

Lecture 17:

# Hardware Specialization and Spatial Programming

---

Parallel Computing  
Stanford CS149, Fall 2020

# Energy-constrained computing

# Performance and Power

$$\text{Power} = \frac{\text{Performance}}{\text{Energy efficiency}} = \frac{\text{Ops}}{\text{second}} \times \frac{\text{Joules}}{\text{Op}}$$

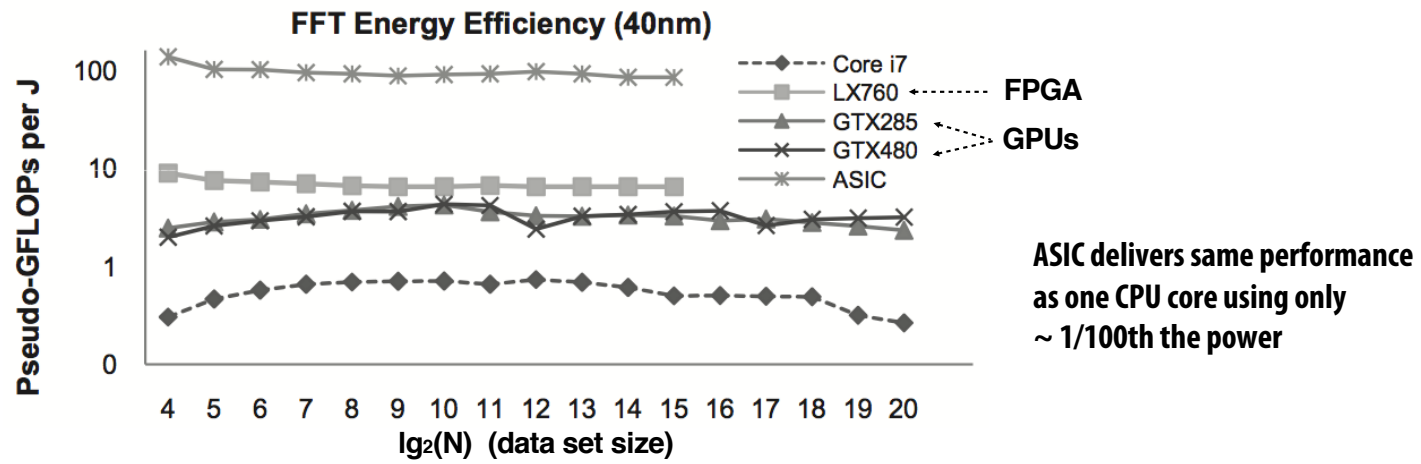
**FIXED**



Specialization (fixed function)  $\Rightarrow$  better energy efficiency

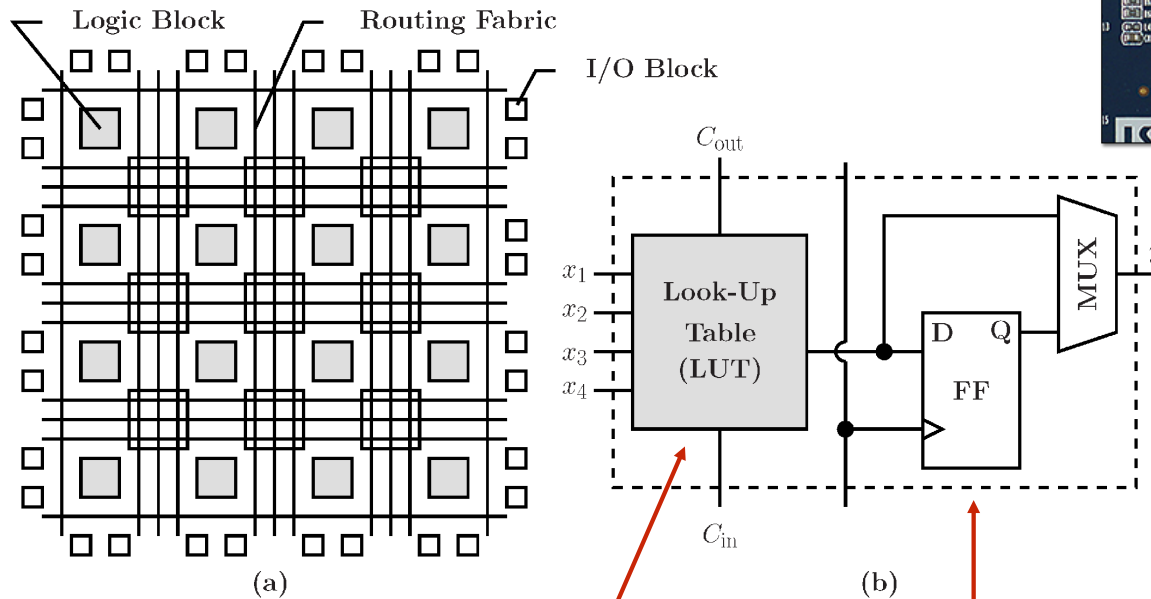
What is the magnitude of improvement from specialization?

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



# FPGAs (Field Programmable Gate Arrays)

- Middle ground between an ASIC and a processor
- FPGA chip provides array of logic blocks, connected by interconnect
- Programmer-defined logic implemented directly by FPGA

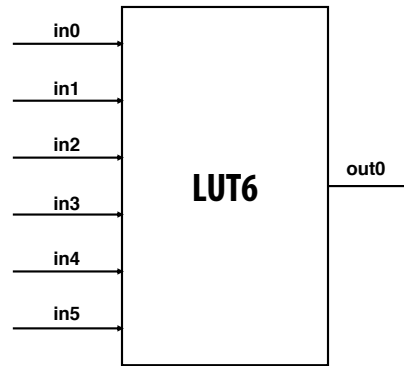


Programmable lookup table (LUT)

Flip flop (a register)

# Specifying combinatorial logic as a LUT

- Example: 6-input, 1 output LUT in Xilinx Virtex-7 FPGAs
  - Think of a LUT6 as a 64 element table



Example:  
6-input AND

| In | Out |
|----|-----|
| 0  | 0   |
| 1  | 0   |
| 2  | 0   |
| 3  | 0   |
| ⋮  | ⋮   |
| 63 | 1   |

40-input AND constructed by chaining  
outputs of eight LUT6's (delay = 3)

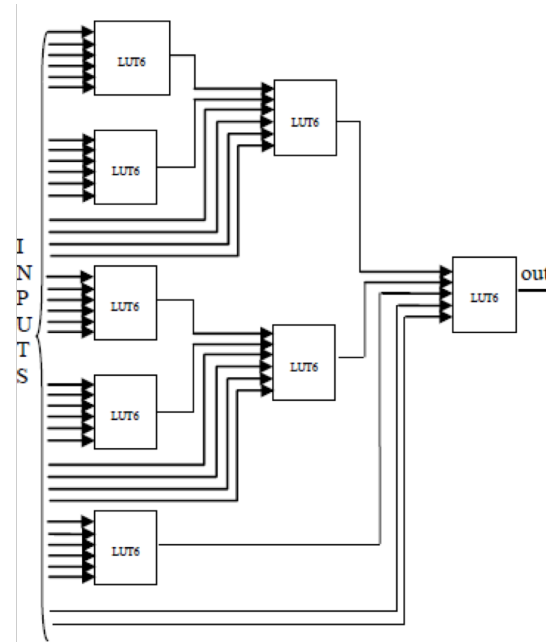
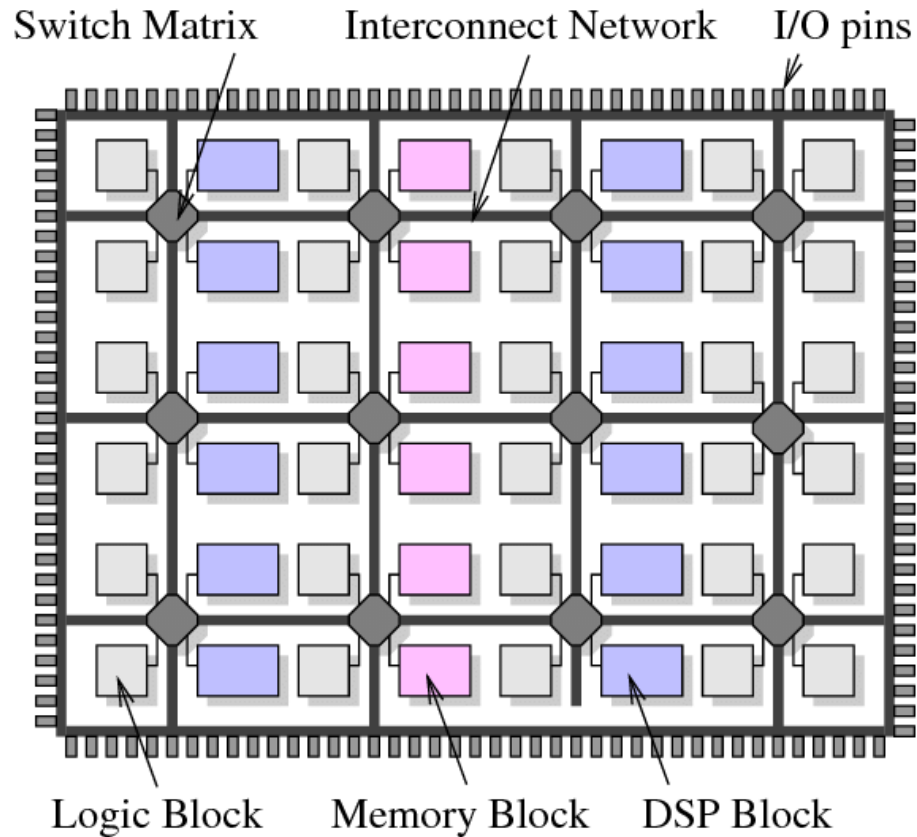


Image credit: [Zia 2013]

# Modern FPGAs



- **A lot of area devoted to hard gates**
  - **Memory blocks (SRAM)**
  - **DSP blocks (multiplier)**

# Project Catapult

[Putnam et al. ISCA 2014]

- Microsoft Research investigation of use of FPGAs to accelerate datacenter workloads
- Demonstrated offload of part of Bing search's document ranking logic

FPGA board



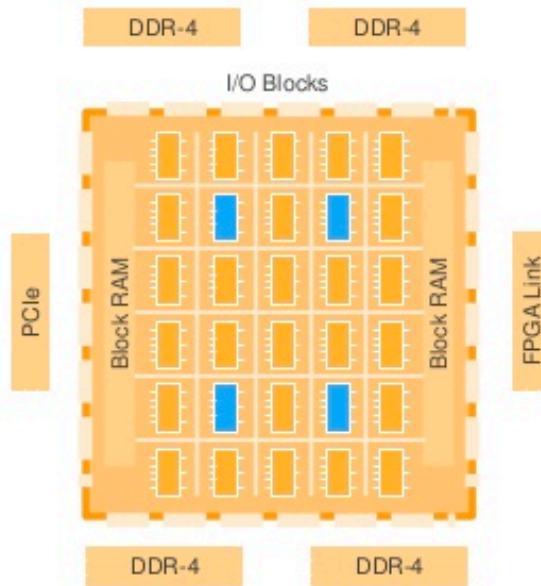
1U server (Dual socket CPU + FPGA connected via PCIe bus)



# Amazon F1

- FPGA's are now available on Amazon cloud services

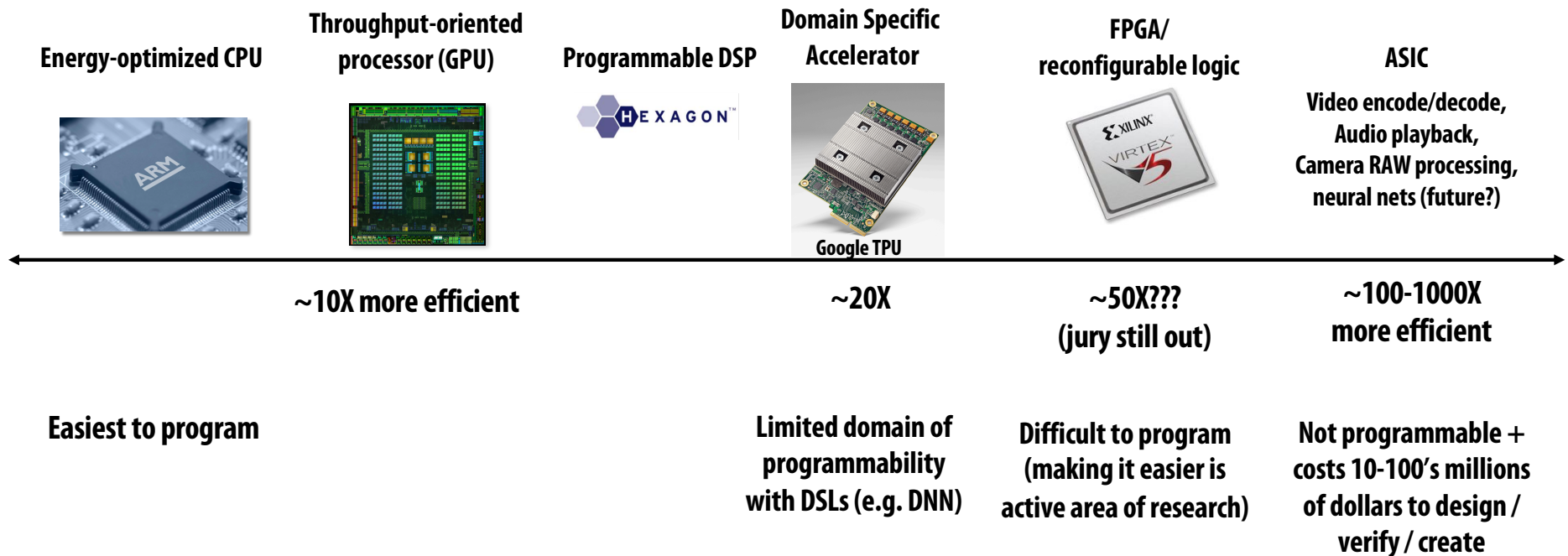
## What's Inside the F1 FPGA?



- System Logic Block:**  
Each FPGA in F1 provides over 2M of these logic blocks
- DSP (Math) Block:**  
Each FPGA in F1 has more than 5000 of these blocks
- I/O Blocks:**  
Used to communicate externally, for example to DDR-4, PCIe, or ring
- Block RAM:**  
Each FPGA in F1 has over 60Mb of internal Block RAM, and over 230Mb of embedded UltraRAM



# Choosing the right tool for the job

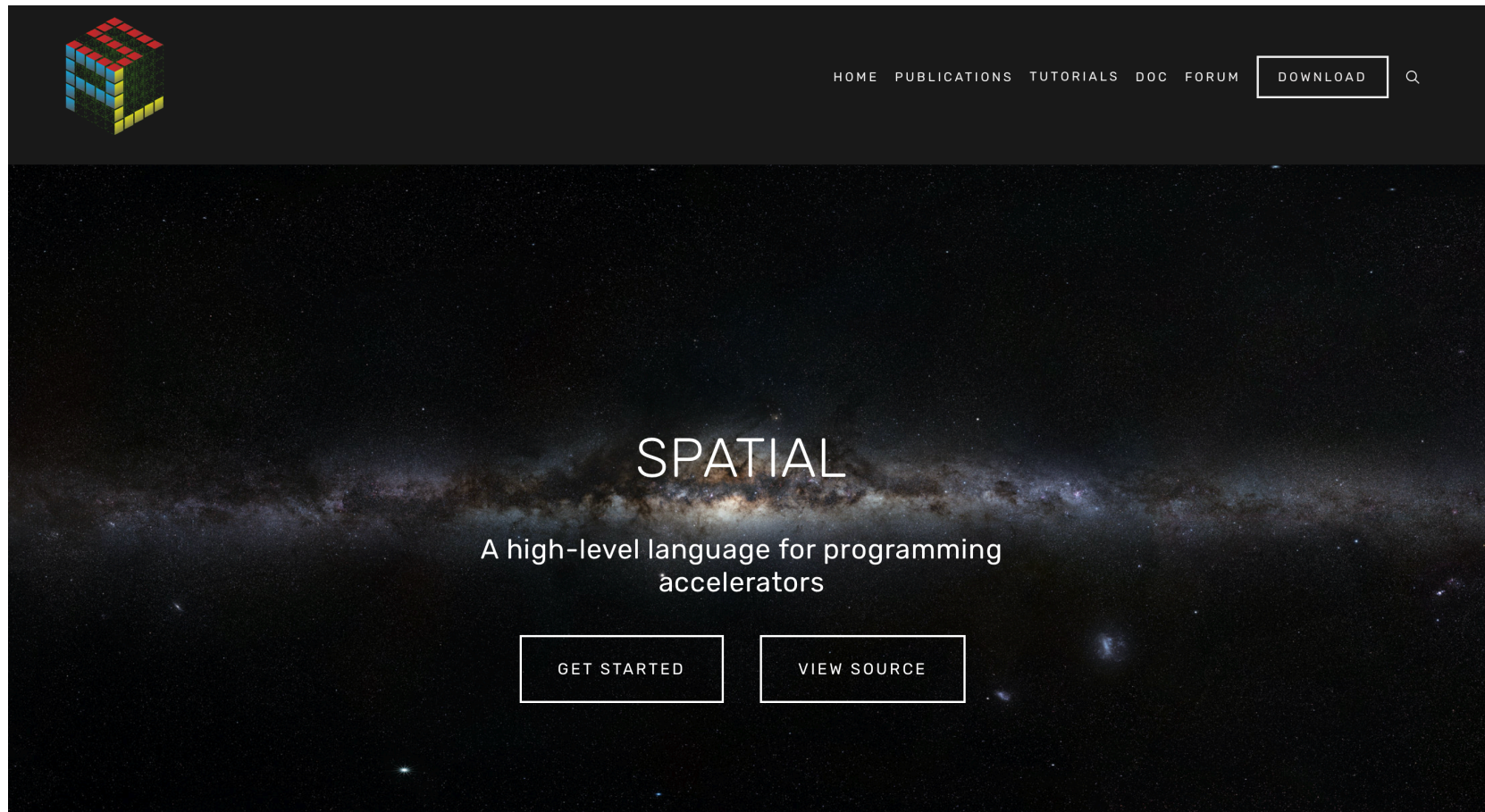



Credit: Pat Hanrahan for this slide design

# So You Want to Design an Accelerator for Your Algorithm

- **Traditionally, you must spend years becoming an expert in VHDL or Verilog...**
- **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**
  - **Parallelism**
  - **Locality**

# Spatial-lang.org



HOME PUBLICATIONS TUTORIALS DOC FORUM **DOWNLOAD** 

# SPATIAL

A high-level language for programming accelerators

[GET STARTED](#) [VIEW SOURCE](#)

# What is Spatial?

**Spatial is a performance-oriented DSL embedded in Scala for programming parallelism and locality**

## Explicit pipelining and parallelization factors

program parallelism

```
Pipeline.Foreach(N by 1 par 4){ i =>
  /* ... */
}
```

## Explicit transfers across memory hierarchy

Program data transfers

```
sram load dram(0::T)
argOut := a + b
```

## Explicit tiling factors

program use of on-chip memories

```
Foreach(N by T){ i =>
  val x = SRAM[Int](T)
  /* ... */
}
```

## Implicit memory banking/buffering schemes

Memory optimizations done automatically to guarantee correctness

```
Foreach(N by 1 par 4){ i =>
  x(i) = ... // Four parallel accesses
  /* ... */
}
```

# Inner Product

Code

Let's build an accelerator to see how Spatial works

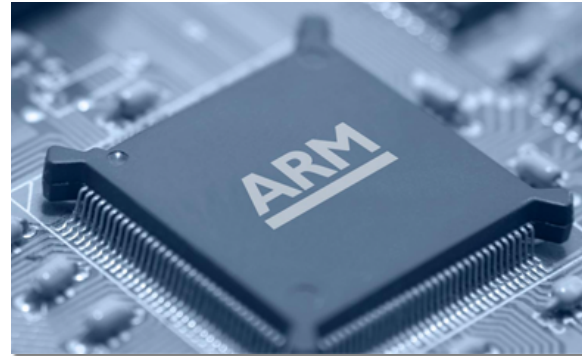
Sketch of generated hardware

# 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



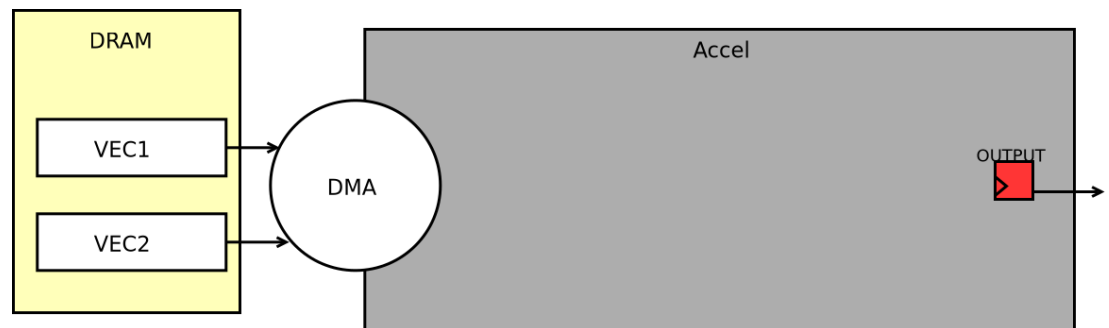
# Inner Product in Spatial

```
// 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 executes “for” loop on the FPGA

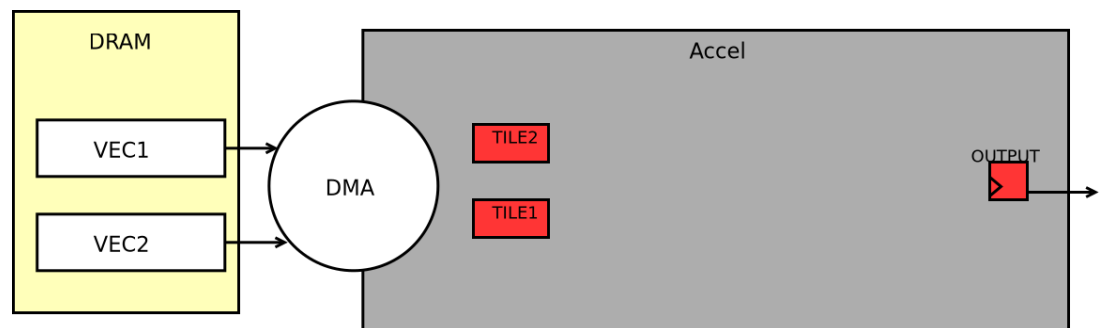




# Inner Product in Spatial

```
// 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)
}
```

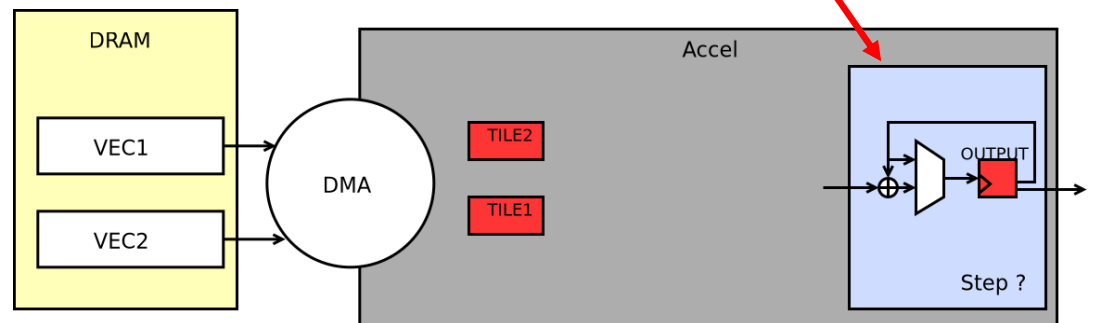


# Inner Product in Spatial

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

```
// 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}
}
```



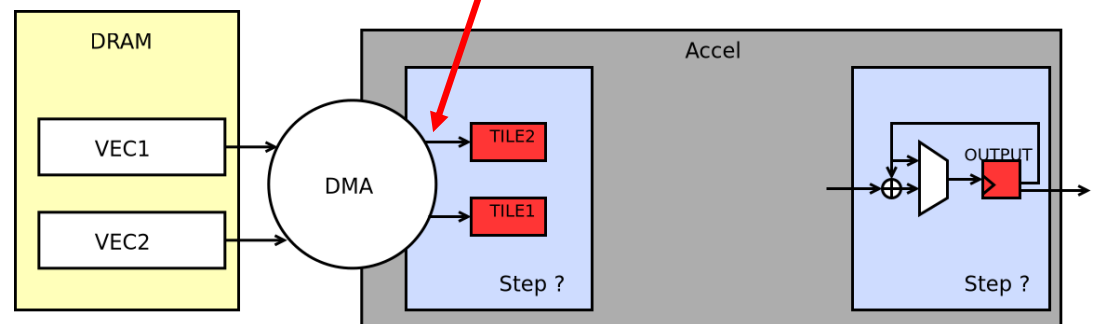
# Inner Product in Spatial

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

```
// 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}
  }
}
```



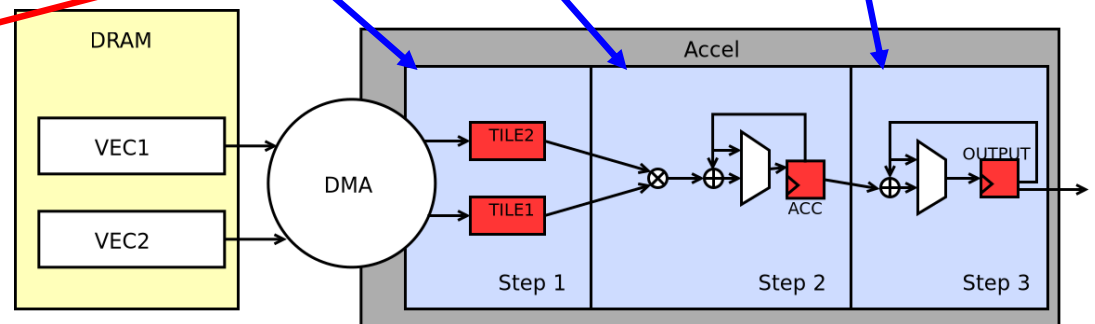
# Inner Product in Spatial

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

The complete app generates a three-step control  
Load → intra-tile accumulate → full accumulate

```
// 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}
}
```

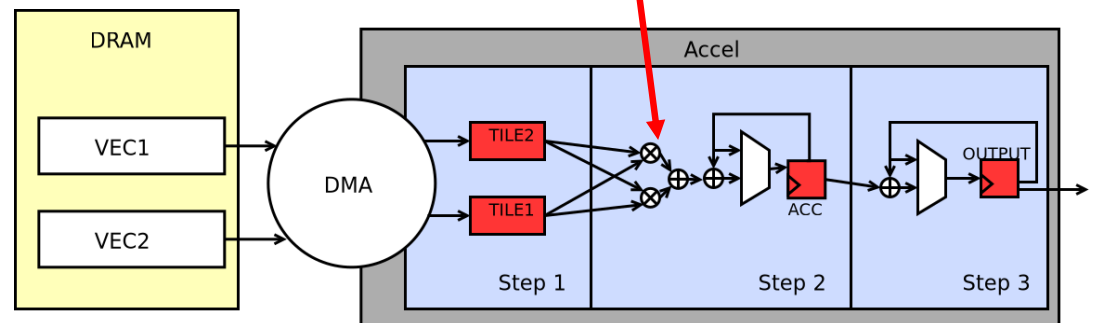


# Inner Product in Spatial

```
// 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)
    }{ _ + _ }
  }{ _ + _ }
}
```

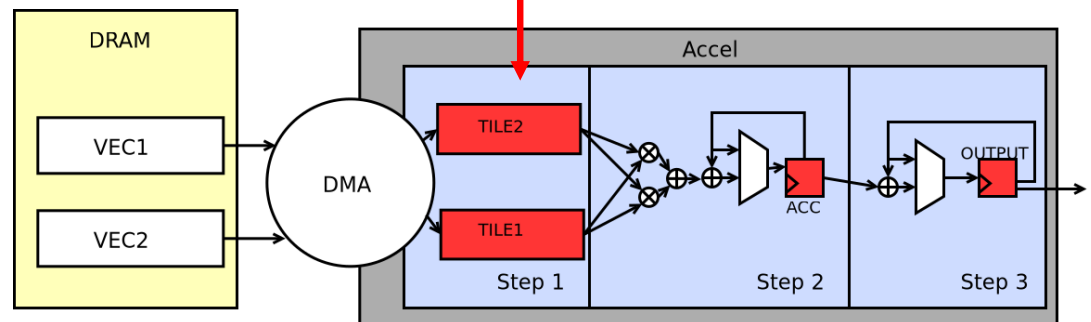
- Spatial generates multi-step controllers
- Spatial manages communication with DRAM
- Spatial helps express hardware datapaths



# 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

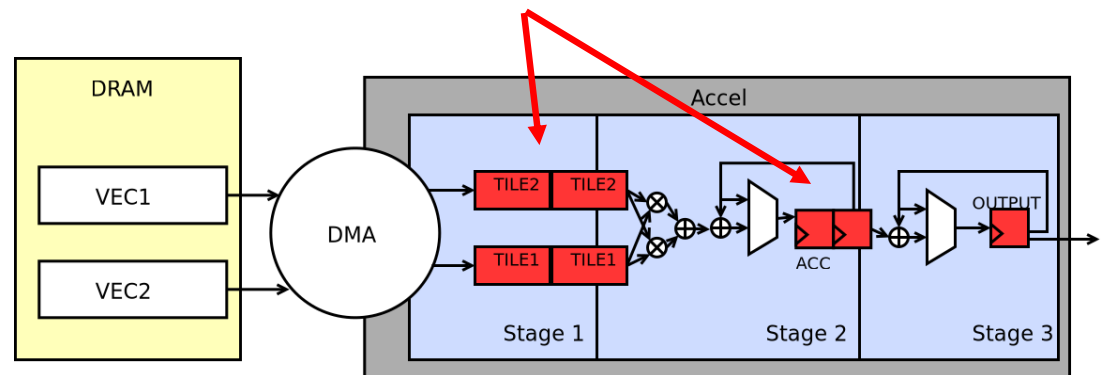


# Inner Product in Spatial

```
// 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 control flow
- With annotation, steps (stages) execute in pipelined fashion. “Buffering” of memories is inferred



# Controllers

- Every “loop” in Spatial is a controller
- **Controller** - A hardware counter chain 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

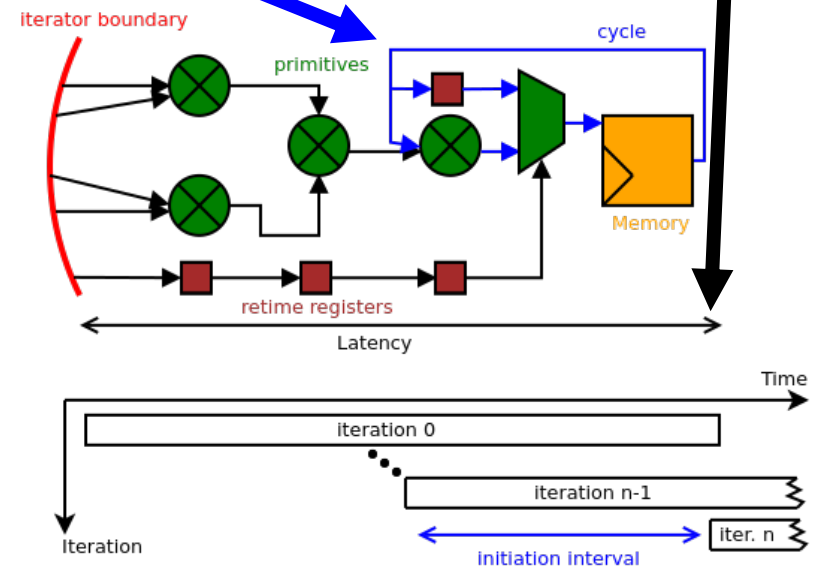
Inner controllers always execute iterations in a pipelined (overlapped) manner

**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

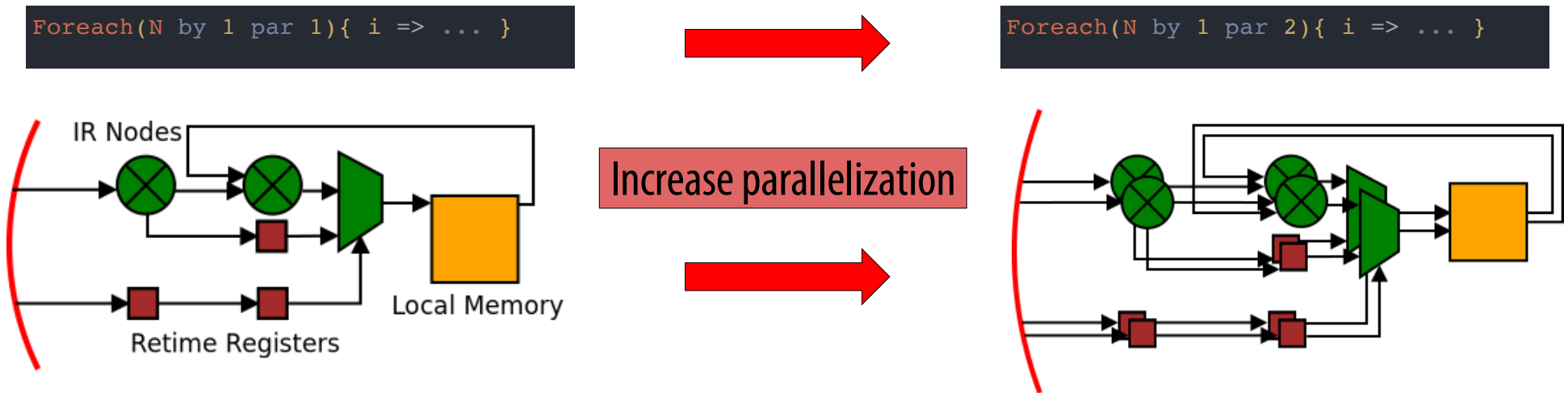
$$T = \mathbf{II} * (iters - 1) + \mathbf{L}$$

```
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



# 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

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

# Scheduling Outer Controllers

There are four major schedules for outer controllers:

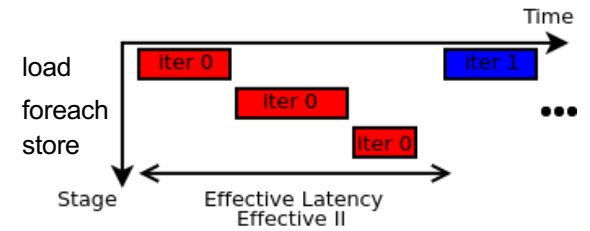
- **Sequential** – No overlapping of inner (child) controllers
- **Pipelined** – Coarse-grained overlapping of inner (child) controllers
- **Fork-Join** – Parallel execution of all inner (child) controllers
- **Stream** – Data-driven execution of inner (child) controllers

# Sequential and Pipelined

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

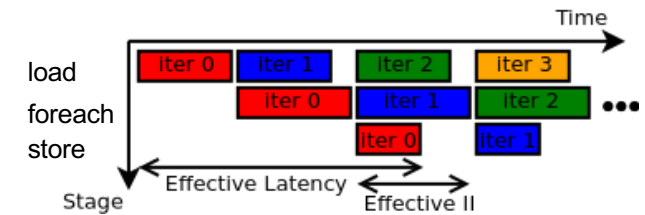
Sequential

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



Pipelined

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

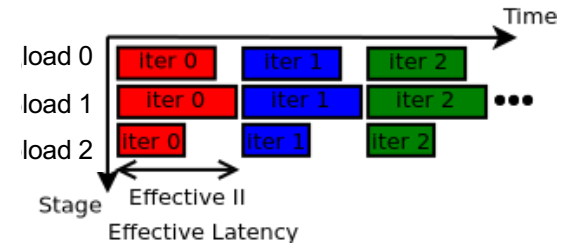


# Fork-Join and Stream

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

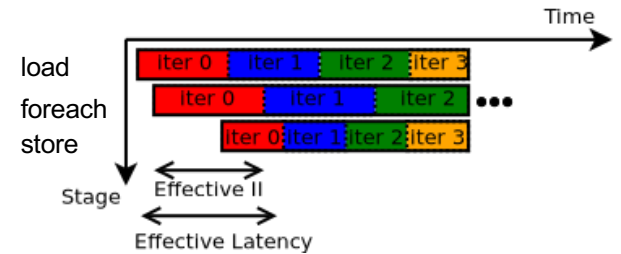
Fork-Join

```
ForkJoin{
  sram0 load dram0
  sram1 load dram1
  sram2 load dram2
}
```



Stream

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







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

**Let's talk about performance debugging**

# Performance Debugging

Performance debugging typically applies to one parent-child slice of the hierarchy at a time

Example parent with three children

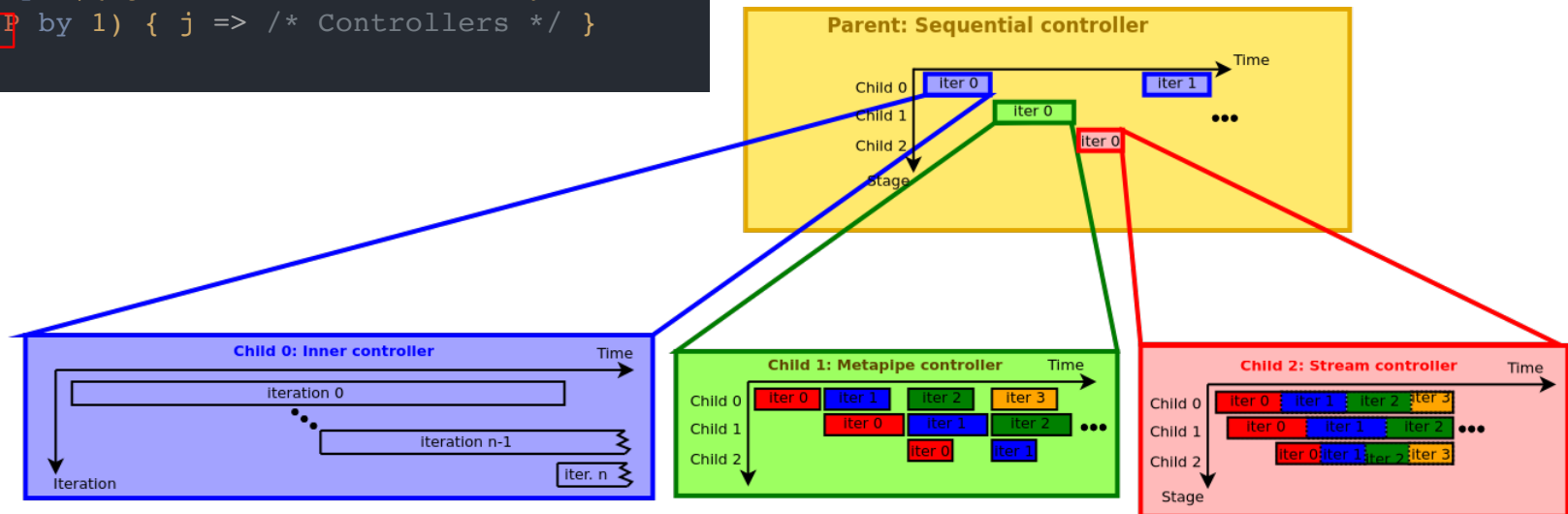
```
Sequential.Foreach(Q by TS){ i =>
    Foreach(N by 1){ j => /* Primitives */ }
    Pipe.Foreach(M by 1){ j => /* Controllers */ }
    Stream.Foreach(P by 1) { j => /* Controllers */ }
}
```

# Performance Debugging 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 */ }
}
```

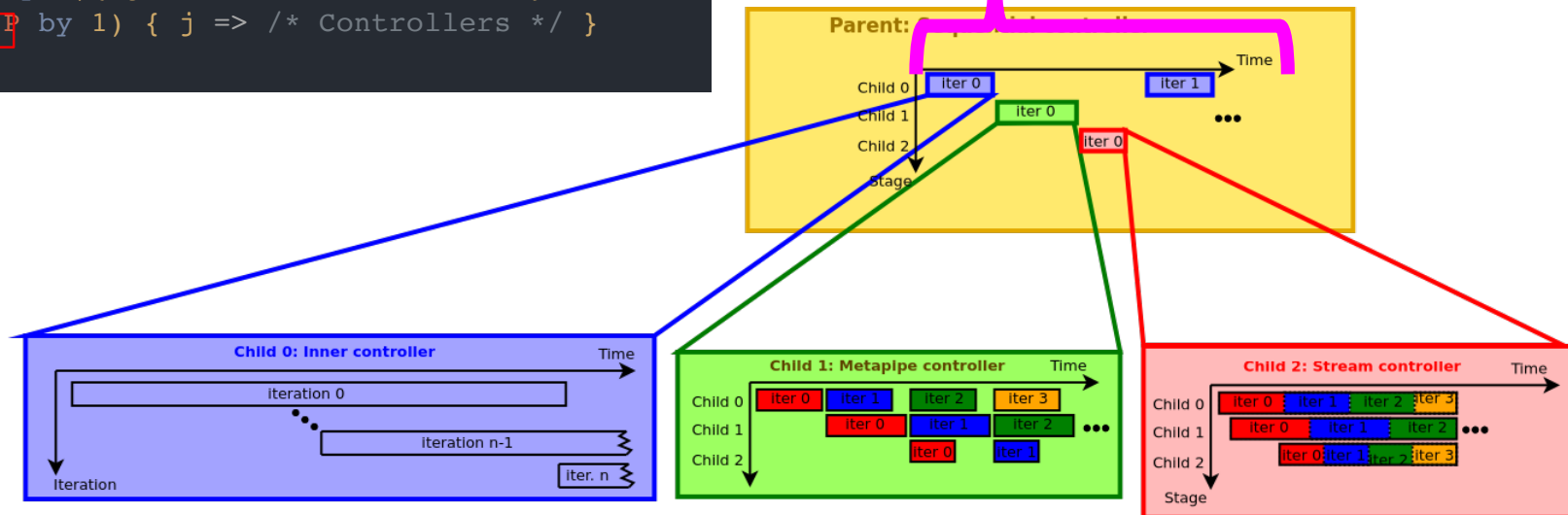


# Performance Debugging

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**

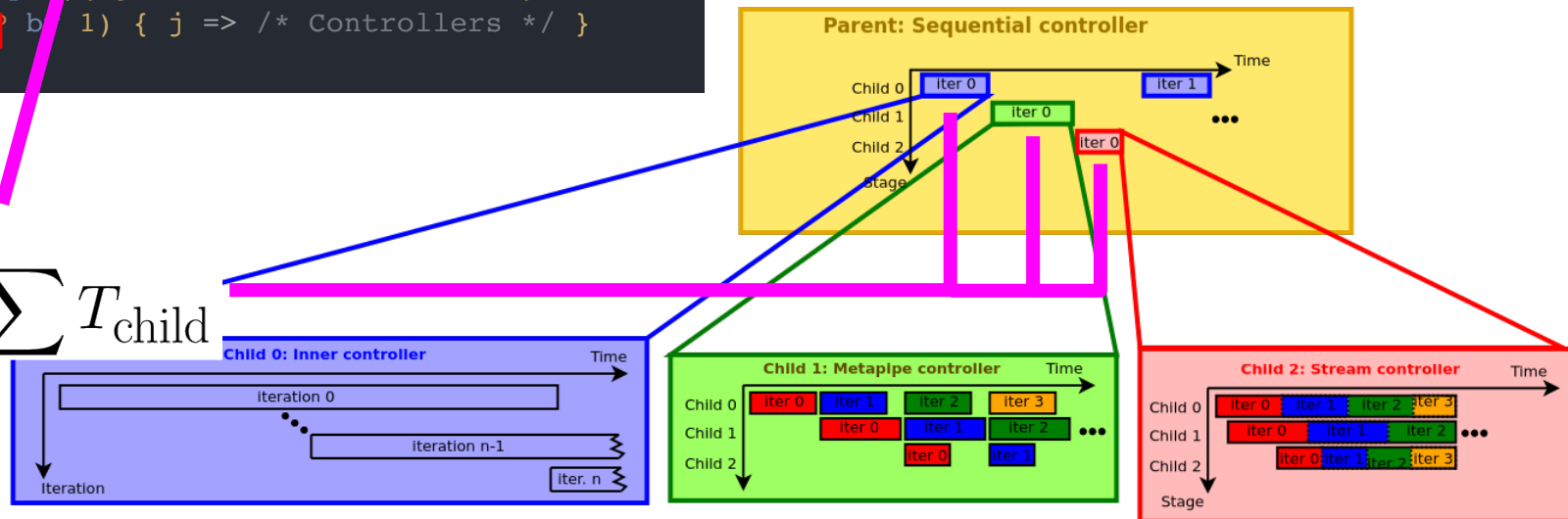
$$T_{\text{parent}} = \mathbf{II}_{\text{eff}} * (\text{iters} - 1) + \mathbf{L}_{\text{eff}}$$

```

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

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

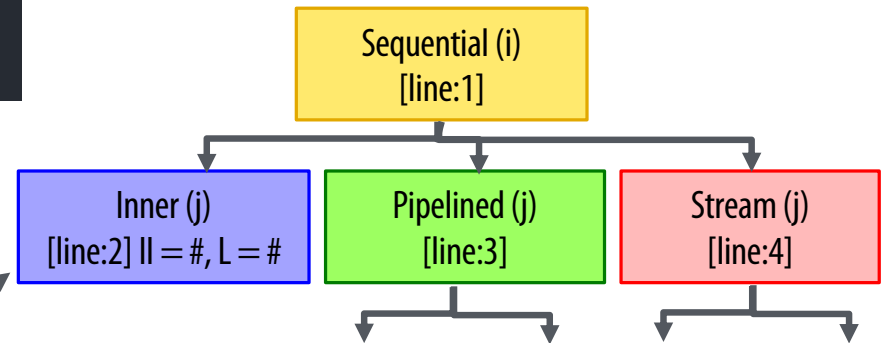
$$\mathbf{II}_{\text{eff}} = \mathbf{L}_{\text{eff}} = \sum T_{\text{child}}$$



# Performance Debugging with Controller Hierarchy

The controller hierarchy is a more concise way to understand performance  
Spatial compiler **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 */ }  
}
```



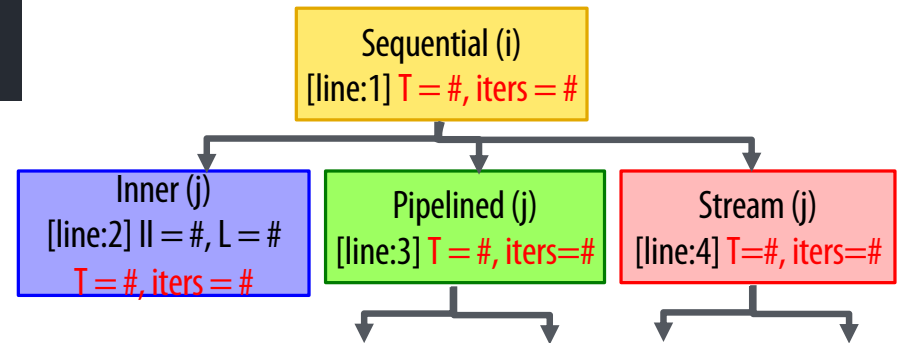
**Line numbers** link controllers back to source code

**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 */ }  
}
```

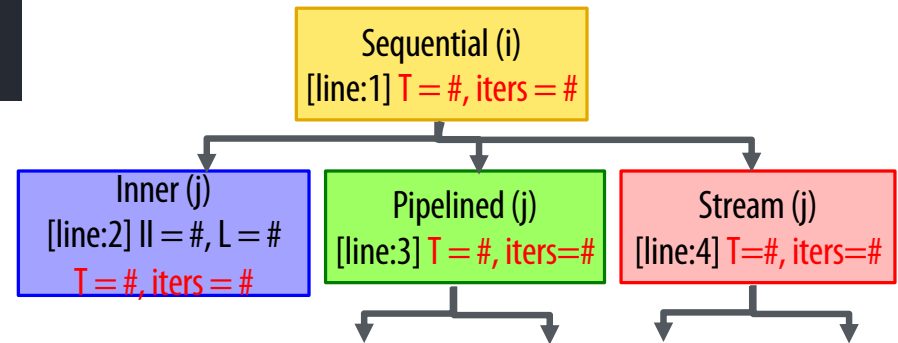


Actual **T and iteration counts** are automatically collected and overlaid after execution

# 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


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

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 ParO){ 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 ParI){ 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:

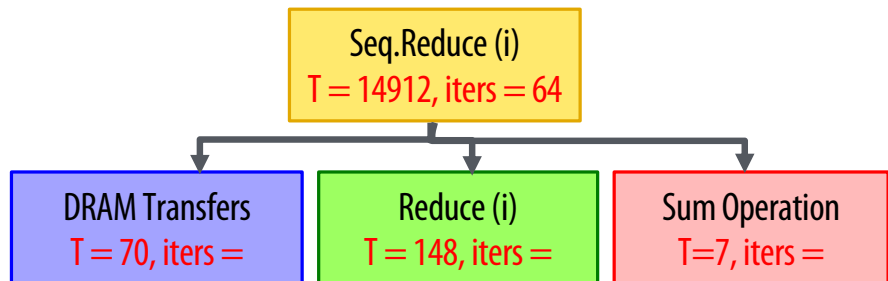
- Outer controller schedule (Reduce)
- ParO
- ParI

# 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 ParO){ 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 ParI){ i =>
      tile1(i) * tile2(i)
    }{a, b => a + b}
  }{a, b => a + b}
}
```

The baseline implementation is  $\text{ParI}=1$ ,  $\text{ParO}=1$ , and  $\text{schedule}=\text{Sequential}$

Our instrumented controller tree will look like this



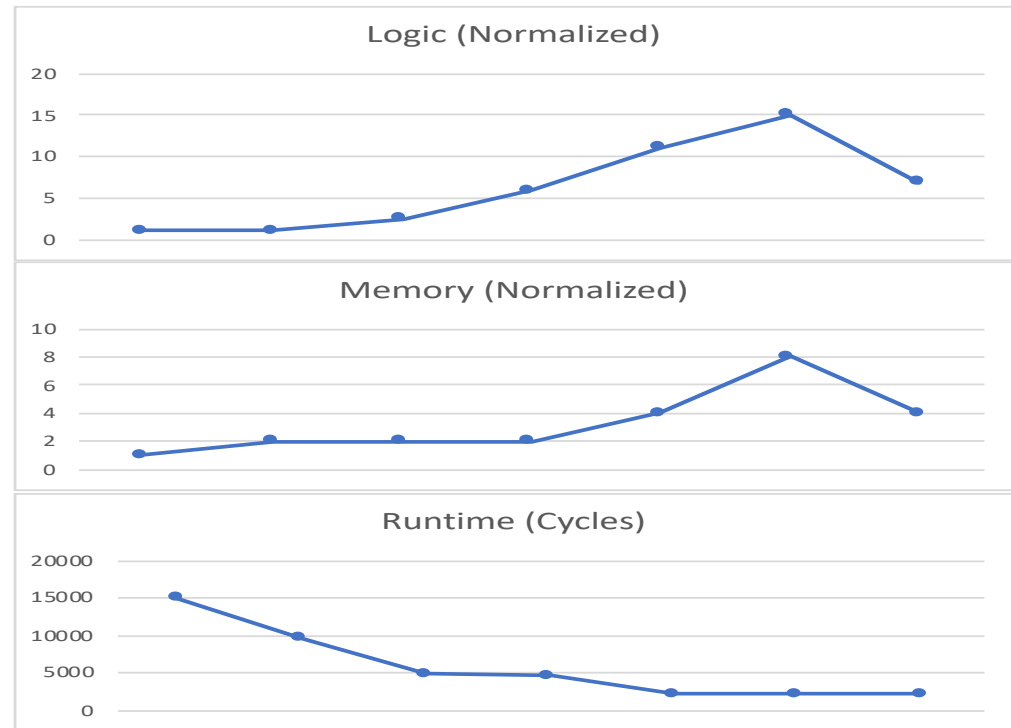
# Optimization Goal

```
// Inner product accelerator
Accel {
  val tile1 = SRAM[Int](tileSize)
  val tile2 = SRAM[Int](tileSize)
  // Outer reduce
  Reduce(output)(N by tileSize par ParO){ 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 ParI){ i =>
      tile1(i) * tile2(i)
    }{a, b => a + b}
  }{a, b => a + b}
}
```

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

# Trailer: Inner Product Optimization

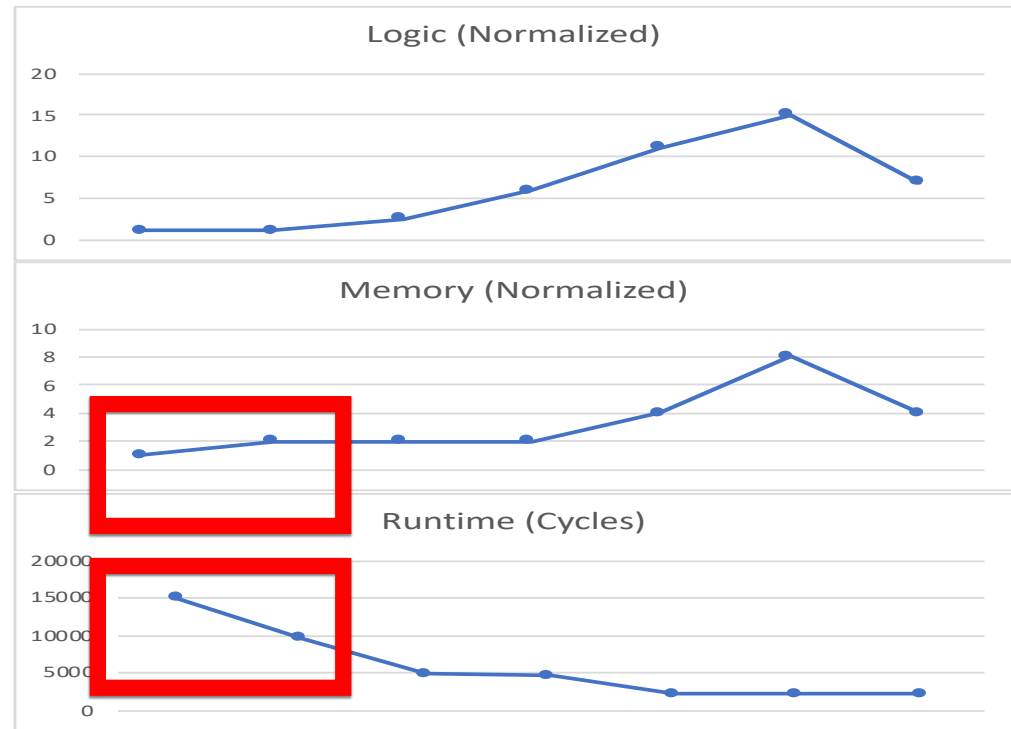
- By optimizing the code, we can improve execution time by  $\sim 7x$
- The best design increases logic by  $\sim 6x$  and memory by  $\sim 4x$



|              |     |      |      |      |      |      |      |
|--------------|-----|------|------|------|------|------|------|
| <b>Parl</b>  | 1   | 1    | 2    | 4    | 4    | 4    | 2    |
| <b>Par0</b>  | 1   | 1    | 1    | 1    | 2    | 4    | 2    |
| <b>Sched</b> | Seq | Pipe | Pipe | Pipe | Pipe | Pipe | Pipe |

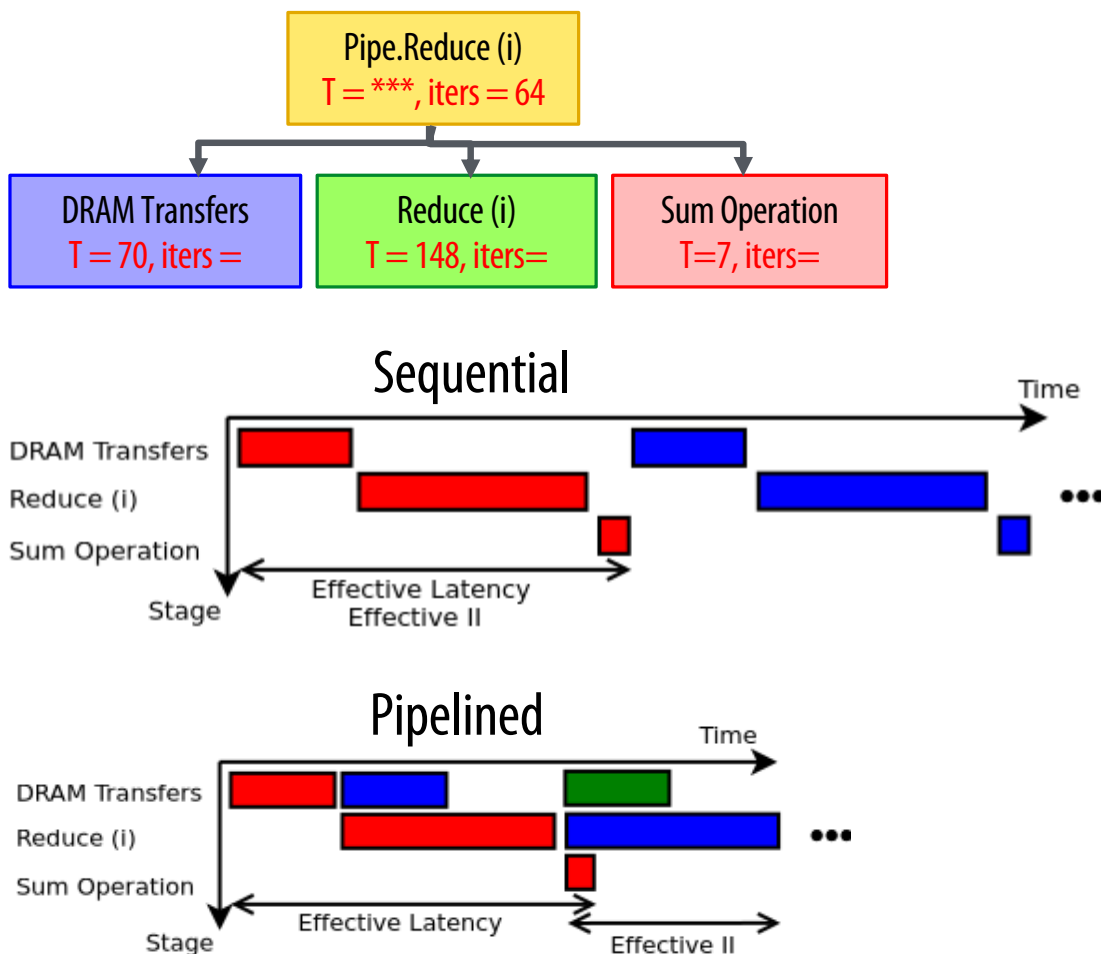
# Sequential vs. Pipelined

- 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



|              |     |      |      |      |      |      |      |
|--------------|-----|------|------|------|------|------|------|
| <b>Par1</b>  | 1   | 1    | 2    | 4    | 4    | 4    | 2    |
| <b>Par0</b>  | 1   | 1    | 1    | 1    | 2    | 4    | 2    |
| <b>Sched</b> | Seq | Pipe | Pipe | Pipe | Pipe | Pipe | Pipe |

# Sequential vs. Pipelined



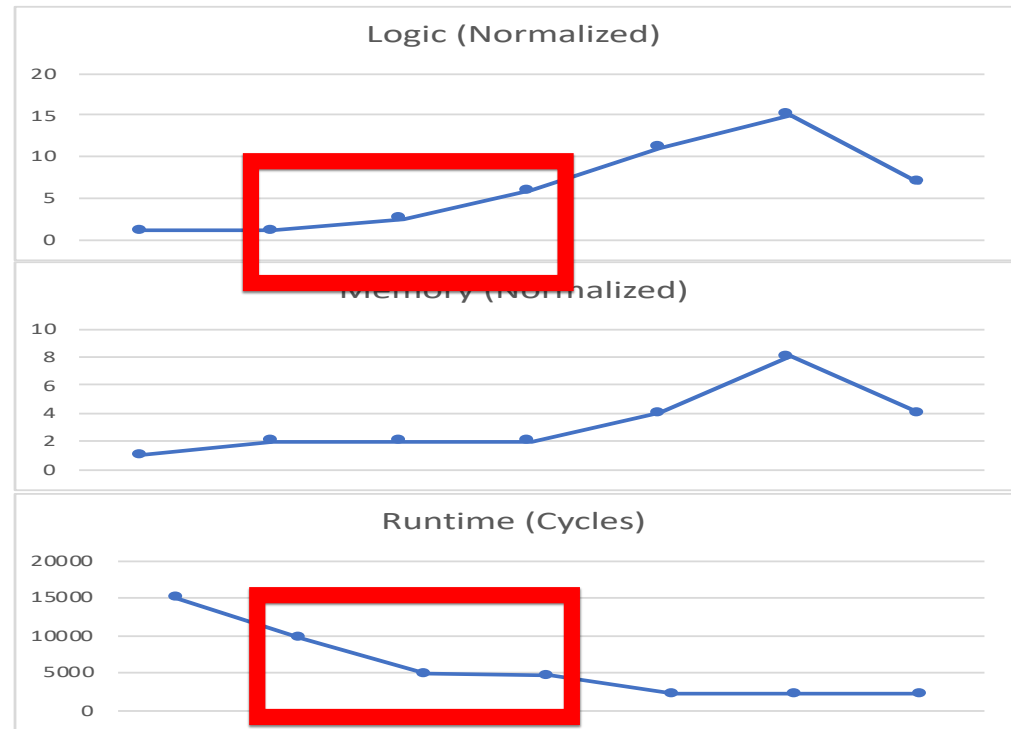
- Understanding how the performance debugger maps to timing diagrams explains this performance boost from pipelining
  - Color corresponds to iteration in diagrams
- For the Sequential case,  $II_{eff} \approx \sum T_c$
- For the Pipelined case,  $II_{eff} \approx \max(T_c)$

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



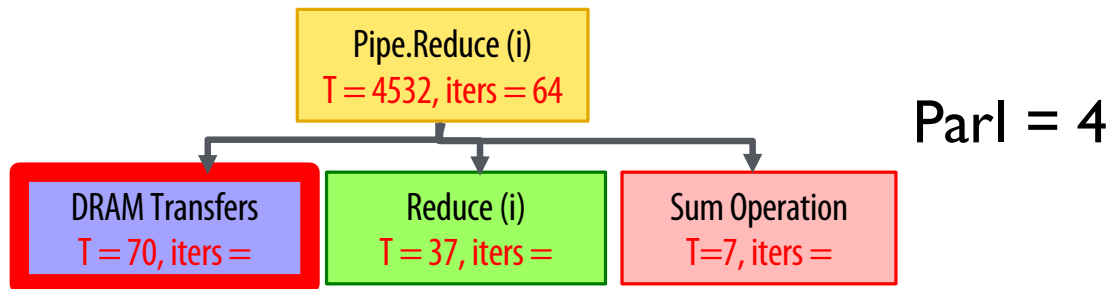
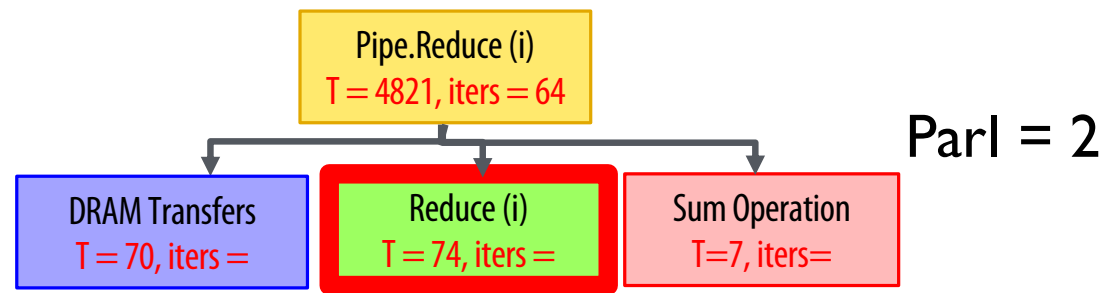
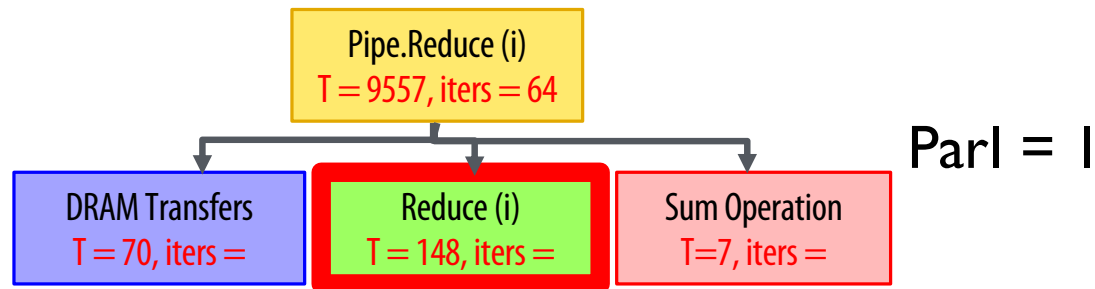
# Inner Parallelization

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



|              |     |      |      |      |      |      |      |
|--------------|-----|------|------|------|------|------|------|
| <b>Parl</b>  | 1   | 1    | 2    | 4    | 4    | 4    | 2    |
| <b>Par0</b>  | 1   | 1    | 1    | 1    | 2    | 4    | 2    |
| <b>Sched</b> | Seq | Pipe | Pipe | Pipe | Pipe | Pipe | Pipe |

# Inner Parallelization

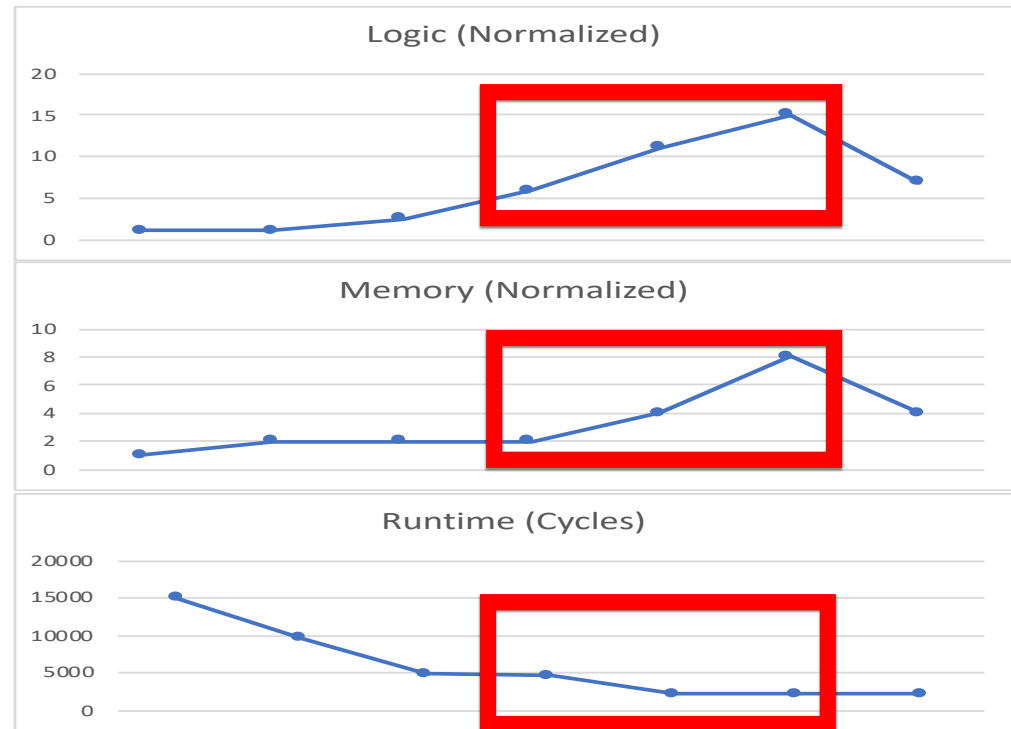


- The performance debugger explains what happened
- The bottleneck stage improves as a result of this parallelization
- There is still a *small* performance improvement for Parl=4 because  $II_{eff} \approx \max(T_c)$  decreases a bit

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

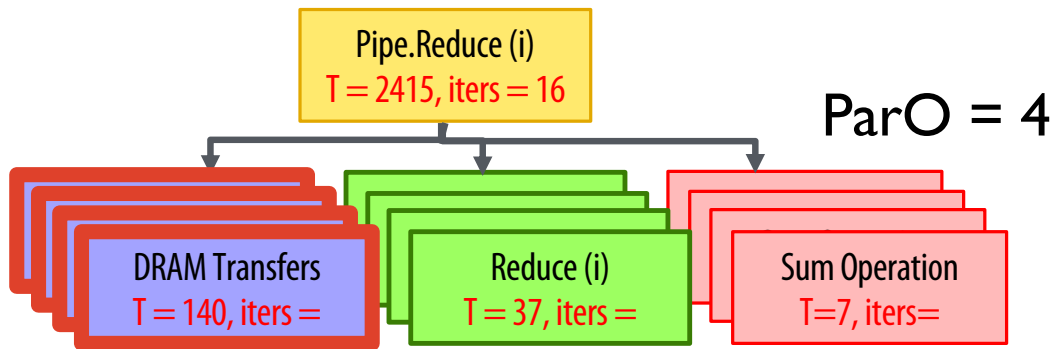
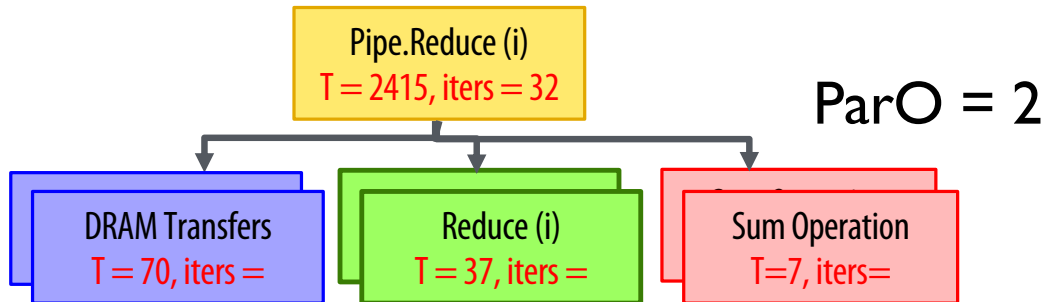
# Outer Parallelization

- There is a performance improvement from Par0=1 to Par0=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 Par0=2 to Par0=4, the app becomes memory-bound
  - Change increases resource utilization without improving performance



|              |     |      |      |      |      |      |      |
|--------------|-----|------|------|------|------|------|------|
| <b>Par1</b>  | 1   | 1    | 2    | 4    | 4    | 4    | 2    |
| <b>Par0</b>  | 1   | 1    | 1    | 1    | 2    | 4    | 2    |
| <b>Sched</b> | Seq | Pipe | Pipe | Pipe | Pipe | Pipe | Pipe |

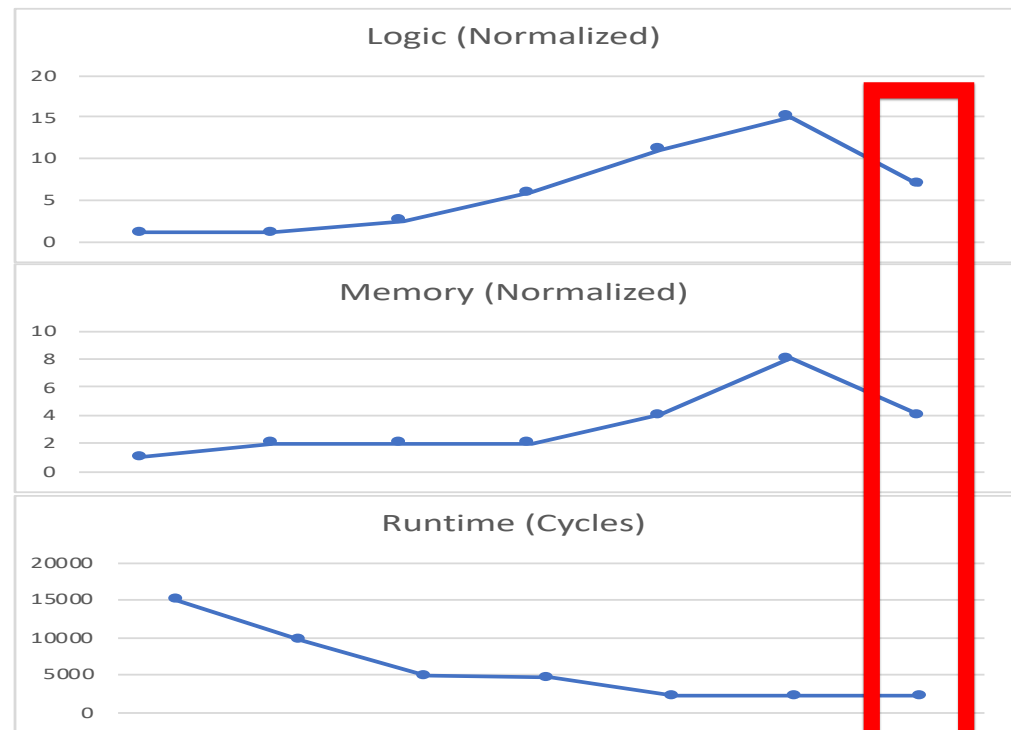
# Outer Parallelization



- The **Reduce** and **Sum Operation** stages are statically scheduled
- The **DRAM Transfers** stages compete for the DRAM and execute when DRAM returns data
- When  $\text{ParO}$  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  $\text{ParO}=2$  but not  $\text{ParO}=4$

# 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  $\sim 7x$ 
  - The best design increases logic by  $\sim 6x$  and memory by  $\sim 4x$



|              |     |      |      |      |      |      |      |
|--------------|-----|------|------|------|------|------|------|
| <b>Par1</b>  | 1   | 1    | 2    | 4    | 4    | 4    | 2    |
| <b>Par0</b>  | 1   | 1    | 1    | 1    | 2    | 4    | 2    |
| <b>Sched</b> | Seq | Pipe | Pipe | Pipe | Pipe | Pipe | Pipe |

# 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**

# Outer Controller Parallelization

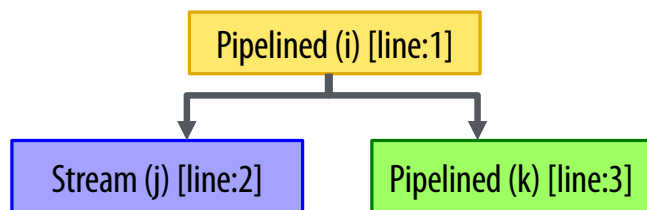
Parallelization of **outer controllers** results in duplication of all child controllers and insertion of synchronization controllers (**ForkJoin**)

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

