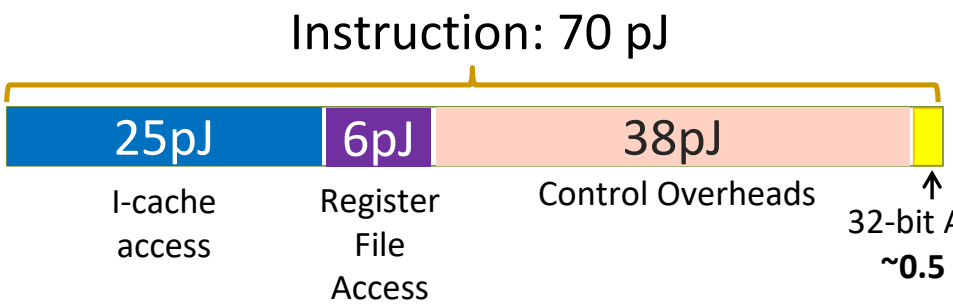
Stanford | ENGINEERING
Electrical Engineering
Computer Science

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

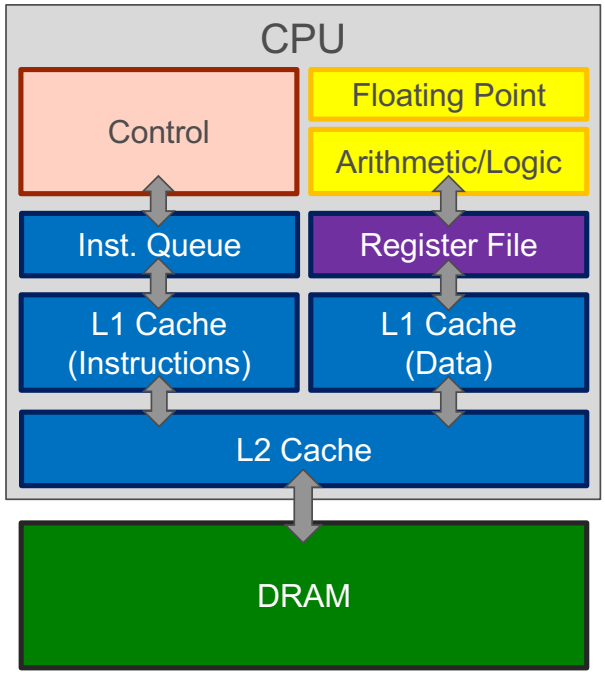
Instructions Add Overheads

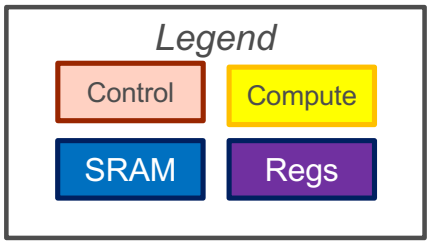


vectorA · vectorB

```
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
```

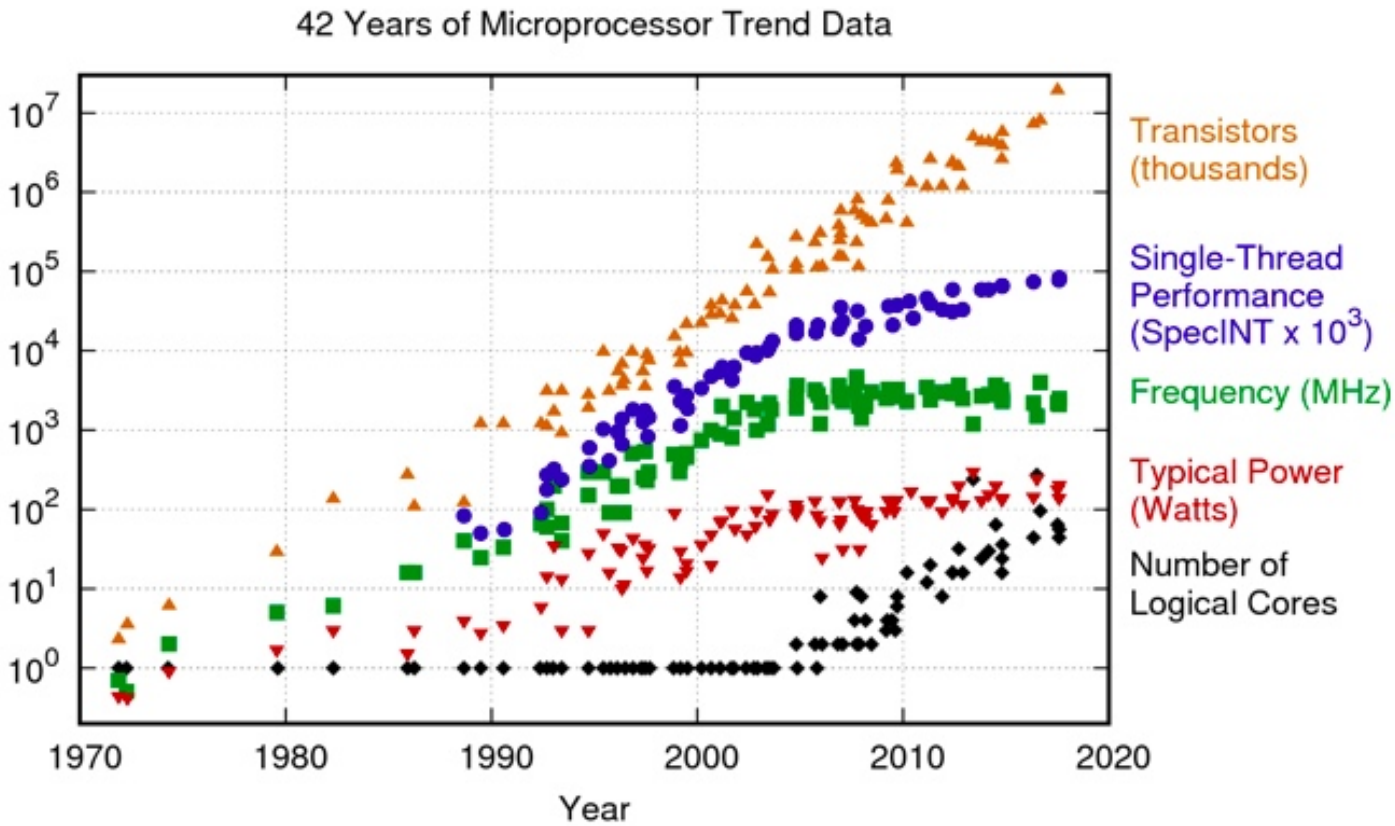
Instruction-Based





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

A Dark Tale: The CPU Power Wall

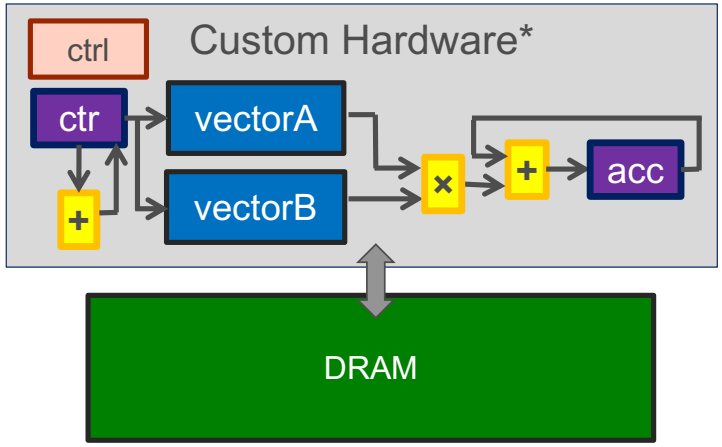


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

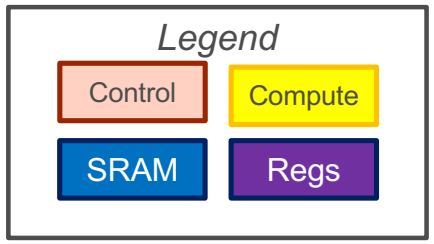
https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/

A More Efficient Way

Configuration-Based



*Also not to scale



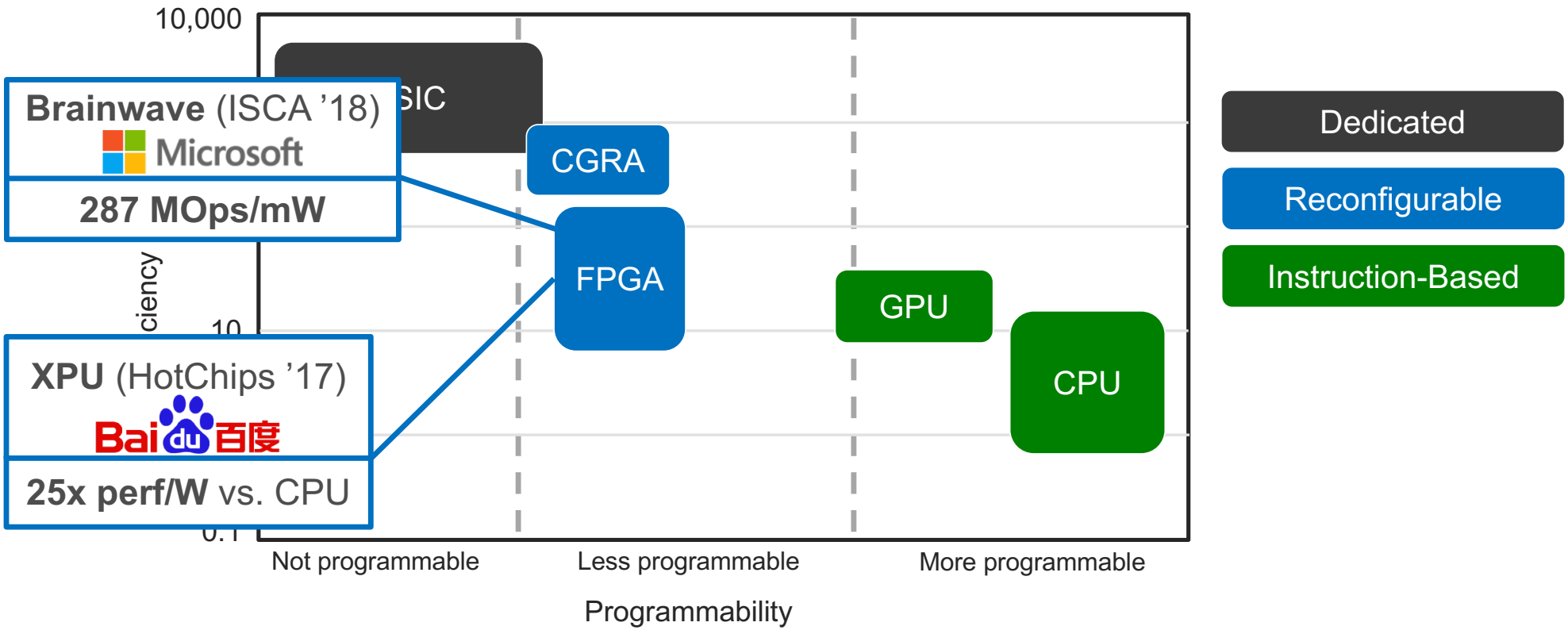
`vectorA · vectorB`

```
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

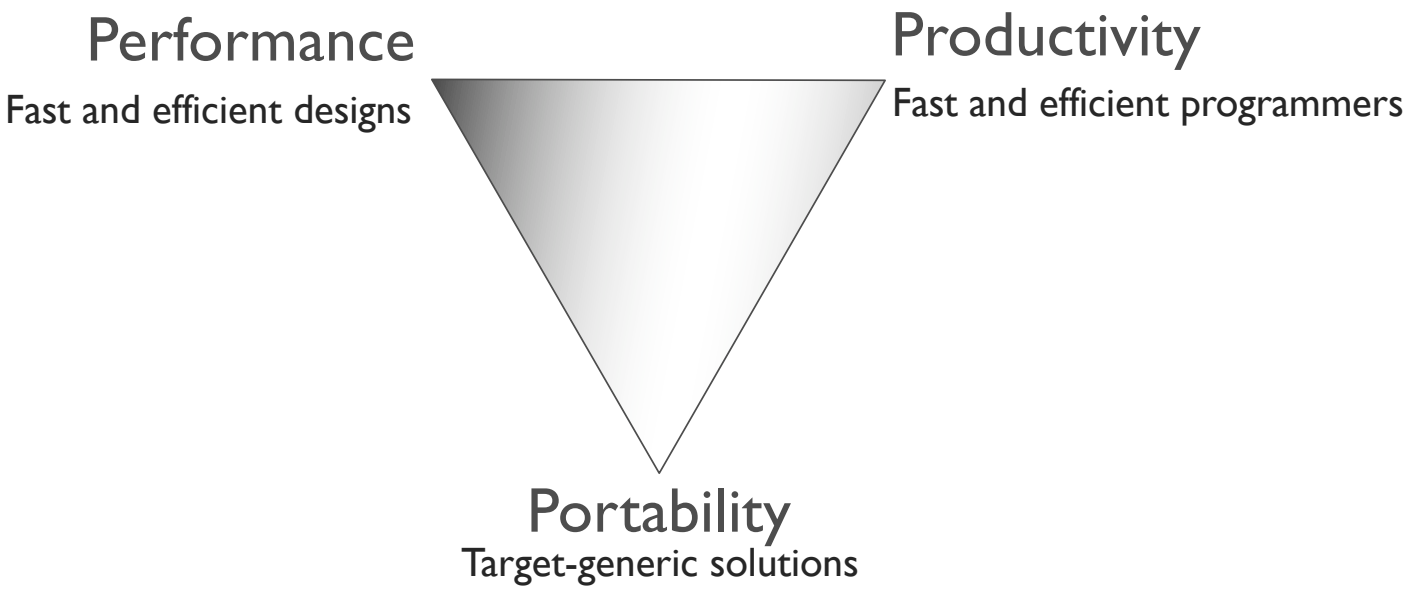
*Not to scale

The Future Is (Probably) Reconfigurable

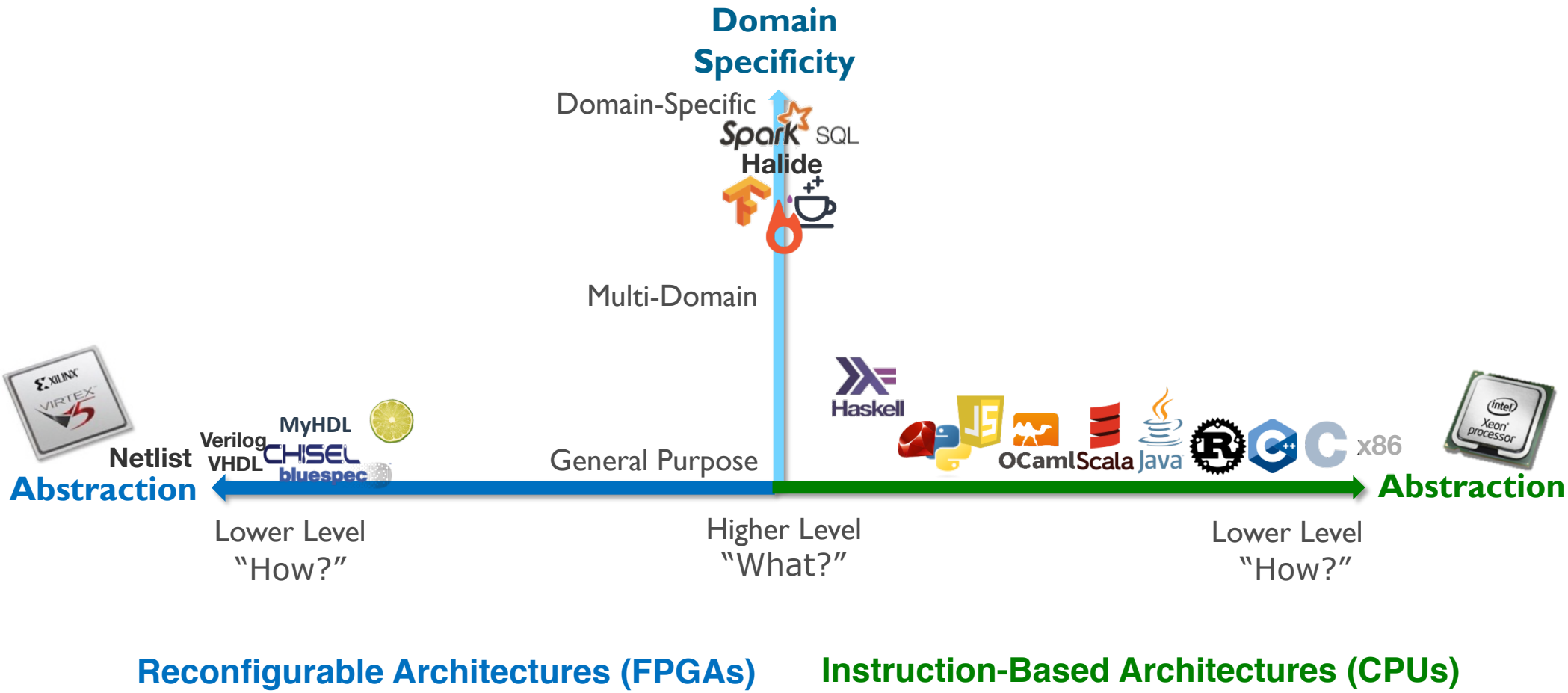


Key Question

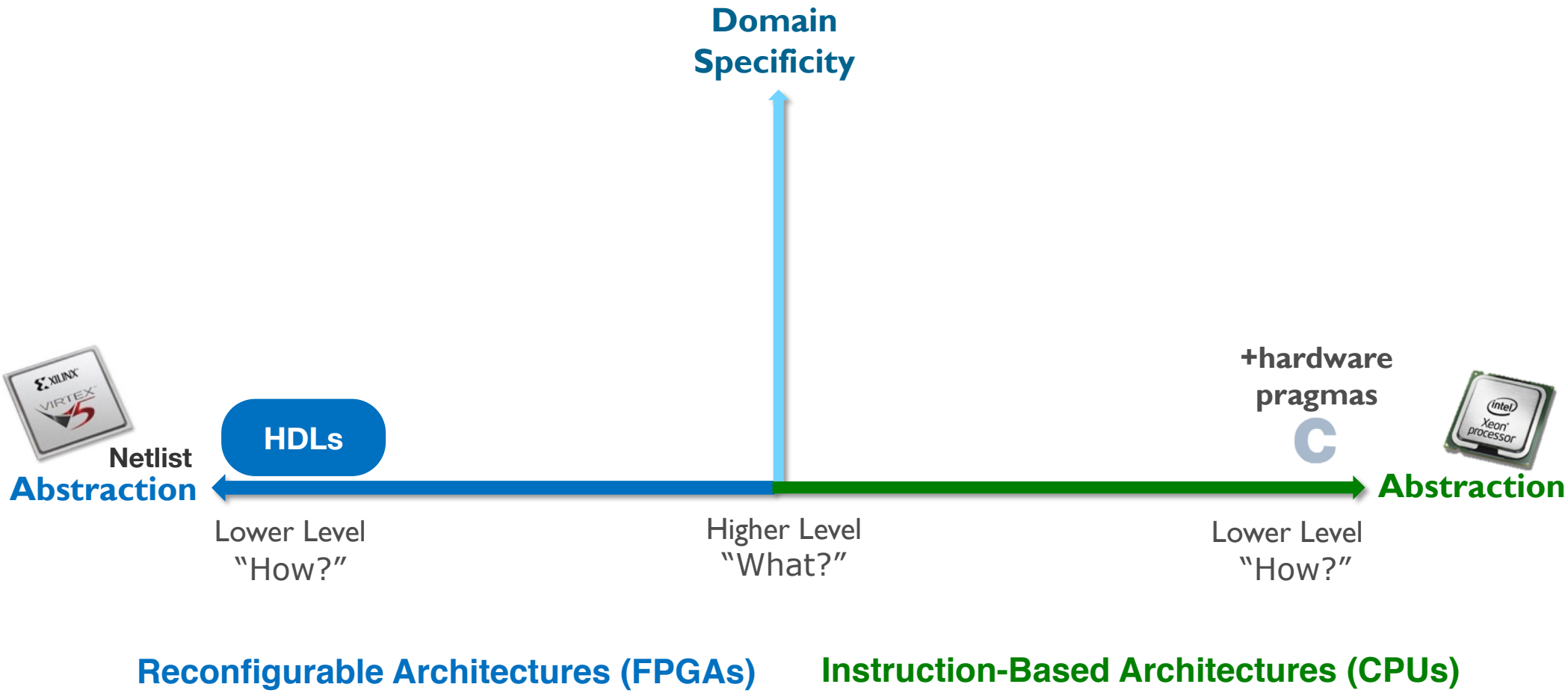
How can we more productively target **reconfigurable architectures** like FPGAs?



Language Taxonomy



Abstracting Hardware Design



HDLs

Hardware Description Languages (HDLs)

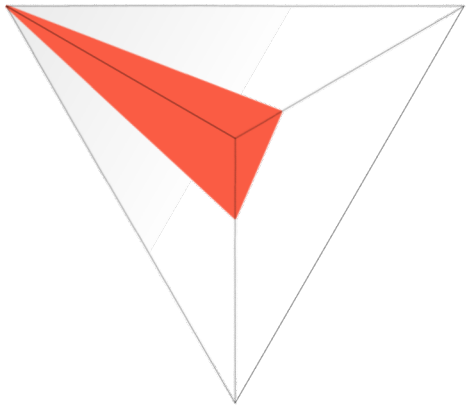
e.g. Verilog, VHDL, Chisel, Bluespec

Performance

✓ Arbitrary RTL

Productivity

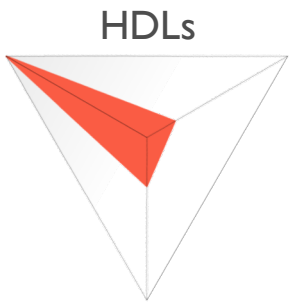
✗ No high-level abstractions



Portability

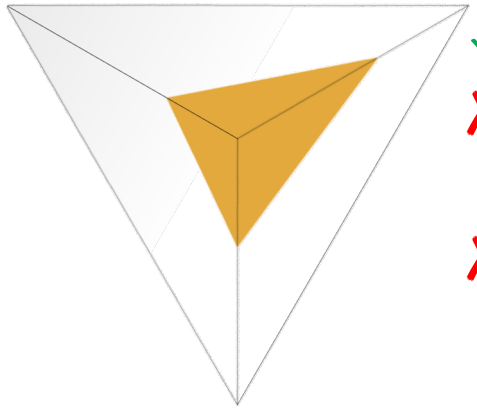
✗ Significant target-specific code

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

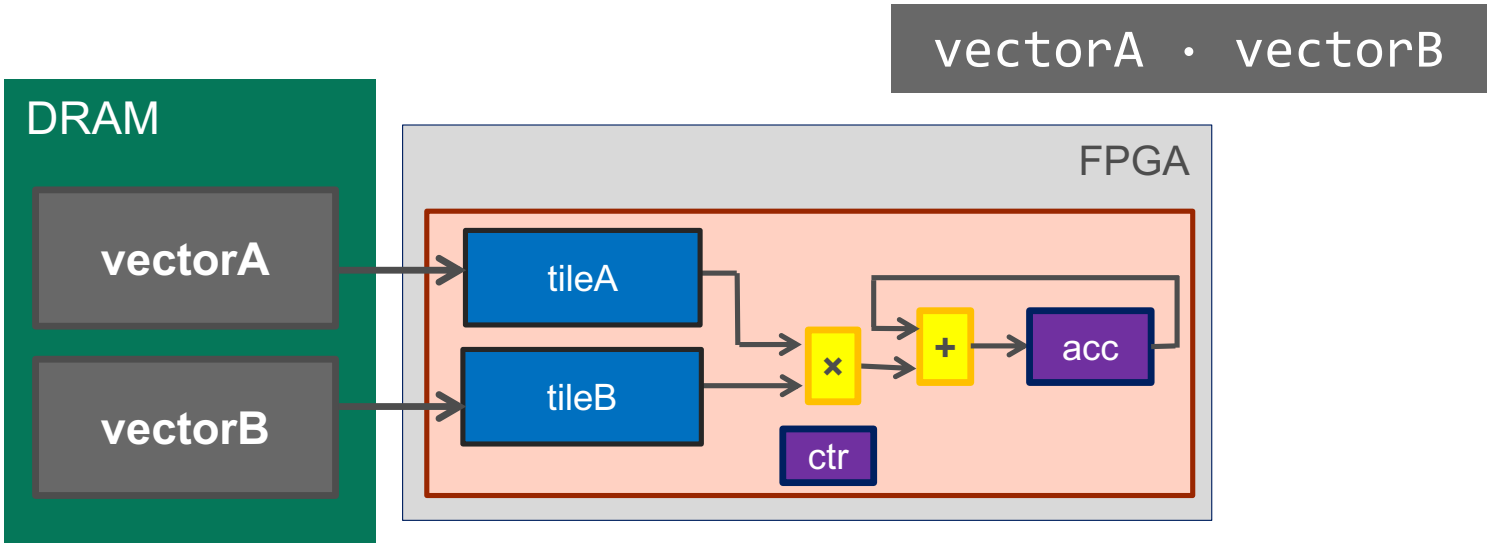
Portability

✓ Portable for single vendor

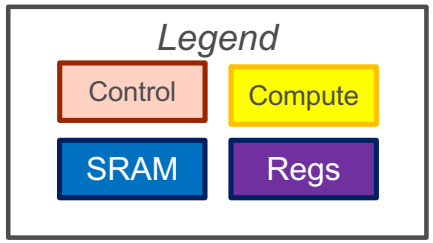
Criteria for Improved HLS

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	✗

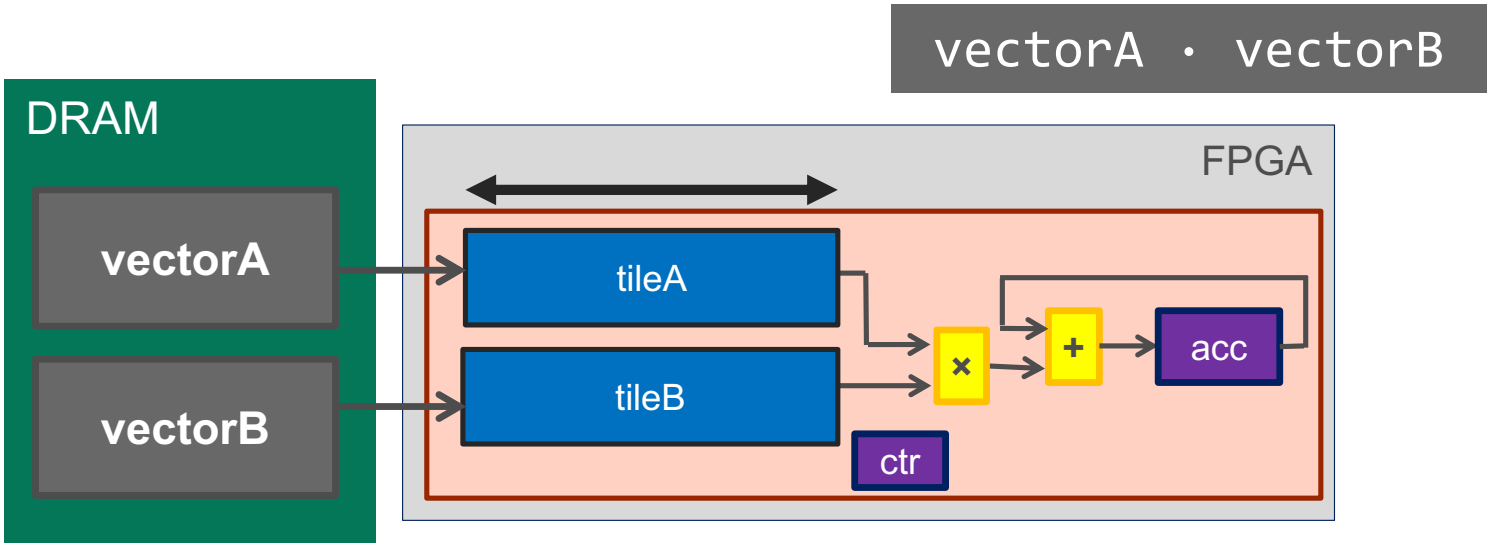
Design Space Parameters Example



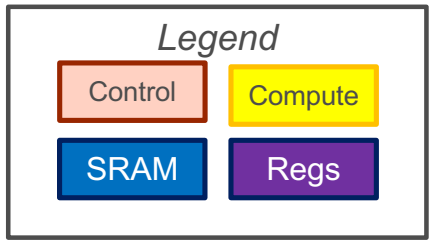
Small and simple, but slow!



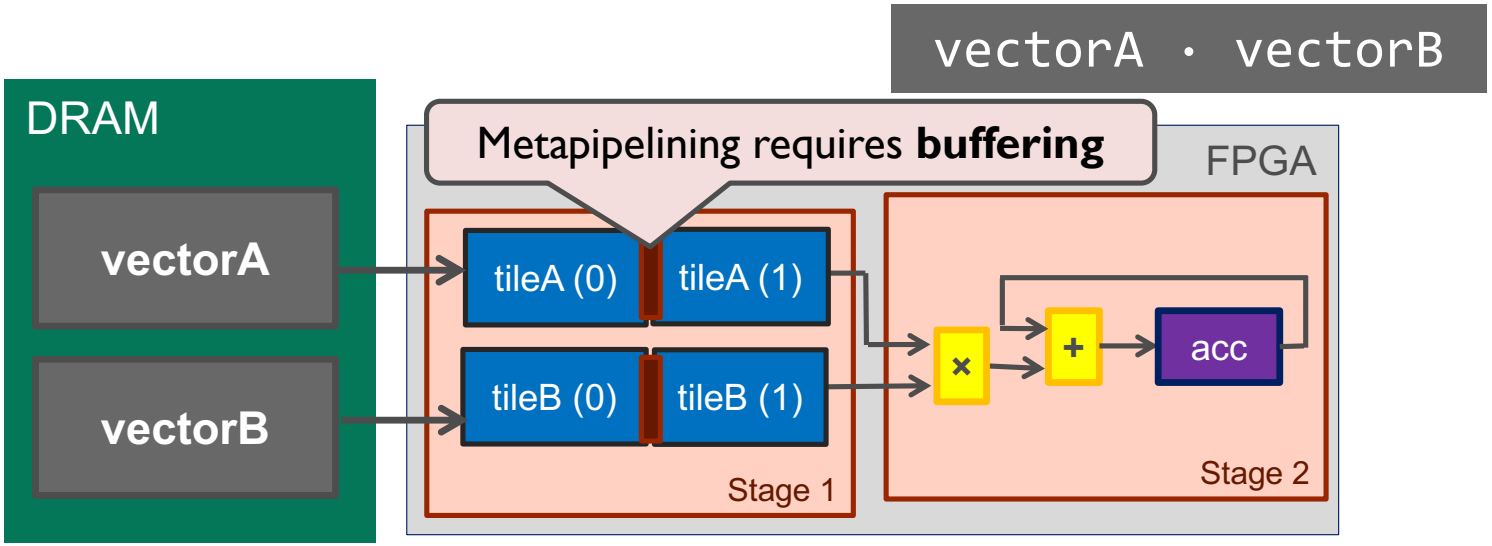
Important Parameters: Buffer Sizes



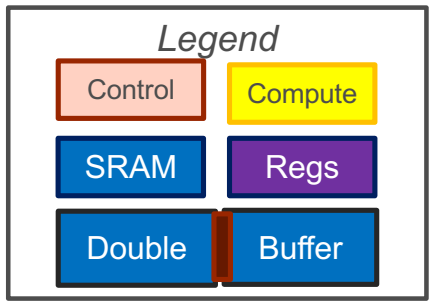
- Increases length of DRAM accesses ↓ Runtime
- Increases exploited locality ↓ Runtime
- Increases local memory sizes ↑ Area



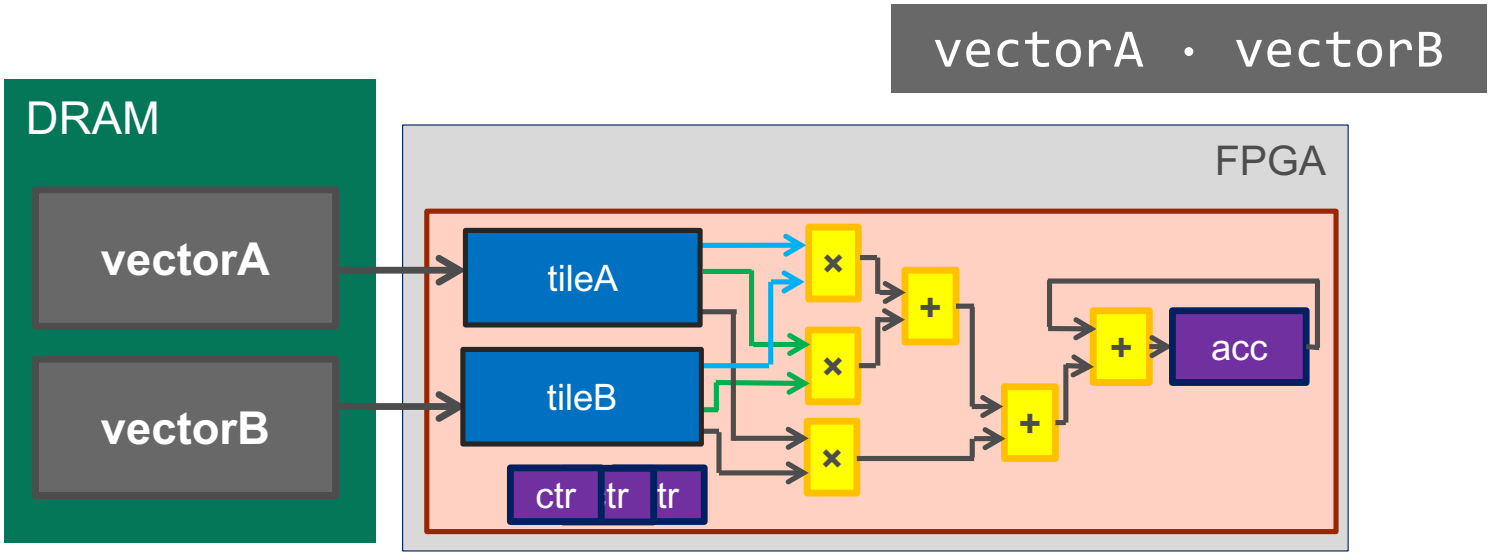
Important Parameters: Pipelining



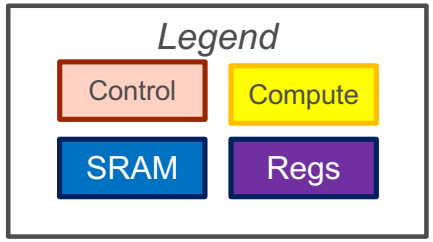
- Overlaps memory and compute ↓ Runtime
- Increases local memory sizes ↑ Area
- Adds synchronization logic ↑ Area



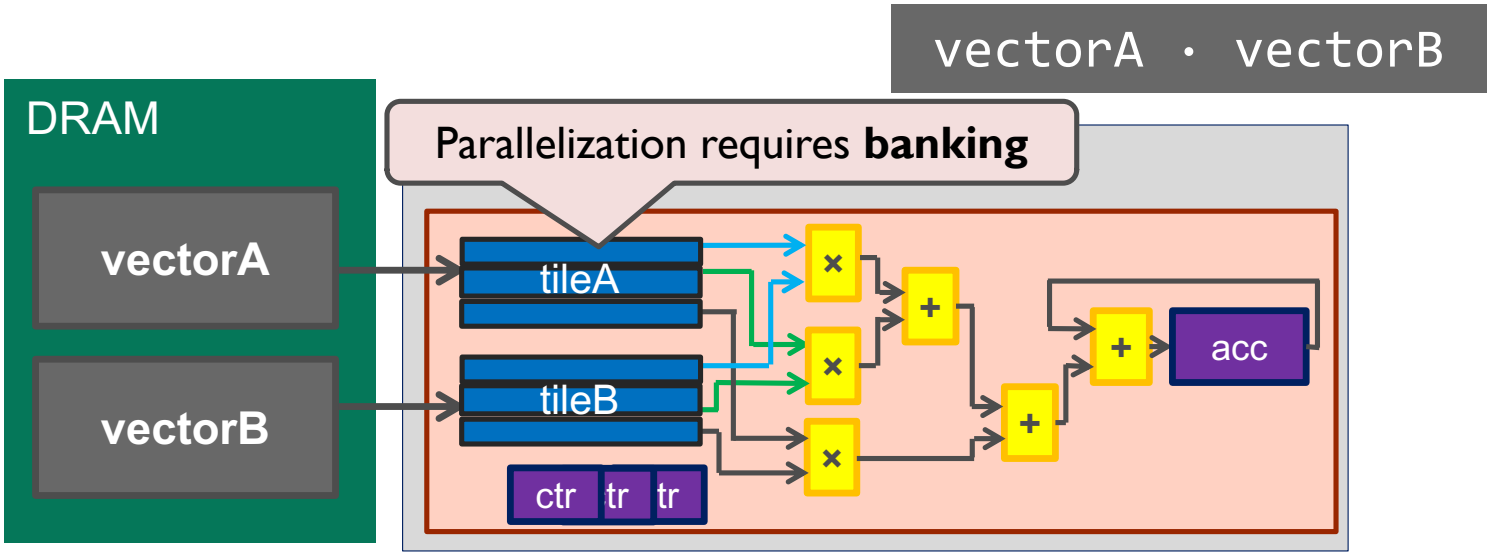
Important Parameters: Parallelization



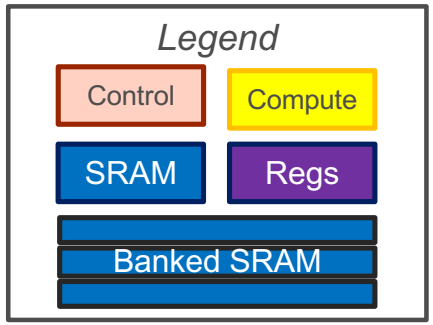
- Improves element throughput ↓ Runtime
- Duplicates compute resources ↑ Area



Important Parameters: Memory Banking



- Improves memory bandwidth ↓ Runtime
- May duplicate memory resources ↑ Area

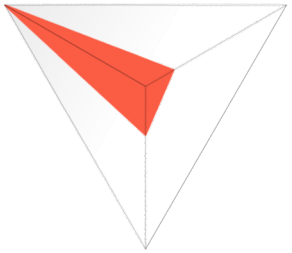


Criteria for Improved HLS

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	✗

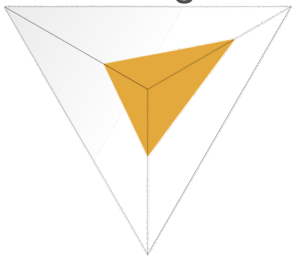
Rethinking HLS

HDLs



Improved HLS

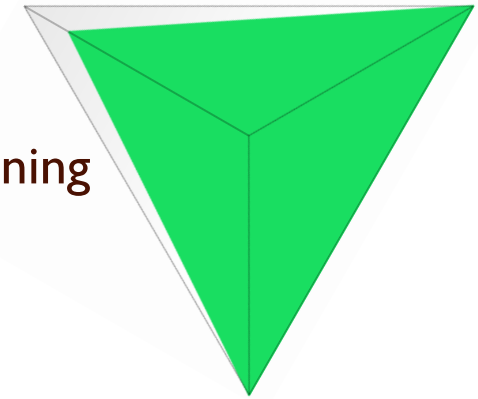
C + Pragmas



Performance

✓ Memory hierarchy

✓ Arbitrary pipelining



Productivity

✓ Nested loops

✓ Automatic memory banking/buffering

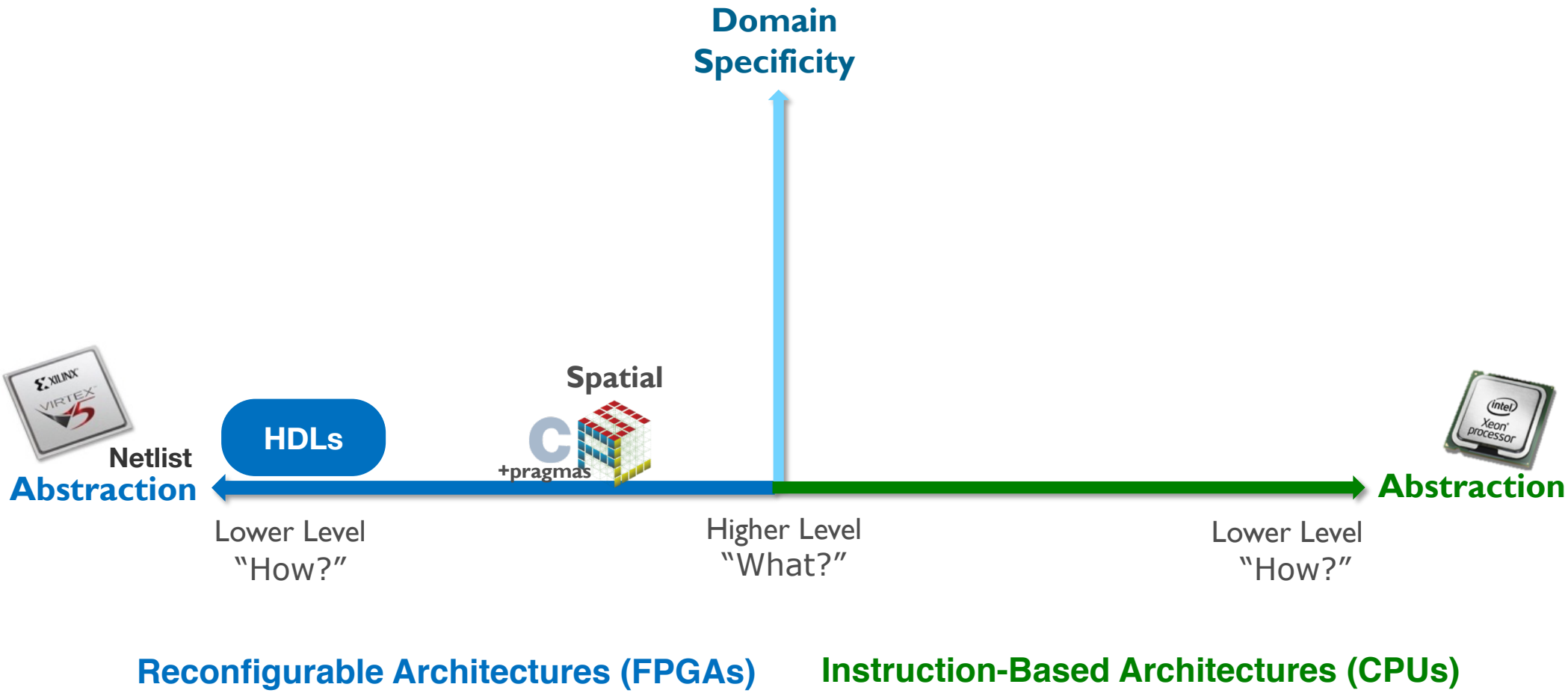
✓ Implicit design parameters (unrolling, banking, etc.)

✓ Automated design tuning

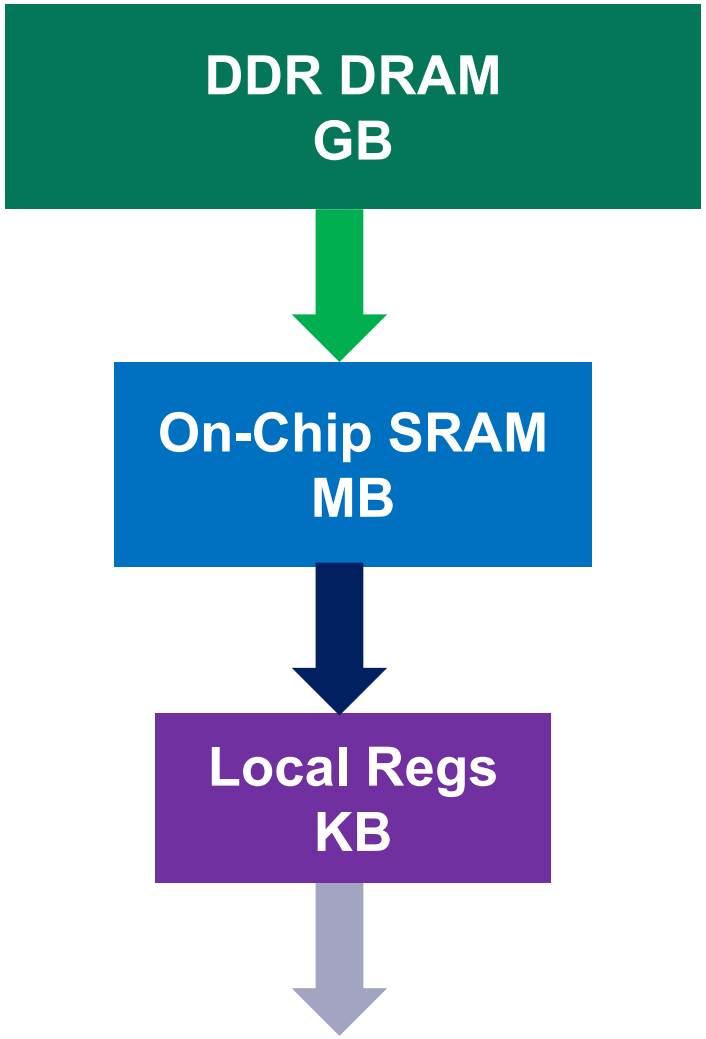
Portability

✓ Target-generic source across reconfigurable architectures

Abstracting Hardware Design



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 =>
  …
}
```

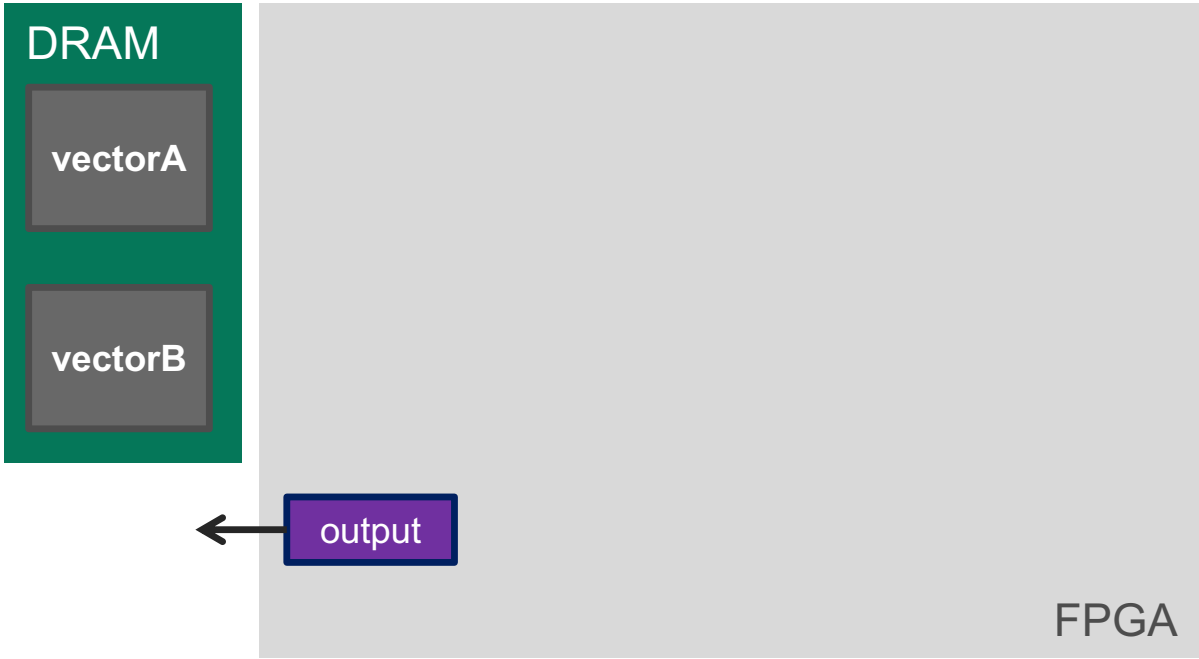
Implicit memory banking and buffering schemes for parallelized access

```
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)
```

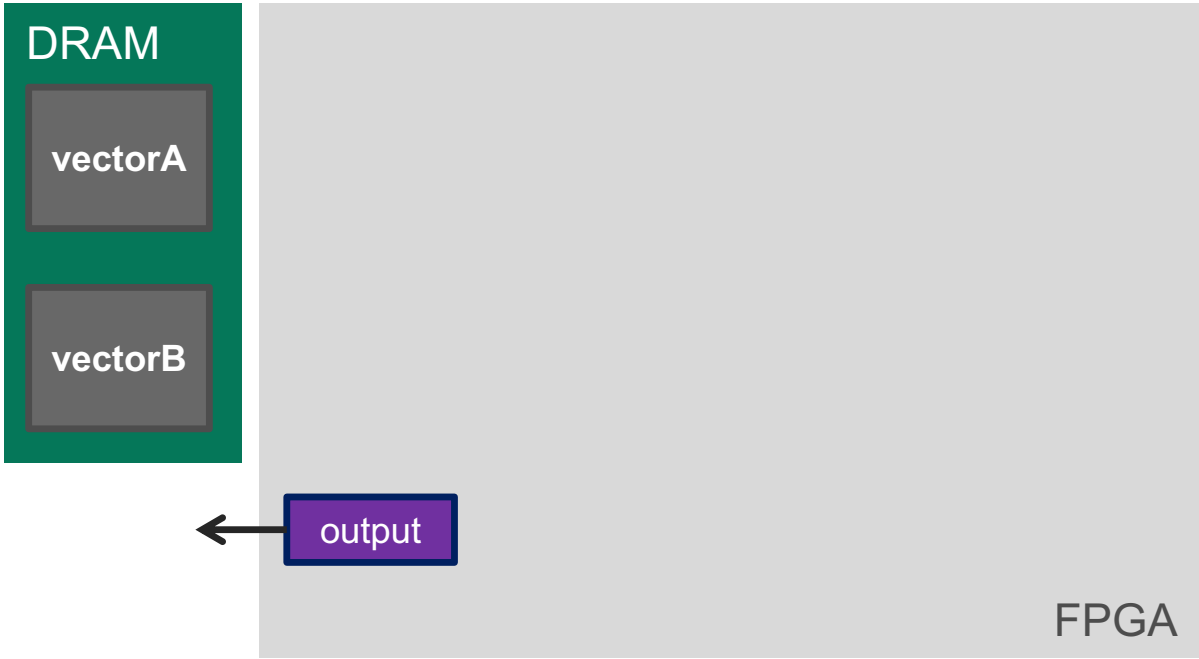
Off-chip memory declarations



Dot Product in Spatial

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

Accel {



}
```

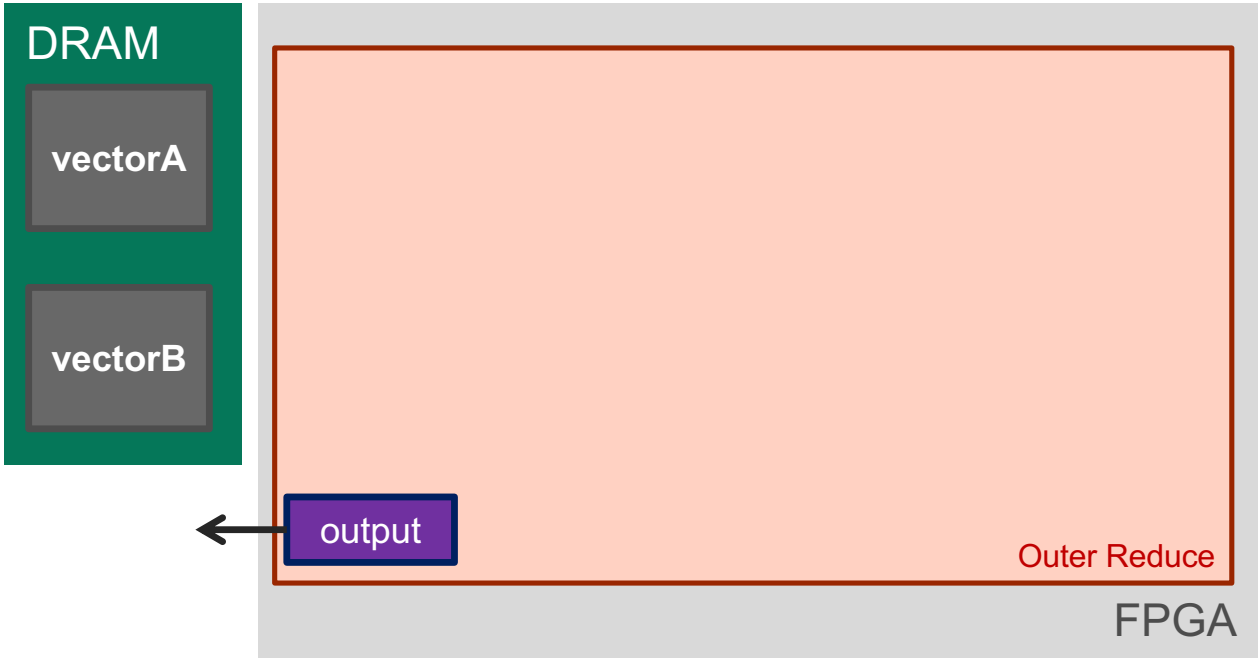
Explicit work division in IR



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 =>



}
```

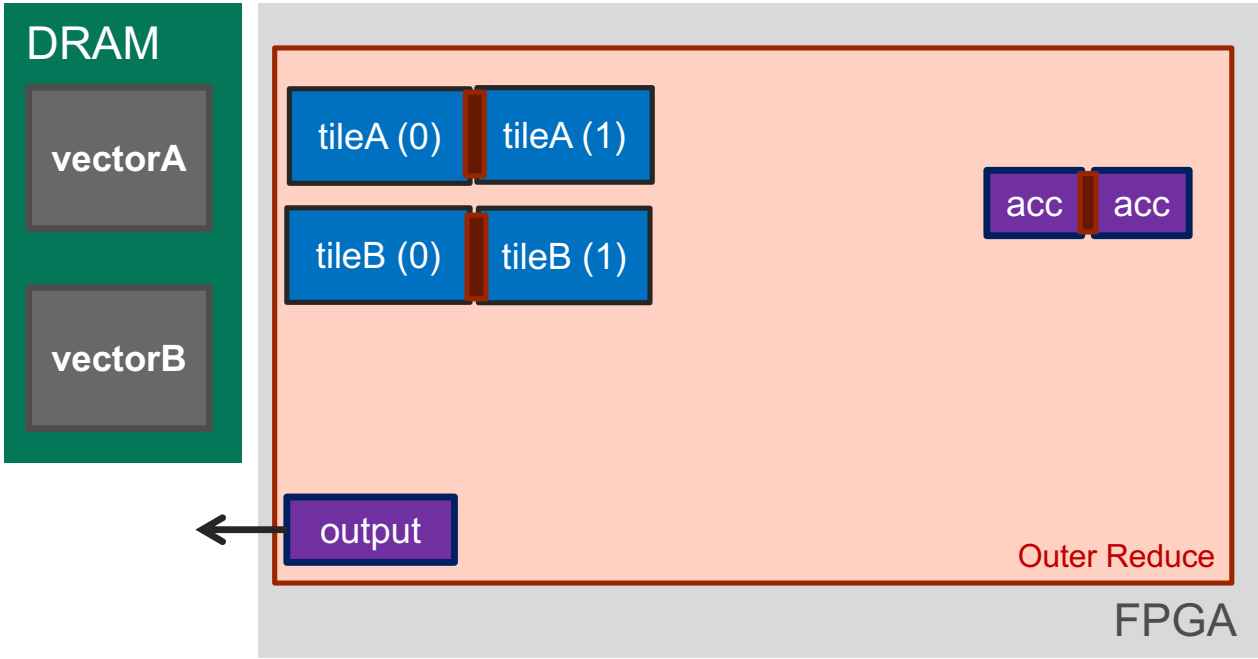
Tiled reduction (outer)



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]



}
```

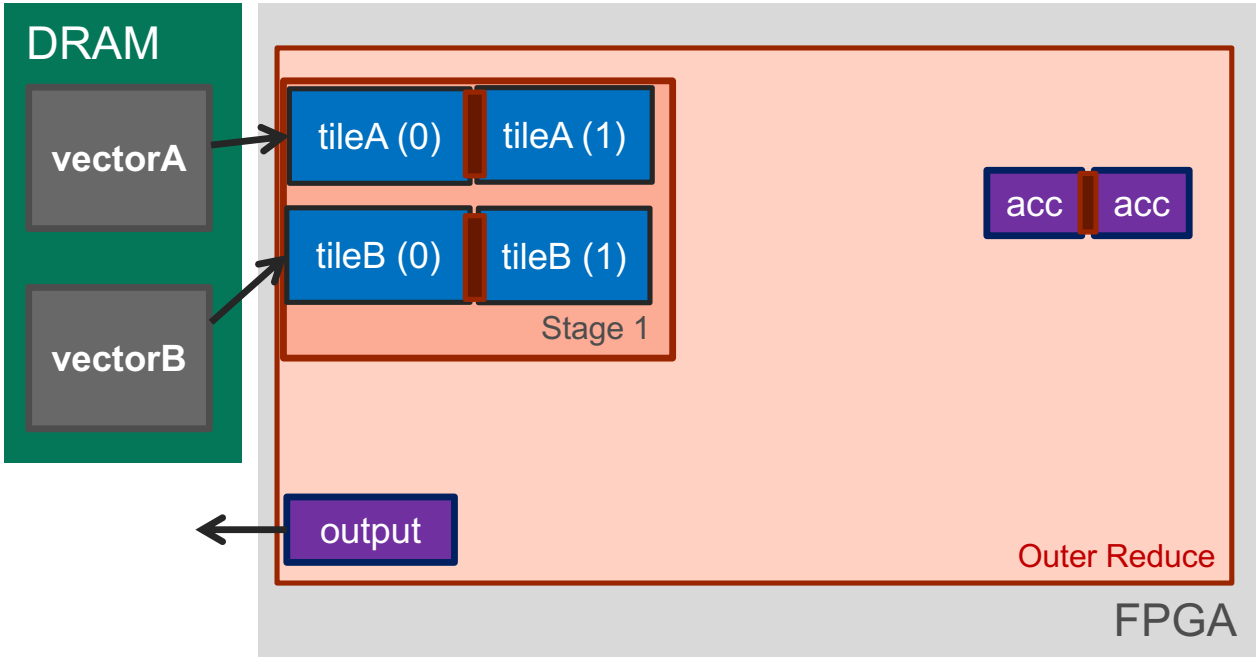
On-chip memory declarations



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)



}
```

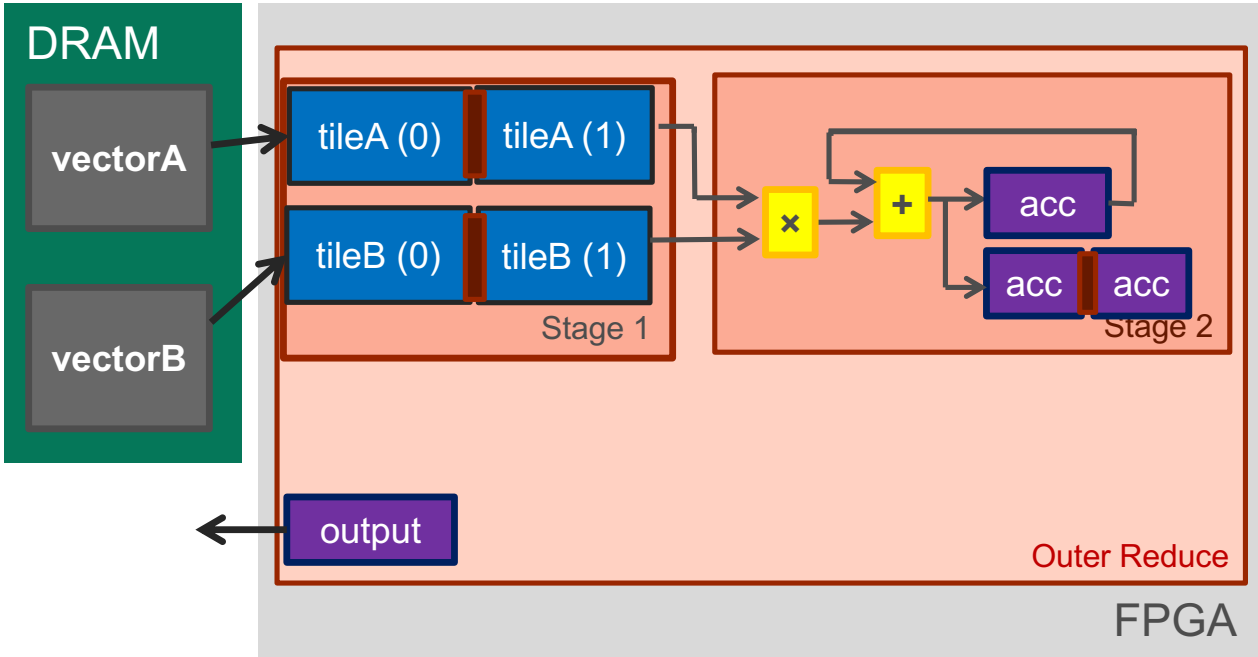
DRAM → SRAM transfers
(also have **store**, **scatter**, and **gather**)



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}

}
```

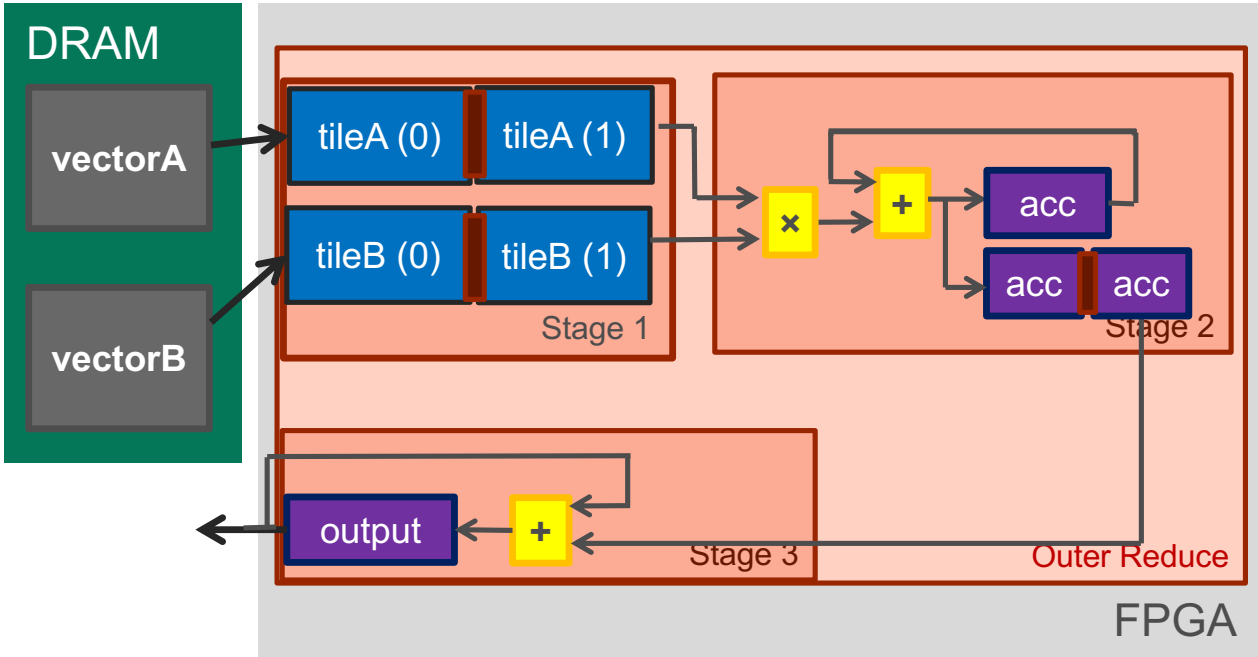
Tiled reduction (pipelined)



Dot Product in Spatial

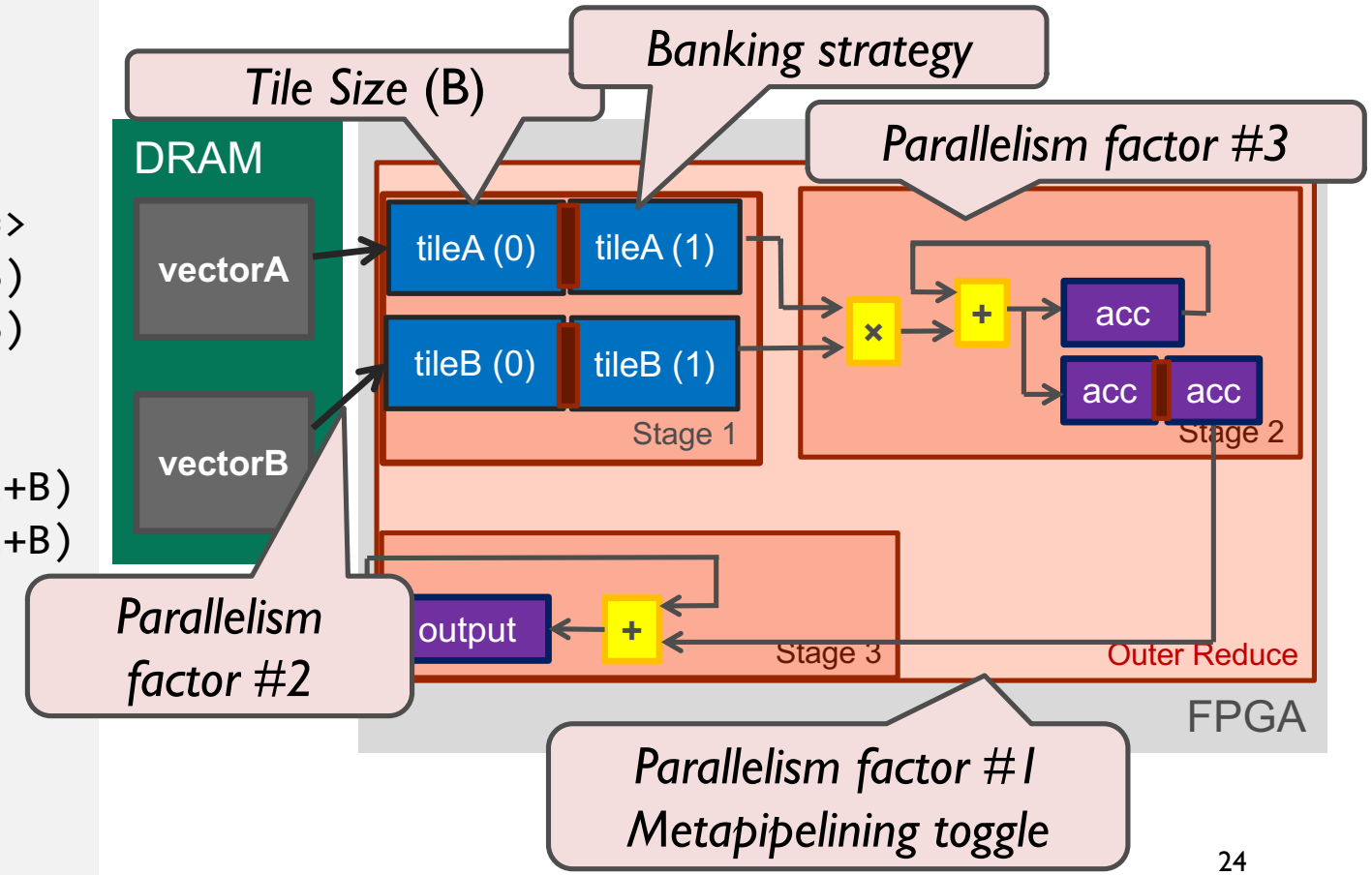
```
val output  = ArgOut[Float]
val vectorA = DRAM[Float](N)
val vectorB = DRAM[Float](N)

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}
}
```

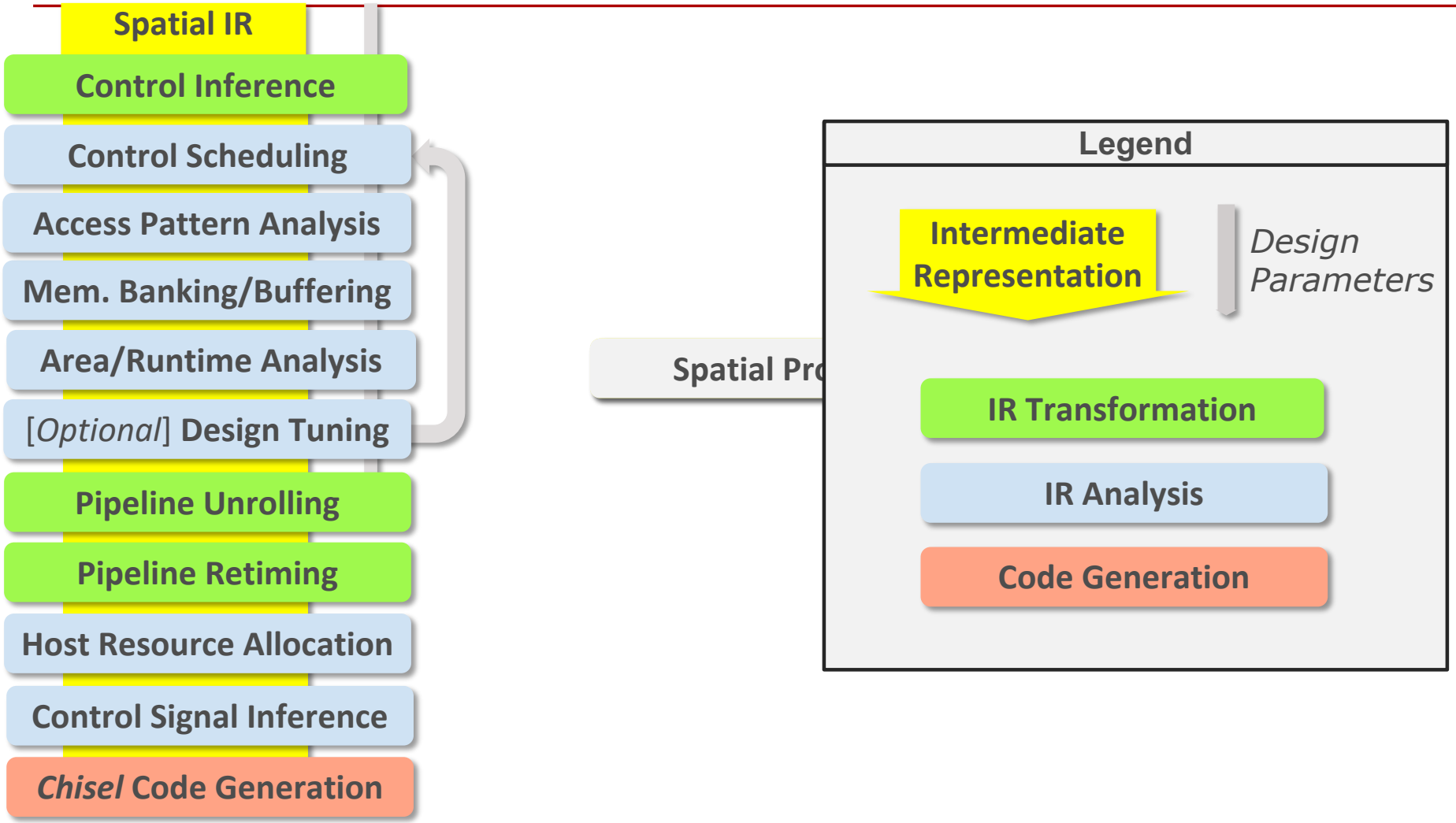


Dot Product in Spatial

```
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



Control Scheduling

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

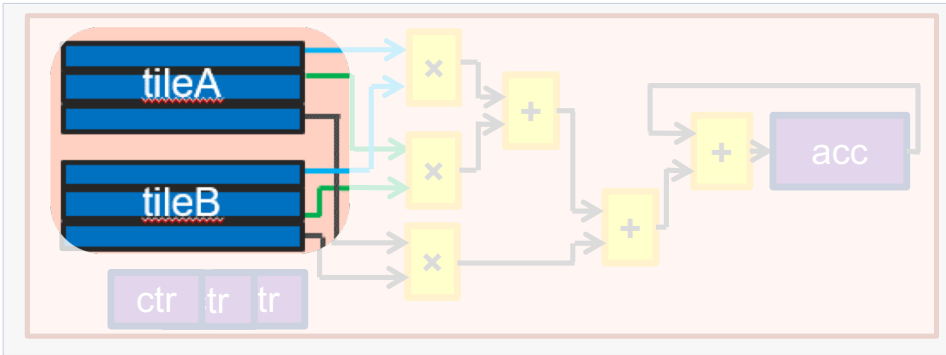
- Creates loop pipeline schedules
  - Detects data dependencies across loop intervals
  - Calculate initiation interval of pipelines
  - Set maximum depth of buffers
- Supports **arbitrarily nested** pipelines

  (Commercial HLS tools don't support this)

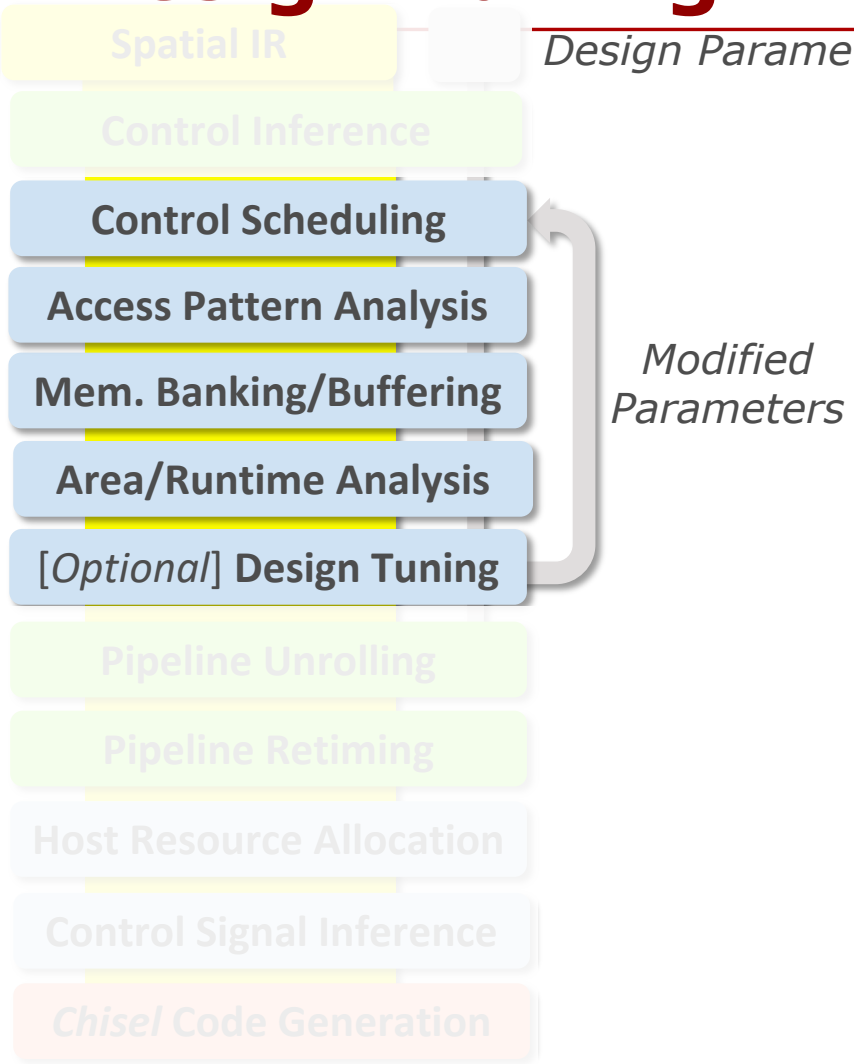
Local Memory Analysis

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

- Active learning: HyperMapper
  (More details in paper)

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

The Spatial Compiler: The Rest

Code generation

- Synthesizable Chisel
- C++ code for host CPU

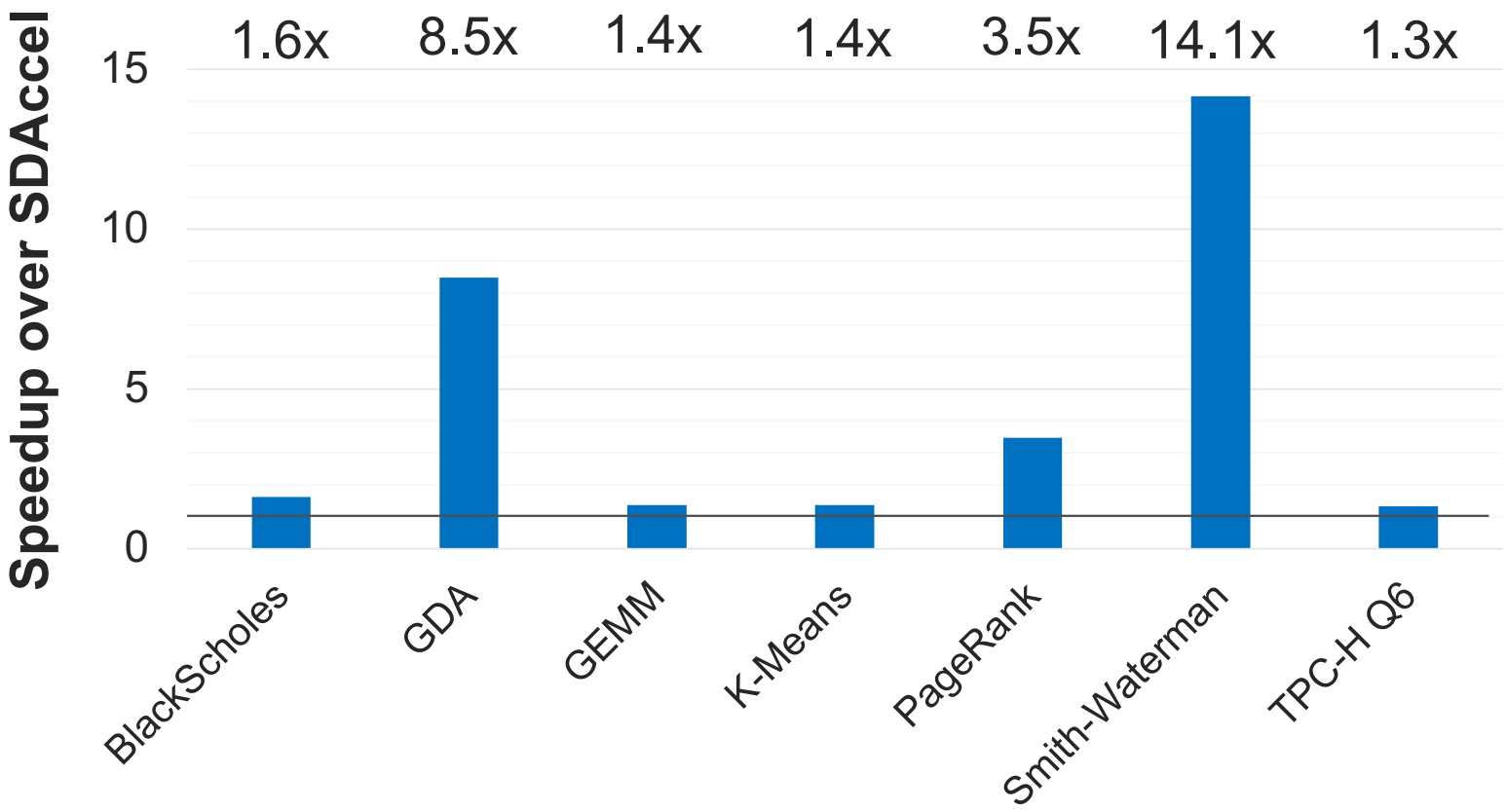
Evaluation: Performance

- 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***

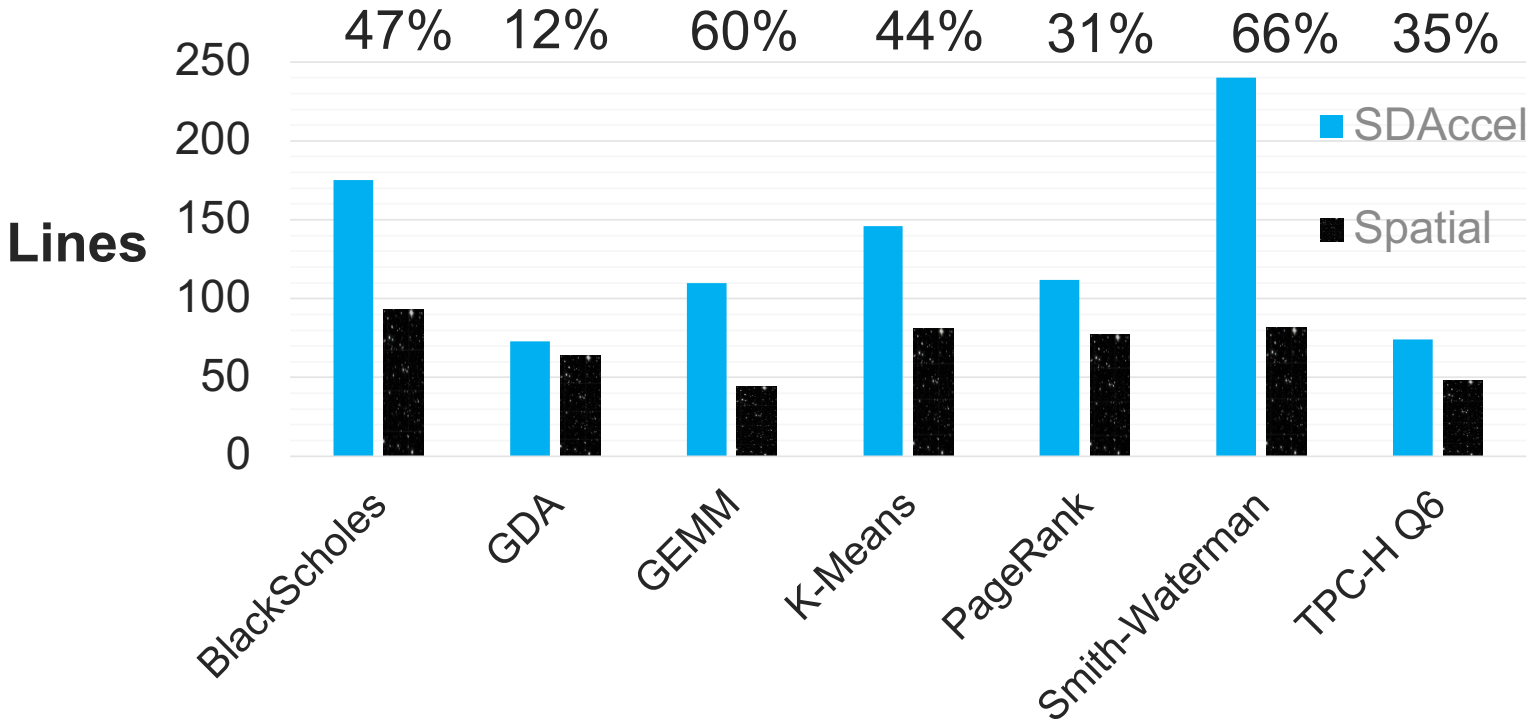
Performance (Spatial vs. SDAccel)

Average **2.9x faster hardware** than SDAccel



Productivity: Lines of Code

Average **42% shorter** programs versus SDAccel

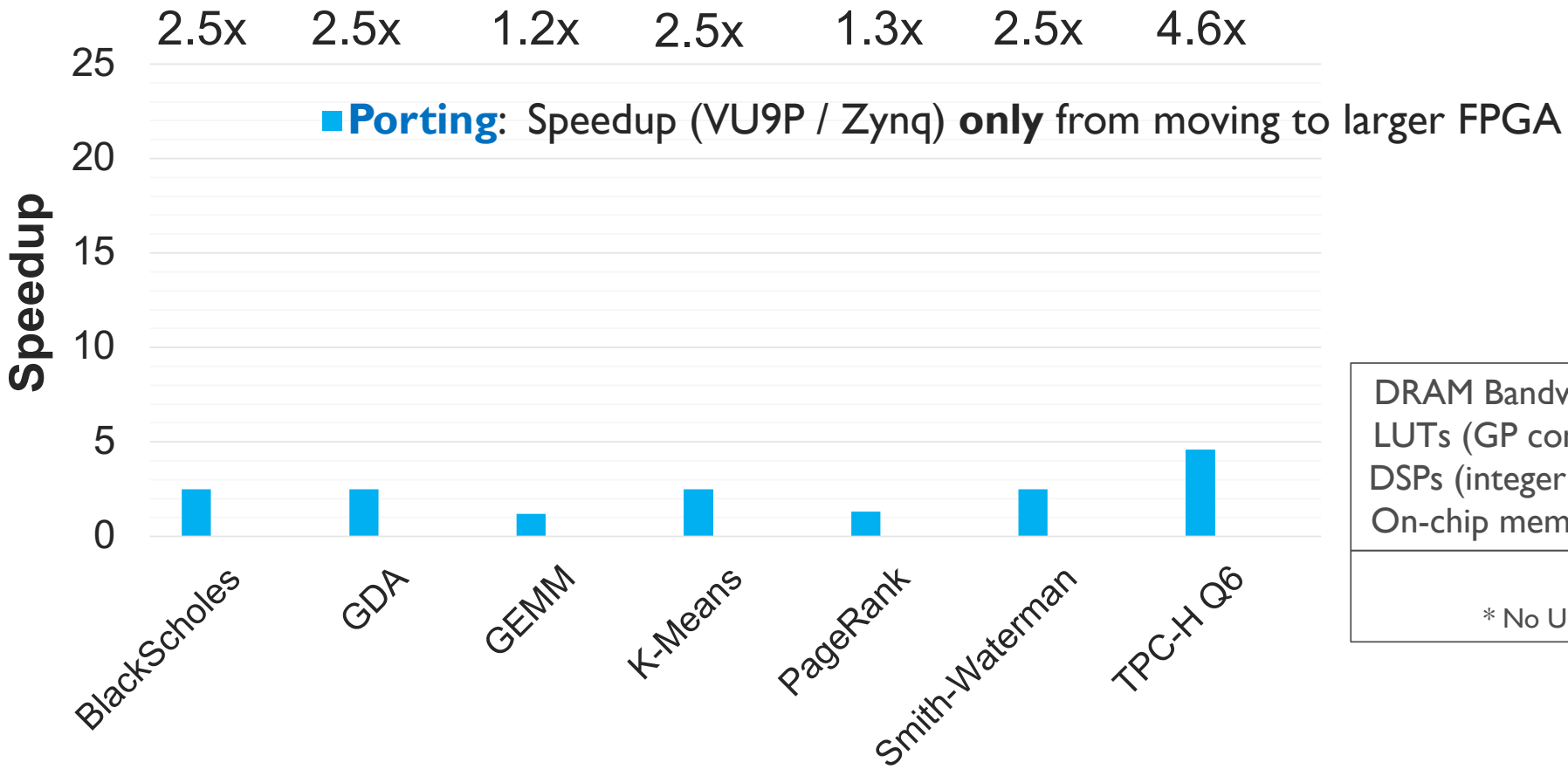


Evaluation: Portability

- 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



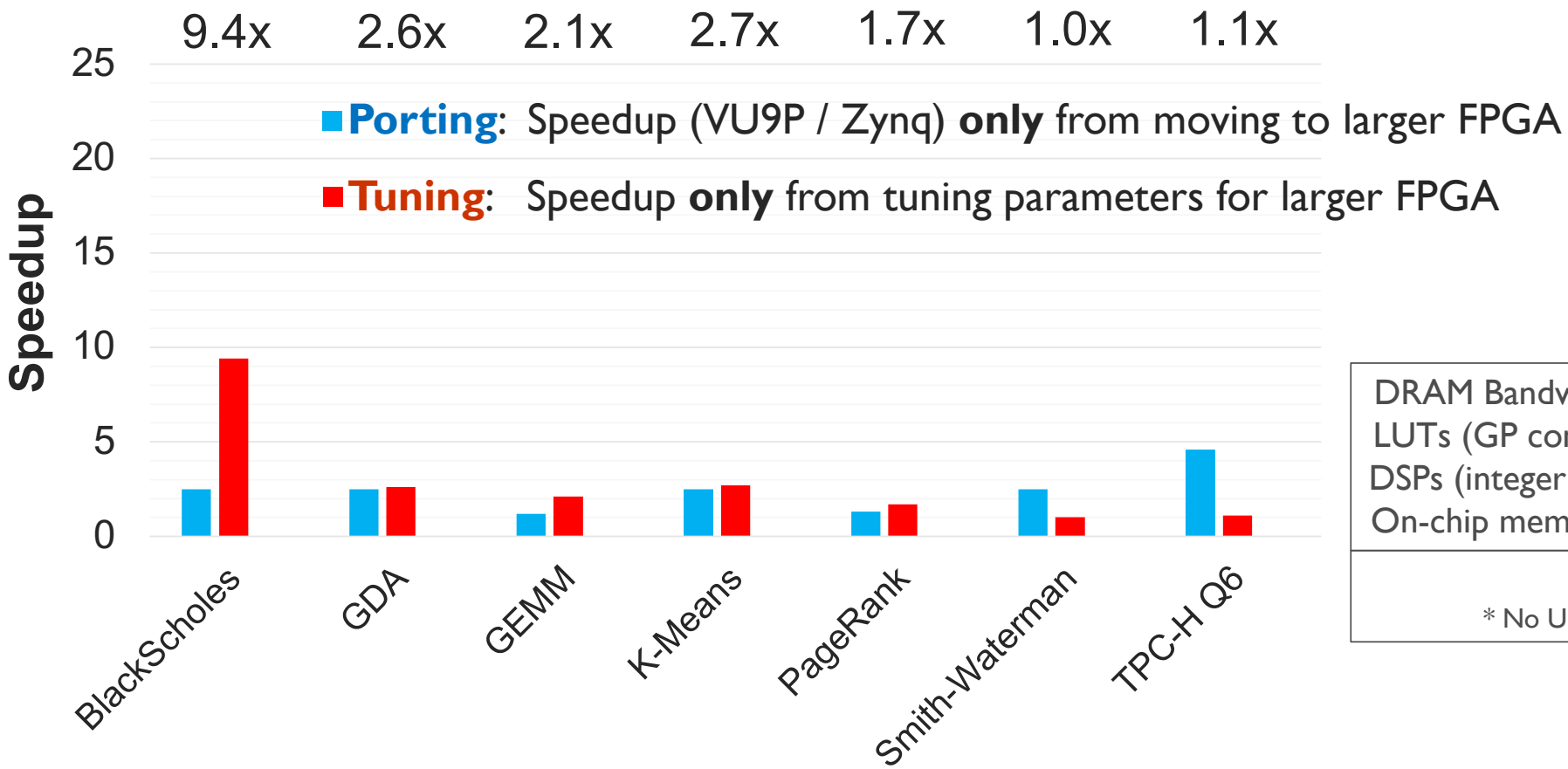
DRAM Bandwidth:	**4.5x**
LUTs (GP compute):	**47.3x**
DSPs (integer FMA):	**7.6x**
On-chip memory*:	**4.0x**

VU9P / ZC706

* No URAM used on VU9P

Portability: VU9P vs. Zynq ZC706

Identical Spatial source, multiple targets



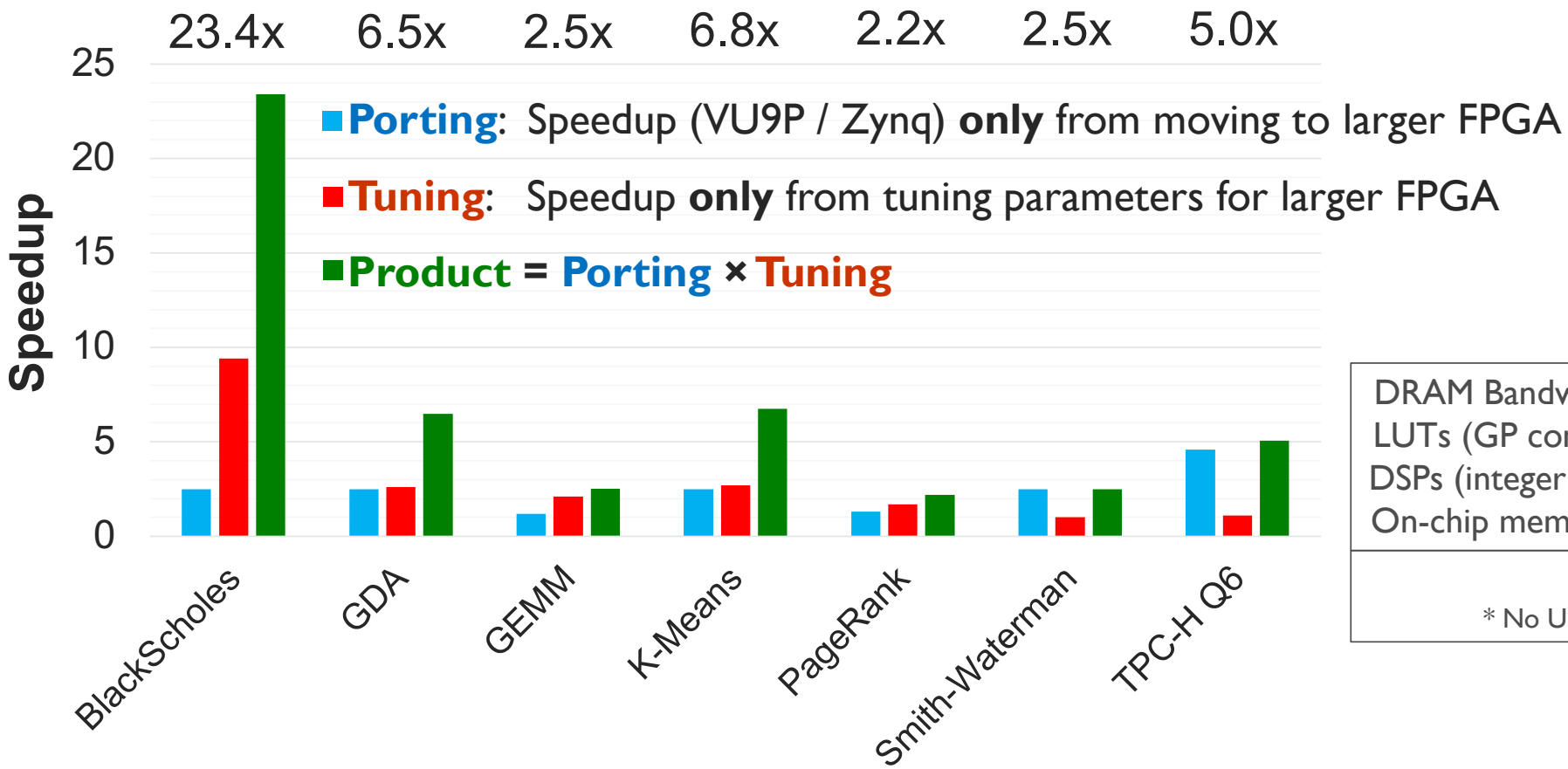
DRAM Bandwidth:	**4.5x**
LUTs (GP compute):	**47.3x**
DSPs (integer FMA):	**7.6x**
On-chip memory*:	**4.0x**

VU9P / ZC706

* No URAM used on VU9P

Portability: VU9P vs. Zynq ZC706

Identical Spatial source, multiple targets



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

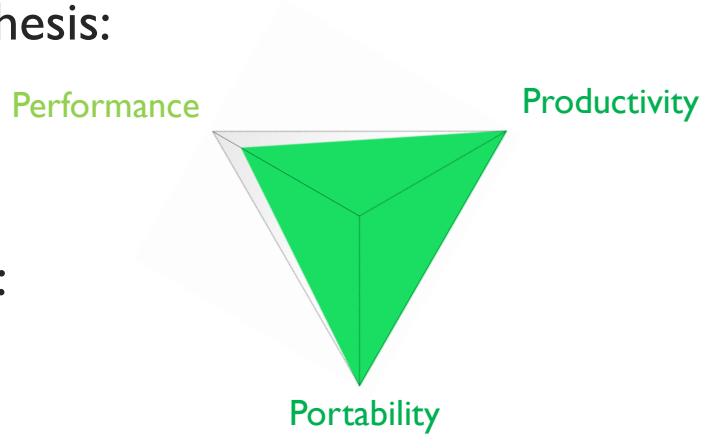
Identical Spatial source, multiple targets
Even reconfigurable hardware that isn't an FPGA!

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)

Conclusion

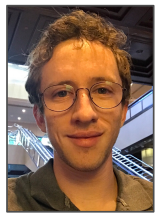
- **Reconfigurable architectures** are becoming key for performance / energy efficiency
- Current programming solutions for reconfigurables are still inadequate
- Need to rethink outside of the C box for high level synthesis:
  - **Memory hierarchy for optimization**
  - **Design parameters for tuning**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

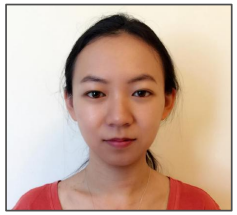
The Team

David
Koeplinger



Matt
Feldman

Raghu
Prabhakar



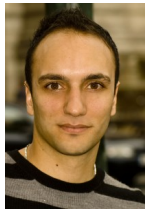
Yaqi
Zhang

Stefan
Hadjis

Ruben
Fiszel

Tian
Zhao

Ardavan
Pedram



Luigi
Nardi

Christos
Kozyrakis

Kunle
Olukotun