Lecture 17:

Hardware Specialization and Algorithm Specific Programming

Parallel Computing
Stanford CS149, Fall 2022

Energy-constrained computing

Performance and Power

Performance

Energy efficiency

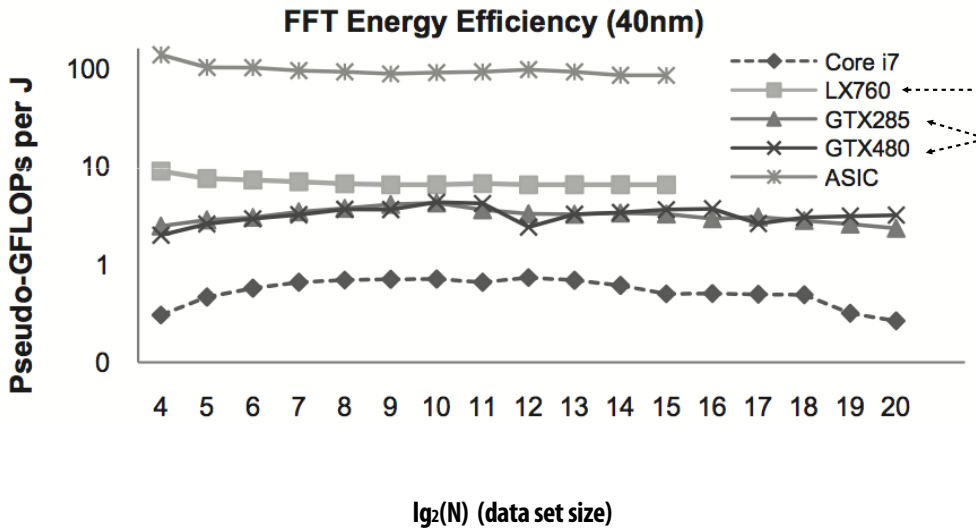
$$Power = \frac{Ops}{second} \times \frac{Joules}{Op}$$

FIXED

What is the magnitude of improvement from specialization?

Specialization (fixed function) ⇒ better energy efficiency

Fast Fourier transform (FFT): throughput and energy benefits of specialization

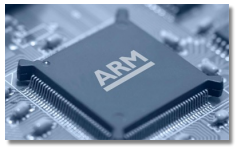


ASIC delivers same performance as one CPU core using only ~ 1/100th the power

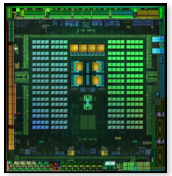
[Chung et al. MICRO 2010]

Choosing the right tool for the job

Energy-optimized CPU



Throughput-oriented processor (GPU)



Programmable DSP

HEXAGON™

Domain Specific Accelerator

Google TPU

FPGA/ reconfigurable logic

ASIC

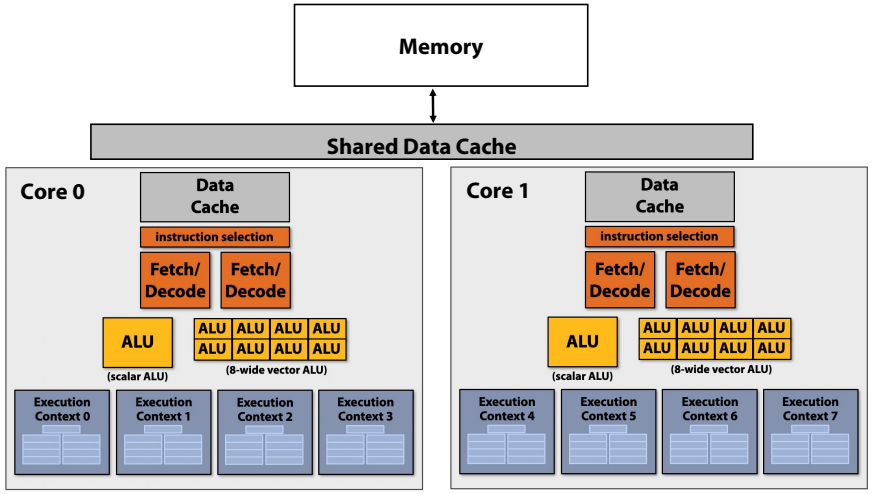
Video encode/decode, Audio playback, Camera RAW processing, neural nets (future?)

	~10X more efficient	~20X	~50X??? (jury still out)	~100-1000X more efficient
Easiest to program		Limited domain of programmability with DSLs (e.g. DNN)	Difficult to program (making it easier is active area of research)	Not programmable + costs 10-100's millions of dollars to design / verify / create

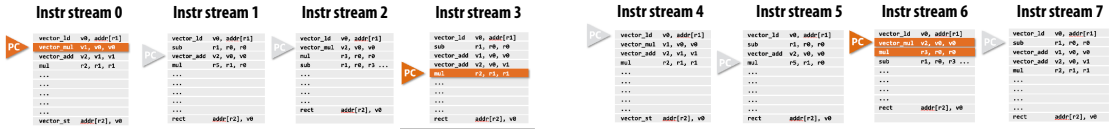
Credit: Pat Hanrahan for this slide design

Mapping Algorithms to Execution Resources

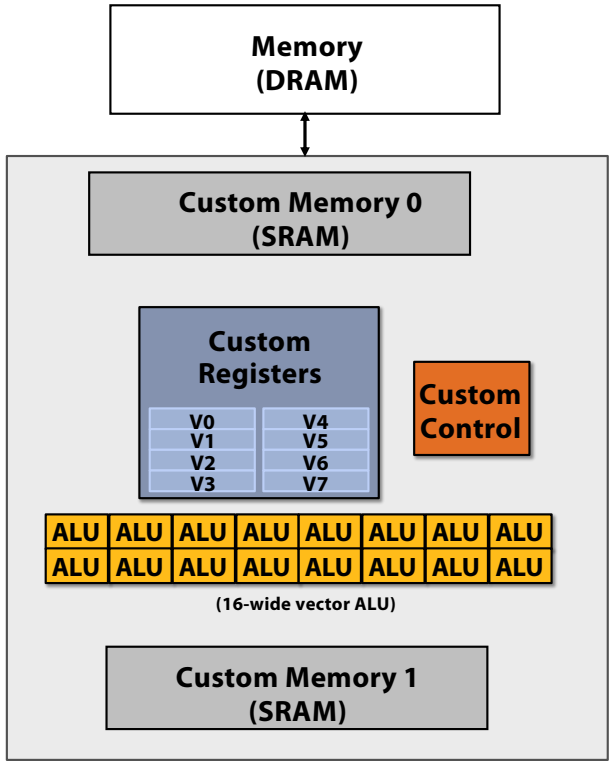
General Purpose Processor



Dual-core processor, multi-threaded cores (4 threads/core).
Two-way superscalar cores: each core can run up to two independent instructions per clock from any of its threads, provided one is scalar and the other is vector



Special Purpose Processor (Accelerator)



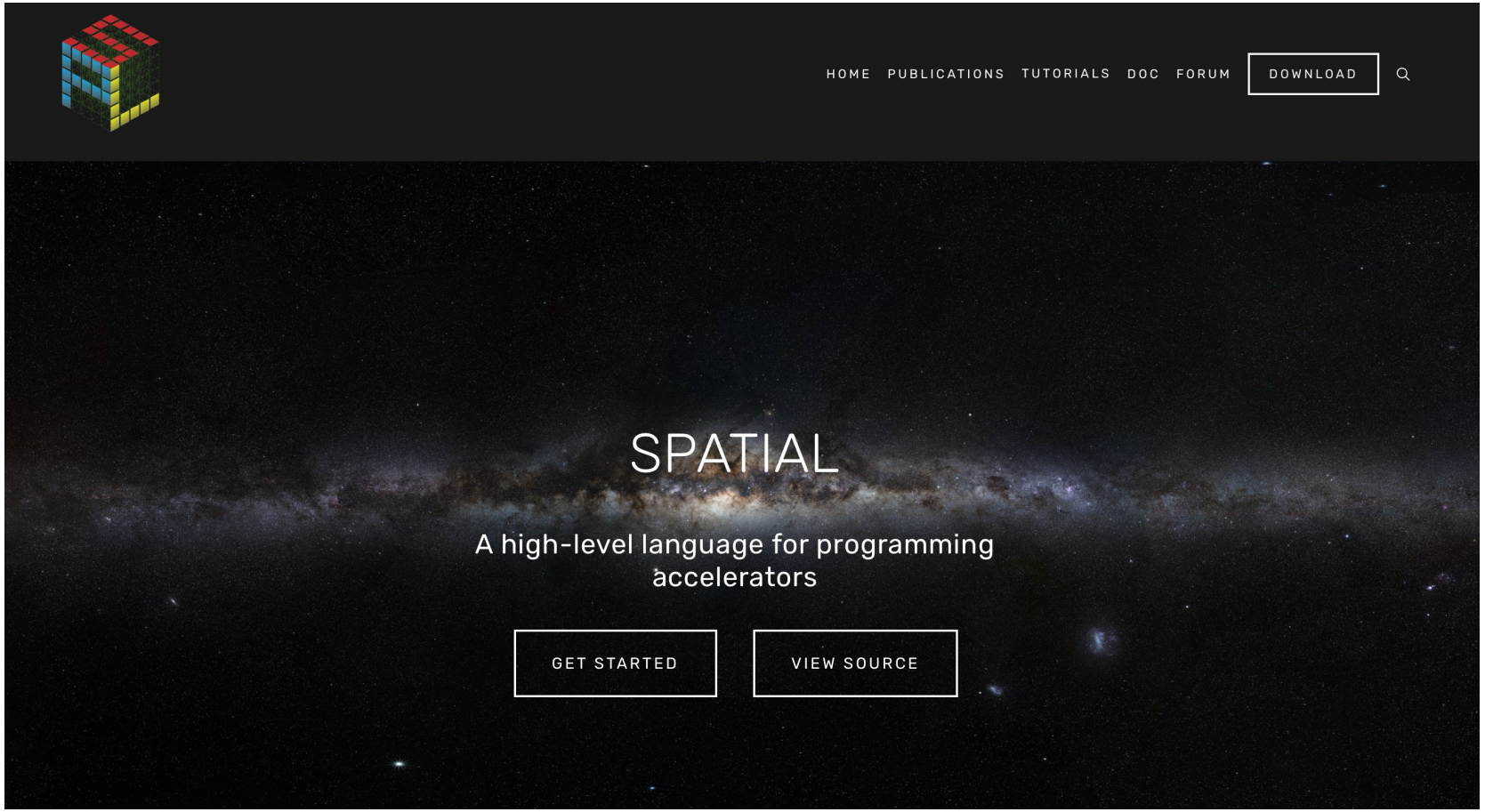
So You Want to Design an Accelerator for Your Algorithm

- Traditionally, you must spend years becoming an expert in VHDL or Verilog, Chisel…

- High-Level Synthesis (HLS): Vivado HLS, Intel OpenCL, and Xilinx SDAccel
  - Restricted C with pragmas
  - These tools sacrifice performance and are difficult to use

- Spatial is a high-level language for designing hardware accelerators that was designed to enable performance-oriented programmers to specify
  - Parallelism: specialized compute
  - Locality: specialized memories and data movement

Spatial-lang.org



Spatial: DSL for Accelerator Design

- Simplify configurable accelerator design
  - Constructs to express:
    - Parallel patterns as parallel and pipelined datapaths
    - hierarchical control
    - explicit memory hierarchies
    - Explicit parameters
  - All parameters exposed to the compiler
  - Simple APIs to manage CPU $\Leftrightarrow$ Accelerator communication

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

The Spatial Language: Memory Templates

Explicit memory hierarchy
Typed storage templates

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

Registers

```
val accum  = Reg[Double]
val fifo   = FIFO[Float](D)
val lbuf   = LineBuffer[Int](R,C)
val pixels = ShiftReg[UInt8](R,C)
```

Explicit transfers across memory hierarchy
Dense and sparse access

```
buffer load image(i, j::j+C)
buffer gather image(a, 10)
```

Streaming abstractions

```
val videoIn  = StreamIn[RGB]
val videoOut = StreamOut[RGB]
```

The Spatial Language: Control Templates

Blocking/non-blocking interaction with CPU

```
Accel { … }

Accel(*) { … }
```

Arbitrary state machine / loop nesting with implicit control signals

```
FSM[Int]{s => s != DONE }{
  case STATE0 =>
    Foreach(C by 1){j => … }
  case STATE1 => …
    Reduce(0)(C by 1){i => … }

}{s => nextState(s) }
```

The Spatial Language: Design Parameters

Spatial templates capture a variety of design parameters:

Explicit parallelization factors

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

Implicit/Explicit control schemes

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

Explicit size parameters for stride and buffer sizes

```
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
}
```

Inner Product

Code

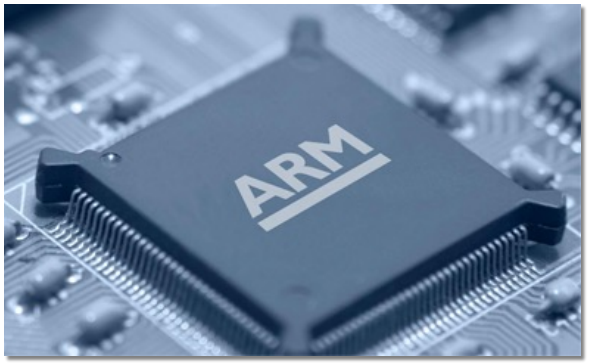
Let's build an accelerator to see how Spatial works

Sketch of generated hardware

Inner Product in C

```
// Set up accumulator and memory pointers
int output = 0;
int* vec1 = (int*)malloc(N * sizeof(int));
int* vec2 = (int*)malloc(N * sizeof(int));

// Iterate through data and accumulate
for (int i = 0; i < N; i++) {
  output = output + (vec1(i) * vec2(i));
}
```

Here is inner product written in C for a CPU



Inner Product in Spatial

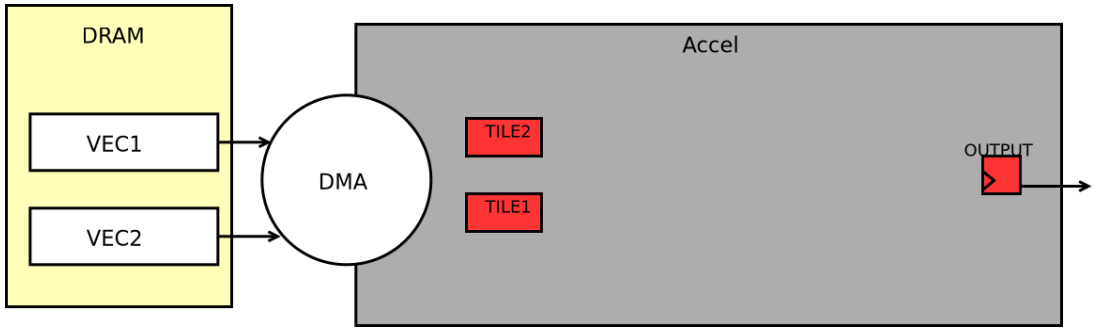
```
// Set up host and memory pointers
val output = ArgOut[Int]
val vec1 = DRAM[Int](N)
val vec2 = DRAM[Int](N)

// Create accelerator (instantiate hardware)
Accel {









}
```

Inner product in Spatial allows the programmer to build a hardware accelerator

- Start of code looks like C example
- Accel instantiates "for" loop in hardware

Inner Product in Spatial

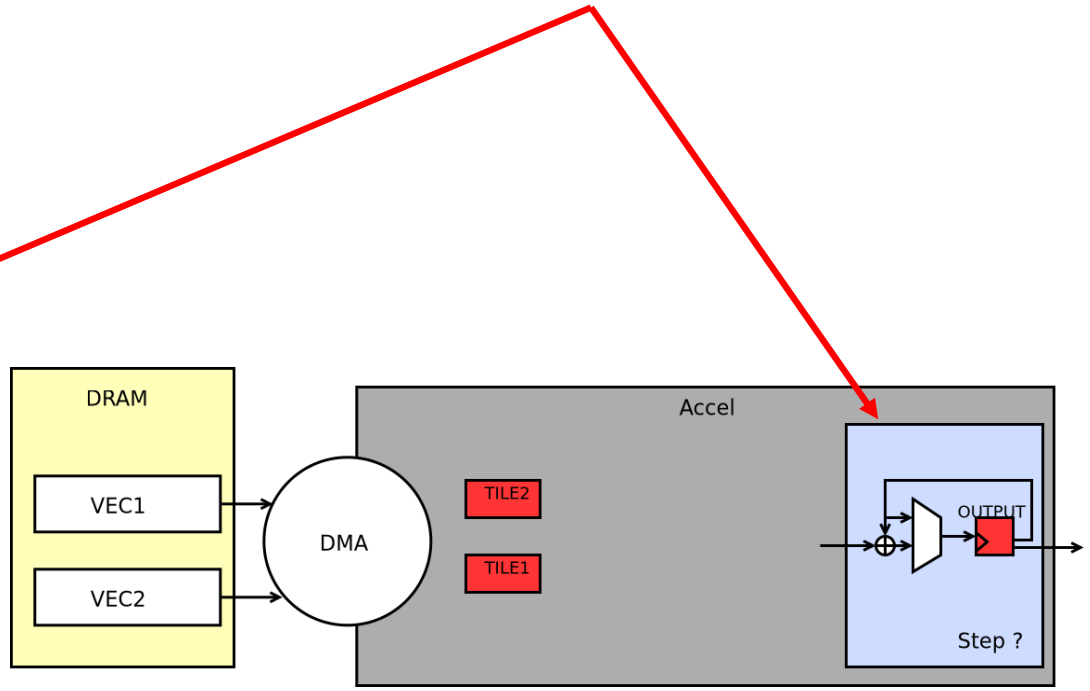
```
// Set up host and memory pointers
val output = ArgOut[Int]
val vec1 = DRAM[Int](N)
val vec2 = DRAM[Int](N)

// Create accelerator
Accel {
  // Allocate on-chip memories
  val tile1 = SRAM[Int](tileSize)
  val tile2 = SRAM[Int](tileSize)









}
```



Inner Product in Spatial

```
// Set up host and memory pointers
val output = ArgOut[Int]
val vec1 = DRAM[Int](N)
val vec2 = DRAM[Int](N)

// Create accelerator
Accel {
  // Allocate on-chip memories
  val tile1 = SRAM[Int](tileSize)
  val tile2 = SRAM[Int](tileSize)
  // Specify outer loop
  Reduce(output)(N by tileSize){ t =>
    // More controllers coming...









  }{a, b => a + b}
}
```

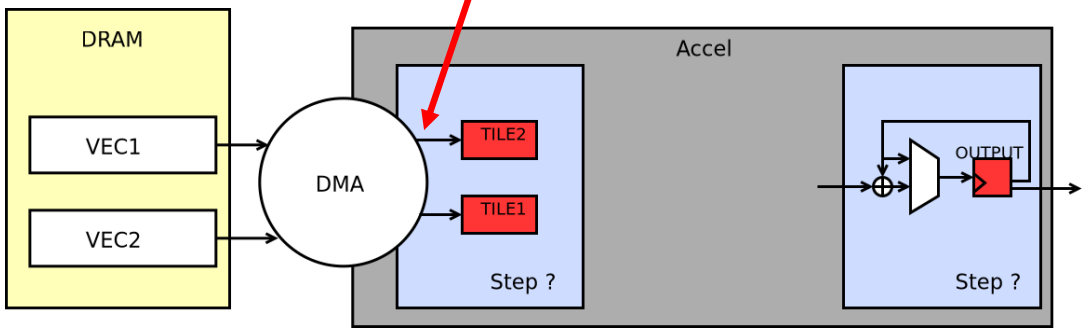
- **Spatial generates multi-step controllers (This Reduce controller's final step will handle the accumulation)**



Inner Product in Spatial

```
// Set up host and memory pointers
val output = ArgOut[Int]
val vec1 = DRAM[Int](N)
val vec2 = DRAM[Int](N)

// Create accelerator
Accel {
  // Allocate on-chip memories
  val tile1 = SRAM[Int](tileSize)
  val tile2 = SRAM[Int](tileSize)
  // Specify outer loop
  Reduce(output)(N by tileSize){ t =>
    // Prefetch data
    tile1 load vec1(t :: t + tileSize)
    tile2 load vec2(t :: t + tileSize)




  }{a, b => a + b}
}
```

- Spatial generates multi-step controllers
- Spatial manages communication with DRAM



Inner Product in Spatial

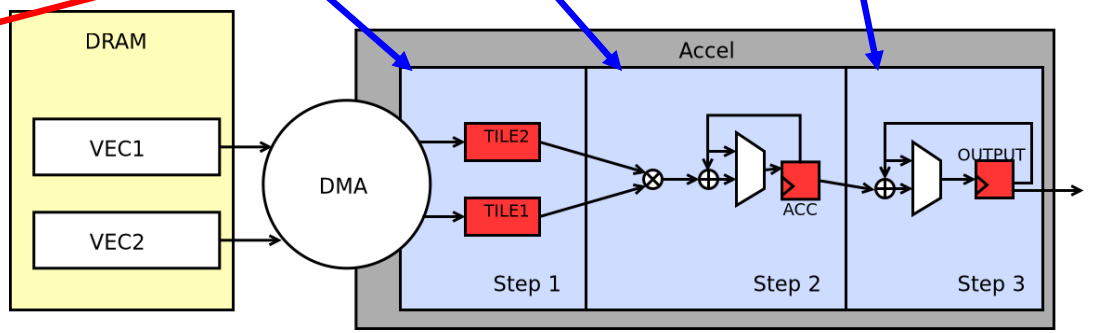
```
// Set up host and memory pointers
val output = ArgOut[Int]
val vec1 = DRAM[Int](N)
val vec2 = DRAM[Int](N)

// Create accelerator
Accel {
  // Allocate on-chip memories
  val tile1 = SRAM[Int](tileSize)
  val tile2 = SRAM[Int](tileSize)
  // Specify outer loop
  Reduce(output)(N by tileSize){ t =>
    // Prefetch data
    tile1 load vec1(t :: t + tileSize)
    tile2 load vec2(t :: t + tileSize)
    // Multiply-accumulate data
    val accum = Reg[Int](0)
    Reduce(accum)(tileSize by 1 par 1){ i =>
      tile1(i) * tile2(i)
    }{a, b => a + b}
  }{a, b => a + b}
}
```

- Spatial generates multi-step controllers
- Spatial manages communication with DRAM

The complete app generates a three-step control

Load → intra-tile accumulate → full accumulate

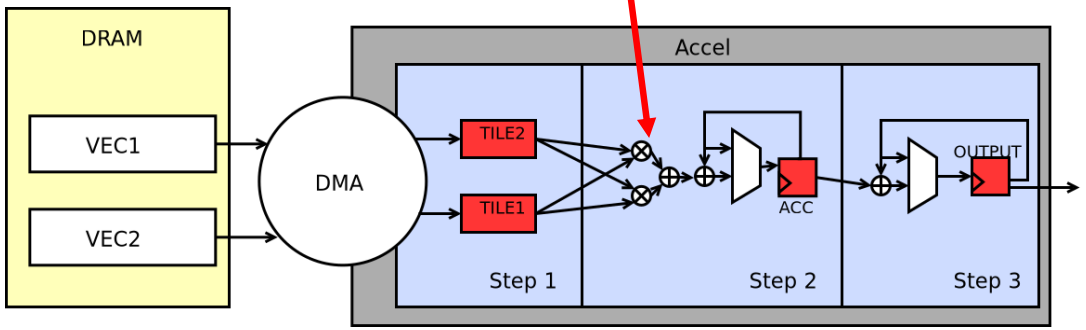
Where is the parallelism?



Inner Product in Spatial

```
// Set up host and memory pointers
val output = ArgOut[Int]
val vec1 = DRAM[Int](N)
val vec2 = DRAM[Int](N)

// Create accelerator
Accel {
  // Allocate on-chip memories
  val tile1 = SRAM[Int](tileSize)
  val tile2 = SRAM[Int](tileSize)
  // Specify outer loop
  Reduce(output)(N by tileSize){ t =>
    // Prefetch data
    tile1 load vec1(t :: t + tileSize)
    tile2 load vec2(t :: t + tileSize)
    // Multiply-accumulate data
    val accum = Reg[Int](0)
    Reduce(accum)(tileSize by 1 par 2){ i =>
      tile1(i) * tile2(i)
    }{ _ + _ }
  }{ _ + _ }
}
```

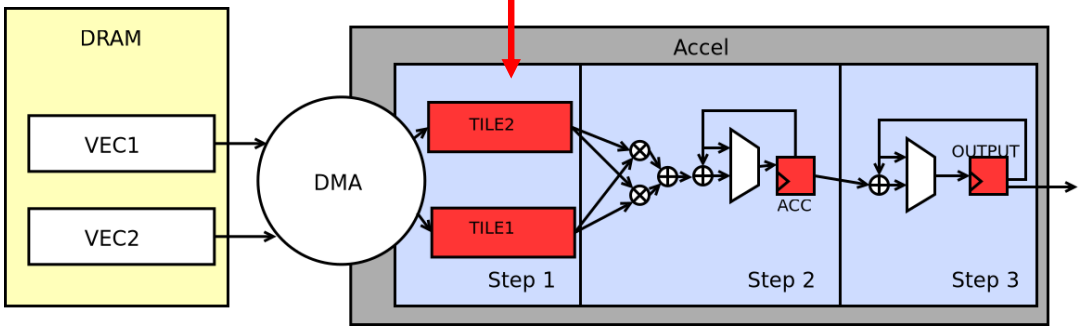
- Spatial generates multi-step controllers
- Spatial manages communication with DRAM
- Spatial helps express hardware datapaths



Inner Product in Spatial

```
// Set up host and memory pointers
val output = ArgOut[Int]
val vec1 = DRAM[Int](N)
val vec2 = DRAM[Int](N)
val bigTileSize = 2*tileSize
// Create accelerator
Accel {
  // Allocate on-chip memories
  val tile1 = SRAM[Int](bigTileSize)
  val tile2 = SRAM[Int](bigTileSize)
  // Specify outer loop
  Reduce(output)(N by bigTileSize){ t =>
    // Prefetch data
    tile1 load vec1(t :: t + bigTileSize)
    tile2 load vec2(t :: t + bigTileSize)
    // Multiply-accumulate data
    val accum = Reg[Int](0)
    Reduce(accum)(bigTileSize by 1 par 2){ i =>
      tile1(i) * tile2(i)
    }{ _ + _ }
  }{ _ + _ }
}
```

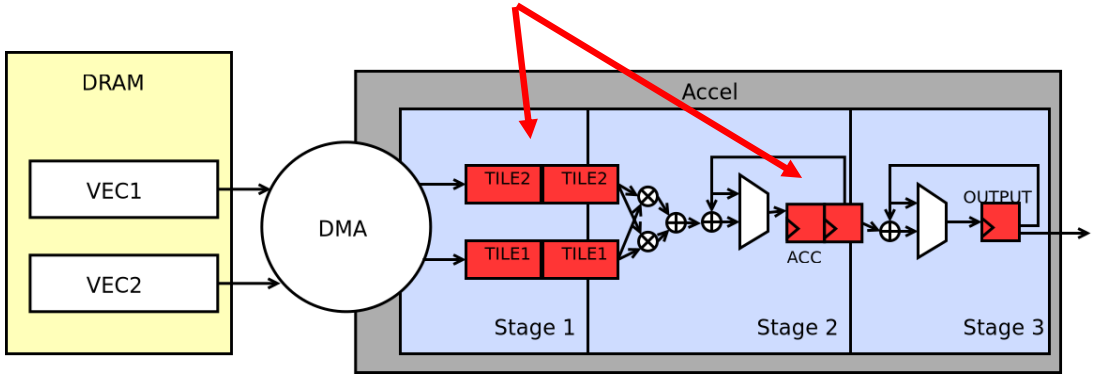
- Spatial generates multi-step controllers
- Spatial manages communication with DRAM
- Spatial helps express hardware datapaths
- Spatial makes it easy to tile



Inner Product in Spatial

```
// Set up host and memory pointers
val output = ArgOut[Int]
val vec1 = DRAM[Int](N)
val vec2 = DRAM[Int](N)

// Create accelerator
Accel {
  // Allocate on-chip memories
  val tile1 = SRAM[Int](tileSize)
  val tile2 = SRAM[Int](tileSize)
  // Specify outer loop
  Pipeline.Reduce(output)(N by tileSize){ t =>
    // Prefetch data
    tile1 load vec1(t :: t + tileSize)
    tile2 load vec2(t :: t + tileSize)
    // Multiply-accumulate data
    val accum = Reg[Int](0)
    Reduce(accum)(tileSize by 1 par 2){ i =>
      tile1(i) * tile2(i)
    }{ _ + _ }
  }{ _ + _ }
}
```

- Spatial generates multi-step controllers
- Spatial manages communication with DRAM
- Spatial helps express hardware datapaths
- Spatial makes it easy to tile
- Spatial lets the user manage scheduling
  - With annotation, steps (stages) execute in pipelined fashion. "Buffering" of memories is inferred



Controllers

- Every "loop" in Spatial is a controller (key parallel abstraction)
- **Controller** - Hardware counters whose values control **datapaths** or **other controllers**
- Controller hierarchy
  - **Inner Controller** - Datapath: consisting of *only* primitive nodes
    - arithmetic, if-then/mux, memory-access, etc.
  - **Outer Controller** - Other controllers

```
Foreach(N by 1) { i =>  // Outer controller
    Foreach(M by 1) { j => mem(i,j) = i+j }            // Inner controller
    Foreach(P by 1) { j => if (j == 0) … = mem(i,j) }  // Inner controller
}
```

Controller Performance

The execution time of a single controller is:

$$T = II * (iters - 1) + L$$

T = Cycles per execution

II = Initiation interval

iters = Number of iterations

L = Latency of the datapath elements

However, II and L have slightly different meanings depending on a controller's level (inner vs outer)

Inner Controllers

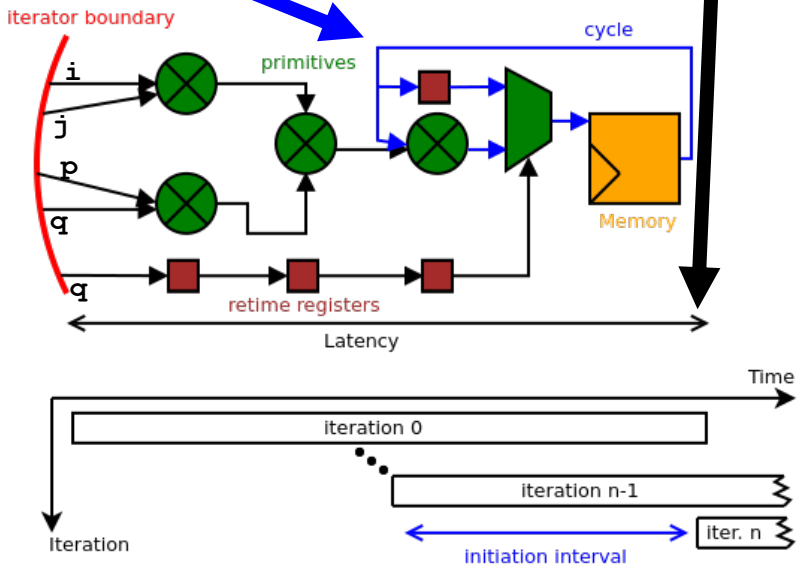
Inner controllers always execute iterations in a pipelined (overlapped) manner

Initiation interval (II): the length of the longest cycle in dataflow graph

Latency (L): the longest path from the loop iterators to the final node

$$T = \mathbf{II} * (iters - 1) + \mathbf{L}$$

```
Foreach(N by 1, M by 1, P by 1, Q by 1){(i,j,p,q) =>
    val sum = i + j + p + q
    val next = reg.value ^ sum
    reg := mux(q == 0, reg.value, next)
}
```



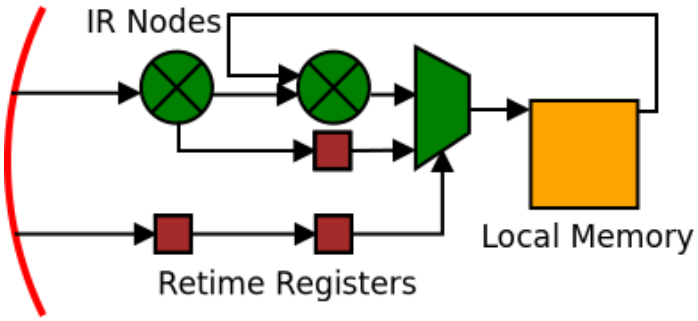
Inner Controller Parallelization

Parallelization of **inner controllers** results in vectorization of the counter chain and duplication of the dataflow graph

Abstraction: parallel

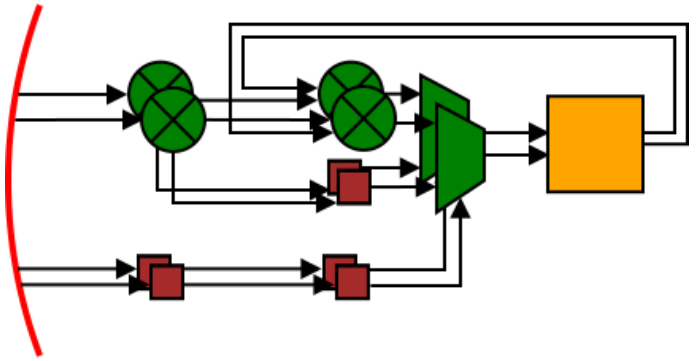
Implementation: vectors

```
Foreach(N by 1 par 1){ i => ... }
```



Increase parallelization

```
Foreach(N by 1 par 2){ i => ... }
```



Outer Controllers

Initiation interval and latency for **outer (parent) controllers** depends on their "schedule," which we will introduce next

We will refer to these properties as "effective" initiation interval and "effective" latency

$$T = II_{\text{eff}} * (iters - 1) + L_{\text{eff}}$$

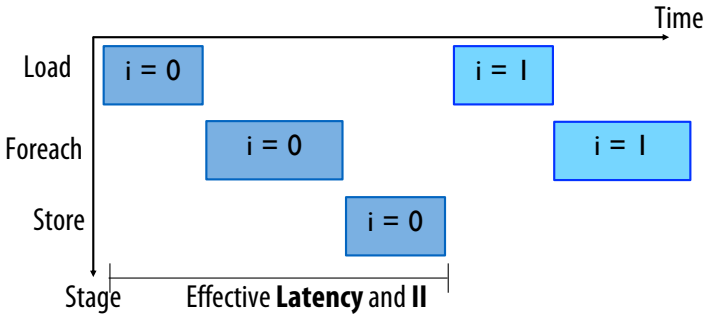
Scheduling Outer Controllers

There are four major schedules for outer controllers:

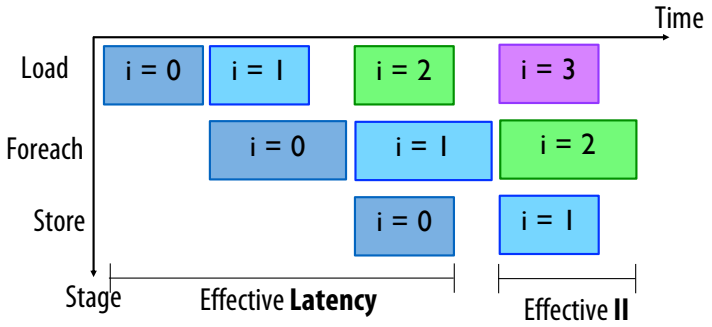
- **Sequential** – No overlapping of inner (child) controllers
- **Pipelined** – Coarse-grained overlapping of inner (child) controllers
- **Stream** – Data-driven execution of inner (child) controllers
- **Fork-Join** – Parallel execution of all inner (child) controllers

A Look at Schedules

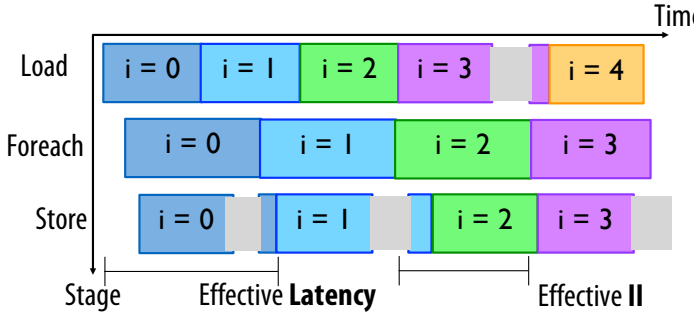
```
Sequential.Foreach(…){i =>
  sram load dram
  Foreach(M by 1){ j => sram2(j) = sram(j) * j }
  dram store sram2
}
```




```
Pipelined.Foreach(…){i =>
  sram load dram
  Foreach(M by 1){ j => sram2(j) = sram(j) * j }
  dram2 store sram2
}
```



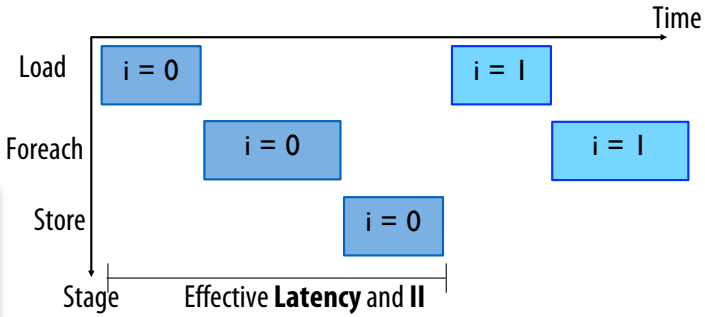

```
Stream.Foreach(…){i =>
  fifoIn load dram
  Foreach(M by 1){ j => fifoOut.enq(fifoIn.deq() * j)
}
  dram2 store fifoOut
}
```



A Closer Look at Schedules

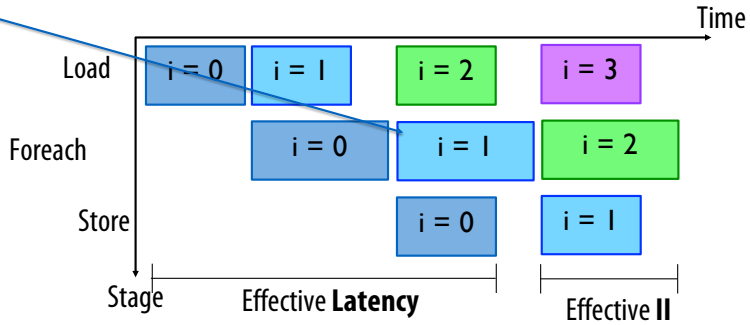
```
Sequential.Foreach(…){i =>
  sram load dram
  Foreach(M by 1){ j => sram2(j) = sram(j) * j }
  dram store sram2
}
```

When the pipeline is full, it is in **steady-state** and the longest stage determines II



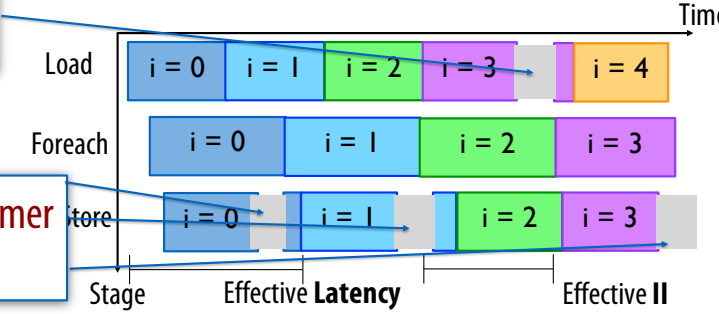

```
Pipelined.Foreach(…){i =>
  sram load dram
  Foreach(M by 1){ j => sram2(j) = sram(j) * j }
  dram2 store sram2
}
```

When an intermediate FIFO is full, the producer stage is **stalled**.




```
Stream.Foreach(…){i
  fifoIn load dram
  Foreach(M by 1){ j => fifoOut.enq(fifoIn.deq() * j)
}
  dram2 store fifoOut
}
```

When an intermediate FIFO is empty, the consumer stage is **starved**.



Spatial vs. Chisel (HDL)

```
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}
}
```

Spatial: ~30 lines

Chisel: ~3200 lines

Spatial Question

- **Spatial programmer's responsibility**

- **Spatial systems responsibility**

The execution time equation and schedules are important, but understanding the controller hierarchy and how to optimize the execution time of the hierarchy is the key to designing good accelerators

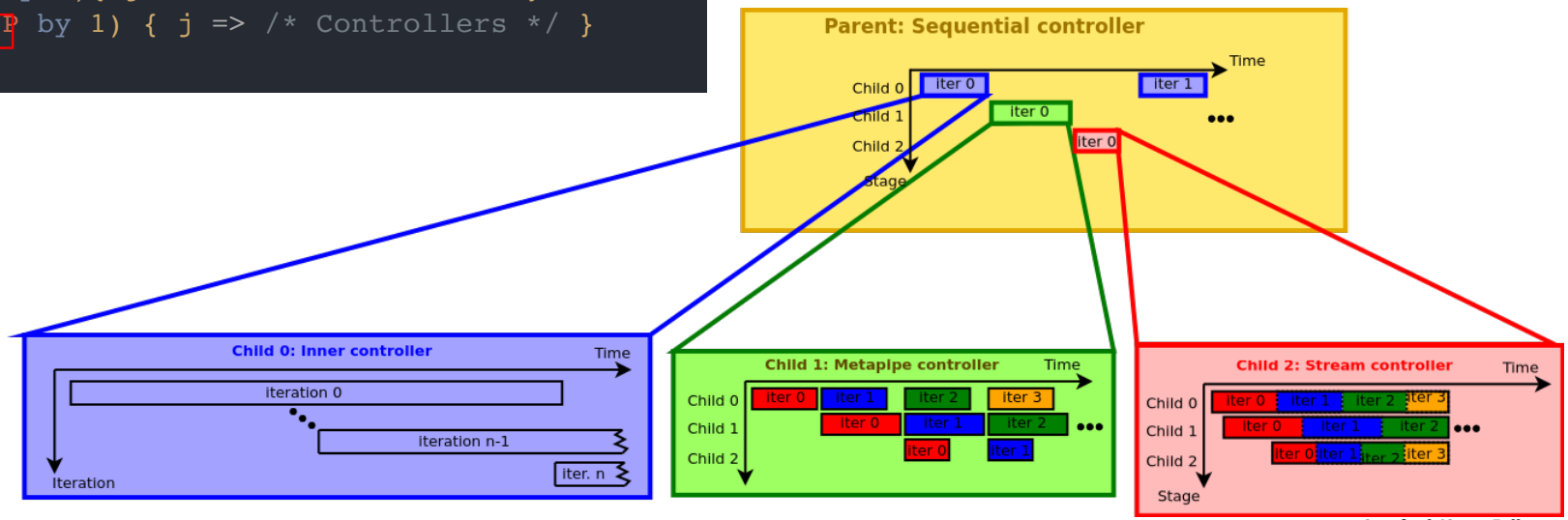
Let's talk about performance debugging

Performance Debugging with Timing Diagrams

We want to minimize the execution time of the **Parent Sequential Controller**

The controller timing diagram looks like this:

```
Sequential.Foreach(Q by TS){ i =>
    Foreach(N by 1){ j => /* Primitives */ }
    Pipe.Foreach(M by 1){ j => /* Controllers */ }
    Stream.Foreach(P by 1) { j => /* Controllers */ }
}
```

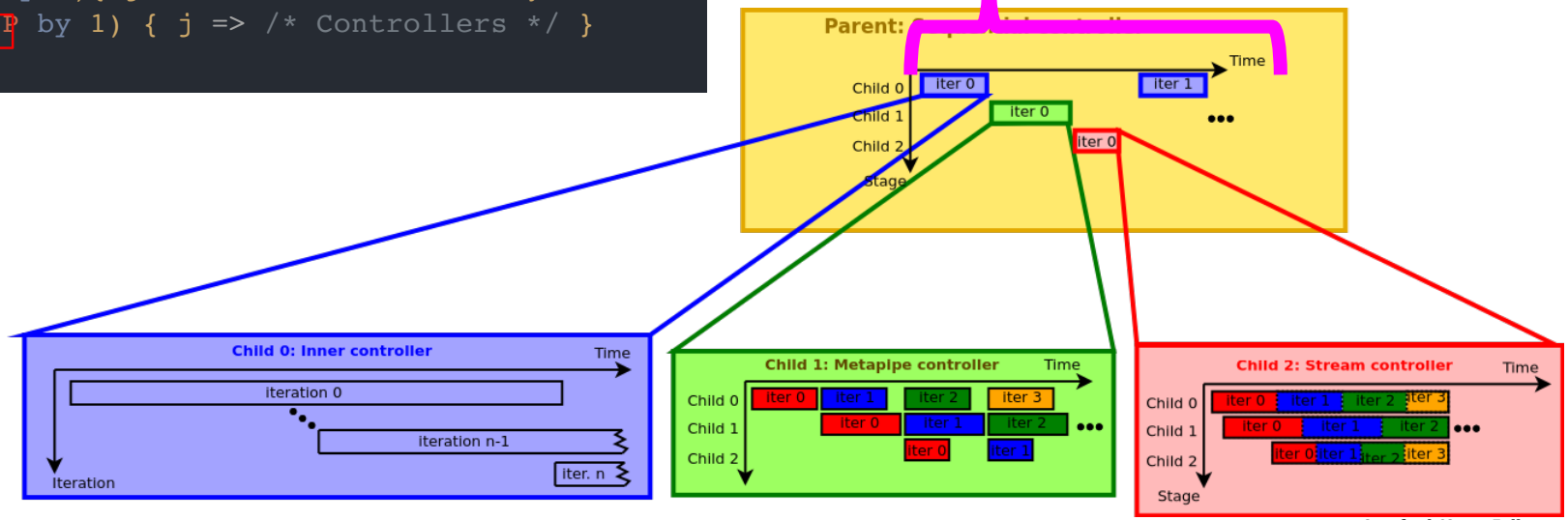


Performance Debugging

We want to minimize the execution time of the **Parent Sequential Controller**

$$T_{\text{parent}} = II_{\text{eff}} * (\text{iters} - 1) + L_{\text{eff}}$$

```
Sequential.Foreach(Q by TS){ i =>
    Foreach(N by 1){ j => /* Primitives */ }
    Pipe.Foreach(M by 1){ j => /* Controllers */ }
    Stream.Foreach(P by 1) { j => /* Controllers */ }
}
```



Performance Debugging

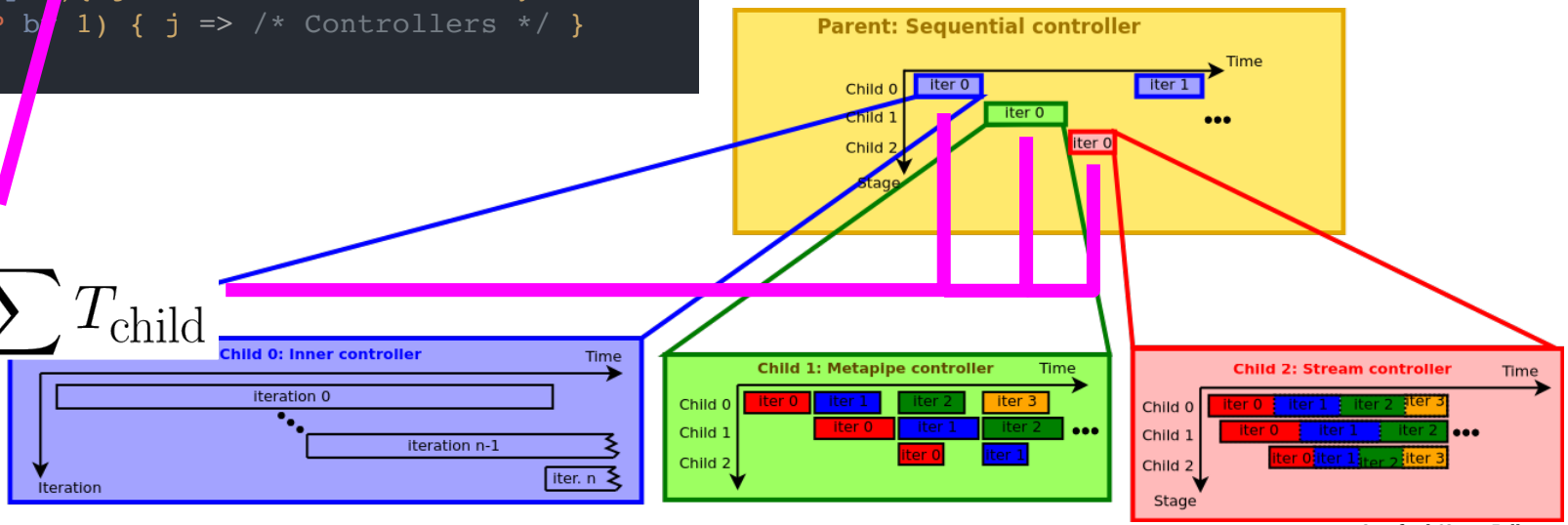
We want to minimize the execution time of the **Parent Sequential Controller**

$$T_{\text{parent}} = \mathbf{II}_{\text{eff}} * (\text{iters} - 1) + \mathbf{L}_{\text{eff}}$$

```
Sequential.Foreach(Q by TS){ i =>
    Foreach(N by 1){ j => /* Primitives */ }
    Pipe.Foreach(N by 1){ j => /* Controllers */ }
    Stream.Foreach(P by 1) { j => /* Controllers */ }
}
```

$$\text{iters} = \frac{Q}{TS}$$

$$\mathbf{II}_{\text{eff}} = \mathbf{L}_{\text{eff}} = \sum T_{\text{child}}$$

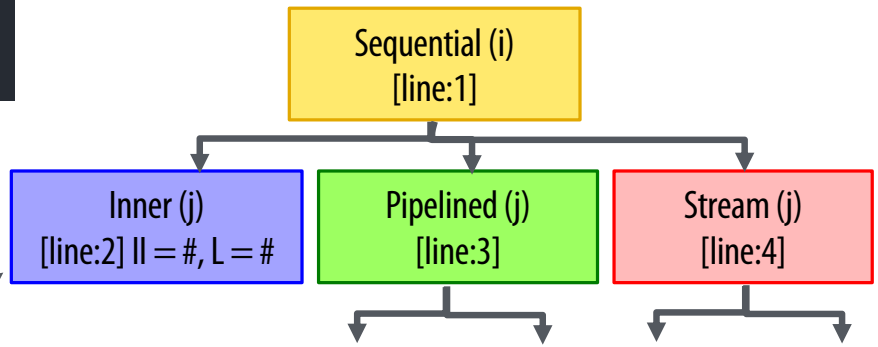


Performance Debugging with Controller Hierarchy

The controller hierarchy is a more concise way to understand performance

Spatial complier **automatically** generates the hierarchy for your application

```
1: Sequential.Foreach(Q by TS){ i =>
2:    Foreach(N by 1){ j => /* Primitives */ }
3:    Pipe.Foreach(M by 1){ j => /* Controllers */ }
4:    Stream.Foreach(P by 1) { j => /* Controllers */ }
}
```



Line numbers link controllers back to source code

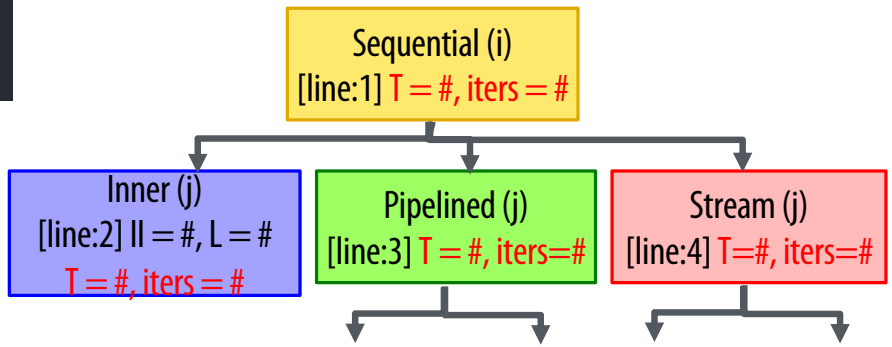
Static properties (II and L for inner controllers) are reported immediately

Controller Hierarchy Performance Debugging

The controller hierarchy is a more concise way to understand performance

Spatial **automatically** generates these trees for your application

```
1: Sequential.Foreach(Q by TS){ i =>
2:    Foreach(N by 1){ j => /* Primitives */ }
3:    Pipe.Foreach(M by 1){ j => /* Controllers */ }
4:    Stream.Foreach(P by 1) { j => /* Controllers */ }
}
```

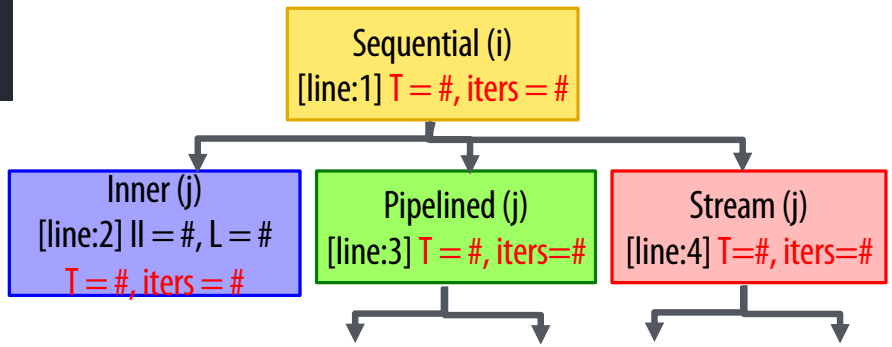


Actual T and iteration counts are automatically collected and overlaid after execution

Controller Hierarchy Performance Debugging

How do you use T and iteration counts effectively?

```
1: Sequential.Foreach(Q by TS){ i =>
2:    Foreach(N by 1){ j => /* Primitives */ }
3:    Pipe.Foreach(M by 1){ j => /* Controllers */ }
4:    Stream.Foreach(P by 1) { j => /* Controllers */ }
}
```



Performance Debugging

One of the most basic tools for improving performance is **parallelization**, which decreases the *iters* of a controller

$$T = II * (iters - 1) + L$$

Parallelization with Spatial's programming model has different meanings for **inner** and **outer** controllers

Optimization Example

- The programmer can use parallelization and controller schedule directives to explore the tradeoff between resource utilization and performance
- Let's revisit our inner product accelerator

Inner Product Optimization Example

```
// Inner product accelerator
Accel {
  val tile1 = SRAM[Int](tileSize)
  val tile2 = SRAM[Int](tileSize)
  // Outer reduce
  Reduce(output)(N by tileSize par ParOut){ t =>
    // Prefetch data
    tile1 load dram1(t :: t + tileSize)
    tile2 load dram2(t :: t + tileSize)
    // Multiply-accumulate data
    val accum = Reg[Int](0)
    Reduce(accum)(tileSize by 1 par ParIn){ i =>
      tile1(i) * tile2(i)
    }{a, b => a + b}
  }{a, b => a + b}
}
```

We will track resource utilization and performance as we tune these parameters:

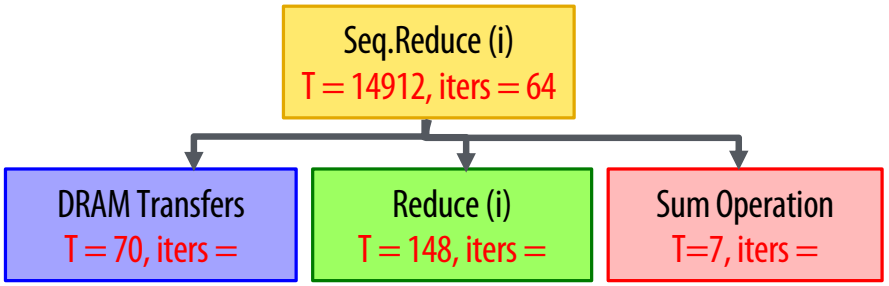
- Outer controller schedule (`Reduce`)
- `ParOut`
- `ParIn`

Inner Product Controller Hierarchy

```
// Inner product accelerator
Accel {
  val tile1 = SRAM[Int](tileSize)
  val tile2 = SRAM[Int](tileSize)
  // Outer reduce
  Reduce(output)(N by tileSize par ParOut){ t =>
    // Prefetch data
    tile1 load dram1(t :: t + tileSize)
    tile2 load dram2(t :: t + tileSize)
    // Multiply-accumulate data
    val accum = Reg[Int](0)
    Reduce(accum)(tileSize by 1 par ParIn){ i =>
      tile1(i) * tile2(i)
    }{a, b => a + b}
  }{a, b => a + b}
}
```

The baseline implementation is ParIn=1, ParOut=1, and schedule=Sequential

Our instrumented controller tree will look like this



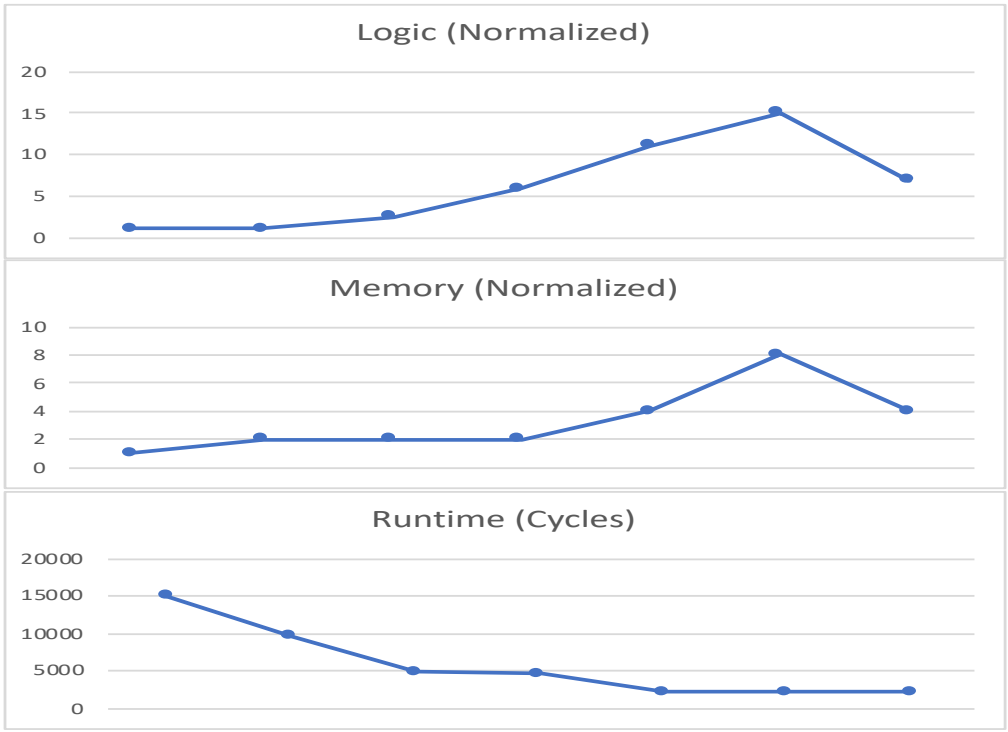
Optimization Goal

```
// Inner product accelerator
Accel {
  val tile1 = SRAM[Int](tileSize)
  val tile2 = SRAM[Int](tileSize)
  // Outer reduce
  Reduce(output)(N by tileSize par ParOut){ t =>
    // Prefetch data
    tile1 load dram1(t :: t + tileSize)
    tile2 load dram2(t :: t + tileSize)
    // Multiply-accumulate data
    val accum = Reg[Int](0)
    Reduce(accum)(tileSize by 1 par ParIn){ i =>
      tile1(i) * tile2(i)
    }{a, b => a + b}
  }{a, b => a + b}
}
```

- Understand impact on
  - execution time (cycles)
  - logic resources (arithmetic nodes)
  - memory resources (bytes)

Trailer: Inner Product Optimization

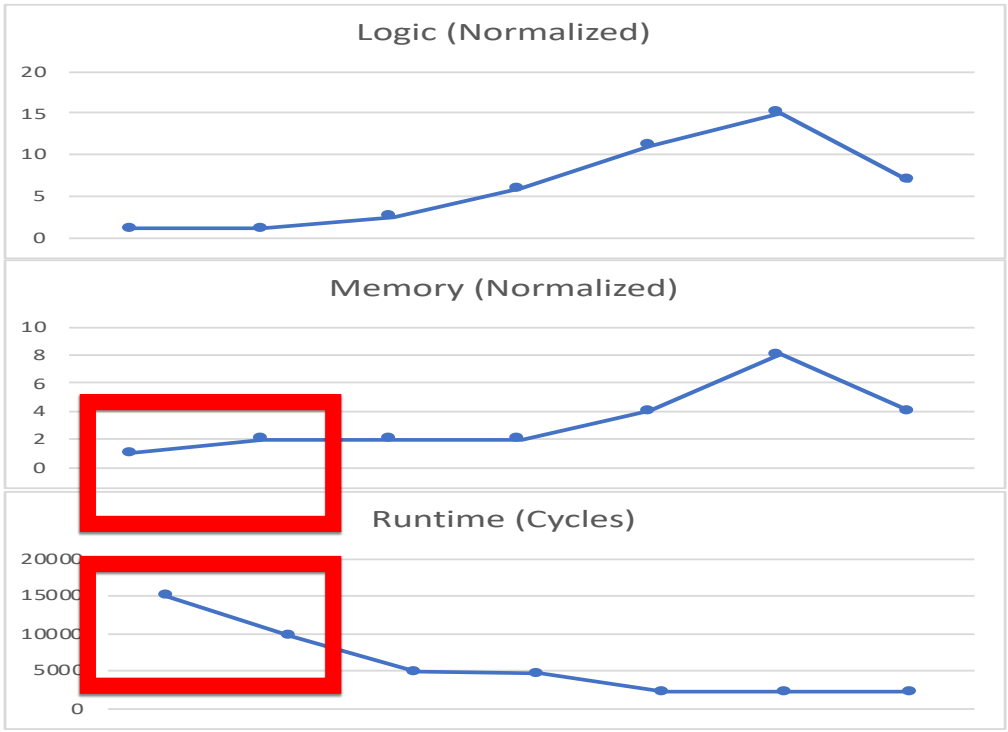
- By optimizing the code, we can improve execution time by ~7x
- The best design increases logic by ~6x and memory by ~4x



ParIn	1	1	2	4	4	4	2
ParOut	1	1	1	1	2	4	2
Sched	Seq	Pipe	Pipe	Pipe	Pipe	Pipe	Pipe

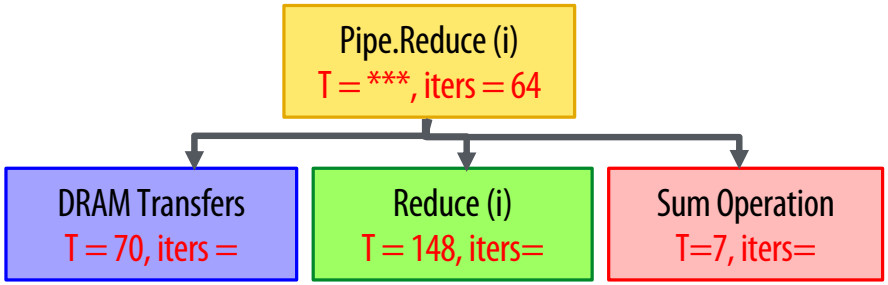
Sequential vs. Pipelined

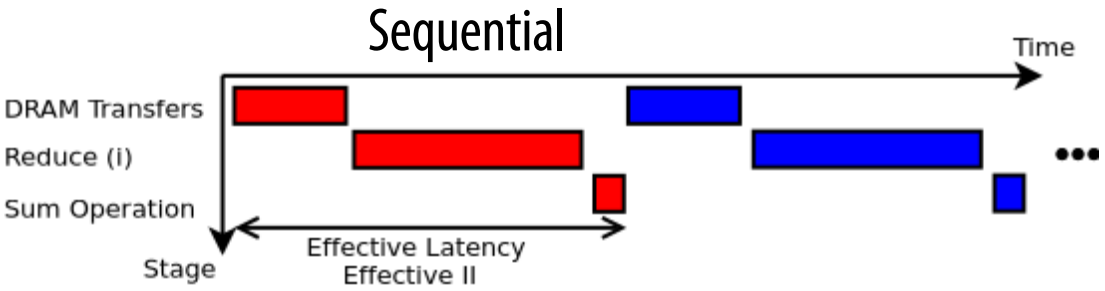
- Scheduling the outer controller as a Pipelined controller, rather than a Sequential controller, yields some performance improvement
- There is an increase in memory utilization due to buffering between stages
- There is no logic increase since we are not changing the datapaths

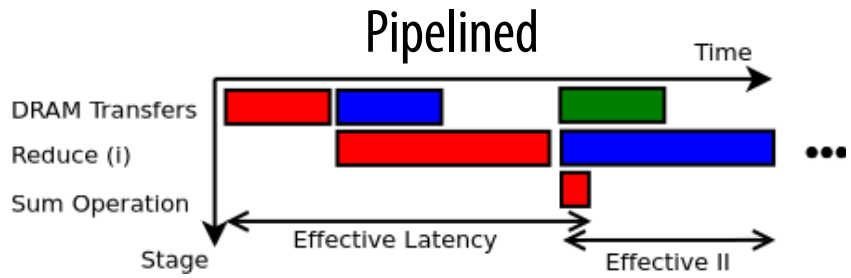


ParIn	1	1	2	4	4	4	2
ParOut	1	1	1	1	2	4	2
Sched	Seq	Pipe	Pipe	Pipe	Pipe	Pipe	Pipe

Sequential vs. Pipelined







- Understanding how the performance debugger maps to timing diagrams explains this performance boost from pipelining
  - Color corresponds to iteration in diagrams
- For the Sequential case, $II_{eff} \approx \sum T_c$
- For the Pipelined case, $II_{eff} \approx \max(T_c)$

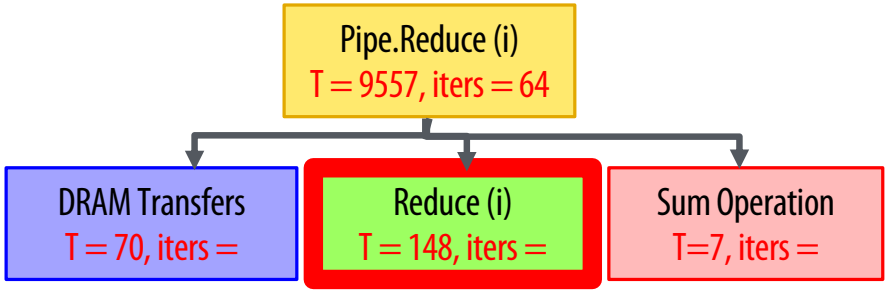
$$T = II_{\text{eff}} * (iters - 1) + L_{\text{eff}}$$

Inner Parallelization

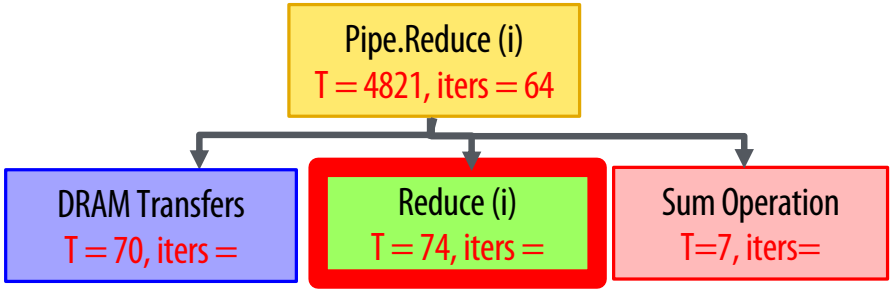
- There was a performance improvement from ParIn=1 to ParIn=2
  - Improved bottleneck of the pipeline
- From ParIn=2 to ParIn=4, we consume more logic but did not see much speedup
  - Inner reduce is no longer the bottleneck

ParIn	1	1	2	4	4	4	2
ParOut	1	1	1	1	2	4	2
Sched	Seq	Pipe	Pipe	Pipe	Pipe	Pipe	Pipe

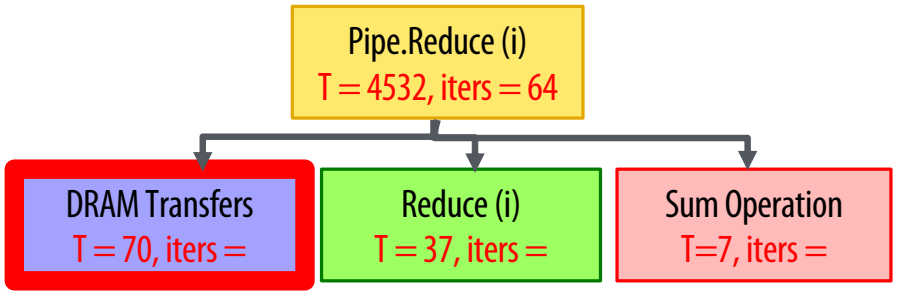
Inner Parallelization



ParIn = 1



ParIn = 2



ParIn = 4

- The performance debugger explains what happened
- The bottleneck stage improves as a result of this parallelization
- There is still a *small* performance improvement for Parl=4 because $II_{eff} \approx \max(T_c)$ decreases a bit

$$T = II_{\text{eff}} * (iters - 1) + L_{\text{eff}}$$

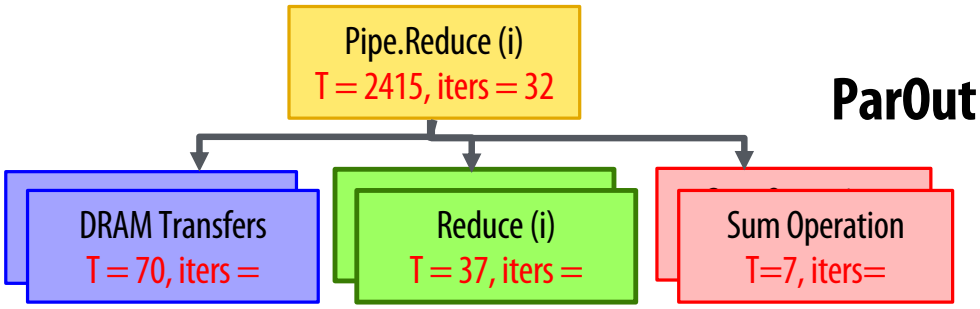
Outer Parallelization

- There is a performance improvement from ParOut=1 to ParOut=2 since we are increasing the off-chip data bandwidth by using more DMA channels
  - Increased both logic and memory since we duplicate the entire accelerator

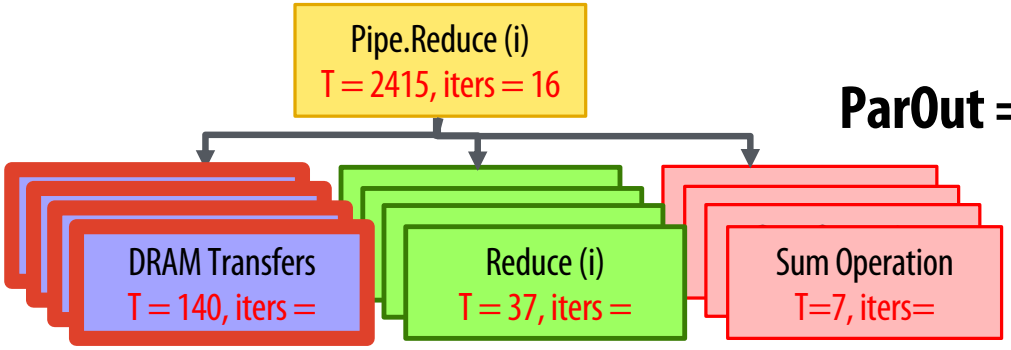
- From ParOut=2 to ParOut=4, the app becomes memory-bound
  - Change increases resource utilization without improving performance

ParIn	1	1	2	4	4	4	2
ParOut	1	1	1	**1**	**2**	**4**	2
Sched	Seq	Pipe	Pipe	Pipe	Pipe	Pipe	Pipe

Outer Parallelization



ParOut = 2

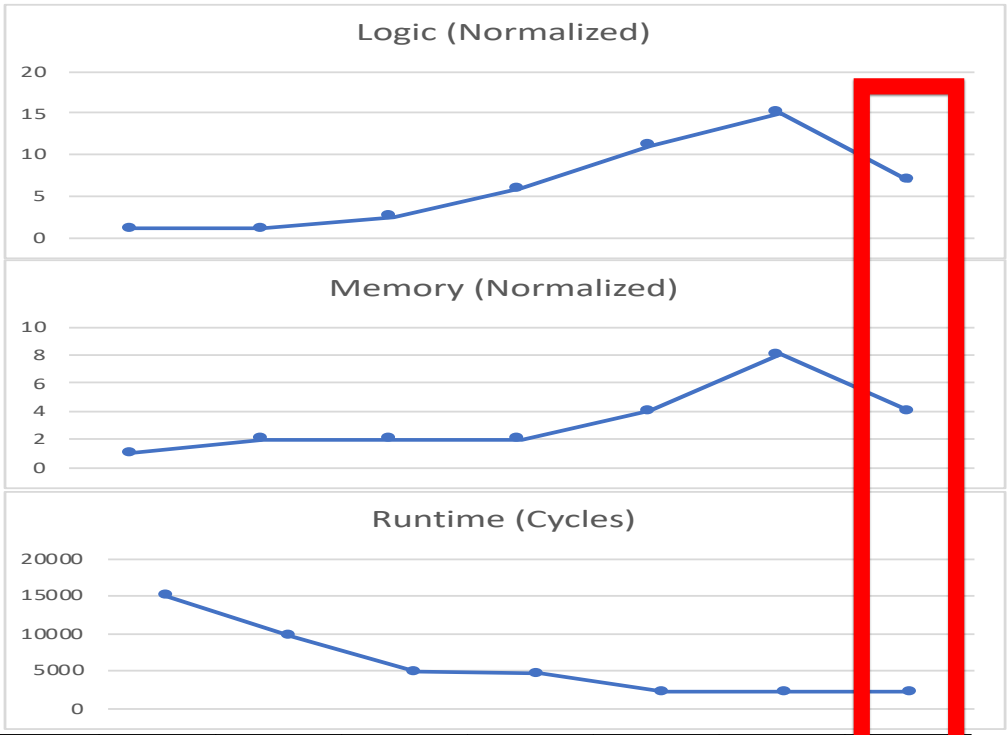


ParOut = 4

- The Reduce and Sum Operation stages are statically scheduled
- The DRAM Transfers stages compete for the DRAM and execute when DRAM returns data
- When ParOut doubles, Pipe.Reduce runs for half as many iterations
  - T does not change because the DRAM Transfers stages runs for twice as long
- This indicates that the DRAM has enough bandwidth to support ParOut=2 but not ParOut=4

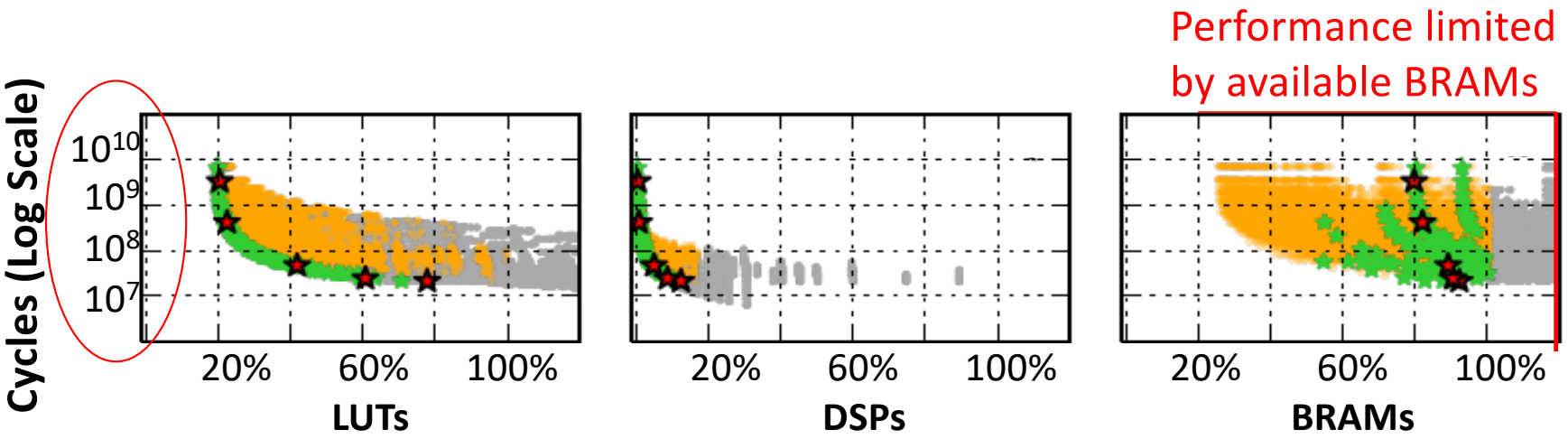
Performance vs. Resources

- The best design has the shortest execution time and uses the fewest resources
- Scale back some parallelization factors to get a better design
- By optimizing the code, we can improve execution time by ~7x
  - The best design increases logic by ~6x and memory by ~4x



ParIn	1	1	2	4	4	4	2
ParOut	1	1	1	1	2	4	2
Sched	Seq	Pipe	Pipe	Pipe	Pipe	Pipe	Pipe

Spatial GDA Design Space Exploration



Space for GDA spans
four orders of magnitude

Valid design point
Invalid design point
Pareto-optimal (ALMs/cycles) design
Synthesized pareto design point

Accelerator Design Summary

- Significant energy efficiency improvements from specialized accelerators (100x–1000x)

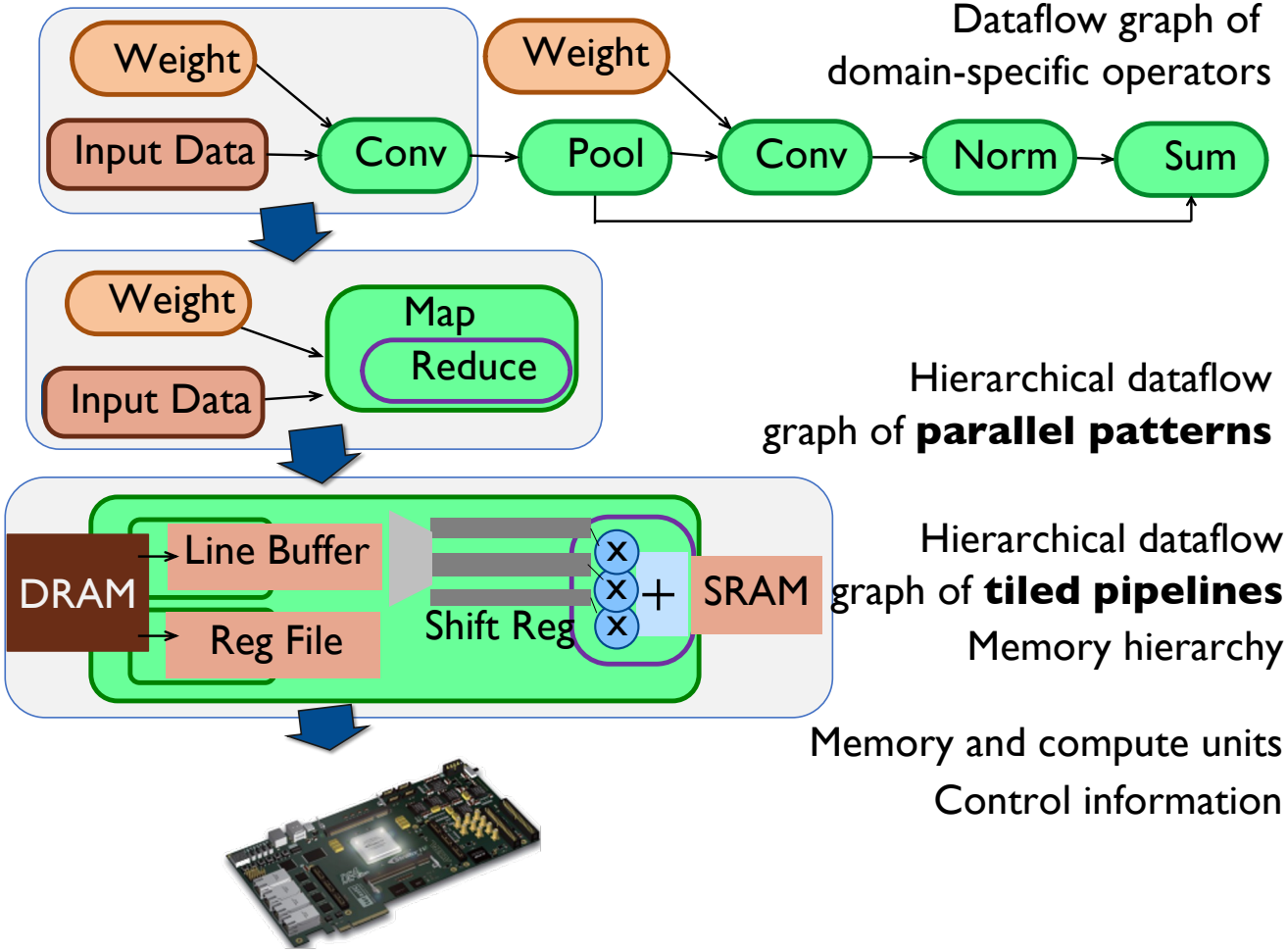
- Designing an accelerator is a tradeoff between performance and resource utilization
  - Parallelism
  - Locality

- It requires the programmer to have insight into the application
  - Where is the bottleneck
  - Is the implementation compute or memory-bound

- Spatial helps you understand the trade-off between performance and resource utilization
  - Allows rapid exploration of your algorithm
  - Enables high-level accelerator design

- ~7x performance improvement for the simple inner product acceleration

TensorFlow to FPGA

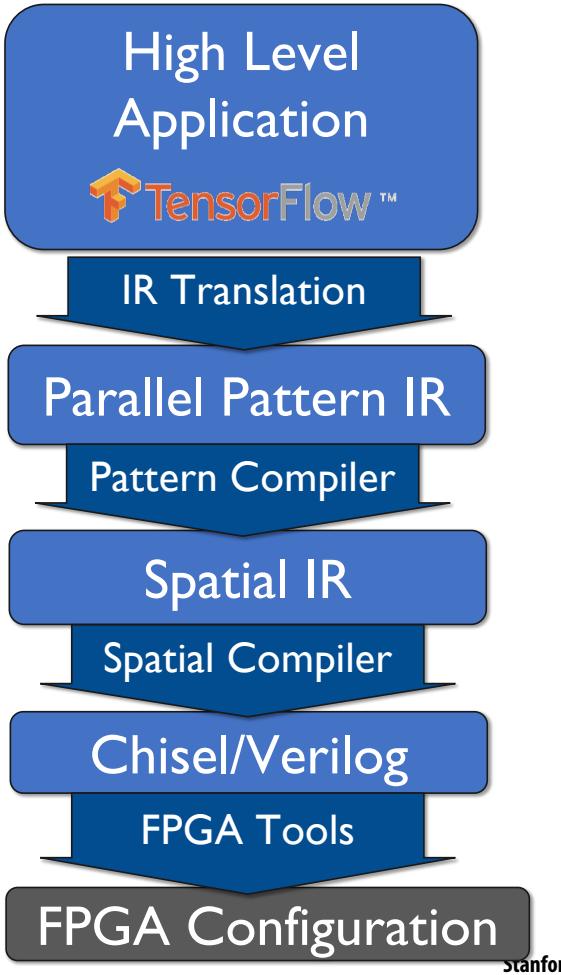


Dataflow graph of domain-specific operators

Hierarchical dataflow graph of **parallel patterns**

Hierarchical dataflow graph of **tiled pipelines**
Memory hierarchy

Memory and compute units
Control information



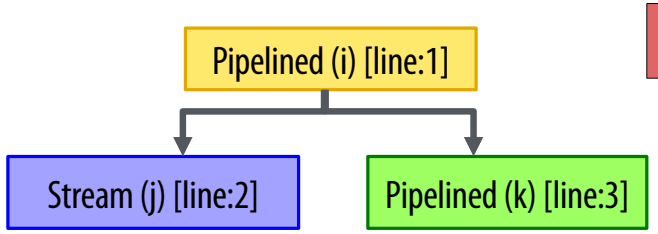
Outer Controller Parallelization

Parallelization of **outer controllers** results in duplication of all child controllers and insertion of synchronization controllers (ForkJoin)

Each duplicate child receives only one lane of the parent counter chain

```
Pipe.Foreach(Q by 1 par 1){ i =>
    Stream.Foreach(M by 1){ j => ... }
    Pipe.Foreach(M by 1){ k => ... }
}
```

```
Pipe.Foreach(Q by 1 par 2){ i =>
    Stream.Foreach(M by 1){ j => ... }
    Pipe.Foreach(M by 1){ k => ... }
}
```



Increase parallelization