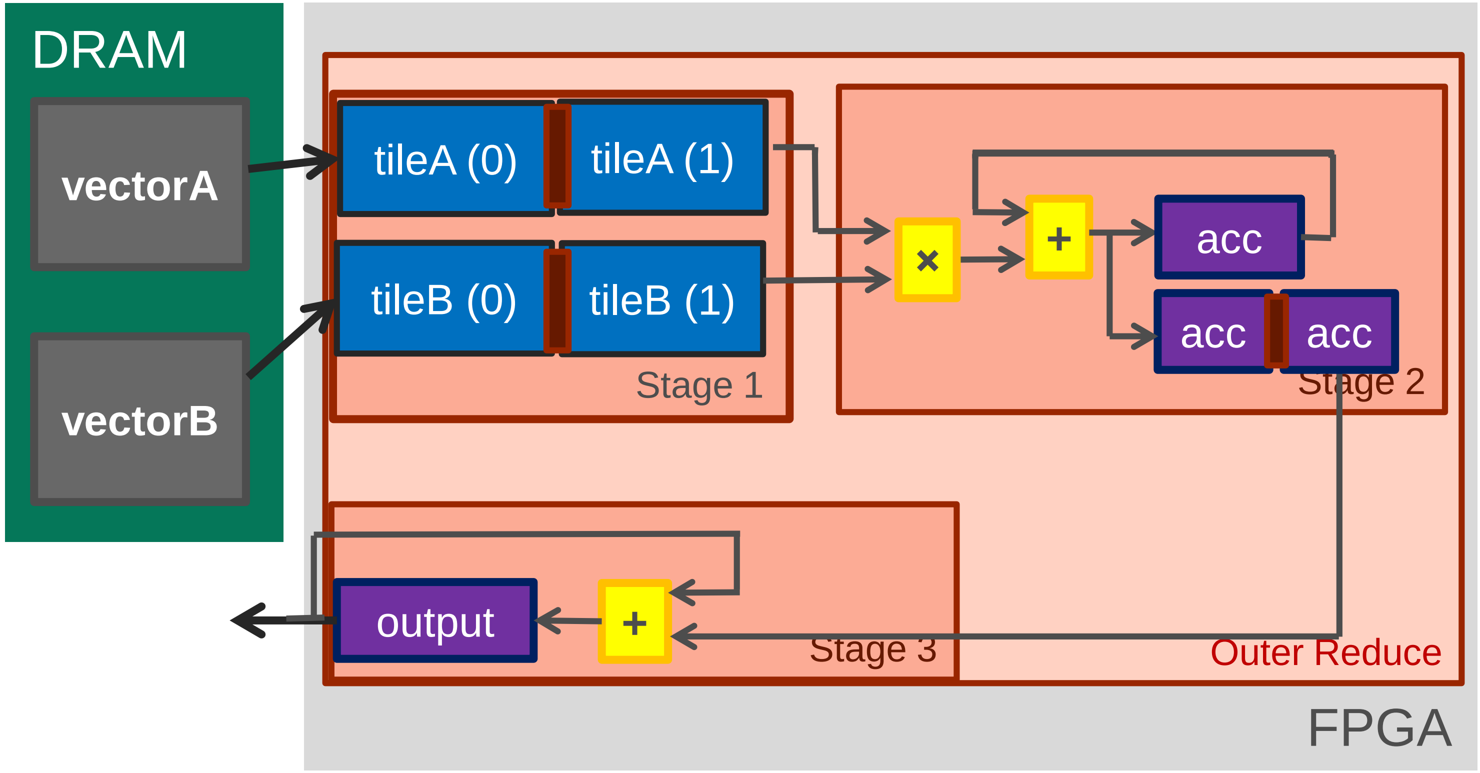
Tiled reduction (pipelined)



Dot Product in Spatial

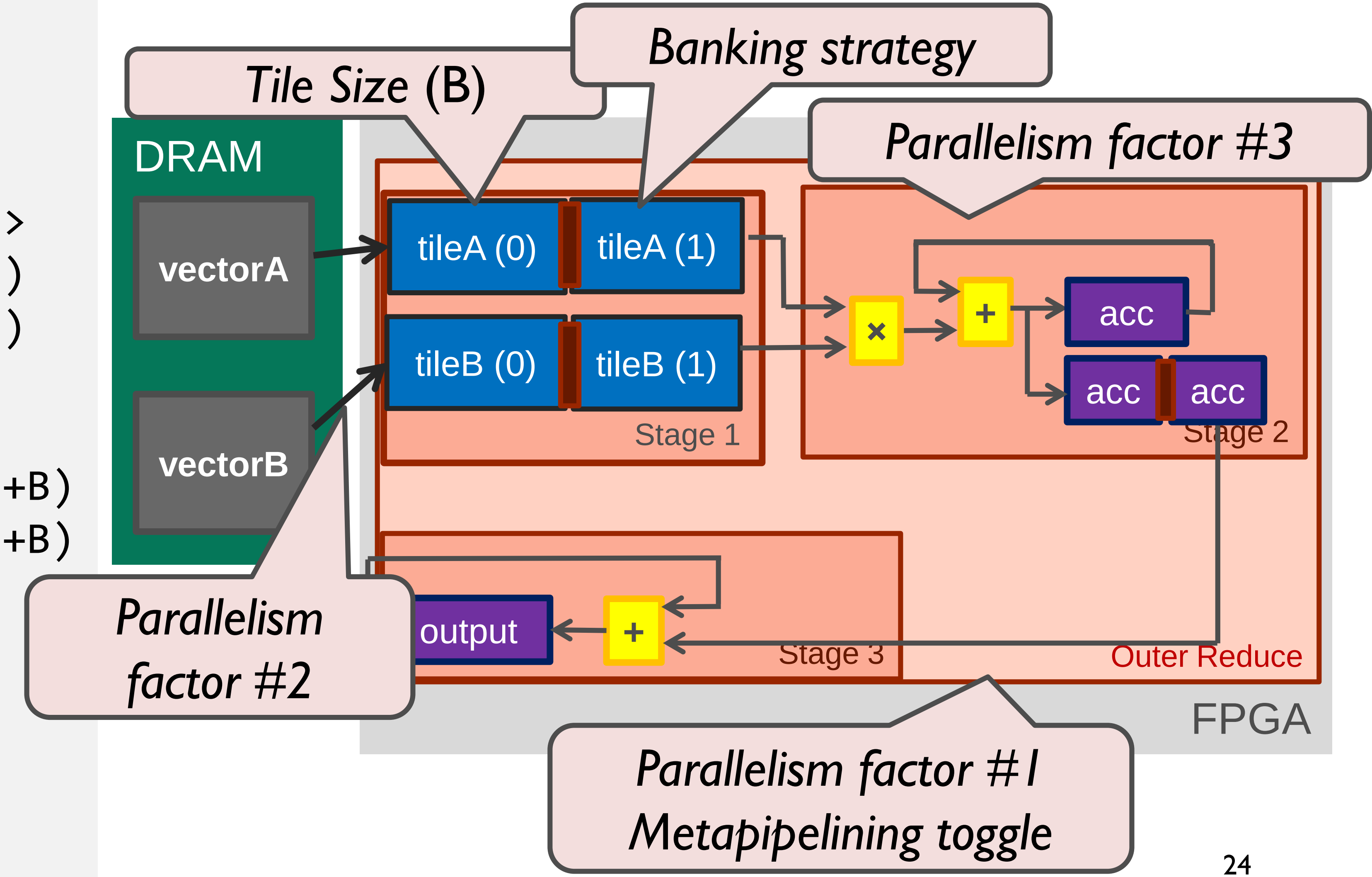
```
val output  = ArgOut[Float]
val vectorA = DRAM[Float](N)
val vectorB = DRAM[Float](N)

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}
}
```

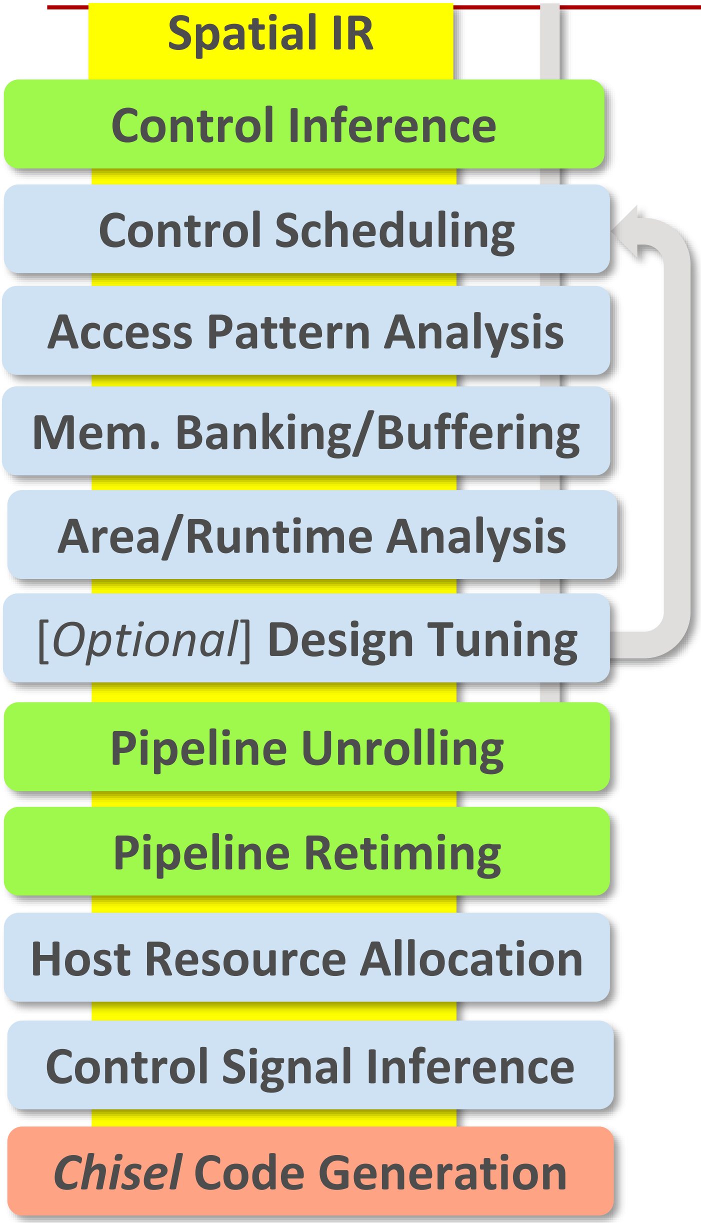


Dot Product in Spatial

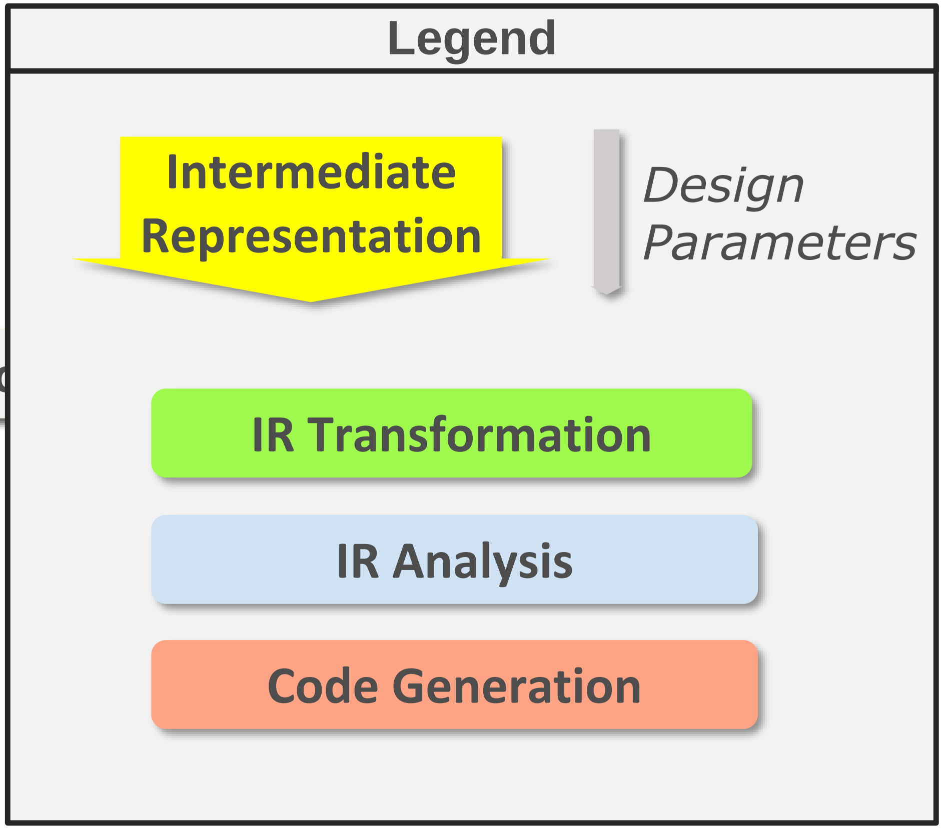
```
val output  = ArgOut[Float]
val vectorA = DRAM[Float](N)
val vectorB = DRAM[Float](N)

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}
}
```

Parameters

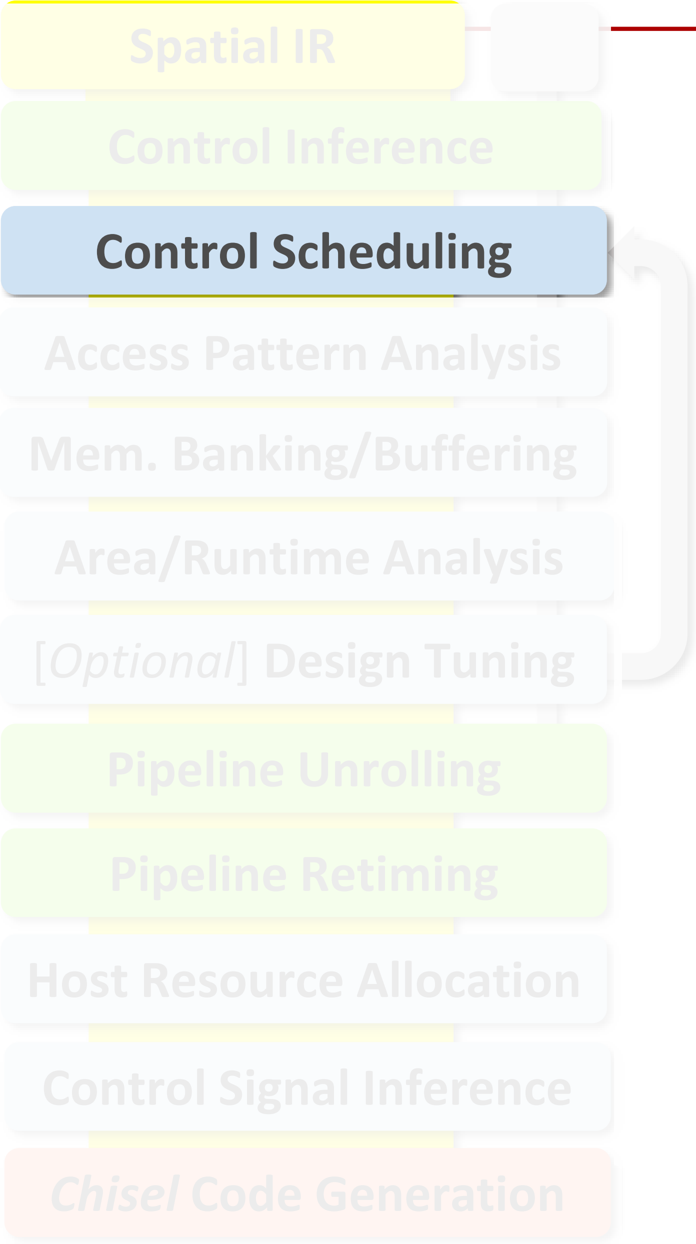
The Spatial Compiler



Spatial Pro



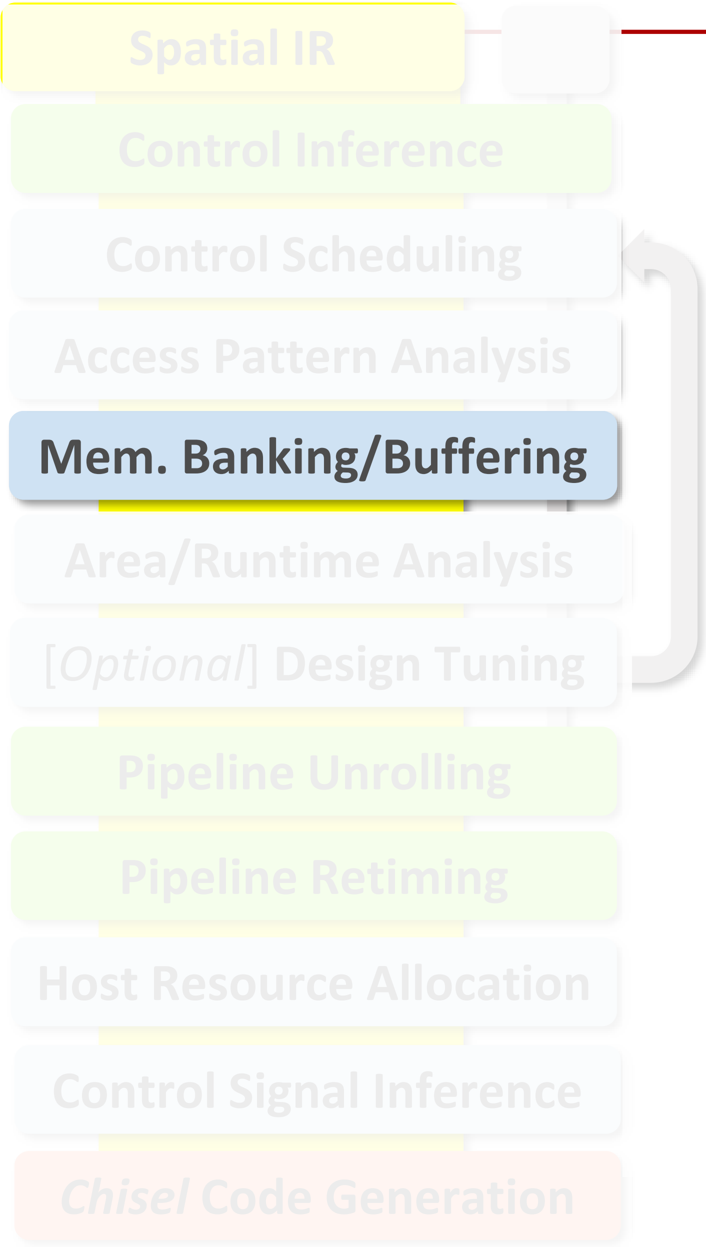
Control Scheduling



- 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)

Local Memory Analysis



Spatial IR

Control Inference

Control Scheduling

Access Pattern Analysis

Mem. Banking/Buffering

Area/Runtime Analysis

[*Optional*] Design Tuning

Pipeline Unrolling

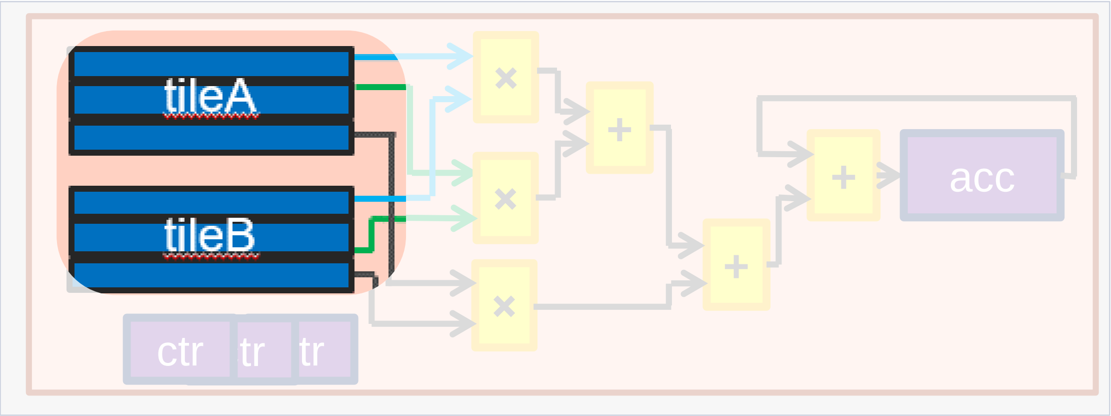
Pipeline Retiming

Host Resource Allocation

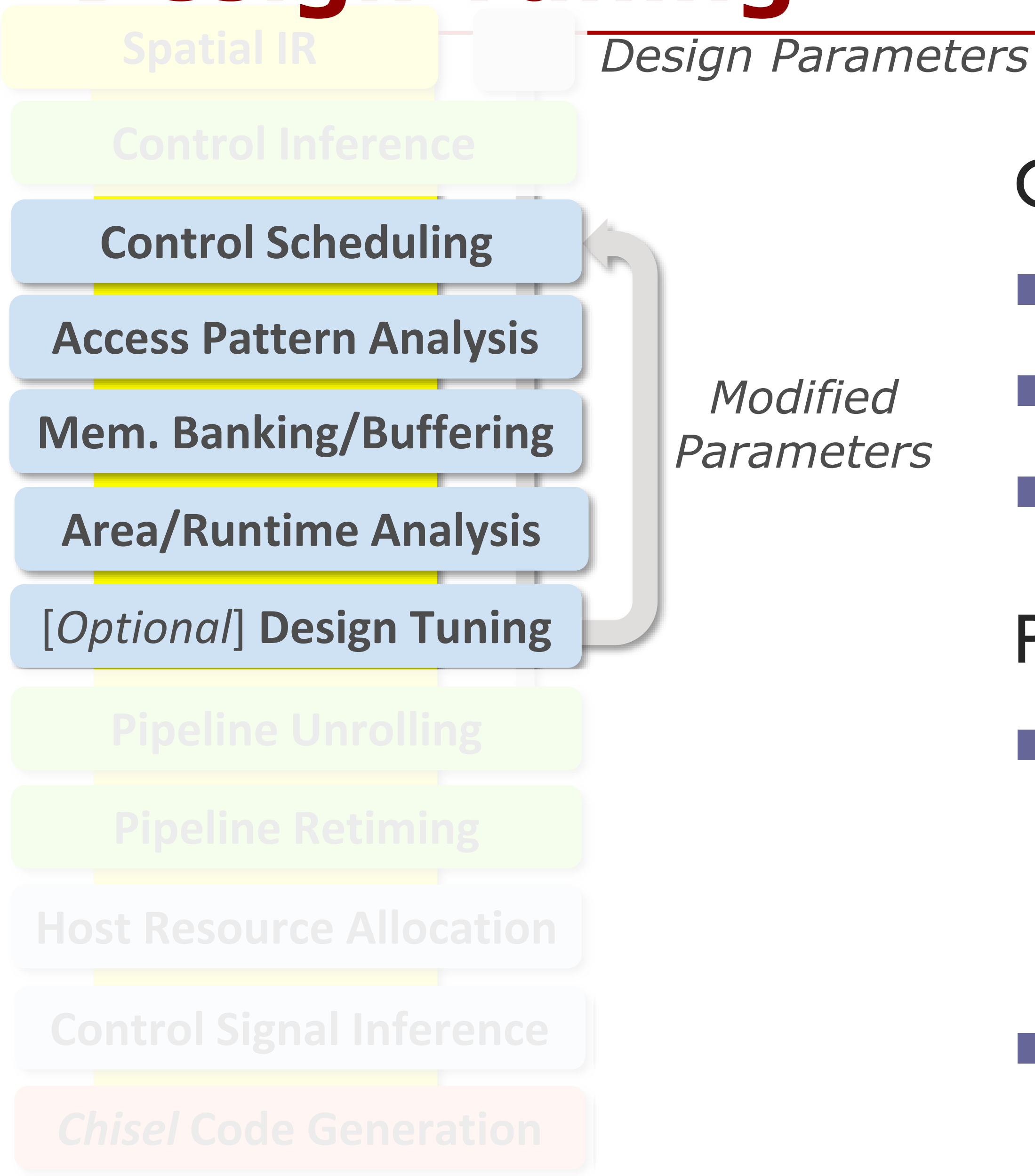
Control Signal Inference

Chisel Code Generation

- Insight: determine banking strategy **in a single loop** nest using **the polyhedral model** [Wang, Li, Cong *FPGA '14*]
- Spatial's contribution: find the (near) optimal banking/buffering strategy **across all loop nests**
- Algorithm in a nutshell:
  1. Bank each reader as a separate coherent copy (accounting for reaching writes)
  2. Greedily merge copies if merging is legal and cheaper



Design Tuning



Original tuning methods:

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

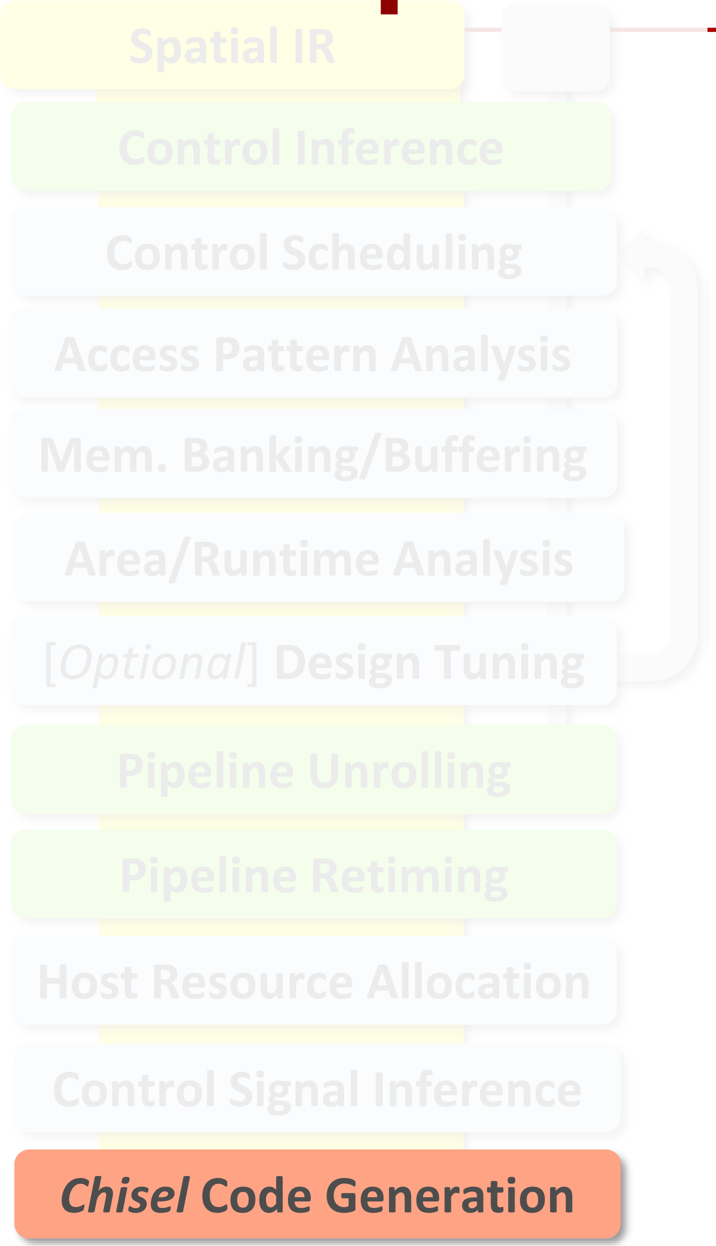
Proposed tuning method

- Reinforcement learning: HyperMapper

(More details in paper)

- **Fast**: No slow transformers in loop

The Spatial Compiler: The Rest



Spatial IR

Control Inference

Control Scheduling

Access Pattern Analysis

Mem. Banking/Buffering

Area/Runtime Analysis

[*Optional*] Design Tuning

Pipeline Unrolling

Pipeline Retiming

Host Resource Allocation

Control Signal Inference

Chisel Code Generation

Code generation

- Synthesizable Chisel
- C++ code for host CPU

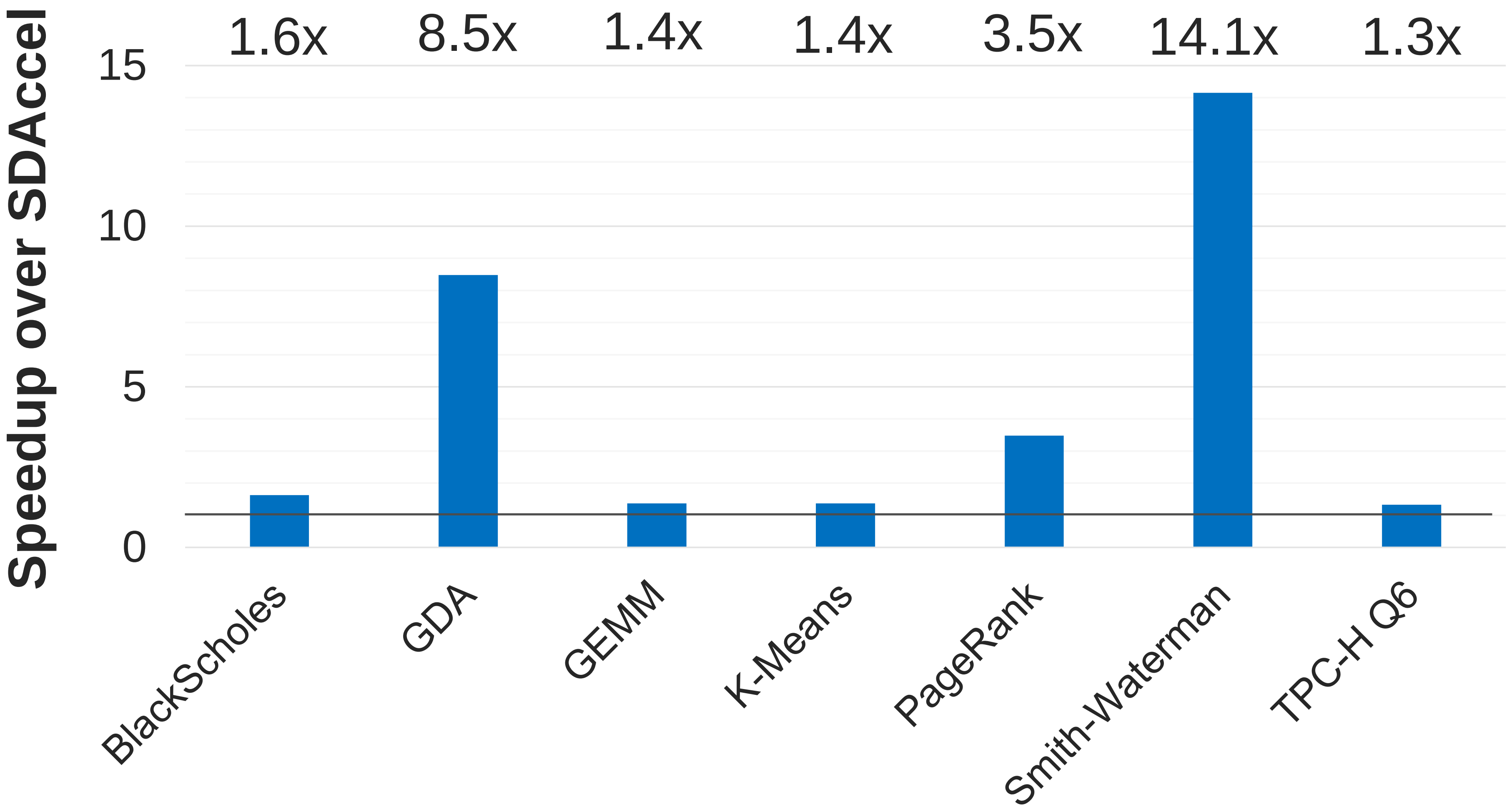
Evaluation: Performance

- 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***

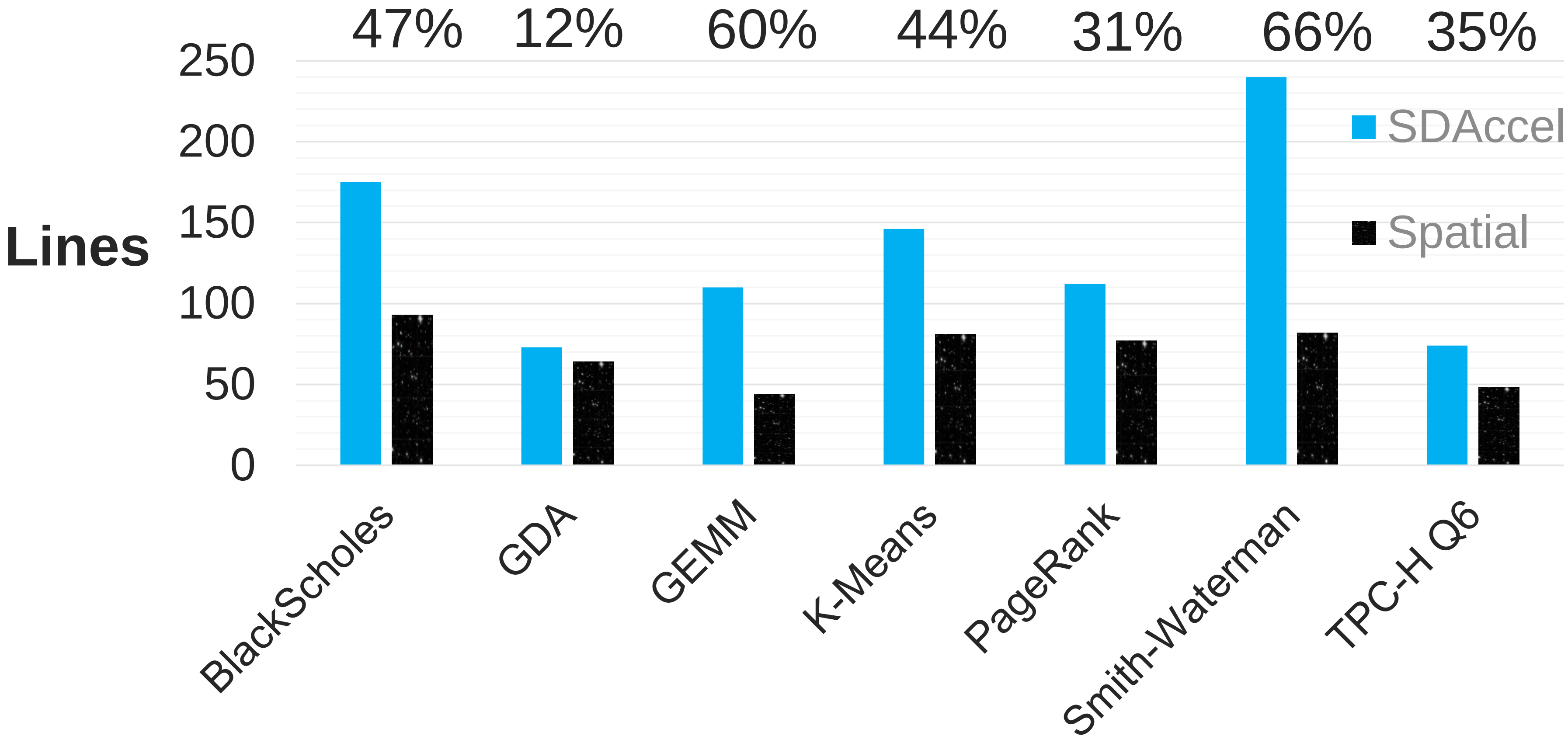
Performance (Spatial vs. SDAccel)

Average **2.9x faster hardware** than SDAccel



Productivity: Lines of Code

Average **42% shorter** programs versus SDAccel

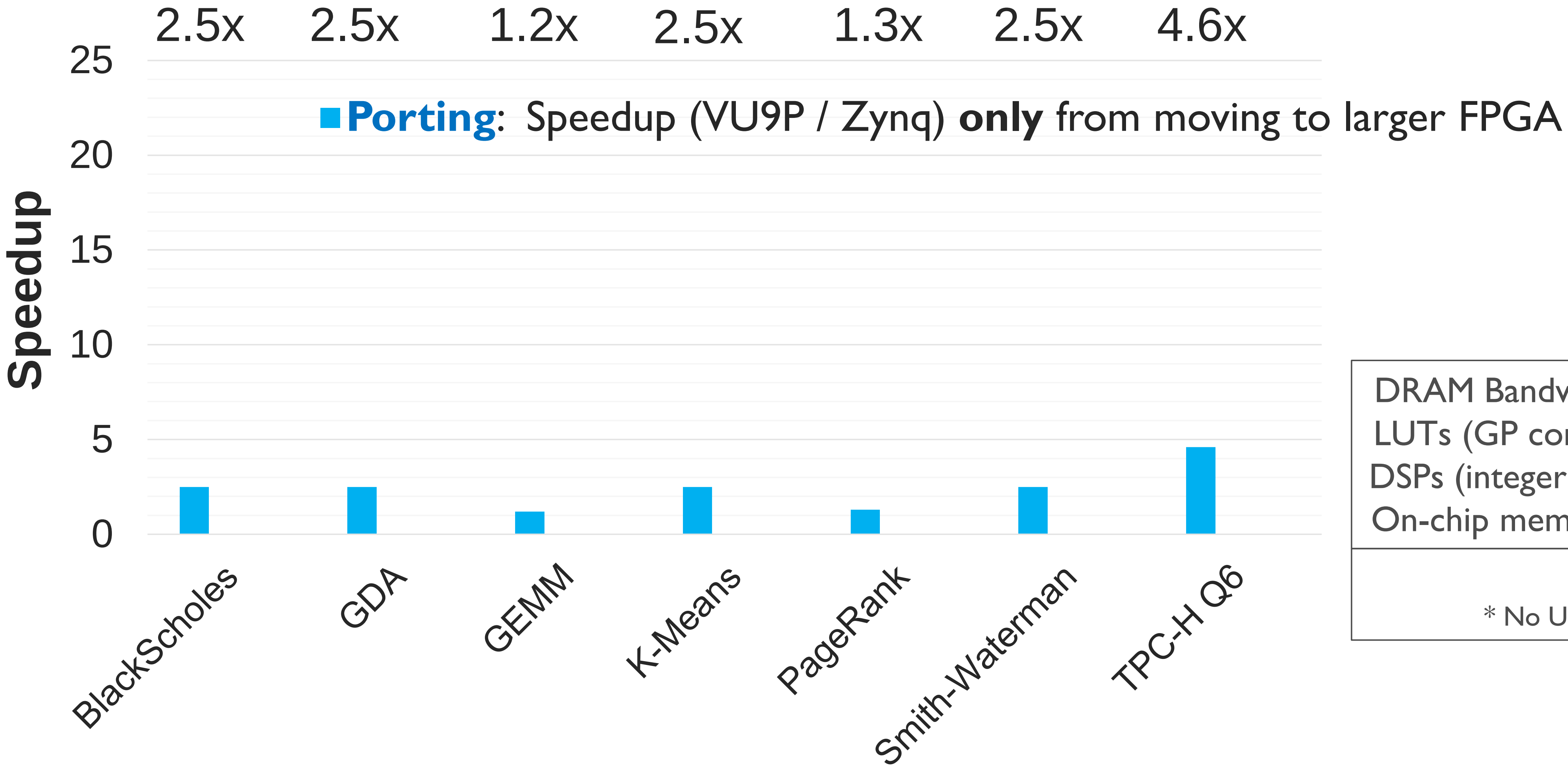


Evaluation: Portability

- 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



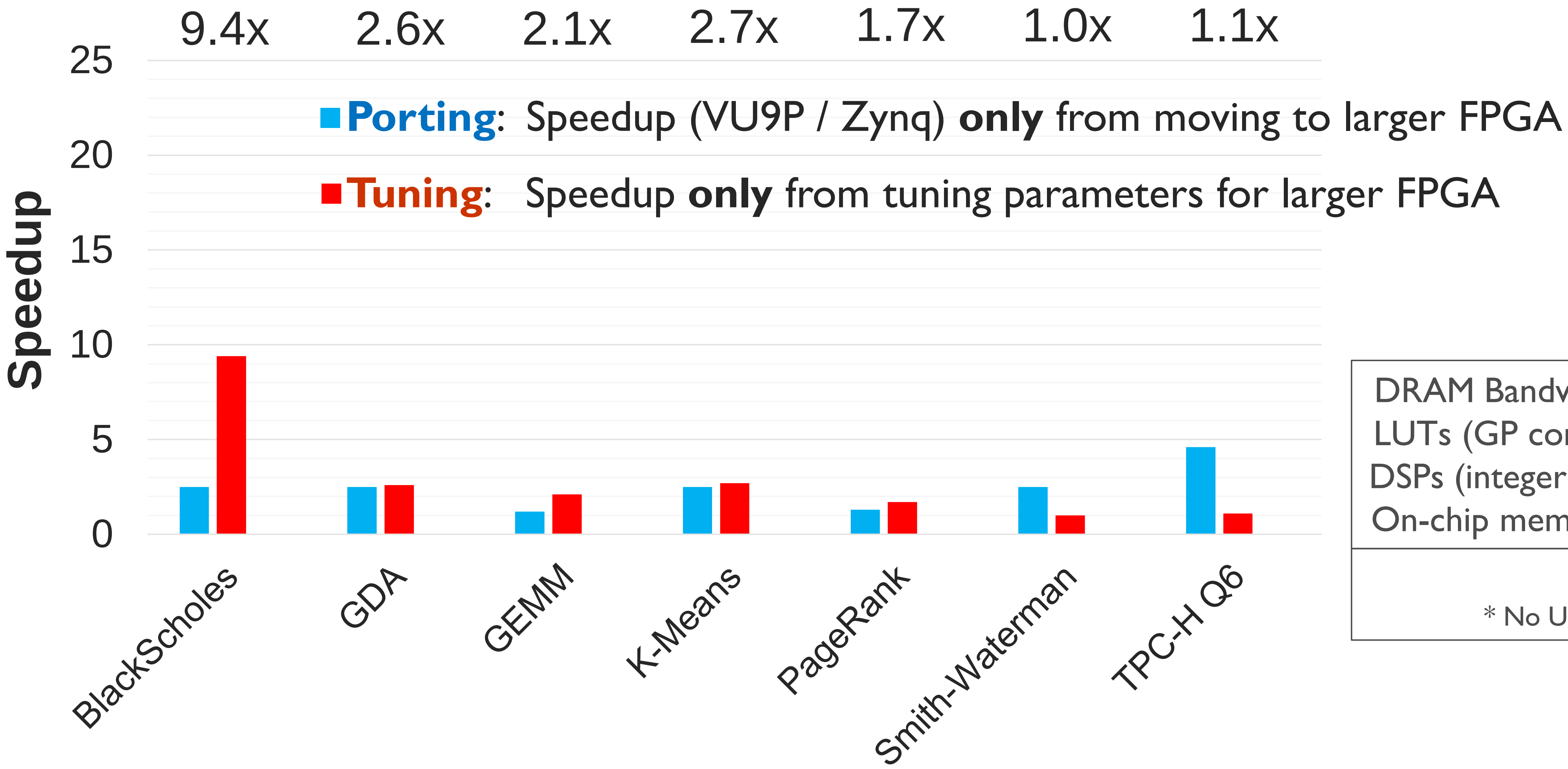
DRAM Bandwidth:	**4.5x**
LUTs (GP compute):	**47.3x**
DSPs (integer FMA):	**7.6x**
On-chip memory*:	**4.0x**

VU9P / ZC706

* No URAM used on VU9P

Portability: VU9P vs. Zynq ZC706

Identical Spatial source, multiple targets



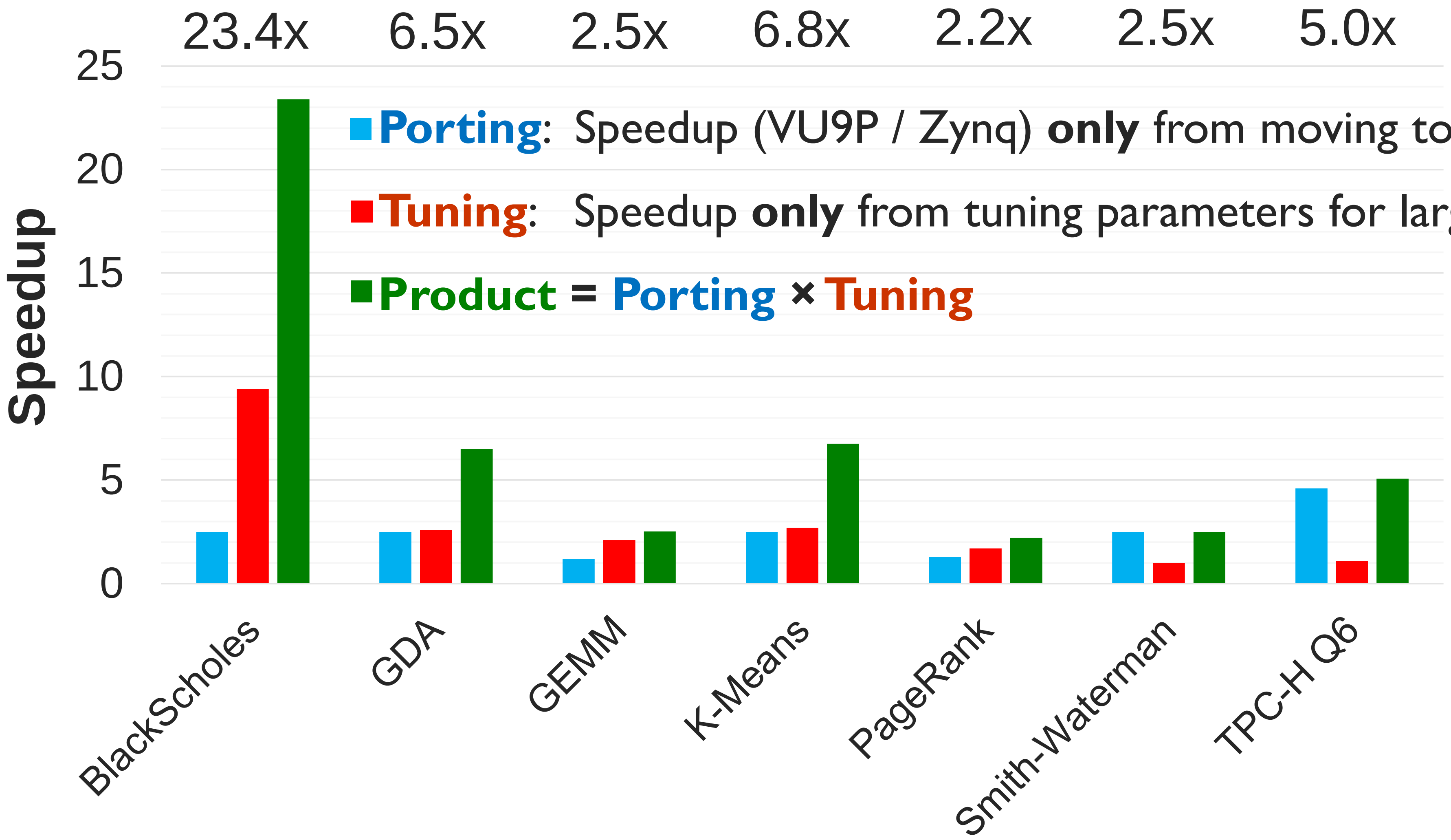
DRAM Bandwidth:	**4.5x**
LUTs (GP compute):	**47.3x**
DSPs (integer FMA):	**7.6x**
On-chip memory*:	**4.0x**

VU9P / ZC706

* No URAM used on VU9P

Portability: VU9P vs. Zynq ZC706

Identical Spatial source, multiple targets



🟦 **Porting**: Speedup (VU9P / Zynq) **only** from moving to larger FPGA

🟥 **Tuning**: Speedup **only** from tuning parameters for larger FPGA

🟩 **Product** = **Porting** × **Tuning**

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

VU9P / ZC706

* No URAM used on VU9P

Portability: Plasticine CGRA

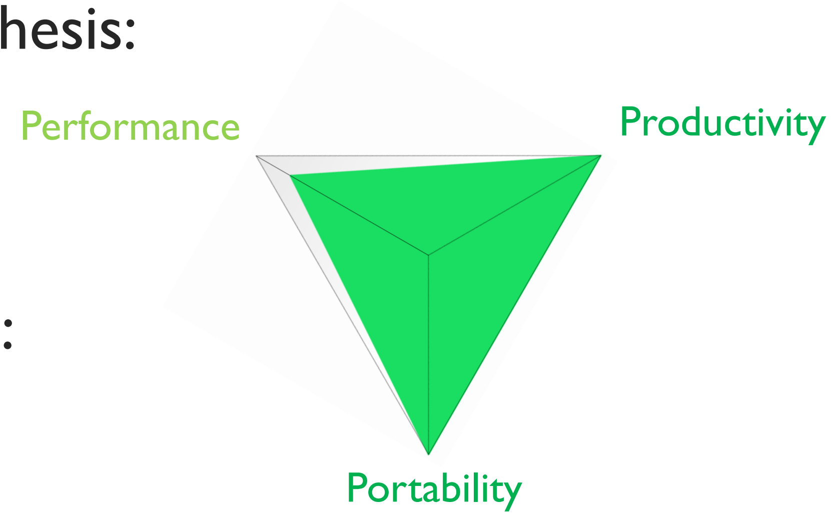
Identical Spatial source, multiple targets
Even reconfigurable hardware that isn't an FPGA!

	DRAM Bandwidth (%)		Resource Utilization (%)			Speedup
Benchmark	**Load**	**Store**	**PCU**	**PMU**	**AG**	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)

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)



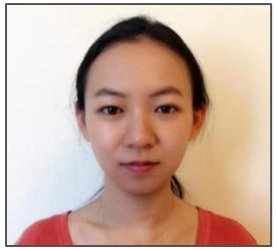
Spatial is open source: spatial.stanford.edu

The Team

David
Koeplinger

Matt
Feldman

Raghu
Prabhakar



Yaqi
Zhang

Stefan
Hadjis

Ruben
Fiszel

Tian
Zhao

Ardavan
Pedram

Luigi
Nardi

Christos
Kozyrakis

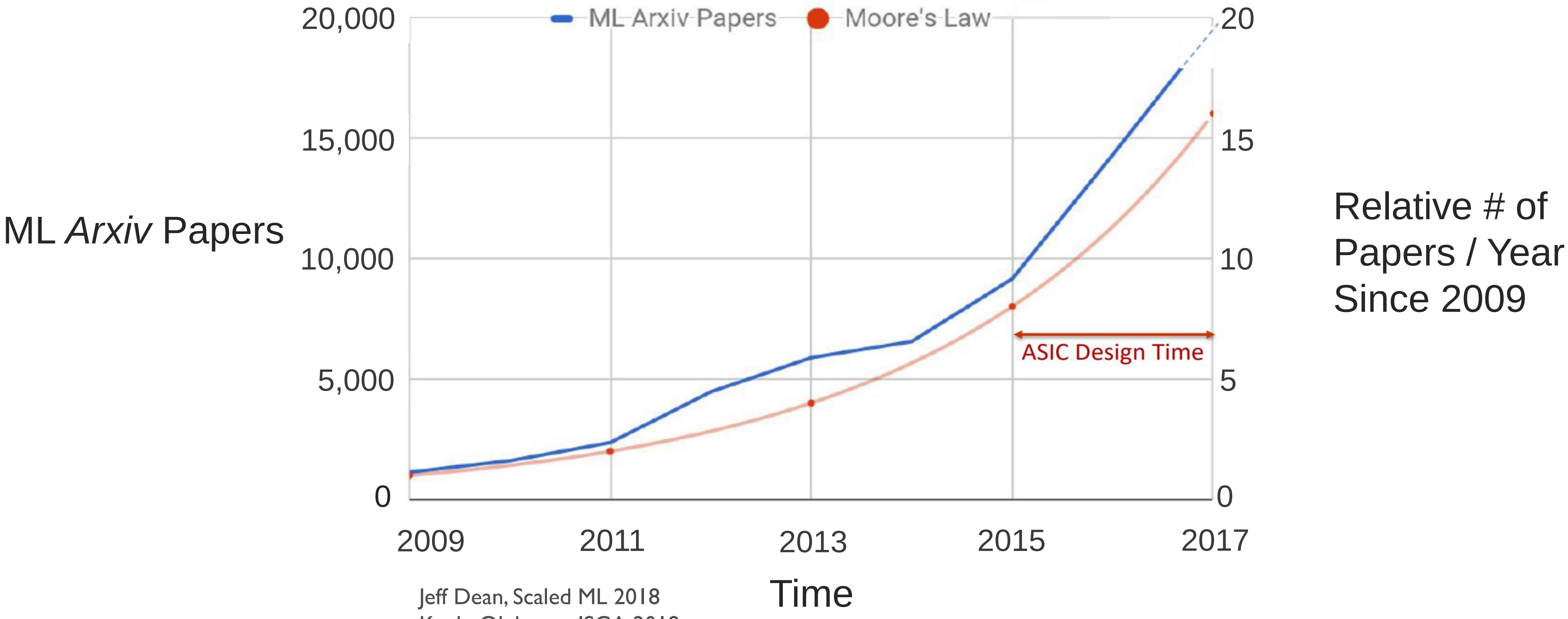
Kunle
Olukotun

Backup Slides

Custom ASICs

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

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



Jeff Dean, Scaled ML 2018
Kunle Olukotun, ISCA 2018

C + Pragmas Example

Add 512 integers originating from accelerator DRAM

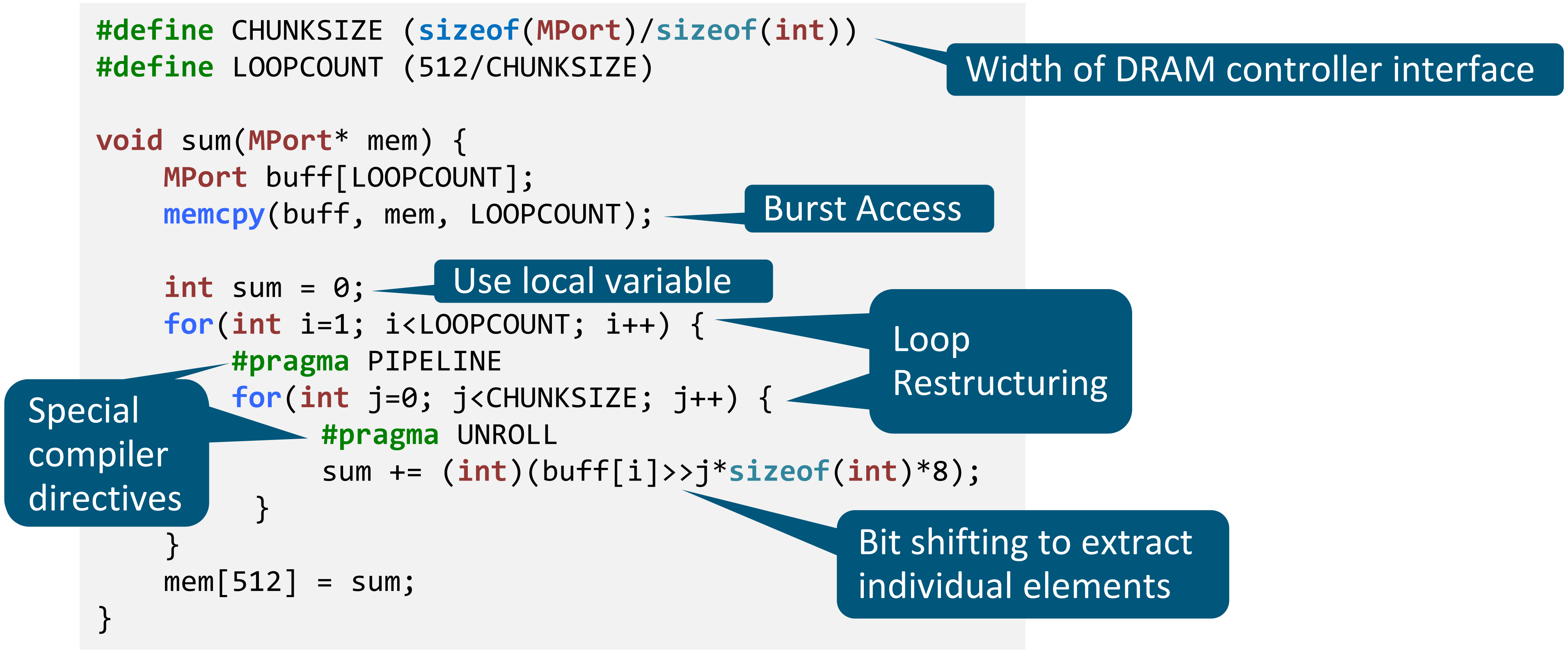
```
void sum(int* mem) {

    mem[512] = 0;

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

}
```

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;
}
```

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

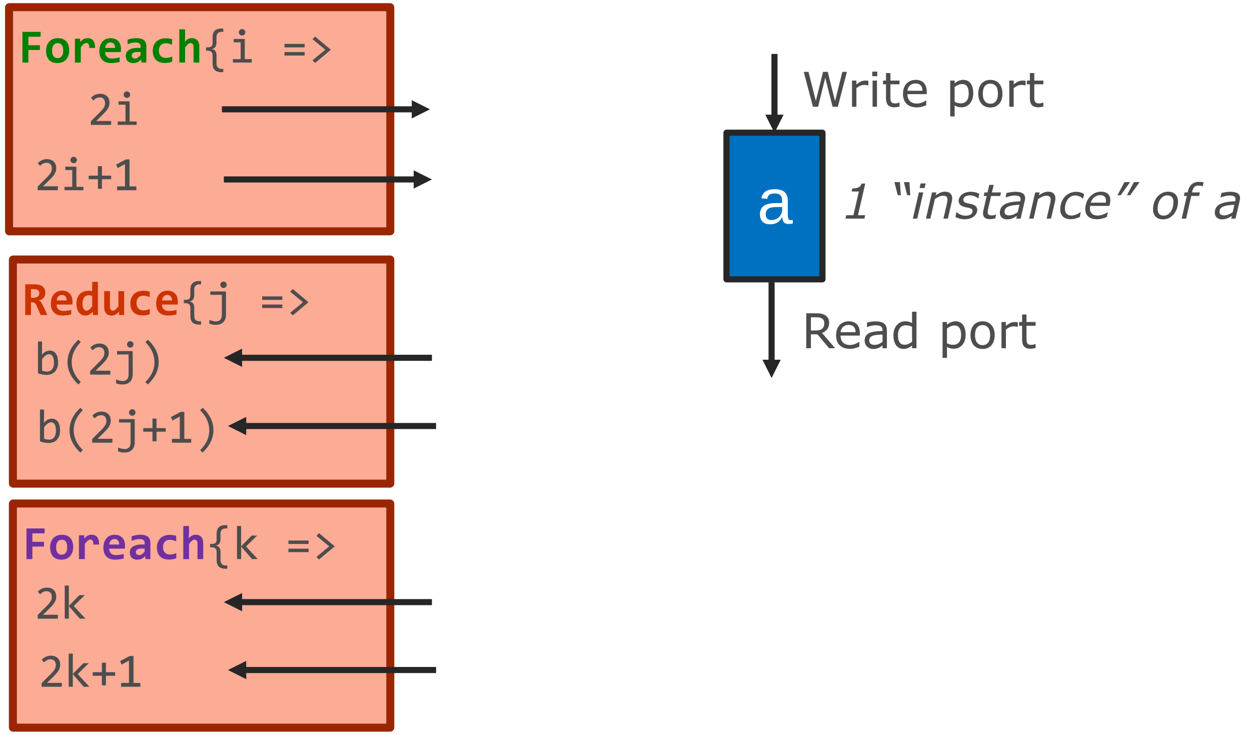
1. Finite physical compute and memory resources
2. Requires aggressive pipelining for performance
   - Maximize useful execution time of compute resources
3. Disjoint memory space
   - No hardware managed memory hierarchy
4. Huge design parameter spaces
   - Parameters are interdependent, change runtime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

Step 1: For each read:

Find the **banking** and **buffering** for that read and all writes that may be visible to that read

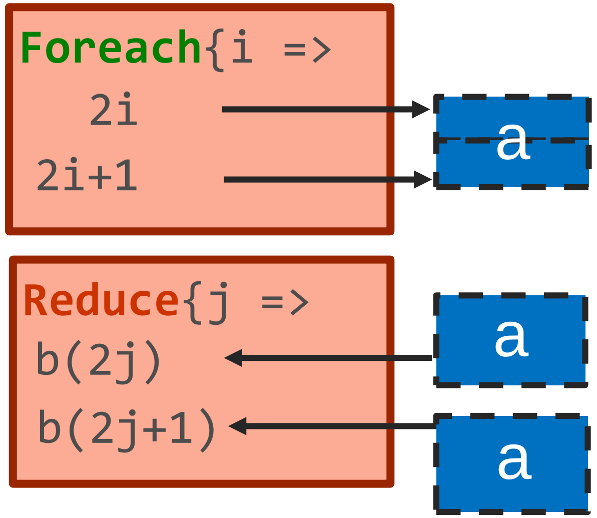


Local Memory Analysis Example (Cont.)

```
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

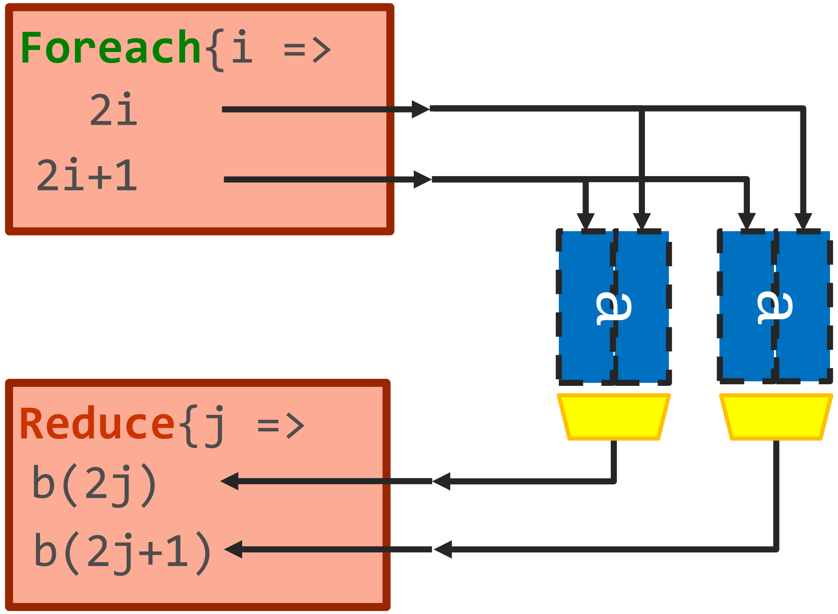


Local Memory Analysis Example (Cont.)

```
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
  }
}
```

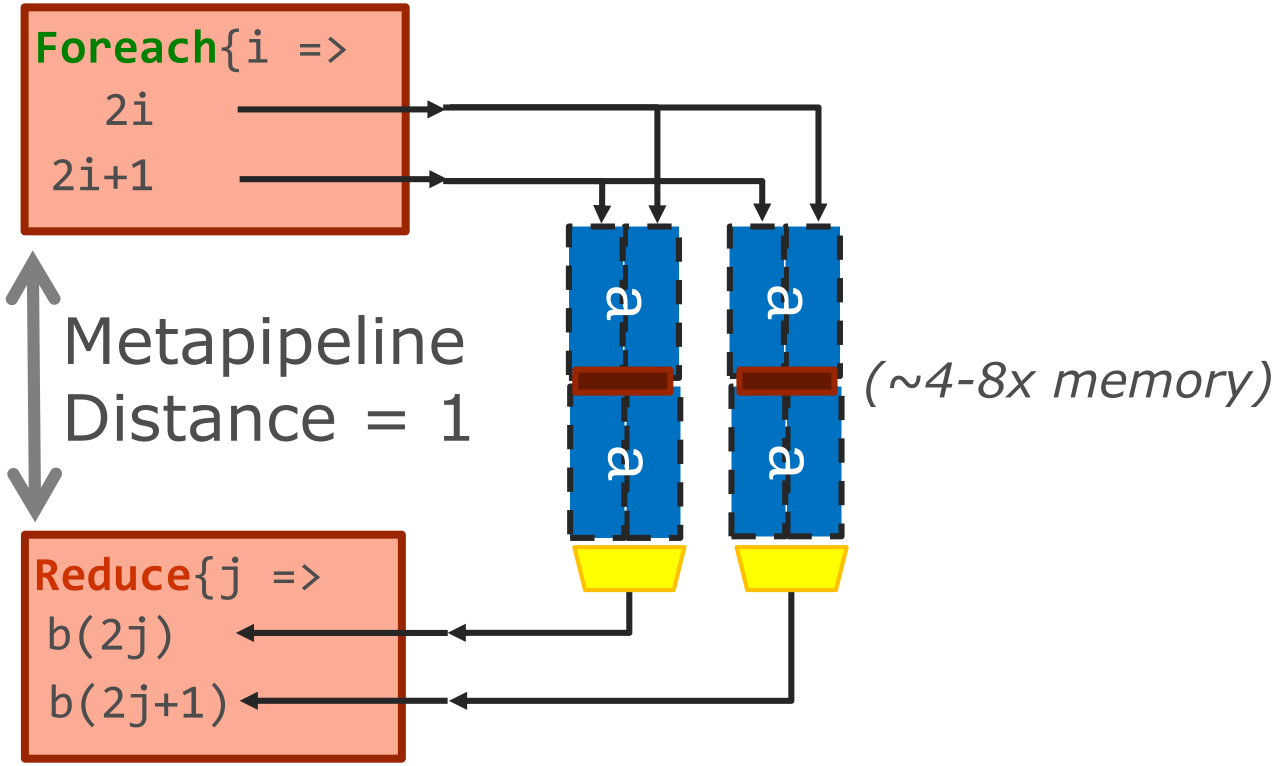
Step 1: For each read:

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
  }
}
```

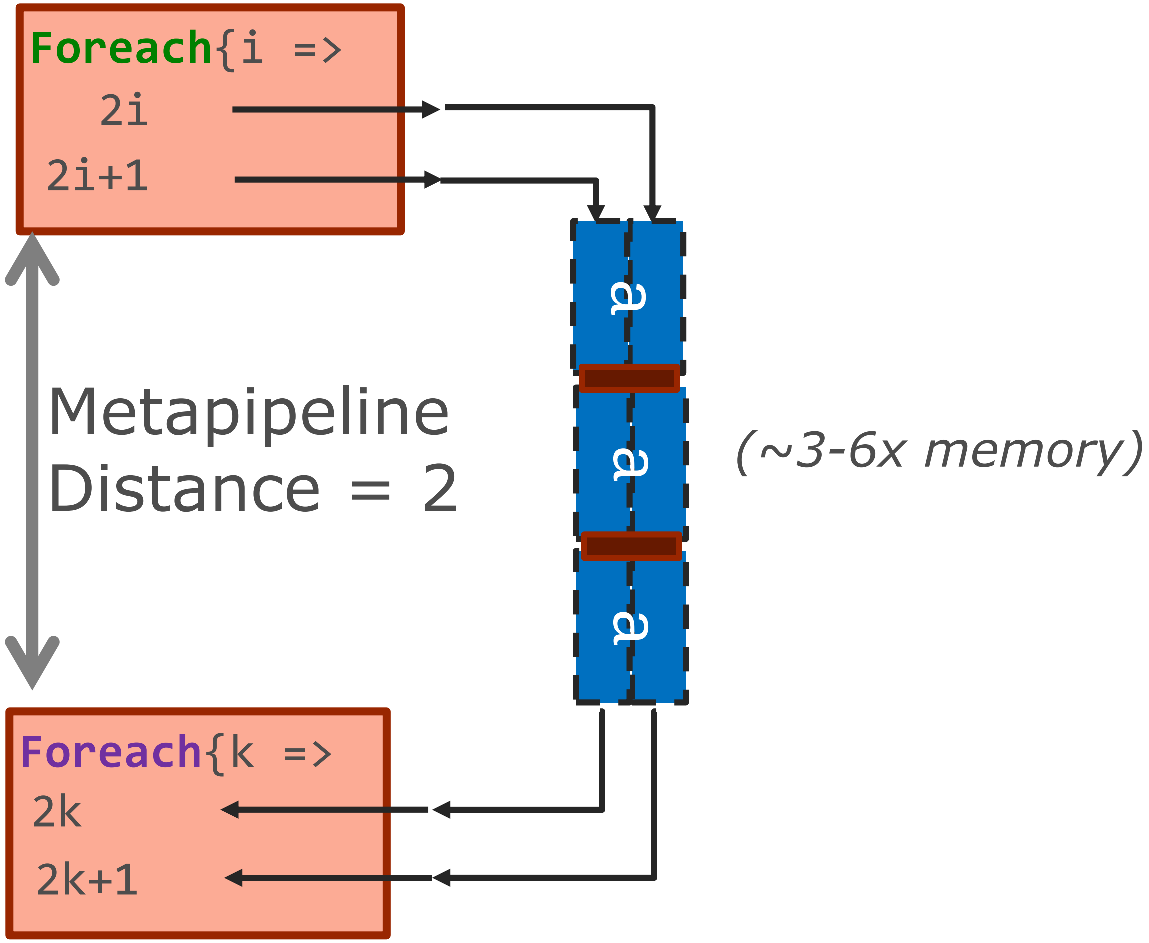


Metapipeline Distance = 1

(~4-8x memory)

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.)

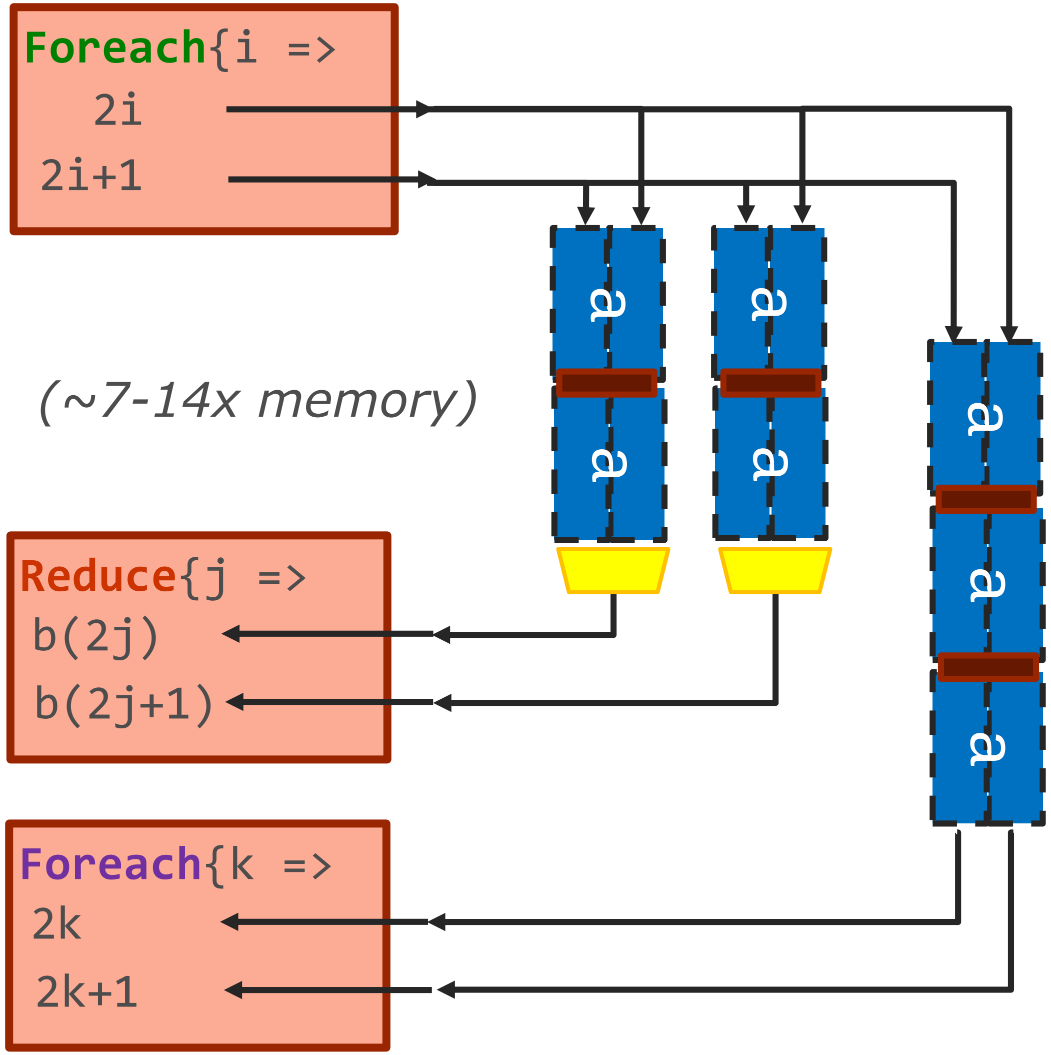
```
Foreach(N by 1){ r =>
  val a = SRAM[Float](D)
  val b = SRAM[Float](D)
```

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!**

```
      c(k) = a(k) * sum
    }
}
```



Foreach{i =>
2i
2i+1

(~7-14x memory)

Reduce{j =>
b(2j)
b(2j+1)

Foreach{k =>
2k
2k+1

Local Memory Analysis

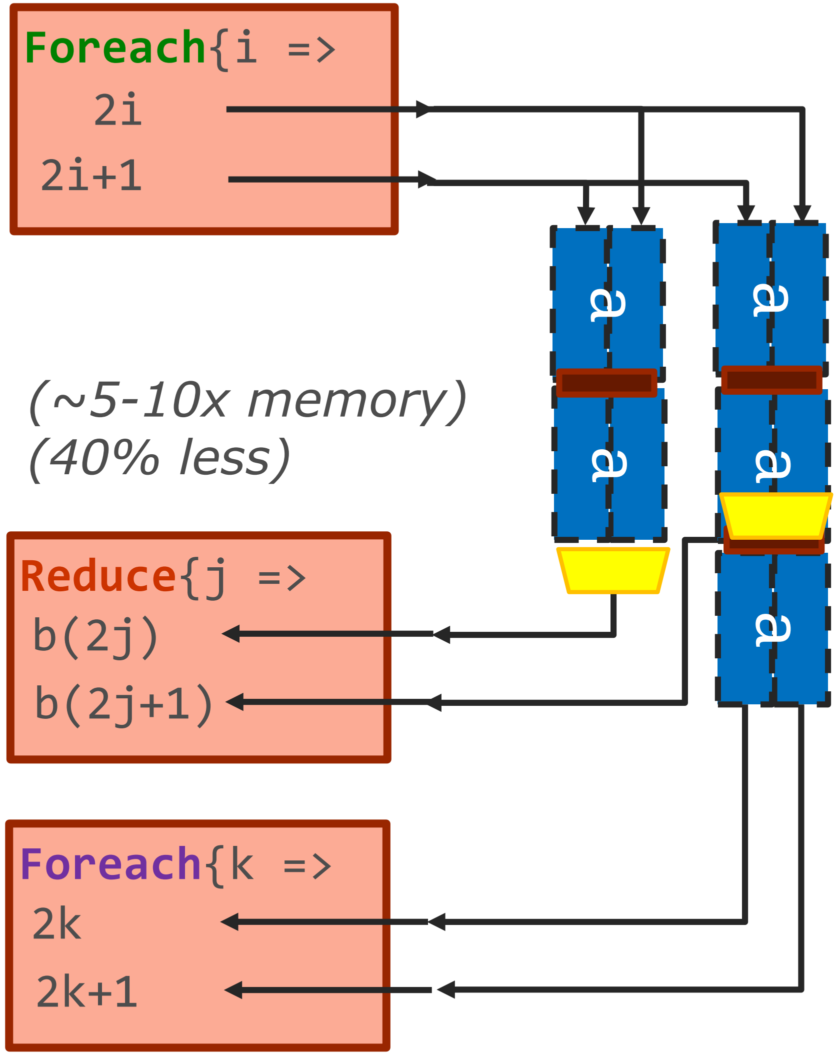
```
Foreach(N by 1){ r =>
  val a = SRAM[Float](D)
  val b = SRAM[Float](D)
```

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!**

```
      c(k) = a(k) * sum
    }
}
```



(~5-10x memory)
(40% less)

Kernel-Based Approach

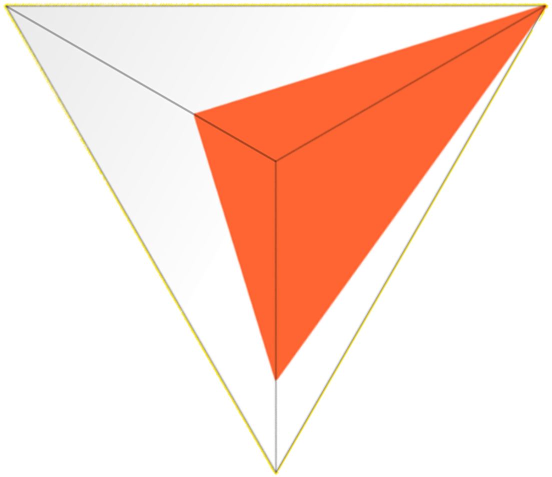
Manually implement each DSL operation;
use a simple compiler to stitch them together

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
}
```

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

```
def epoch(i: Int, ...): Unit = {
  val yPt = Reg[TM]
  if (i >= yAddr & i < yAddr+CSIZE & yAddr != -1) {
    yPt := yCache(i - yAddr)
  }
  else {
    yAddr := i - (i % CSIZE)
    yCache load y(yAddr::yAddr + CSIZE)
    yPt := yCache(i % CSIZE)
  }

  val x = SRAM[TX](D)
  x load data(i, 0::D)

  // Compute gradient against wK_t
  val yHat = Reg[TM]
  Reduce(yHat)(D by 1){j => wK(j) * x(j).to[TM] }{_+_}
  val yErr = yHat - yPt

  // Update wK_t with reduced variance update
  Foreach(D by 1){i =>
    wK(i) = wK(i) - (A.to[TM] * yErr * x(i).to[TM])
  }
}
```

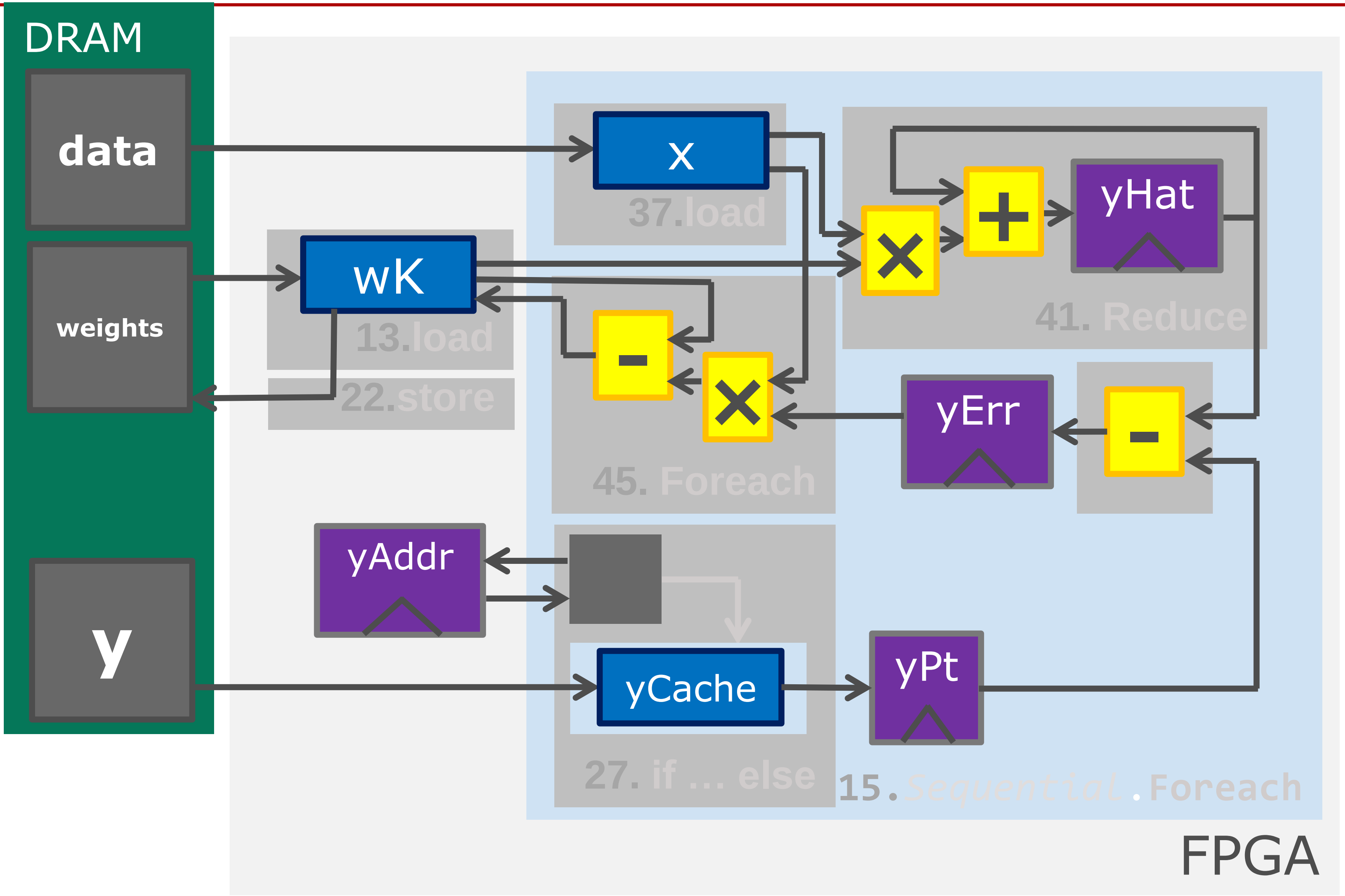
Custom caching for random access on *y*

Explicit memory transfer

Gradient computation

Weight update

SGD in Spatial: Hardware


DRAM
data
weights
y
x
37.load
wK
13.load
22.store
yHat
41. Reduce
yErr
45. Foreach
yAddr
yCache
27. if … else
yPt
15.Sequential.Foreach
FPGA