

Lecture 18:

Hardware Specialization and Spatial Programming

**Parallel Computing
Stanford CS149, Fall 2021**

Energy-constrained computing

Performance and Power

$$\text{Power} = \frac{\text{Performance}}{\text{Energy efficiency}}$$

Performance *Energy efficiency*

Ops *Joules*

second *Op*

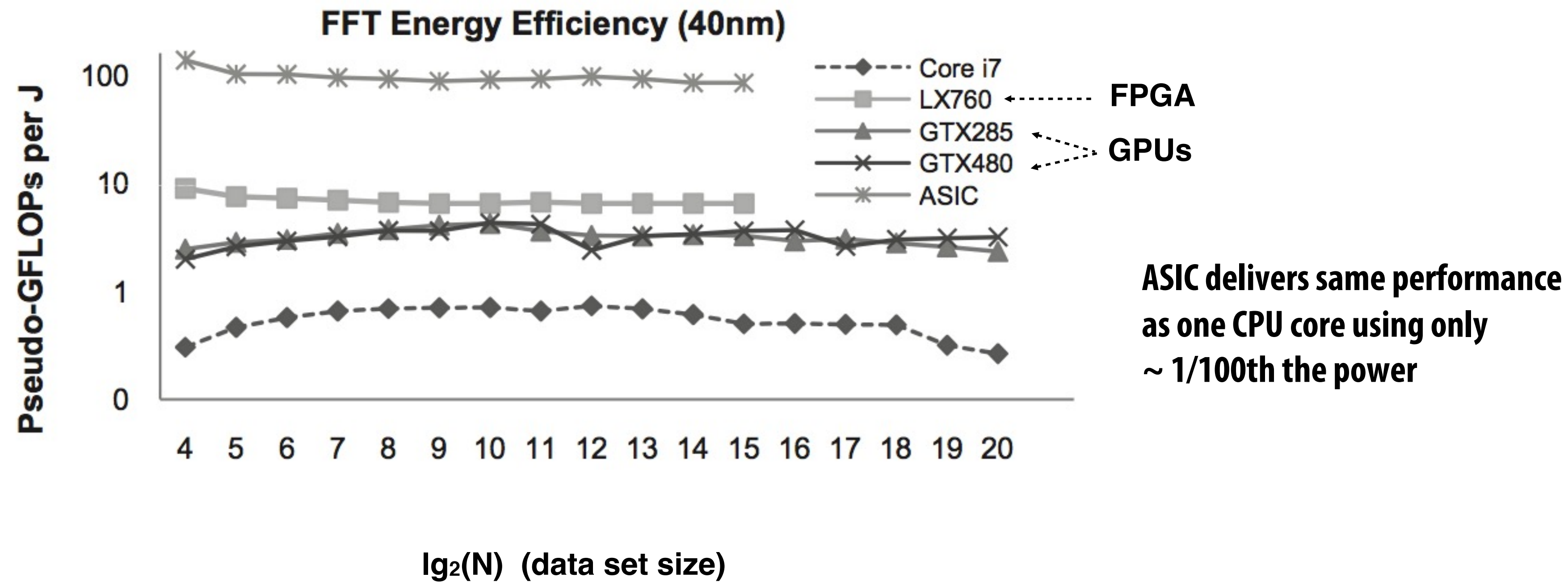
FIXED



Specialization (fixed function) \Rightarrow better energy efficiency

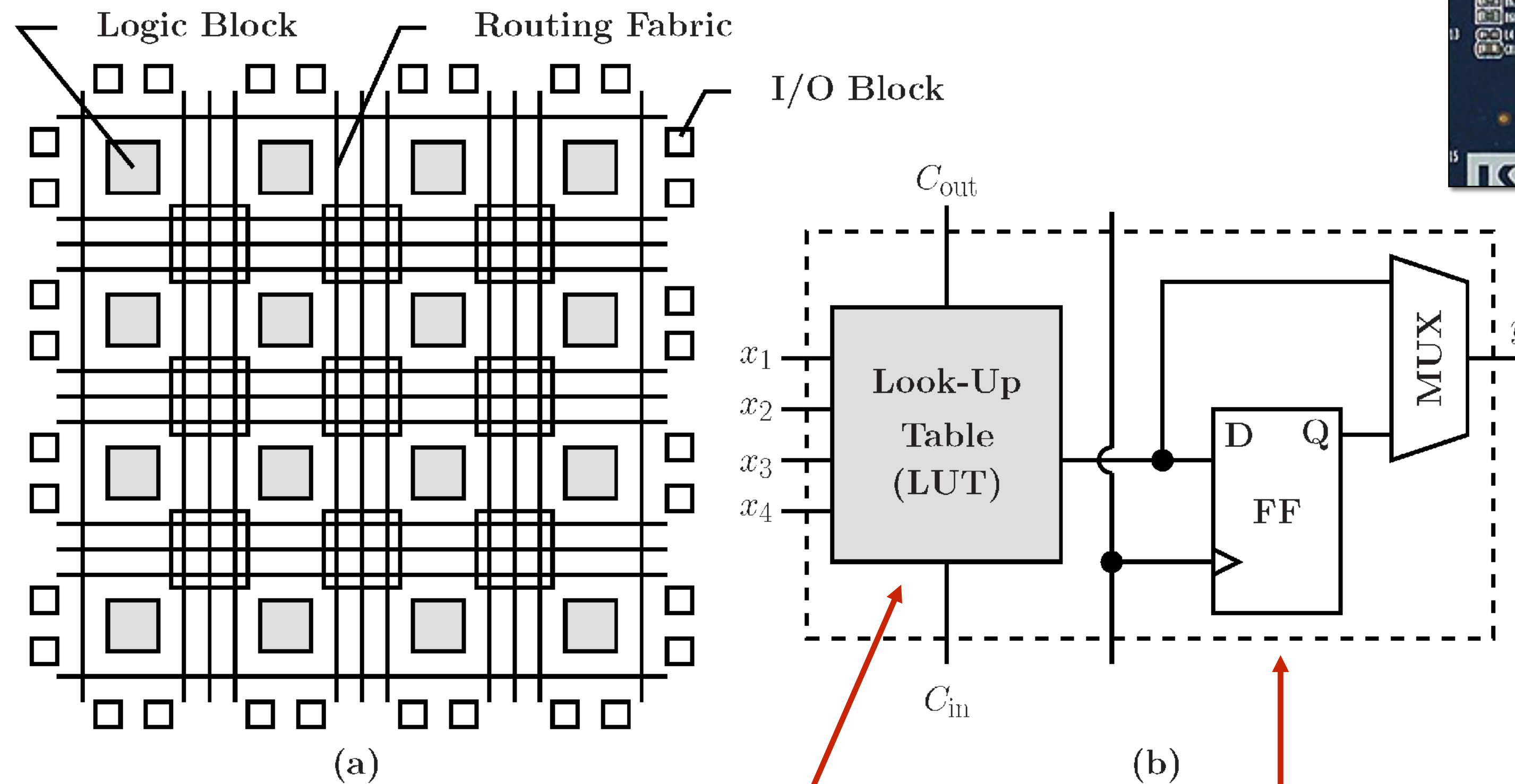
What is the magnitude of improvement from specialization?

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

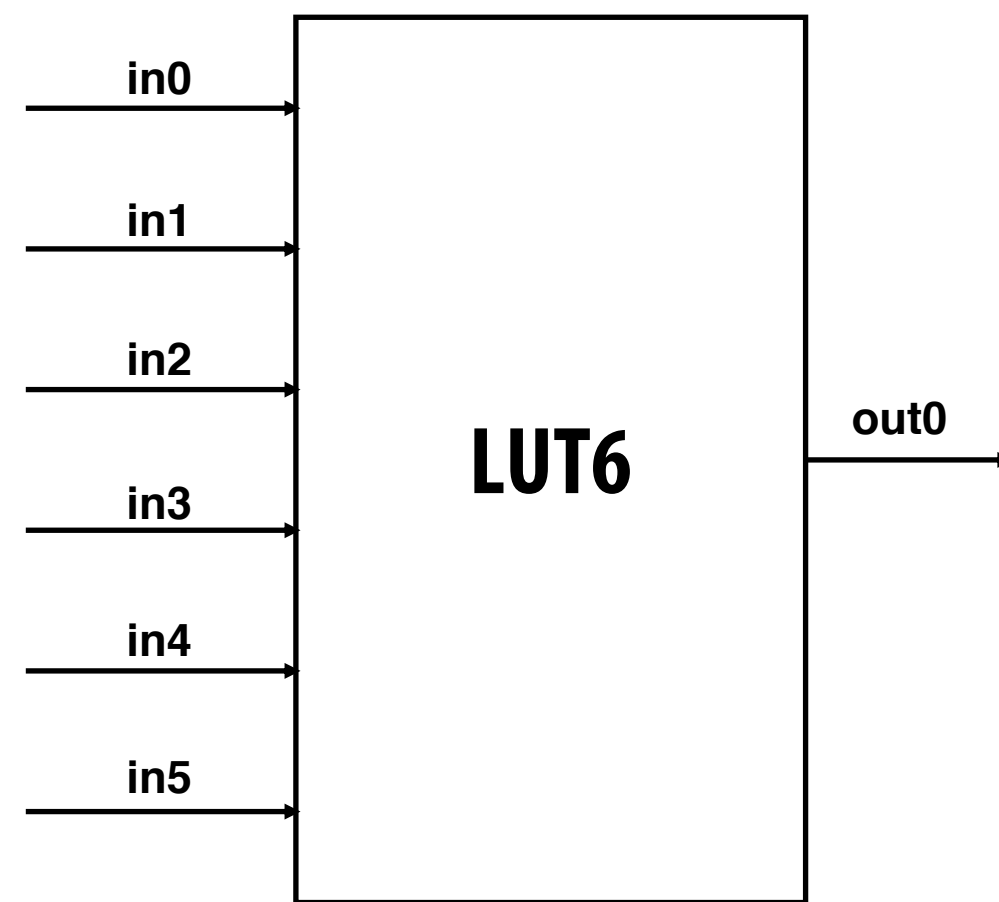


Programmable lookup table (LUT)

Flip flop (a register)

Specifying combinational 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)

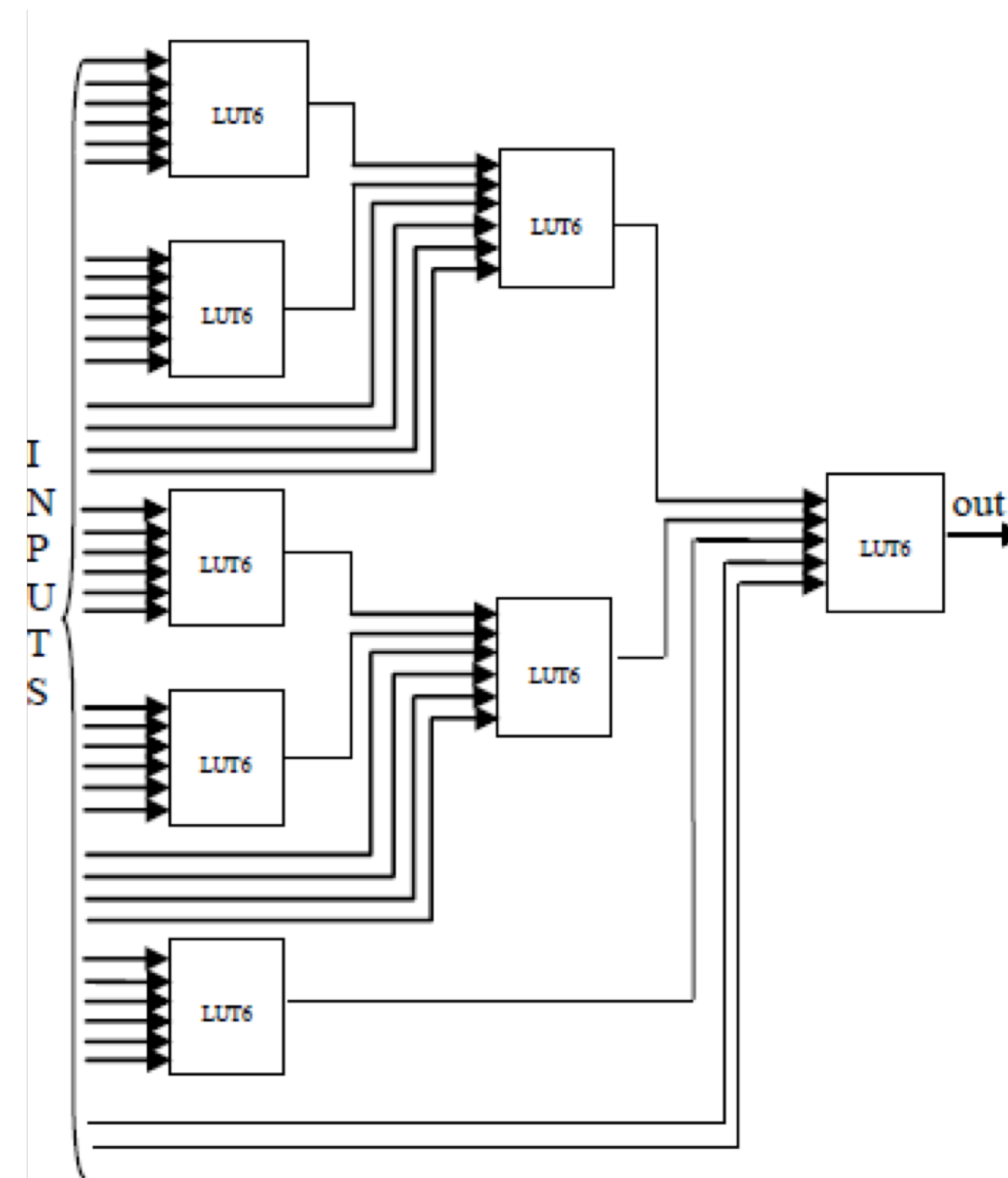
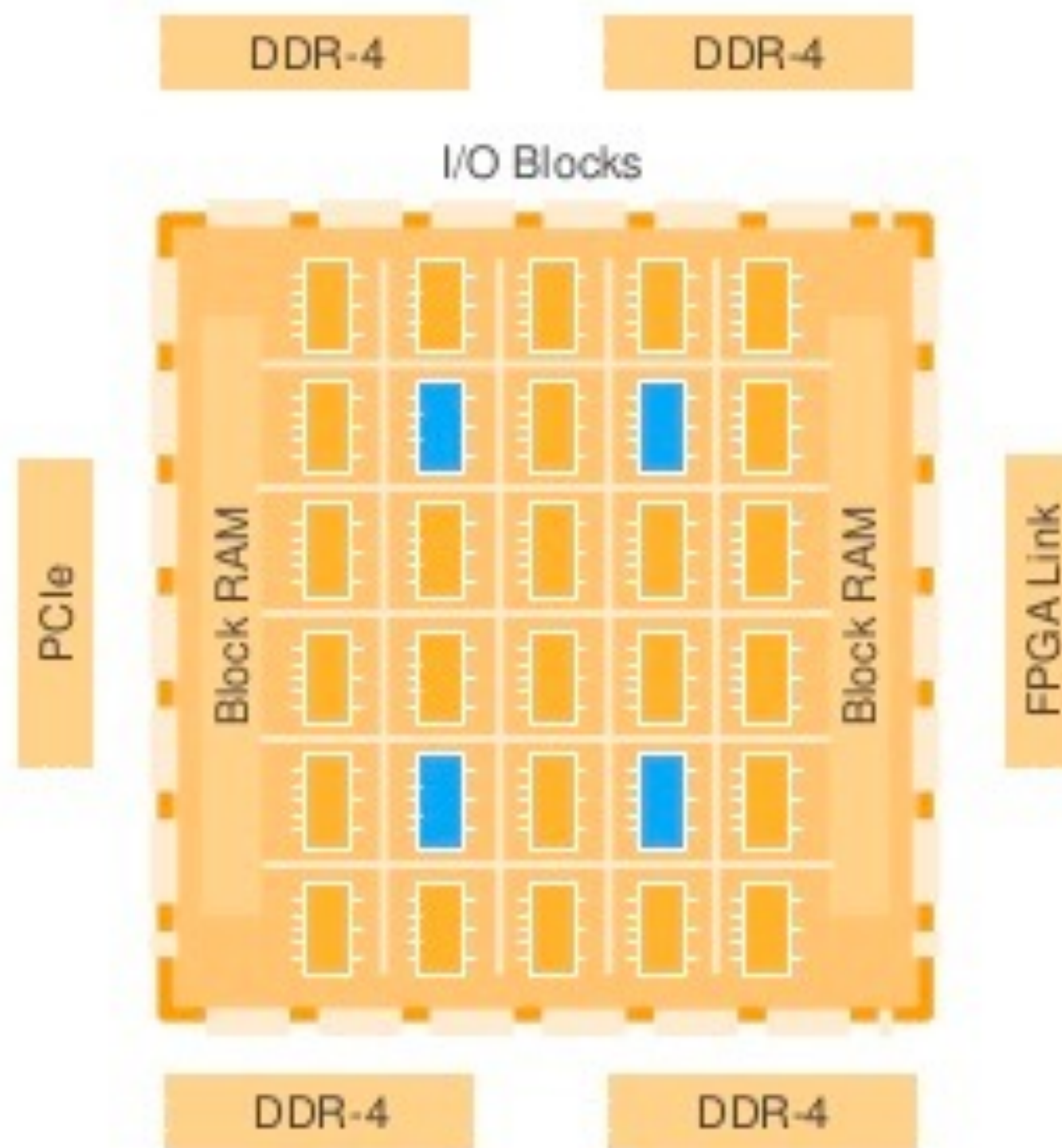


Image credit: [Zia 2013]

Amazon F1

- **FPGA's are now available on Amazon cloud services**

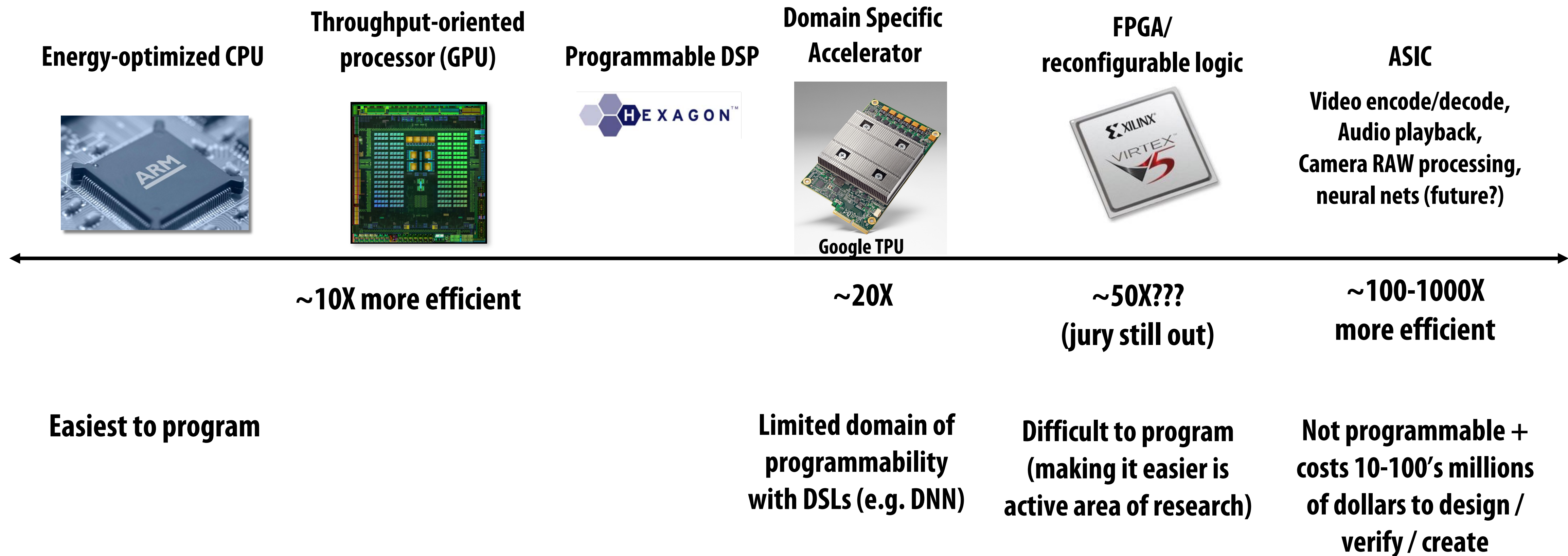
What's Inside the F1 FPGA?



- 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

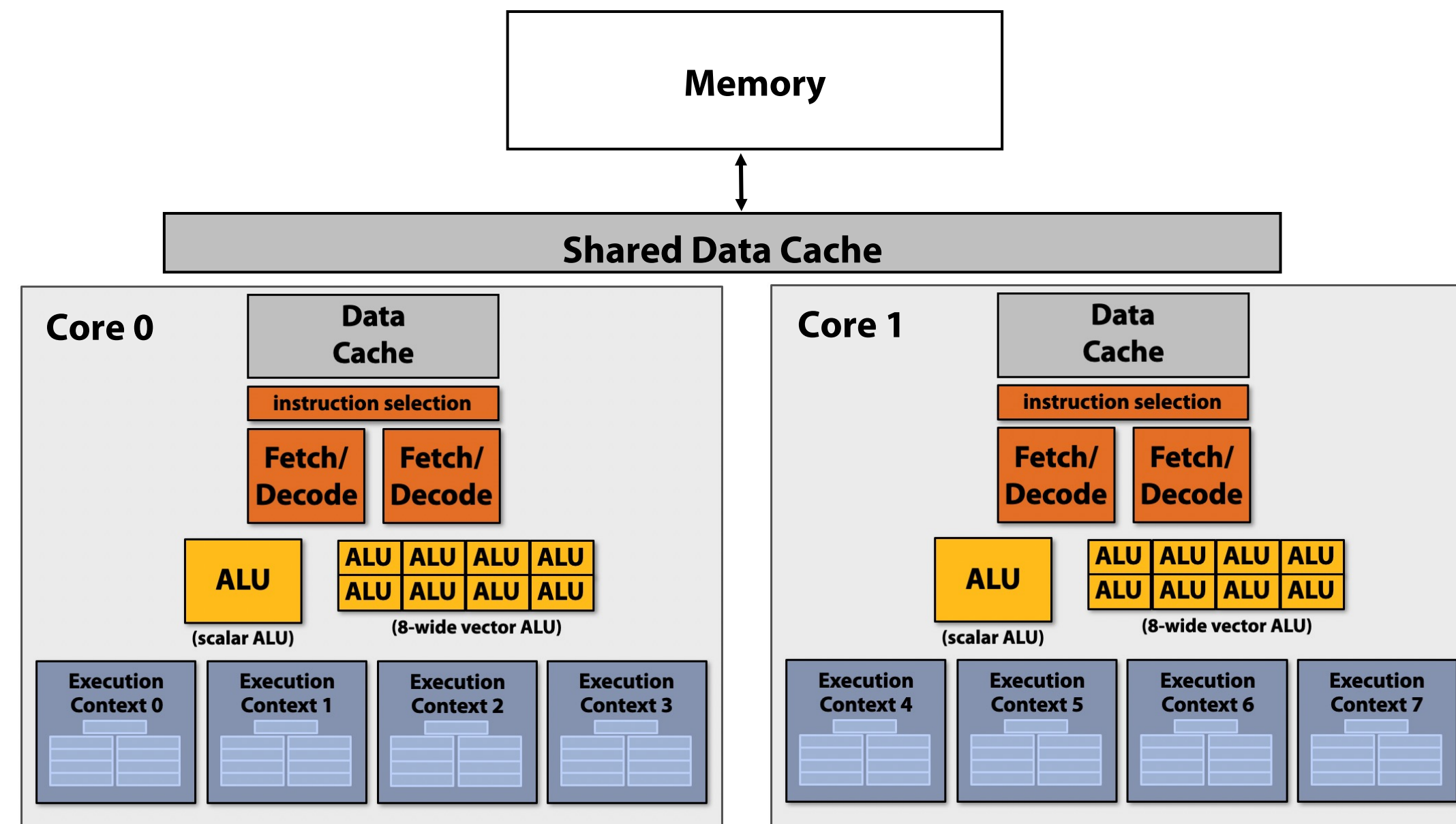


Choosing the right tool for the job

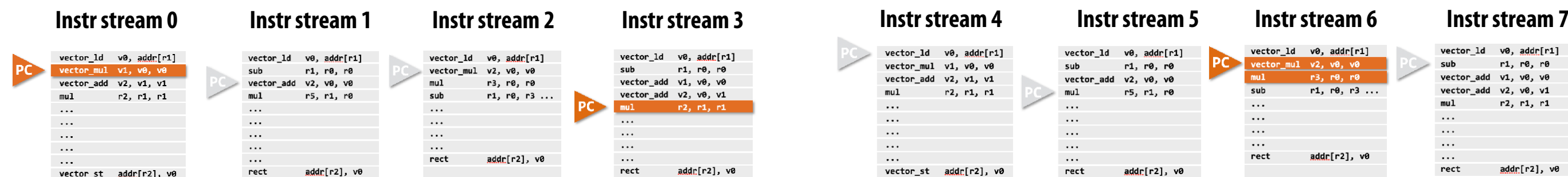


Mapping Algorithms to Execution Resources

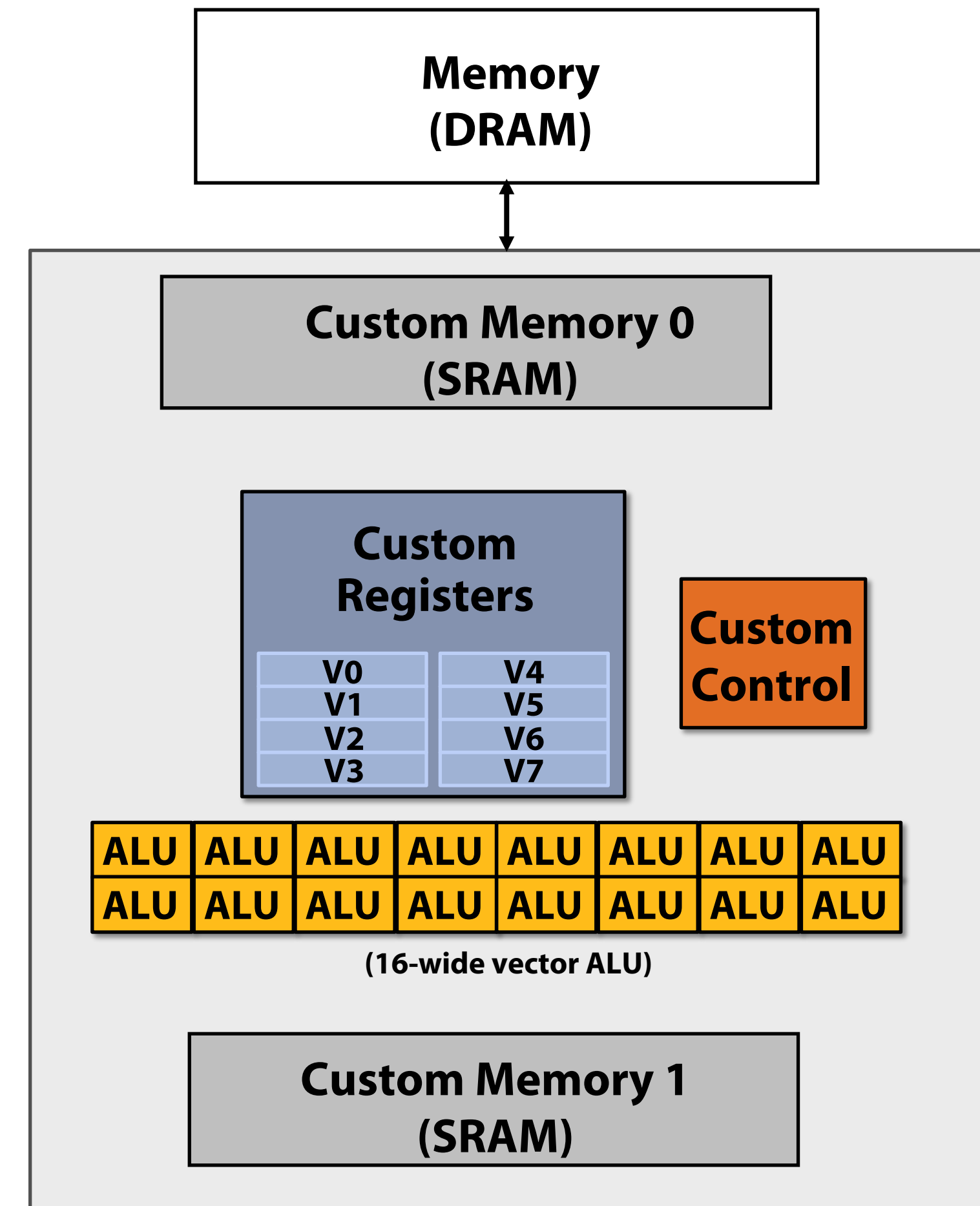
General Purpose Processor



**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**



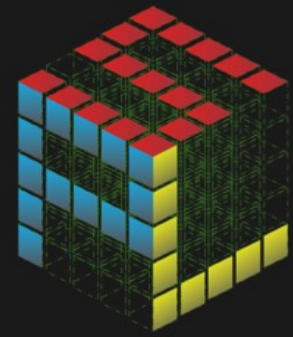
Special Purpose Processor (Accelerator)



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



HOME PUBLICATIONS TUTORIALS DOC FORUM

DOWNLOAD



SPATIAL

A high-level language for programming
accelerators

GET STARTED

VIEW SOURCE

Introducing Spatial

- **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

Typed storage templates

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

Explicit memory hierarchy

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

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

Let's build an accelerator to see how Spatial works

Code

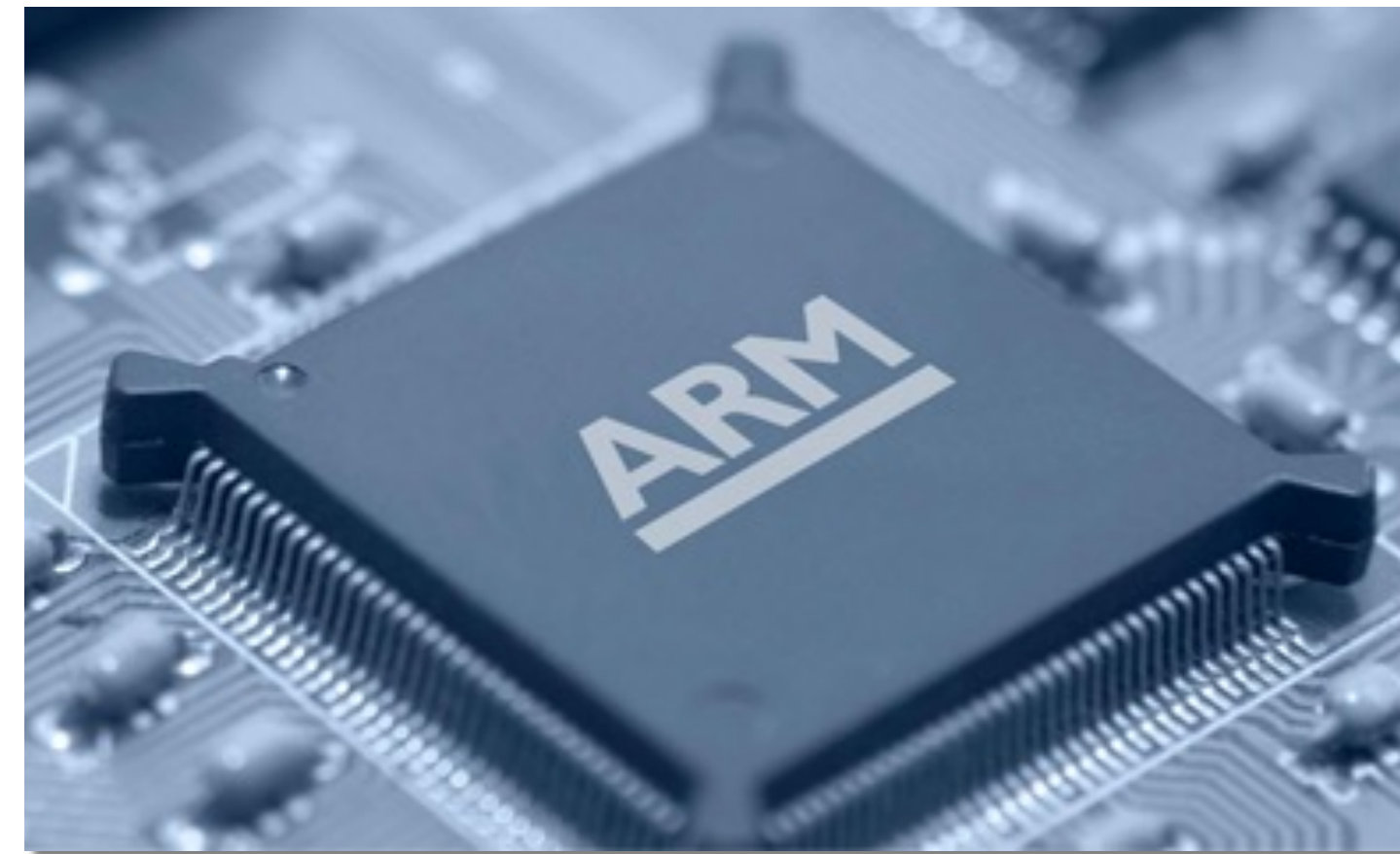
Sketch of generated hardware

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



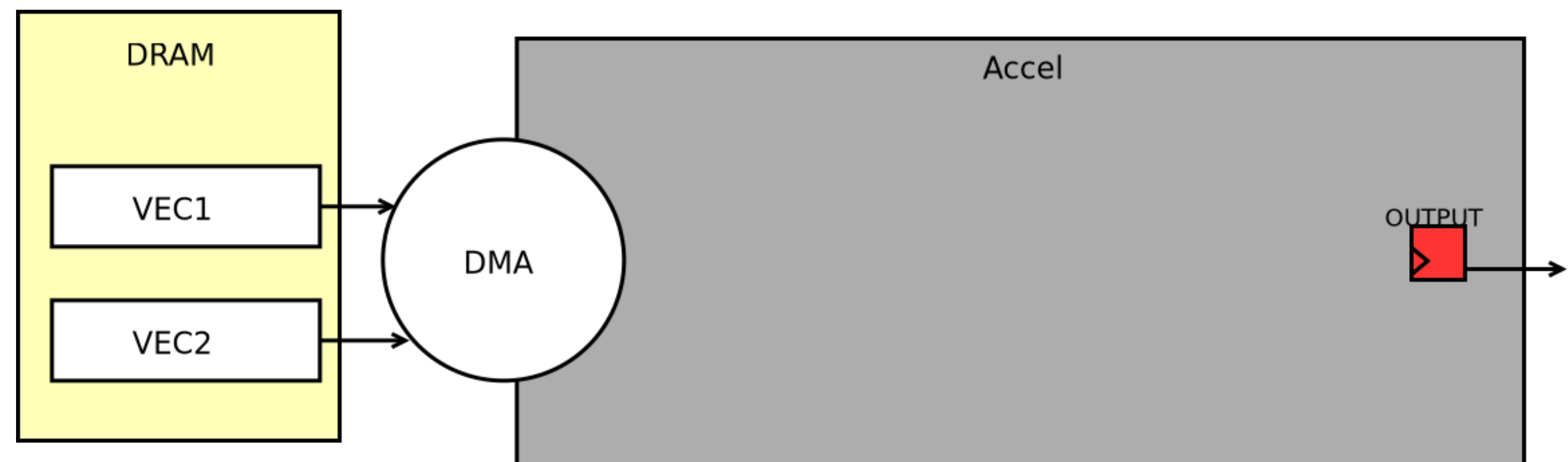
Inner Product in Spatial

```
// 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

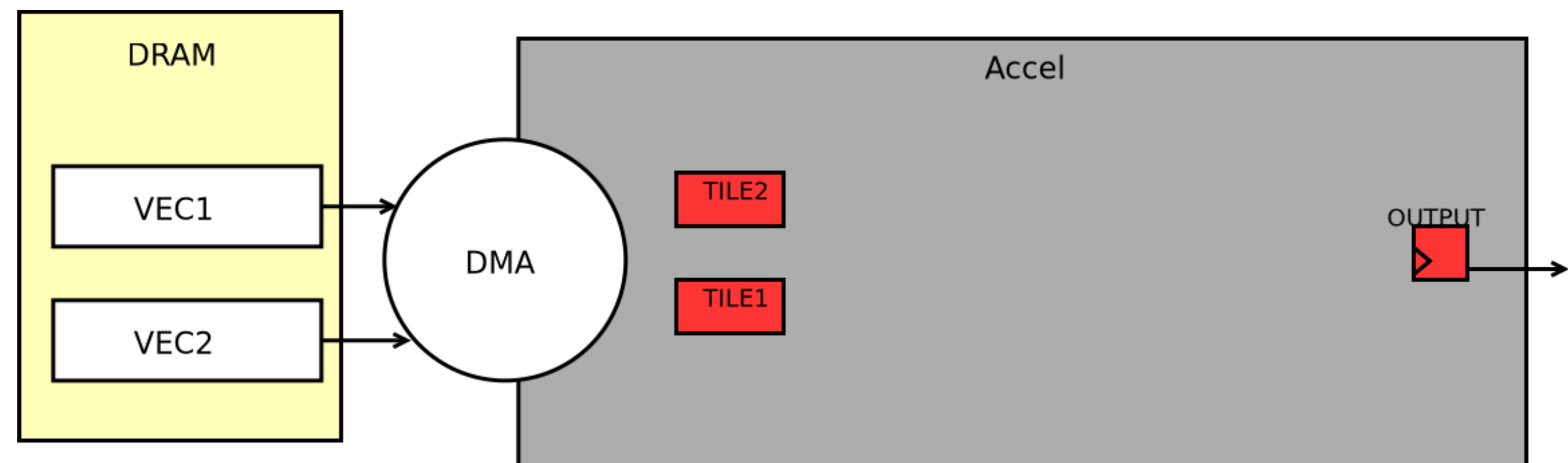
- Start of code looks like C example
- Accel executes “for” loop on the FPGA



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)
}
```

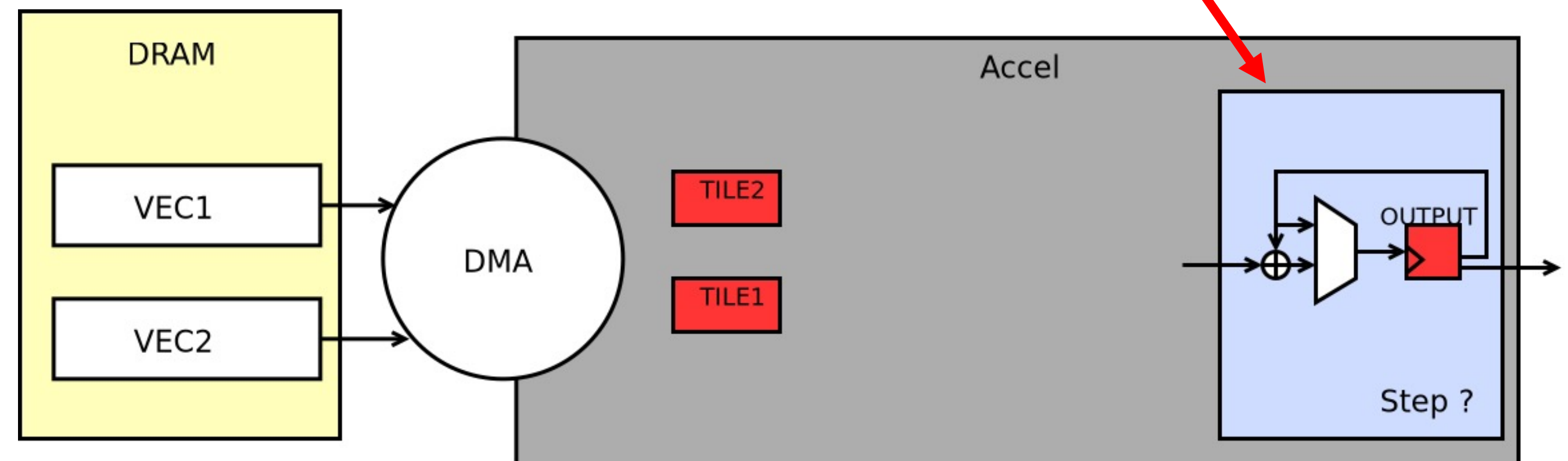


Inner Product in Spatial

```
// 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...
  }
}
```

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



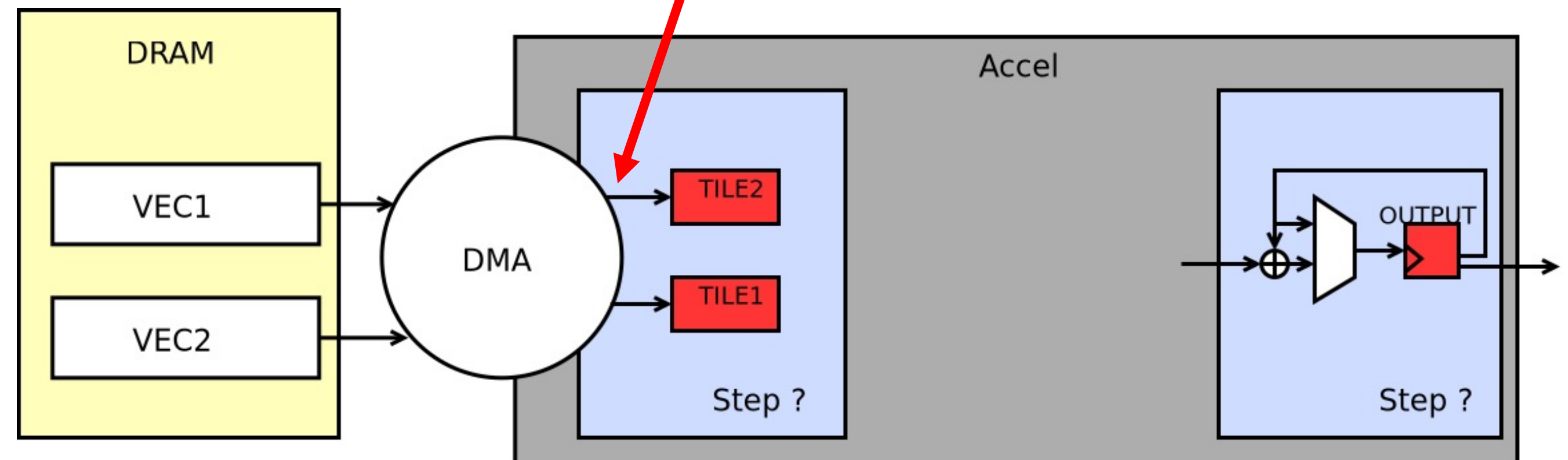
Inner Product in Spatial

- **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

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

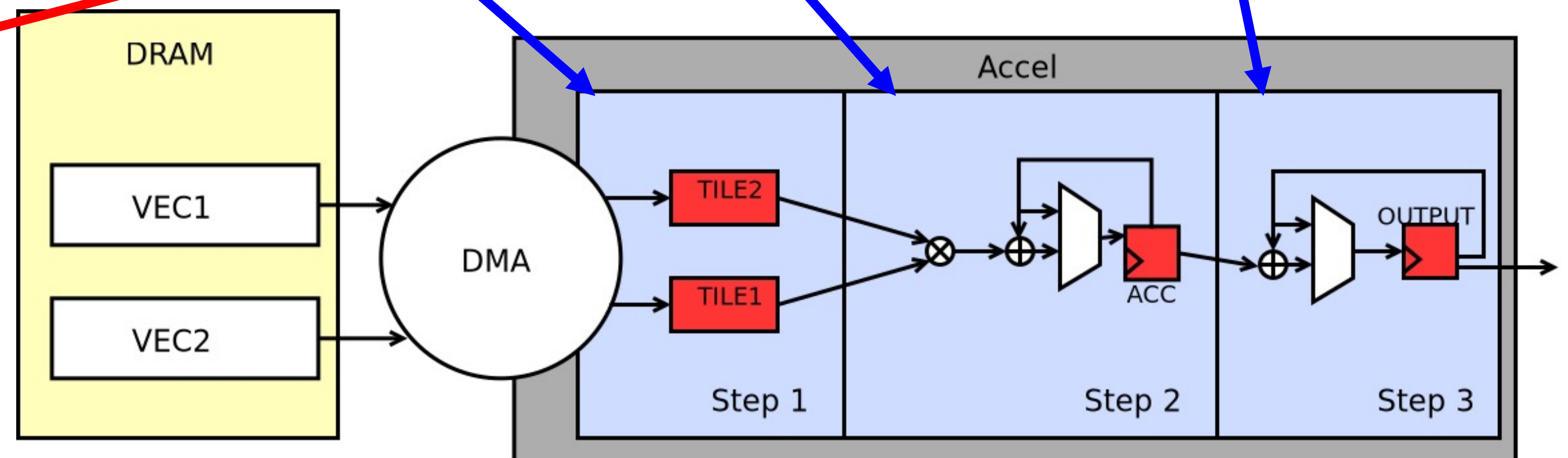
// Create accelerator
Accel {
  // Allocate on-chip memories
  val tile1 = SRAM[Int](tileSize)
  val tile2 = SRAM[Int](tileSize)
  // Specify outer loop
  Reduce(output)(N by tileSize){ t =>
    // Prefetch data
    tile1 load vec1(t :: t + tileSize)
    tile2 load vec2(t :: t + tileSize)
    // Multiply-accumulate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

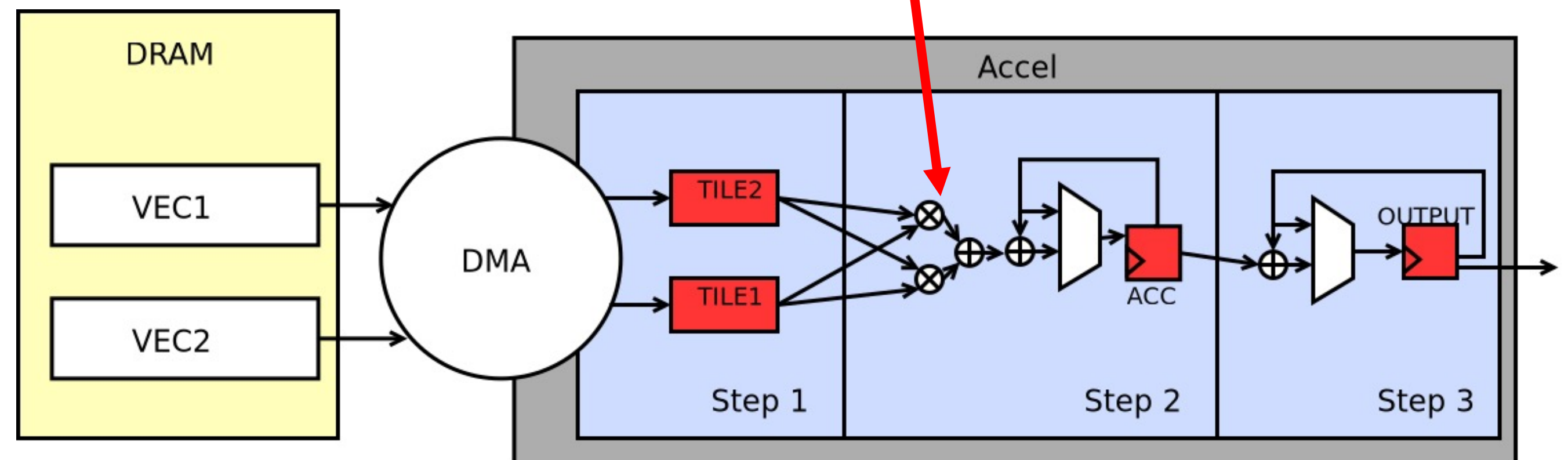


Inner Product in Spatial

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

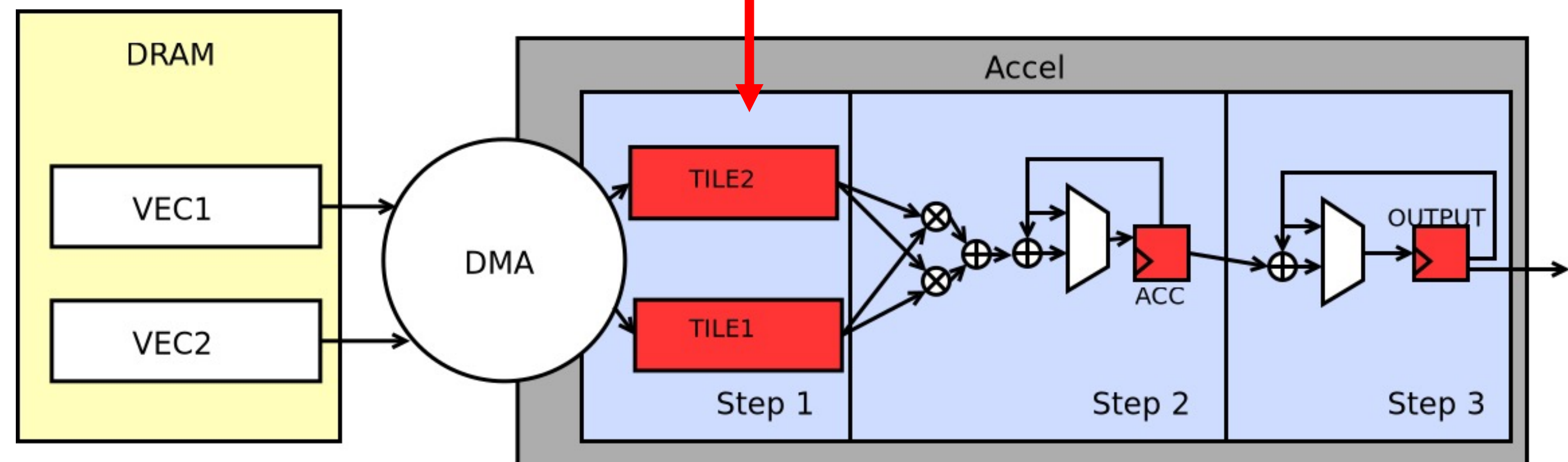
- **Spatial generates multi-step controllers**
- **Spatial manages communication with DRAM**
- **Spatial helps express hardware datapaths**



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

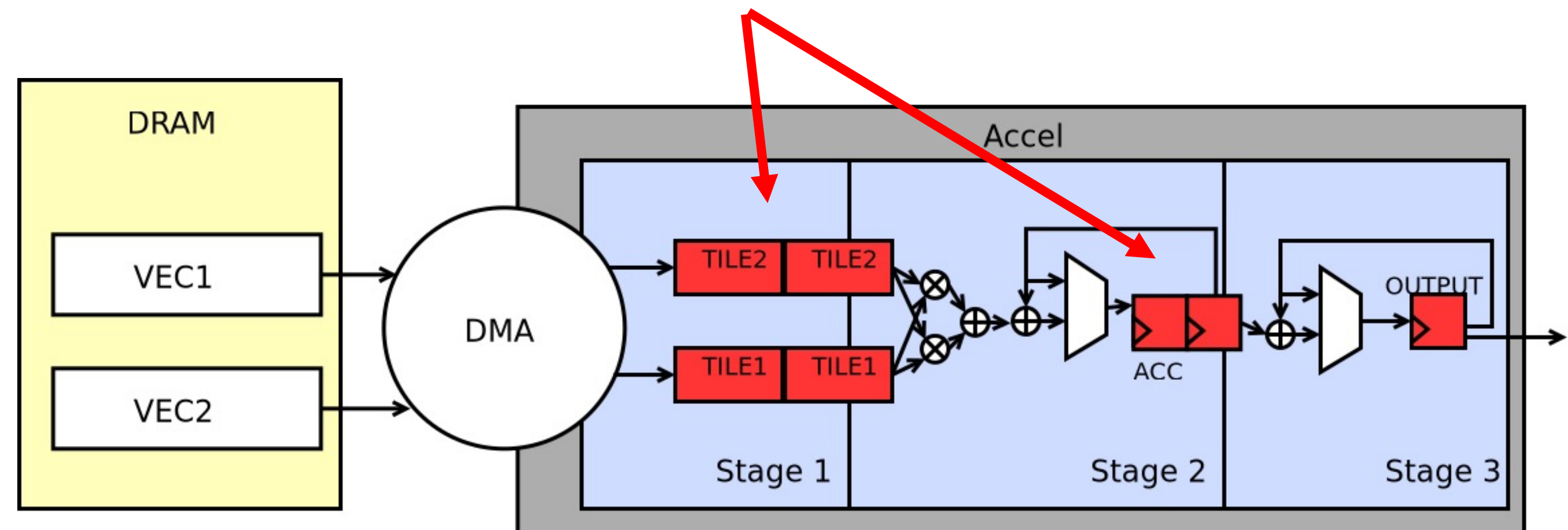


Inner Product in Spatial

```
// 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 control flow
 - 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** - 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

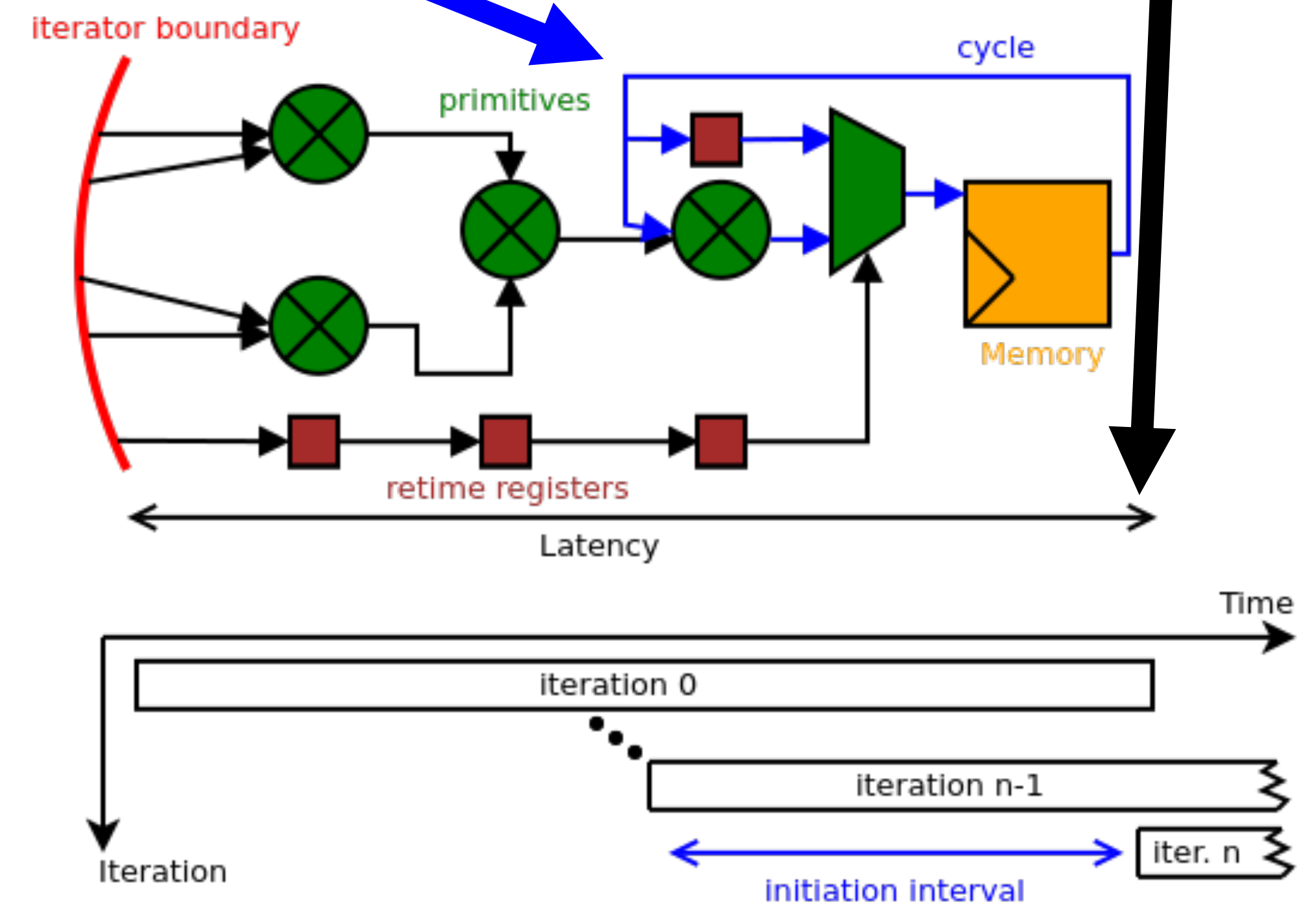
Inner controllers always execute iterations in a pipelined (overlapped) manner

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

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

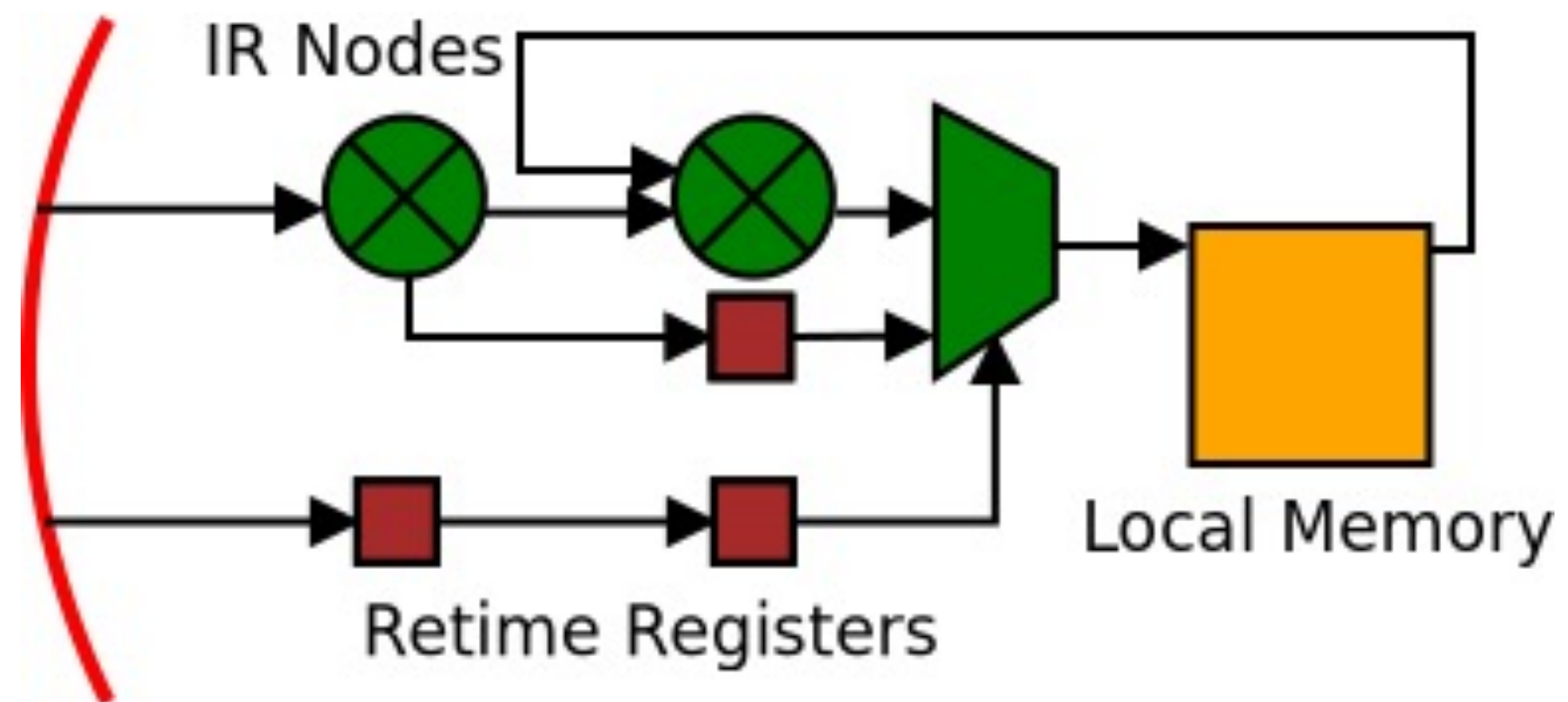
```
Foreach(N by 1, M by 1, P by 1, Q by 1){(i,j,p,q) =>
  val sum = i + j + p + q
  val next = reg.value ^ sum
  reg := mux(q == 0, reg.value, next)
}
```



Inner Controller Parallelization

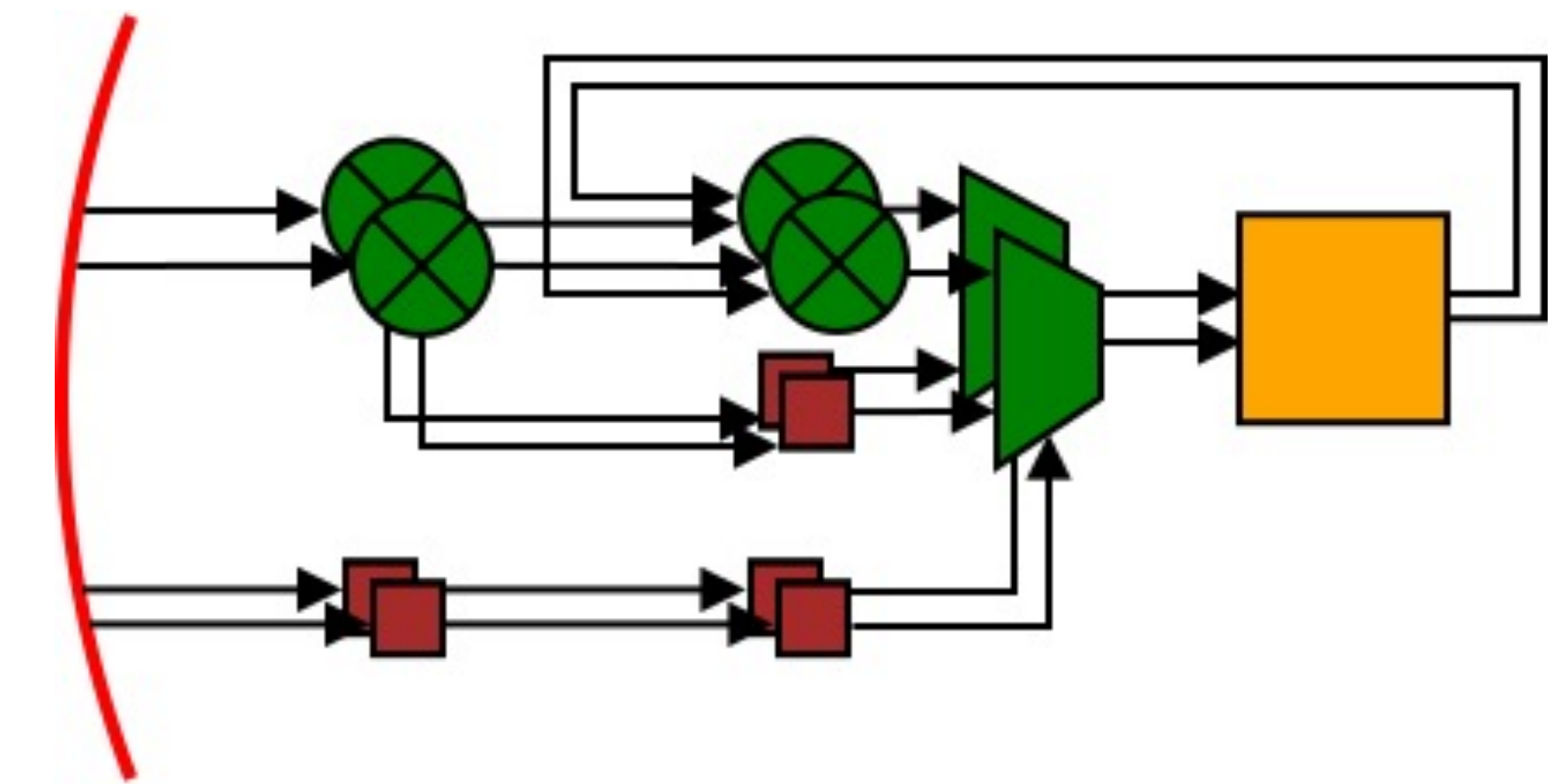
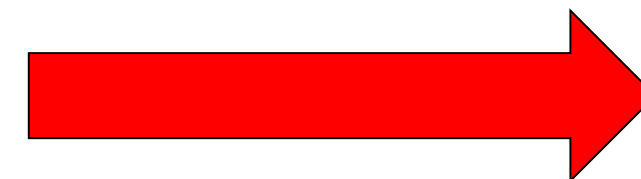
Parallelization of **inner controllers** results in vectorization of the counter chain and duplication of the dataflow graph

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



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

Increase parallelization



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:

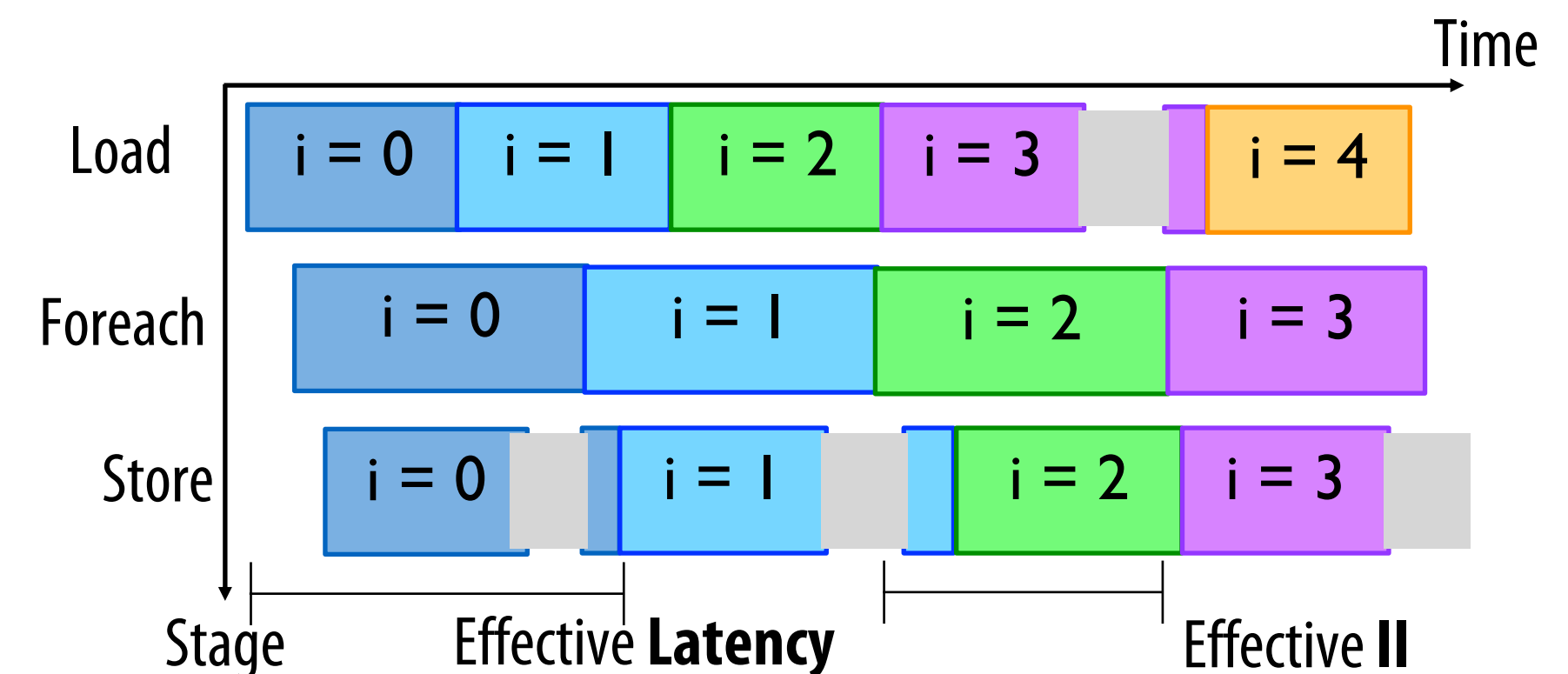
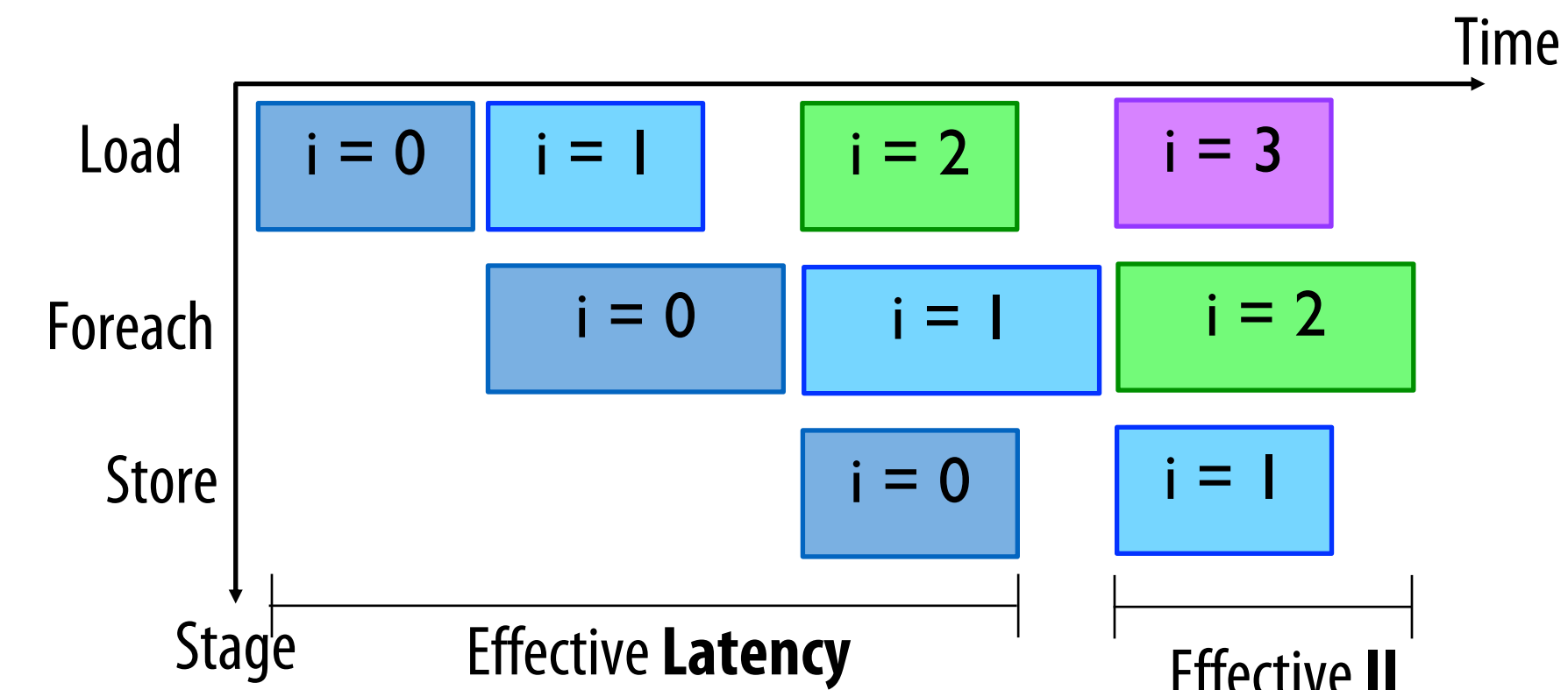
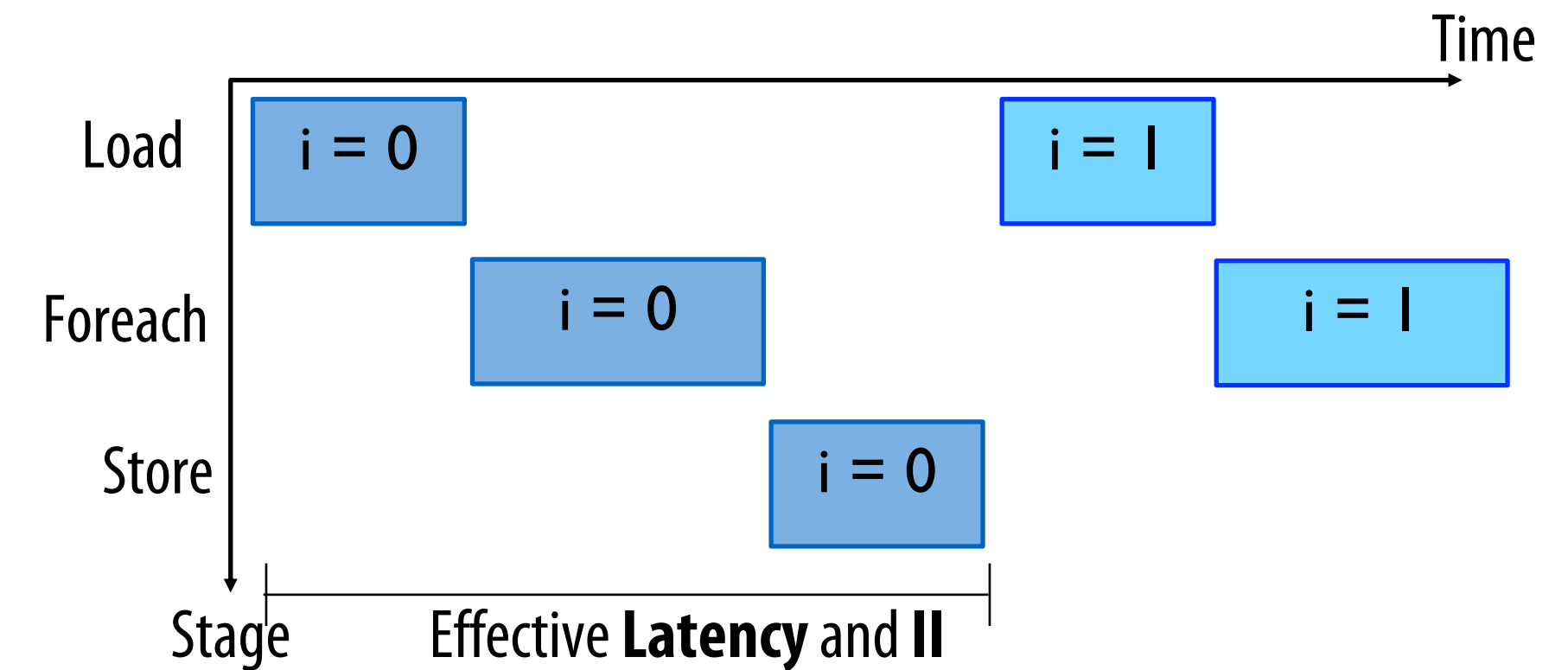
- **Sequential** – No overlapping of inner (child) controllers
- **Pipelined** – Coarse-grained overlapping of inner (child) controllers
- **Stream** – Data-driven execution of inner (child) controllers
- **Fork-Join** – Parallel execution of all inner (child) controllers

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
}
```

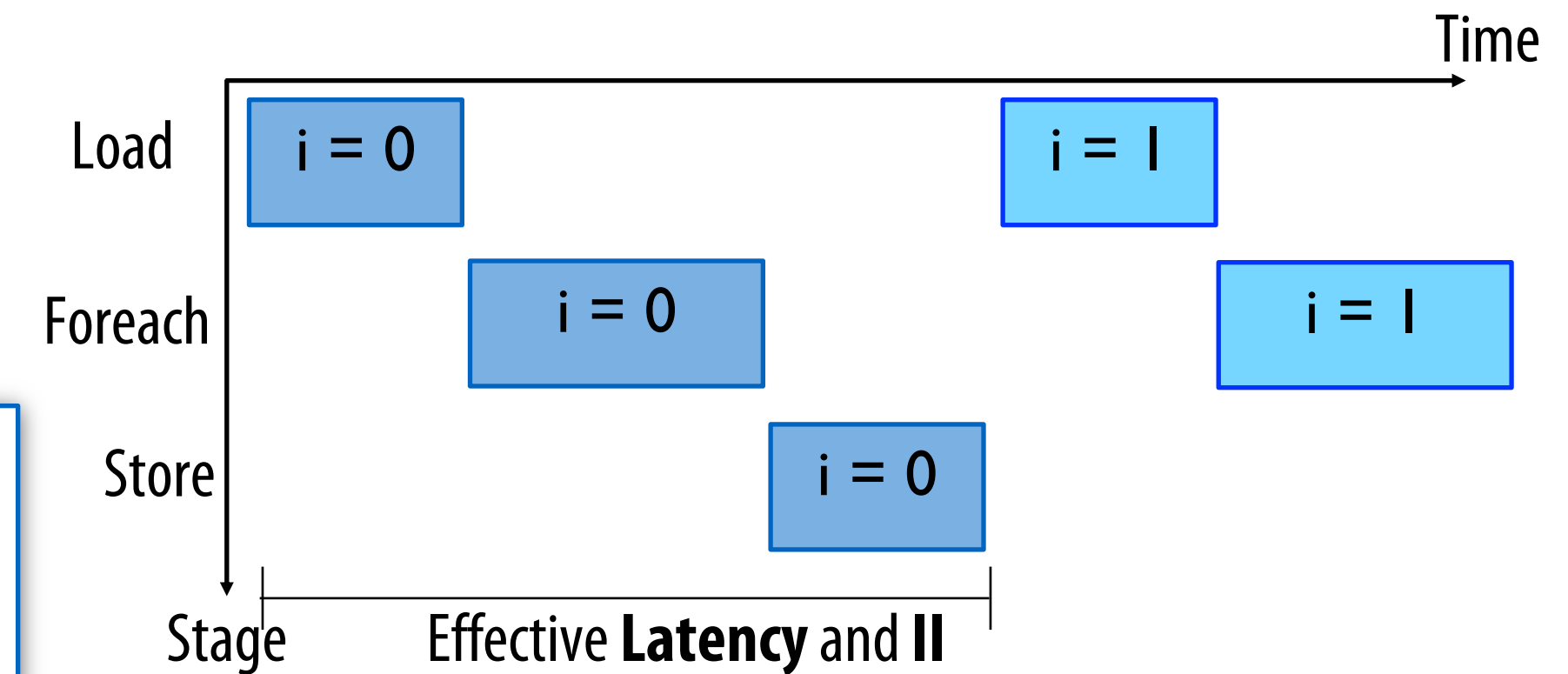
```
Stream.Foreach(...) { i =>
  fifoIn load dram
  Foreach(M by 1) { j => fifoOut.enq(fifoIn.deq() * j) }
  dram2 store fifoOut
}
```



A Closer Look at Schedules

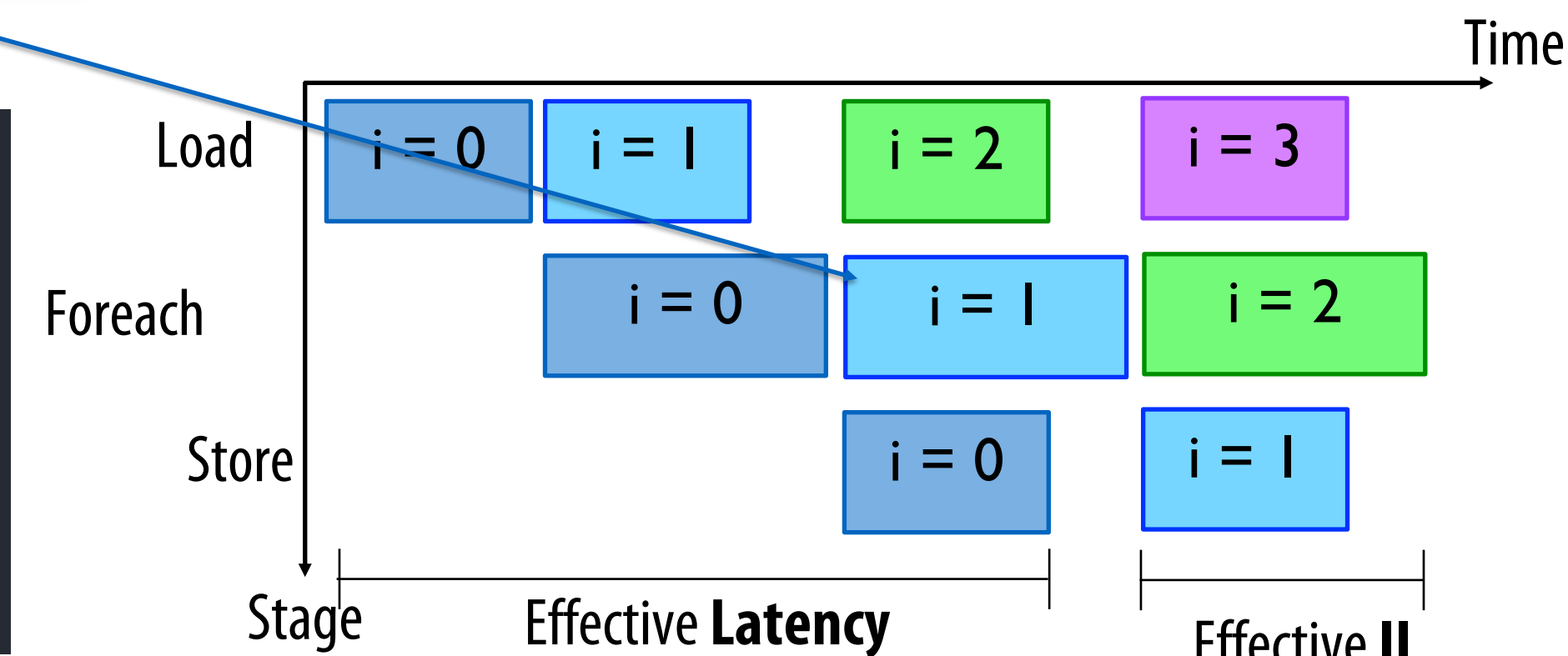
```
Sequential.Foreach(...) { i =>
  sram load dram
  Foreach(M by 1) { j => sram2(i) = sram(j) * j }
  dram store sram2
}
```

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



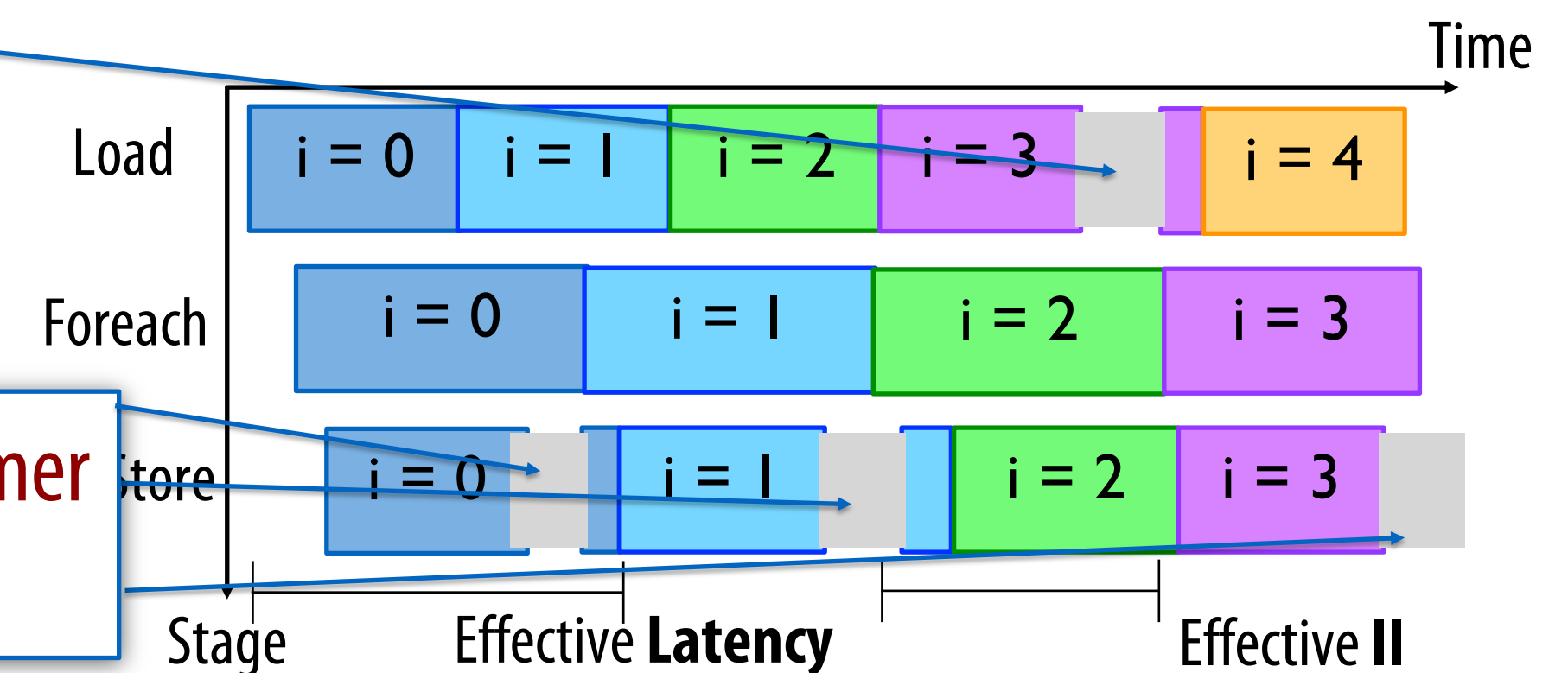
```
Pipelined.Foreach(...) { i =>
  sram load dram
  Foreach(M by 1) { j => sram2(j) = sram(j) * j }
  dram2 store sram2
}
```

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 Question

- **Spatial programmer's responsibility**

- **Spatial systems responsibility**

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

```
it x2024_UnitPipe extends x2045 {
  al x2006_Fifo_wdata = Wire(Vec(1, UInt(32.W)))
  al x2006_Fifo_readEn = Wire(Bool())
  al x2006_Fifo_writeEn = Wire(Bool())
  al x2006_Fifo_rdata = x2006_Fifo.io.out
  2006_Fifo.io.in := x2006_Fifo_wdata
  2006_Fifo.io.pop := x2006_Fifo_readEn
  2006_Fifo.io.push := x2006_Fifo_writeEn
  al x2012_UnitPipe_offset = 0
  al x2012_UnitPipe_sm = Module(new InnerPipe(1))
  2012_UnitPipe_sm.io.input.enable := x2012_UnitPipe_en;
  2012_UnitPipe_done := Utils.delay(x2012_UnitPipe_sm.io.output.done, x2012_UnitPipe_offset)
  al x2012_UnitPipe_rst_en = x2012_UnitPipe_sm.io.output.rst_en
  al x2012_UnitPipe_datapath_en = x2012_UnitPipe_en & ~x2012_UnitPipe_rst_en
  2012_UnitPipe_sm.io.input.ctr_done := Utils.delay(x2012_UnitPipe_sm.io.output.ctr_en, 1 + x2012_UnitPipe_offset)
  al x2012_UnitPipe_ctr_en = x2012_UnitPipe_sm.io.output.ctr_inc
  2012_UnitPipe_sm.io.input.forever := false.B
  al x2007_data = Vec(List(io.memStreams(1).rdata.bits(0), io.memStreams(1).rdata.bits(1), io.memStreams(1).rdata.bits(2),
  o.memStreams(1).rdata.bits(3), io.memStreams(1).rdata.bits(4), io.memStreams(1).rdata.bits(5),
  o.memStreams(1).rdata.bits(6), io.memStreams(1).rdata.bits(7), io.memStreams(1).rdata.bits(8),
  o.memStreams(1).rdata.bits(9), io.memStreams(1).rdata.bits(10), io.memStreams(1).rdata.bits(11),
  o.memStreams(1).rdata.bits(12), io.memStreams(1).rdata.bits(13), io.memStreams(1).rdata.bits(14), io.memStreams(1).rdata.bits(15)))
  o.memStreams(1).cmd.bits.addr(0) := x2005_data(64, 33)
  o.memStreams(1).cmd.bits.size := x2005_data(32, 1)
  o.memStreams(1).cmd.valid := x2005_valid
  o.memStreams(1).cmd.bits.isWr := ~x2005_data(0)
  al x2023_UnitPipe_offset = 0
  al x2023_UnitPipe_sm = Module(new MetaPipe(2))
  2023_UnitPipe_sm.io.input.enable := x2023_UnitPipe_en;
  2023_UnitPipe_done := Utils.delay(x2023_UnitPipe_sm.io.output.done, x2023_UnitPipe_offset)
  al x2023_UnitPipe_rst_en = x2023_UnitPipe_sm.io.output.rst_en
  2023_UnitPipe_sm.io.input.numIter := (1.U)
  2023_UnitPipe_sm.io.input.rst := x2023_UnitPipe_resetter
  al x2023_UnitPipe_datapath_en = x2023_UnitPipe_en & ~x2023_UnitPipe_rst_en
  2023_UnitPipe_sm.io.input.forever := false.B
  2023_UnitPipe_sm.io.input.stageDone(0) := x2015_UnitPipe_done;
  2015_UnitPipe_en := x2023_UnitPipe_sm.io.output.stageEnable(0) & ~x2006_Fifo.io.empty
  2015_UnitPipe_resetter := x2023_UnitPipe_sm.io.output.rst_en
  2023_UnitPipe_sm.io.input.stageDone(1) := x2022_unrForeach_done;
  2022_unrForeach_en := x2023_UnitPipe_sm.io.output.stageEnable(1)
  2022_unrForeach_resetter := x2023_UnitPipe_sm.io.output.rst_en

  it x2032_UnitPipe extends x2044_UnitPipe {
    al x2028 = io.argIns(3)
    al x2029_tuple = util.Cat(x2028, 256.U(32.W), true.B)
    2025_valid := x2044_UnitPipe_done & true.B
    2025_data := x2029_tuple
    2026_Fifo_writeEn := x2032_UnitPipe_ctr_en & true.B
    2026_Fifo_wdata := Vec(List(64.U(32.W)))

    it x2035_UnitPipe extends x2043_UnitPipe {
      2026_Fifo_readEn := x2035_UnitPipe_ctr_en & true.B
      al x2034_deqFrom2026 = x2026_Fifo_rdata(0)

      it x2042_unrForeach extends x2043_UnitPipe {
        1234 := x2036_ctr(0)
        al x1236_vecified = Array(b1235)
        al x2038 = x1236_vecified.zipWithIndex.map(case (en, i) => Mux(en, x2027_data(i), 0.U(32.W)))
        al x2039_elem0 = x2038.apply(0)
        al x2040_vecified = Array(x2039_elem0)
        al x2041_parSt_wVec = Wire(Vec(1, new multidimM(1, 32)))
        2041_parSt_wVec.zip(x2040_vecified).foreach case (port, dat) => port.data := dat
        2041_parSt_wVec(0).en := x1236_vecified(0)
        2041_parSt_wVec(0).addr(0) := b1234
        2004_sram_0.connectWPort(x2041_parSt_wVec, x2042_unrForeach_eng, List(0))

        it x2043_UnitPipe extends x2044_UnitPipe {
          al x2035_UnitPipe_offset = 0
          al x2035_UnitPipe_sm = Module(new InnerPipe(1))
          2035_UnitPipe_sm.io.input.enable := x2035_UnitPipe_en;
          2035_UnitPipe_done := Utils.delay(x2035_UnitPipe_sm.io.output.done, x2035_UnitPipe_offset)
          al x2035_UnitPipe_rst_en = x2035_UnitPipe_sm.io.output.rst_en
          al x2035_UnitPipe_datapath_en = x2035_UnitPipe_en & ~x2035_UnitPipe_rst_en
          2035_UnitPipe_sm.io.input.ctr_done := Utils.delay(x2035_UnitPipe_sm.io.output.ctr_en, 1 + x2035_UnitPipe_offset)
          al x2035_UnitPipe_ctr_en = x2035_UnitPipe_sm.io.output.ctr_inc
          2035_UnitPipe_sm.io.input.forever := false.B
          al x2037_ctrchain_strides = List(1.U(32.W))
          al x2037_ctrchain_maxes = List(64.U(32.W))
          al x2037_ctrchain = Module(new templates.Counter(List(1)))
          2037_ctrchain.io.input.maxes.zip(x2037_ctrchain_maxes).foreach { case (port, max) => port := max }
          2037_ctrchain.io.input.strides.zip(x2037_ctrchain_strides).foreach { case (port, stride) => port := stride }
          2037_ctrchain.io.input.enable := x2037_ctrchain_en
          2037_ctrchain_done := x2037_ctrchain.io.output.done
          2037_ctrchain.io.input.reset := x2037_ctrchain_resetter
          al x2037_ctrchain_maxed = x2037_ctrchain.io.output.saturated
          al x2036_ctr = (0 until 1).map(j => x2037_ctrchain.io.output.counts(0 + j))
          al x2042_unrForeach_level0_iters = (64.U(32.W) - 0.U(32.W)) / (1.U(32.W) * 1.U(32.W))
          ux(((64.U(32.W) - 0.U(32.W)) % (1.U(32.W) * 1.U(32.W))) == 0.U, 0.U, 1.U)
          al x2042_unrForeach_offset = 0
          al x2042_unrForeach_sm = Module(new InnerPipe(1))
          2042_unrForeach_sm.io.input.enable := x2042_unrForeach_en;
          2042_unrForeach_done := Utils.delay(x2042_unrForeach_sm.io.output.done, x2042_unrForeach_offset)
          al x2042_unrForeach_rst_en = x2042_unrForeach_sm.io.output.rst_en
          al x2042_unrForeach_datapath_en = x2042_unrForeach_sm.io.output.datapath_en
          2037_ctrchain_en := x2042_unrForeach_sm.io.output.ctr_inc
          2037_ctrchain_resetter := x2042_unrForeach_rst_en
          2042_unrForeach_sm.io.input.ctr_done := Utils.delay(x2037_ctrchain_done, 1 + x2042_unrForeach_offset)
          2042_unrForeach_sm.io.input.forever := false.B
          al b1235 = b1234 < x2037_ctrchain_maxes(0)
        }
      }
    }
  }
}
```

Chisel: ~3200 lines

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

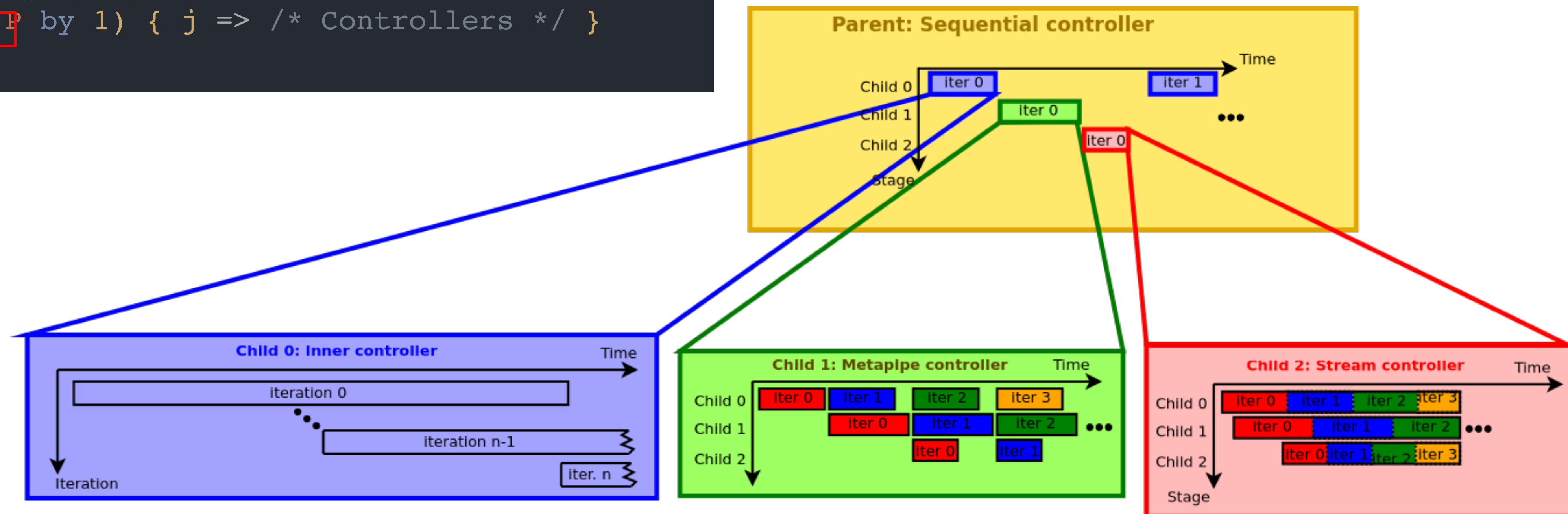
```
Sequential.Foreach(Q by TS){ i =>
  Foreach(N by 1){ j => /* Primitives */ }
  Pipe.Foreach(M by 1){ j => /* Controllers */ }
  Stream.Foreach(P by 1) { j => /* Controllers */ }
}
```

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 */ }
}
```



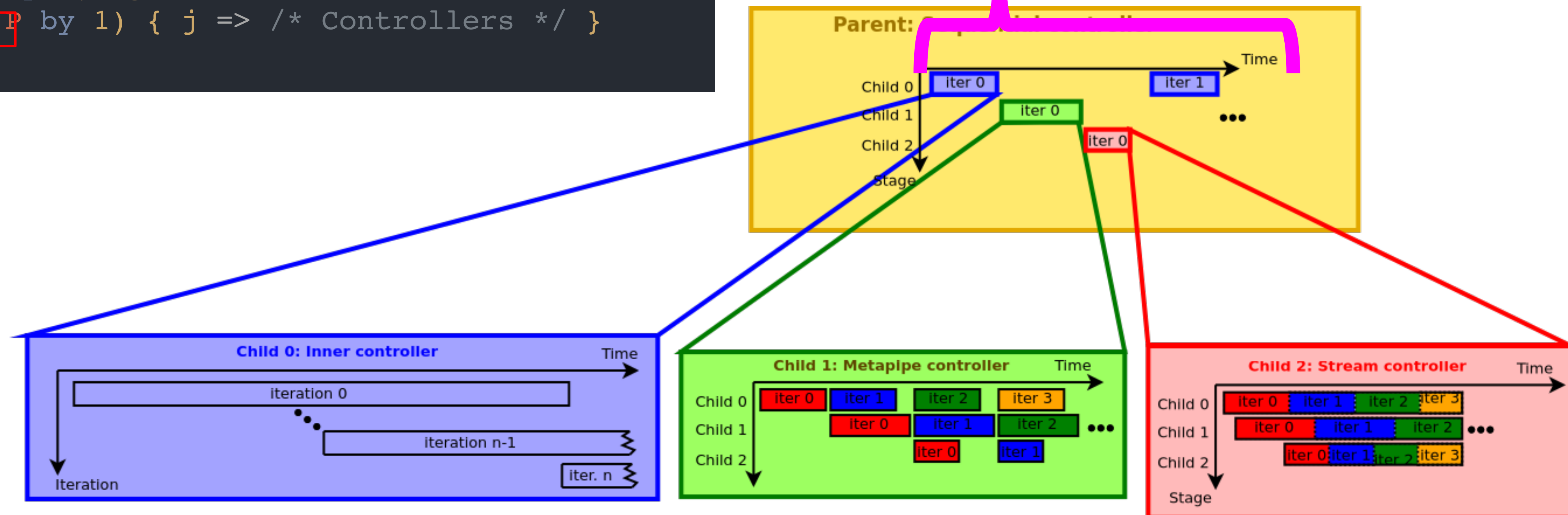
Performance Debugging

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

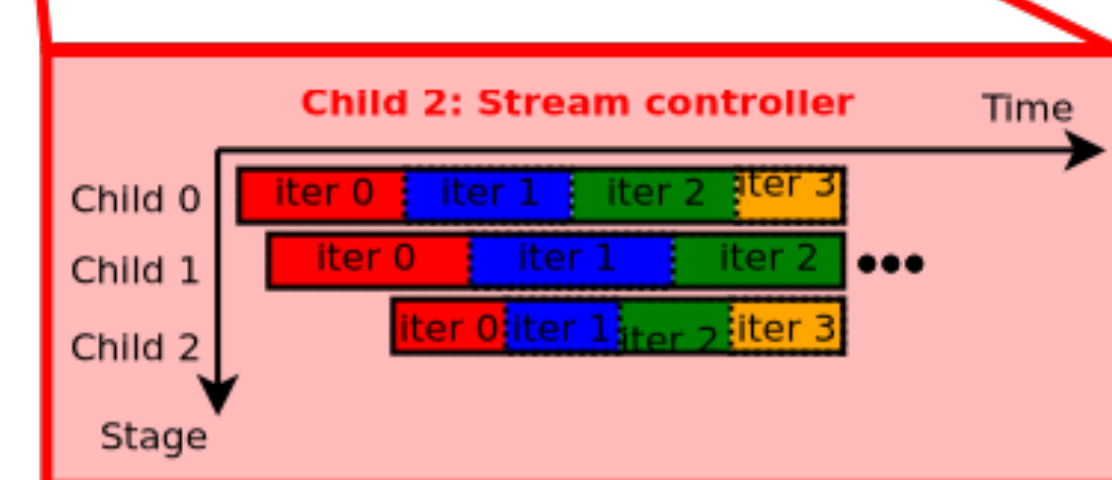
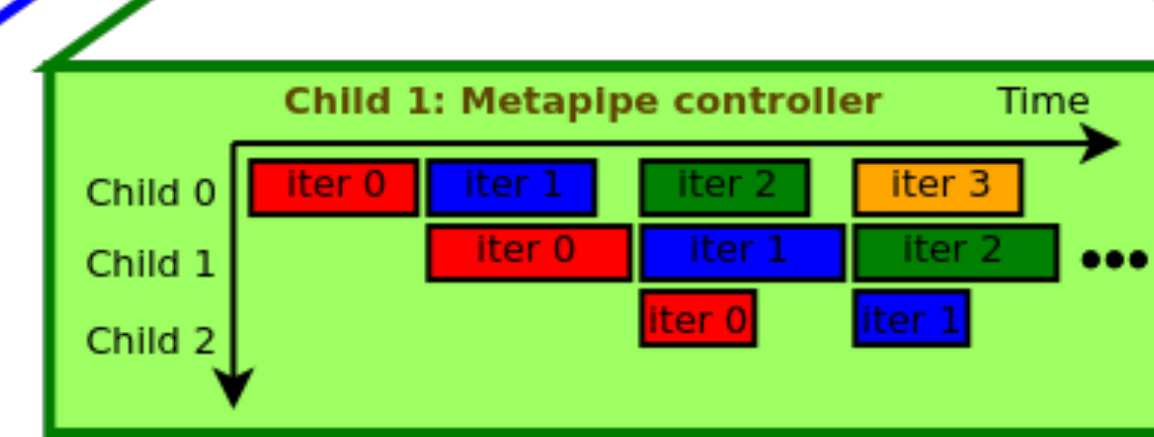
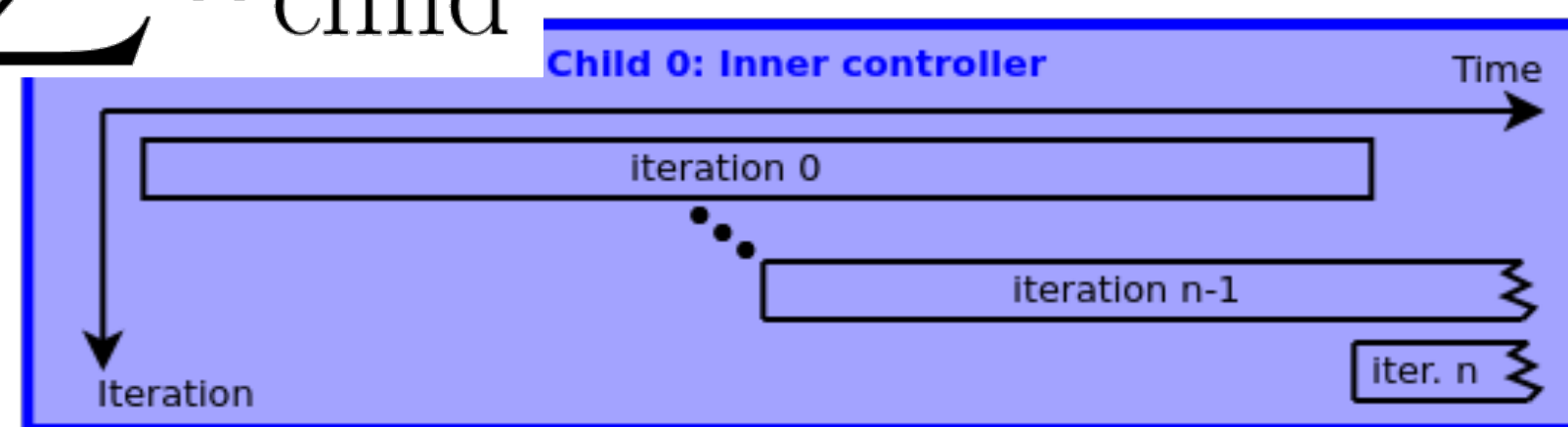
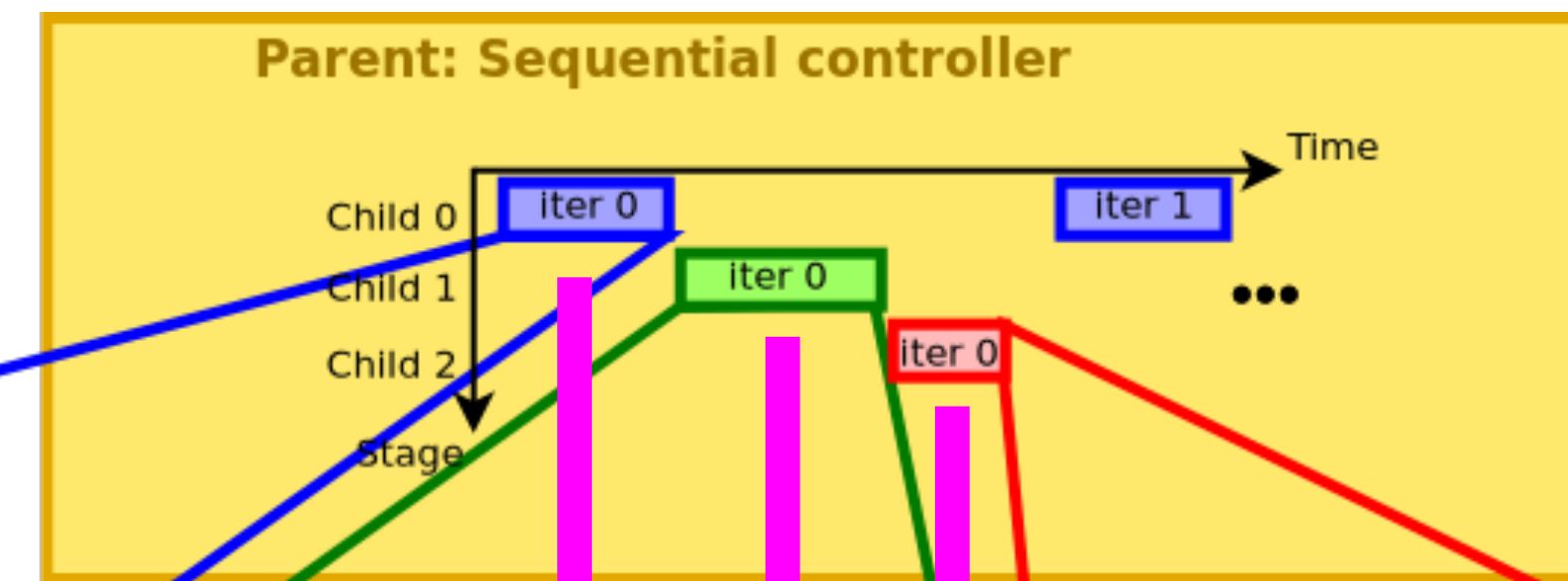
We want to minimize the execution time of the **Parent Sequential Controller**

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

```
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 */ }
}
```

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

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

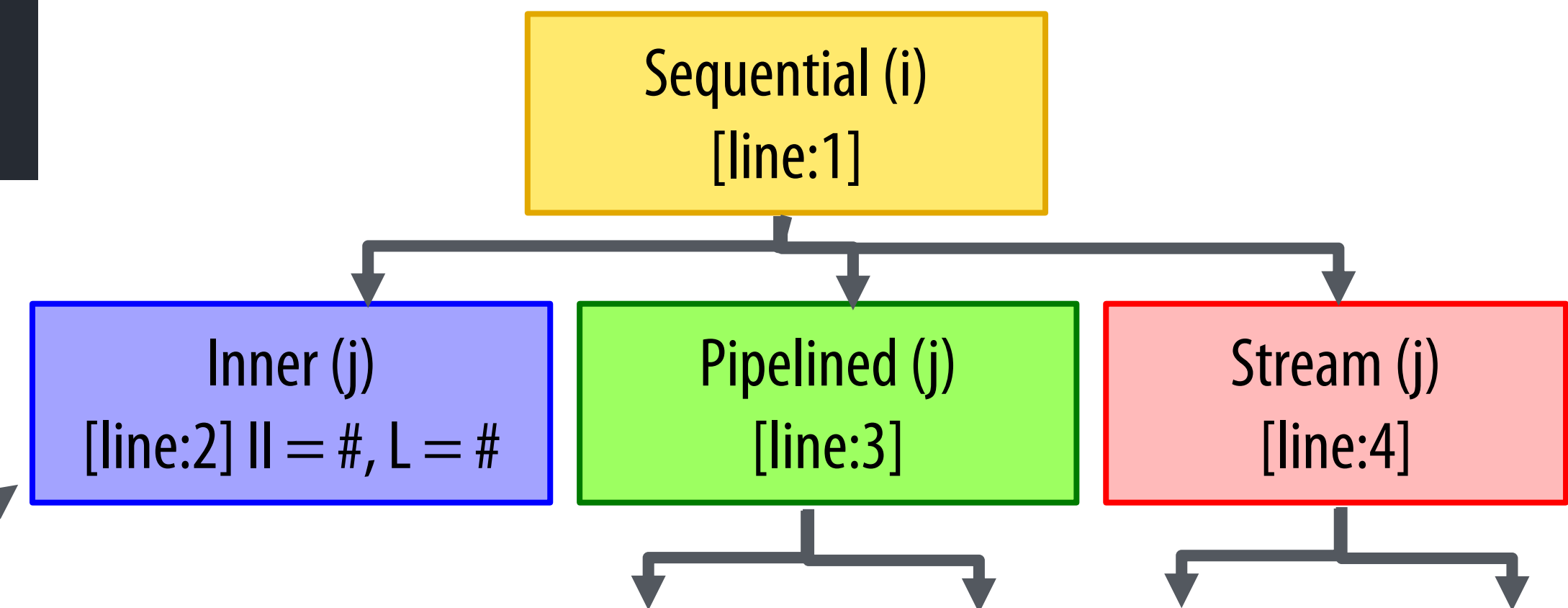


Performance Debugging with Controller Hierarchy

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 */ }
}
```



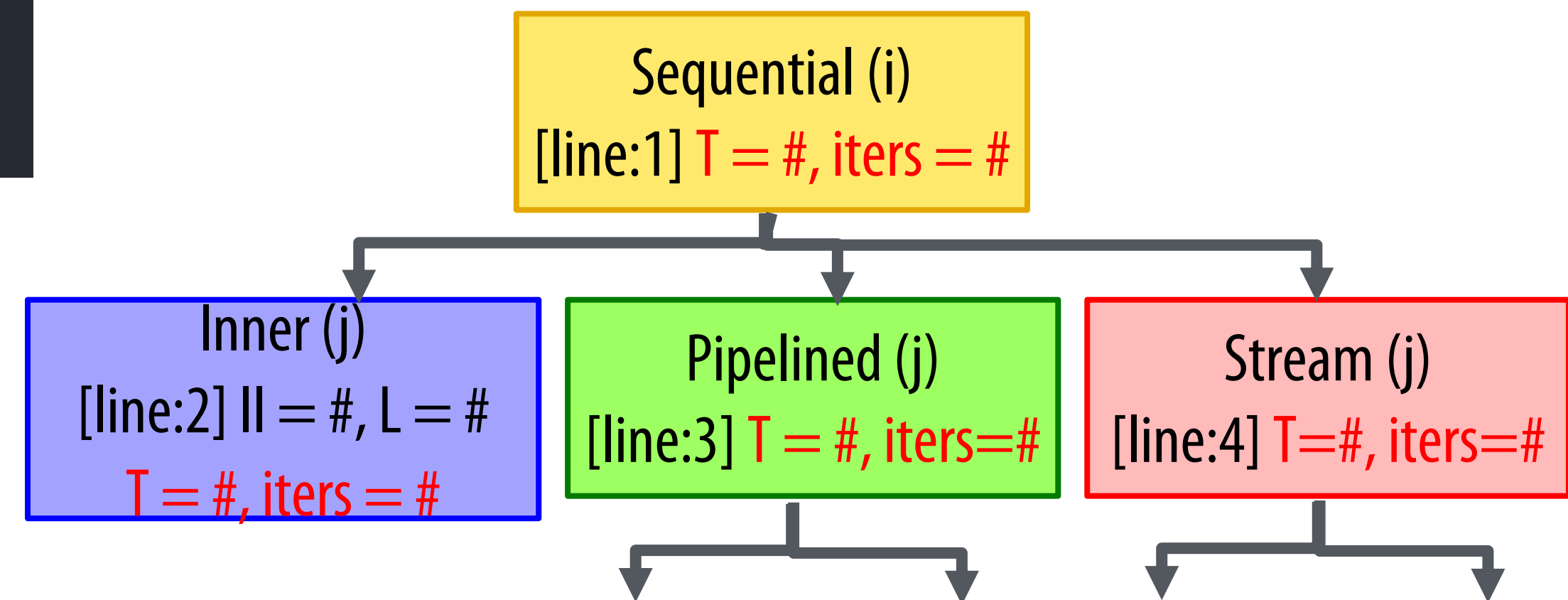
Line numbers link controllers back to source code

Static properties (ll 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 */ }  
}
```

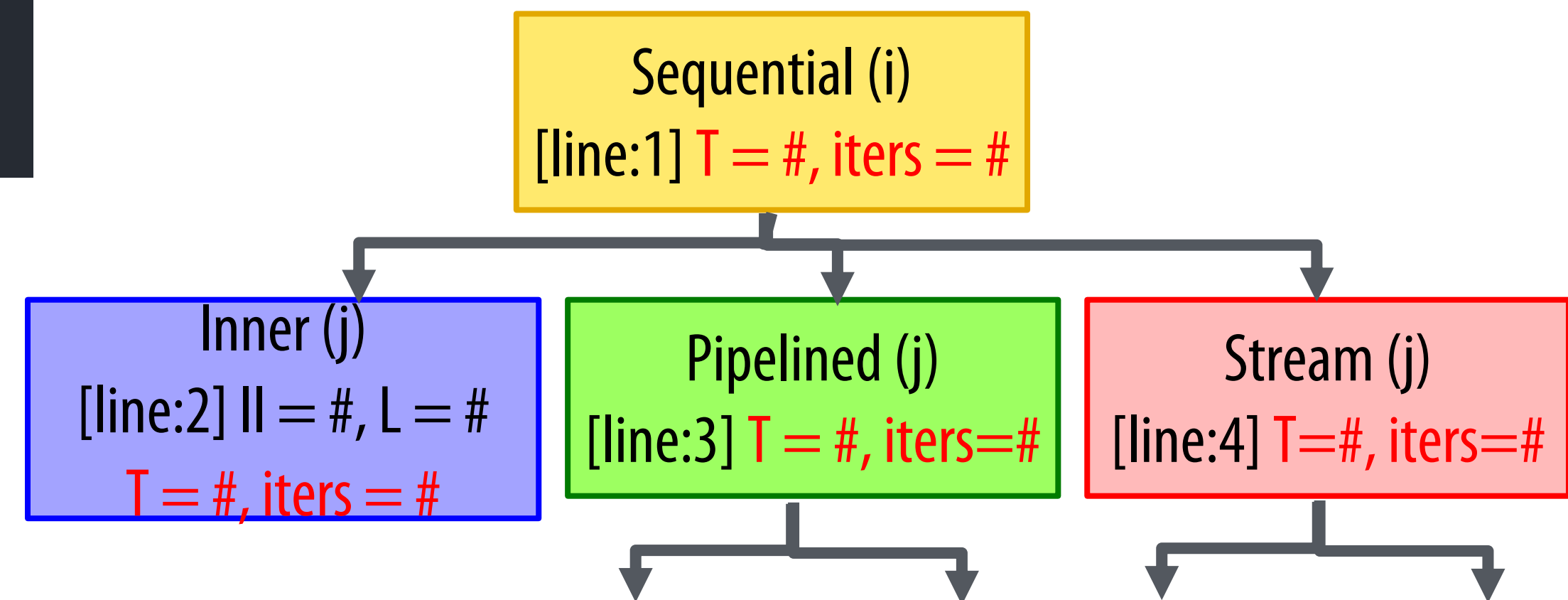


Actual **T and iteration counts** are automatically collected and overlaid after execution

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:

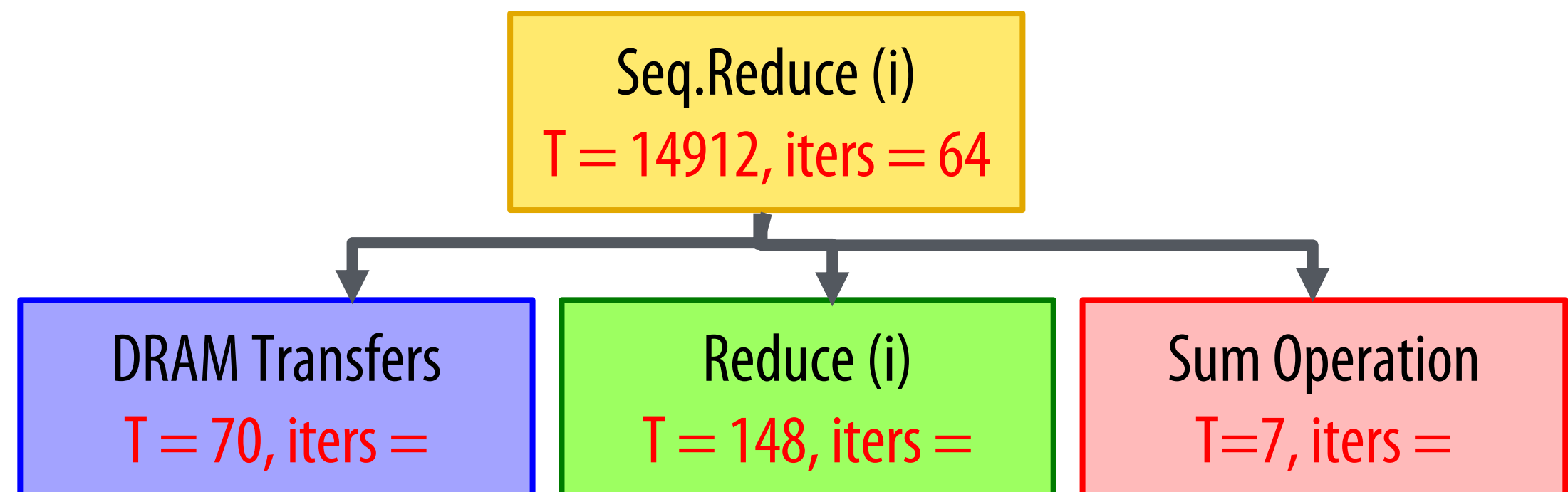
- Outer controller schedule (Reduce)
- ParO
- ParI

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 Par0){ 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 Par1){ i =>
      tile1(i) * tile2(i)
    }{a, b => a + b}
  }{a, b => a + b}
}
```

The baseline implementation is $\text{Par1}=1$, $\text{Par0}=1$, and $\text{schedule}=\text{Sequential}$

Our instrumented controller tree will look like this



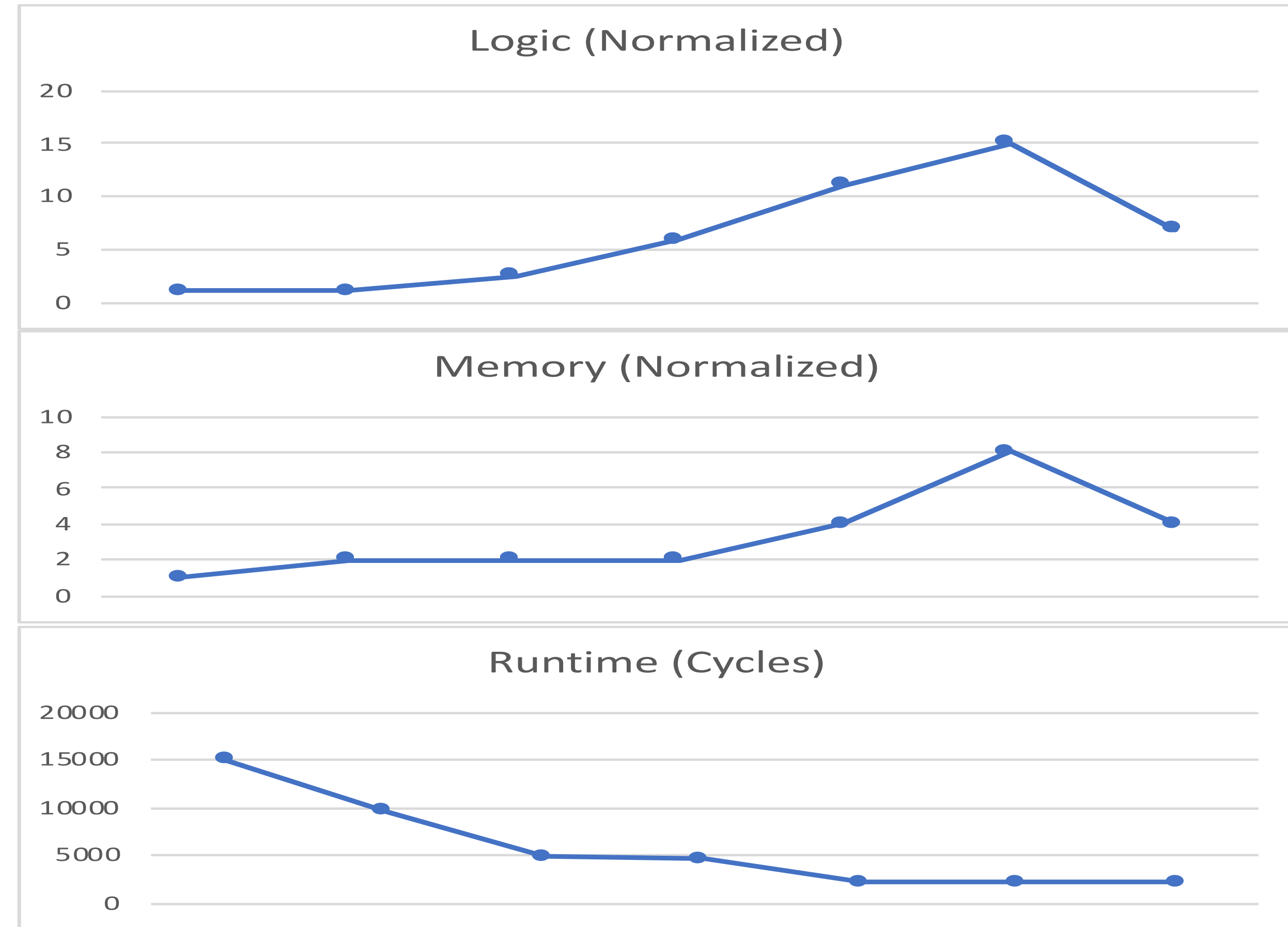
Optimization Goal

```
// 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

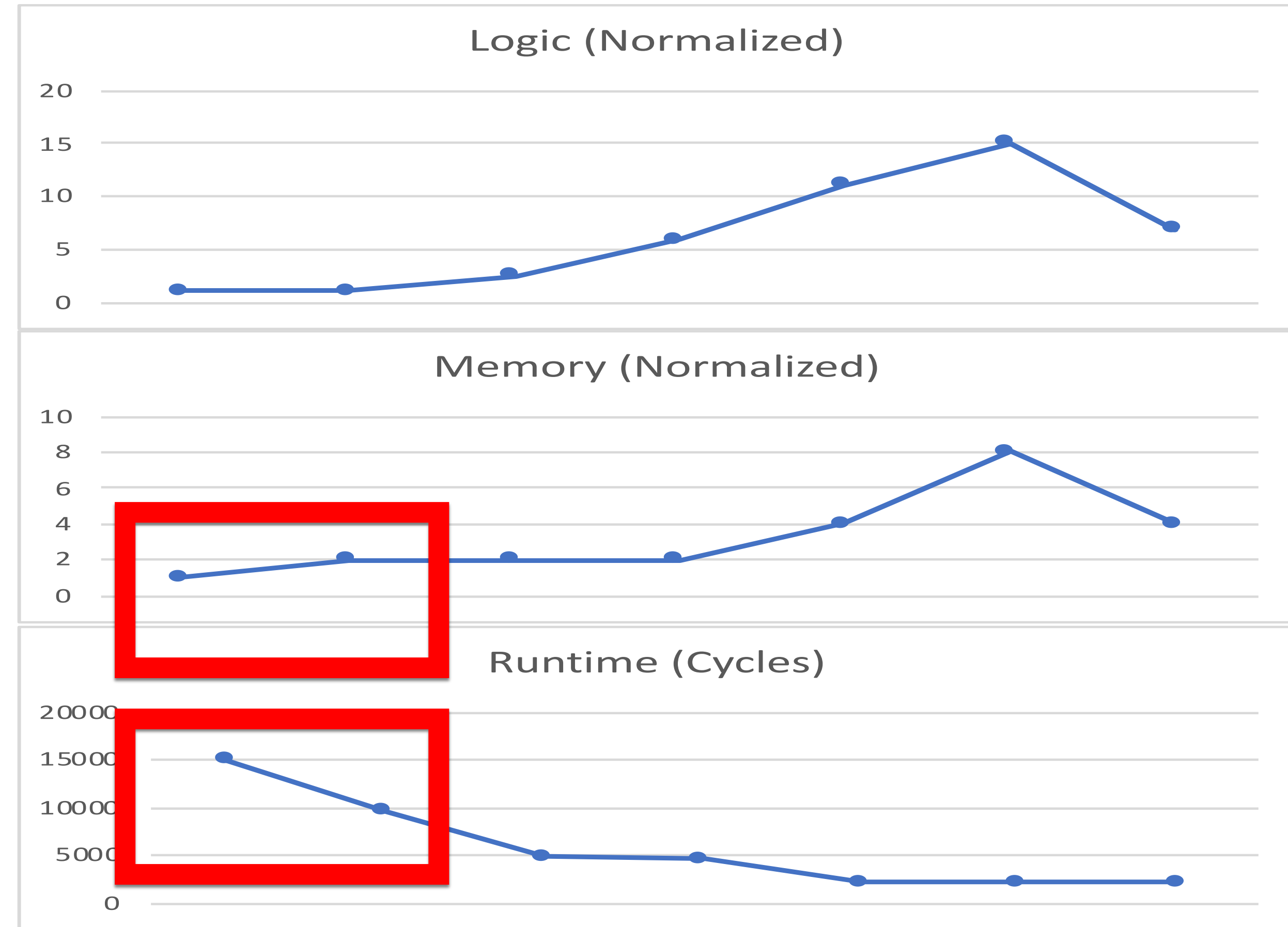
- By optimizing the code, we can improve execution time by $\sim 7x$
- The best design increases logic by $\sim 6x$ and memory by $\sim 4x$



Parl	1	1	2	4	4	4	2
Par0	1	1	1	1	2	4	2
Sched	Seq	Pipe	Pipe	Pipe	Pipe	Pipe	Pipe

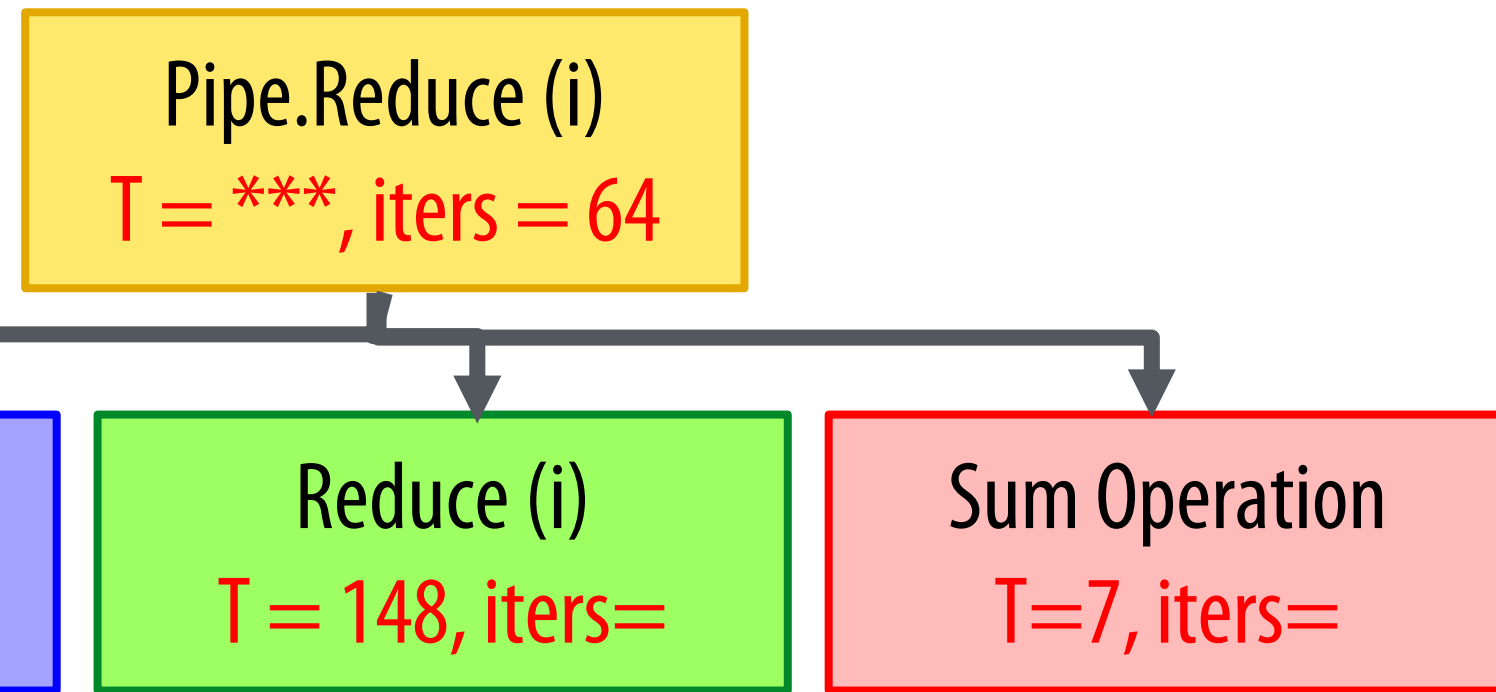
Sequential vs. Pipelined

- 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

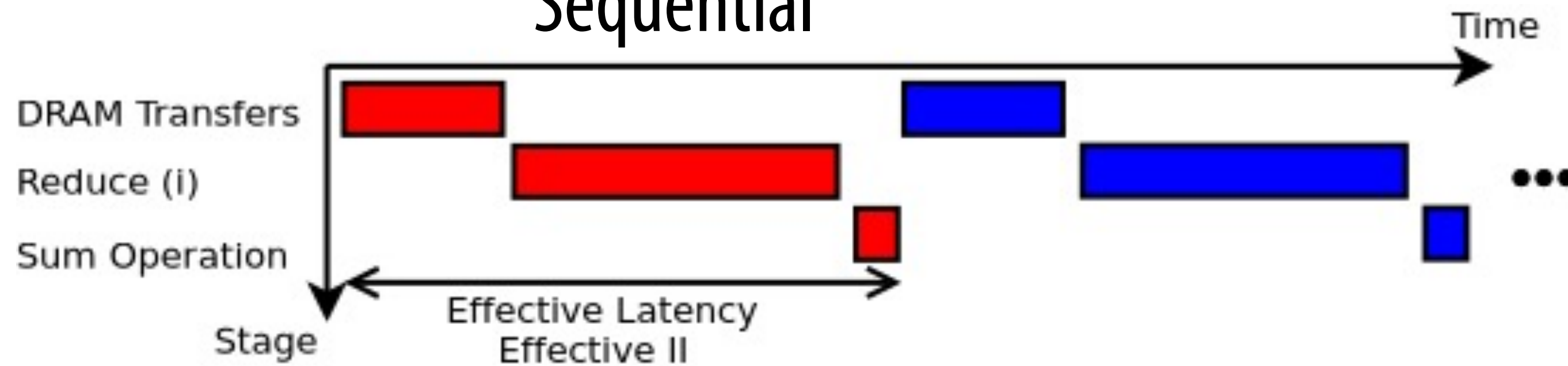


Par1	1	1	2	4	4	4	2
Par0	1	1	1	1	2	4	2
Sched	Seq	Pipe	Pipe	Pipe	Pipe	Pipe	Pipe

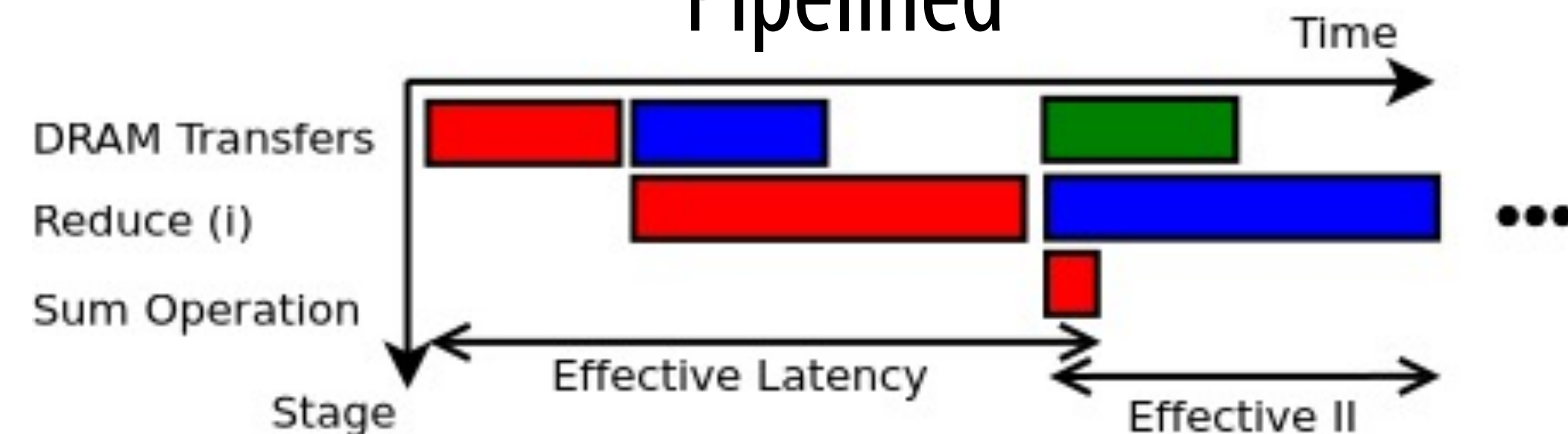
Sequential vs. Pipelined



Sequential



Pipelined

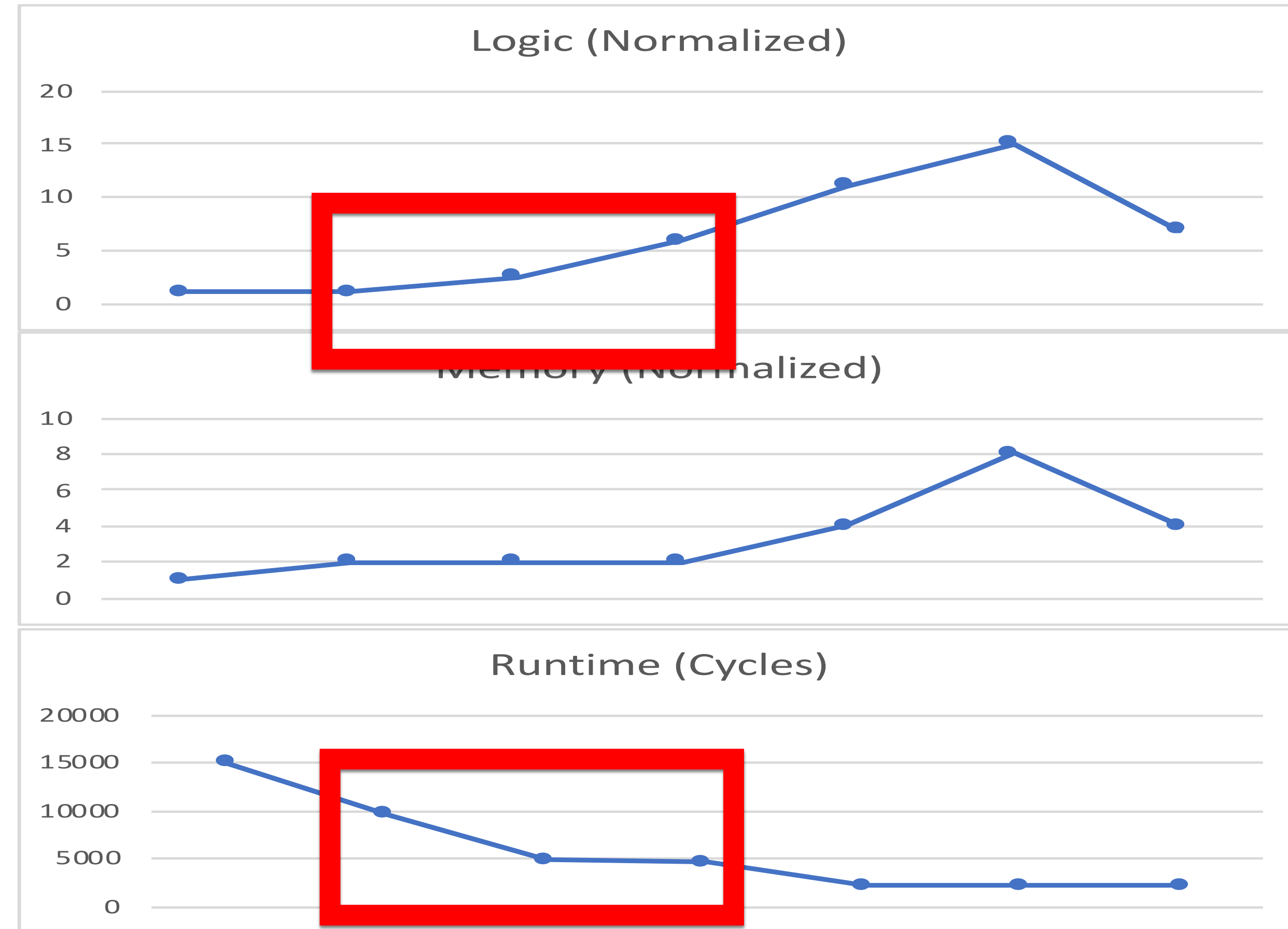


- Understanding how the performance debugger maps to timing diagrams explains this performance boost from pipelining
 - Color corresponds to iteration in diagrams
- For the Sequential case, $II_{eff} \approx \sum T_c$
- For the Pipelined case, $II_{eff} \approx \max(T_c)$

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

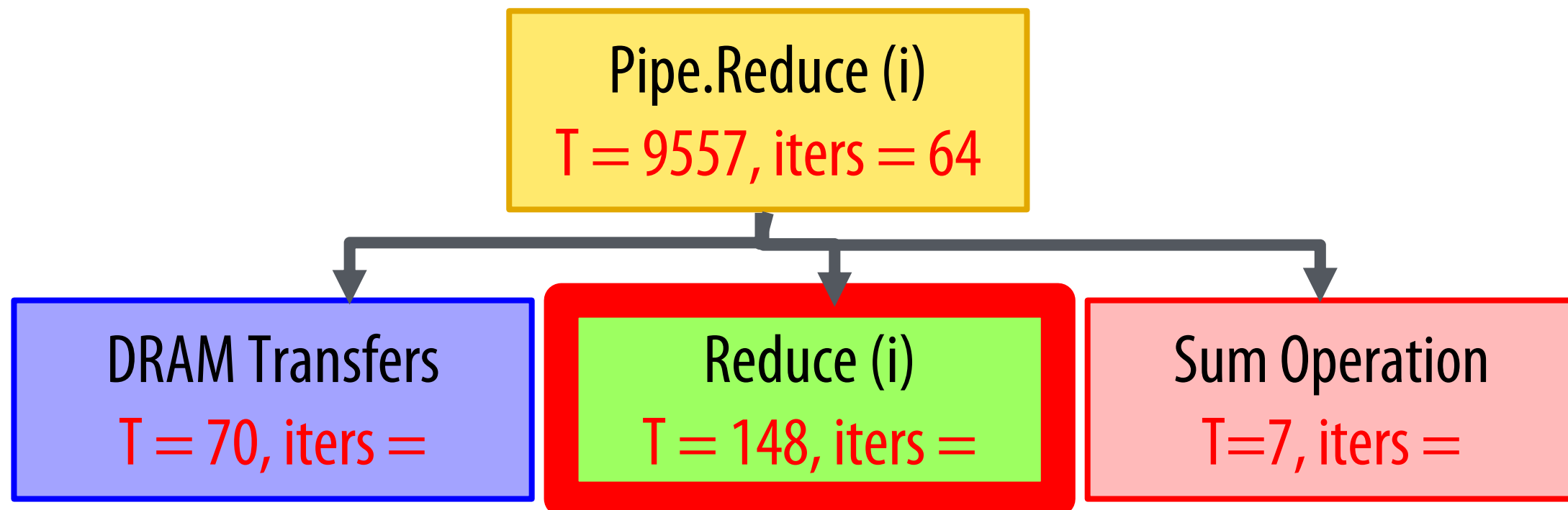
Inner Parallelization

- 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

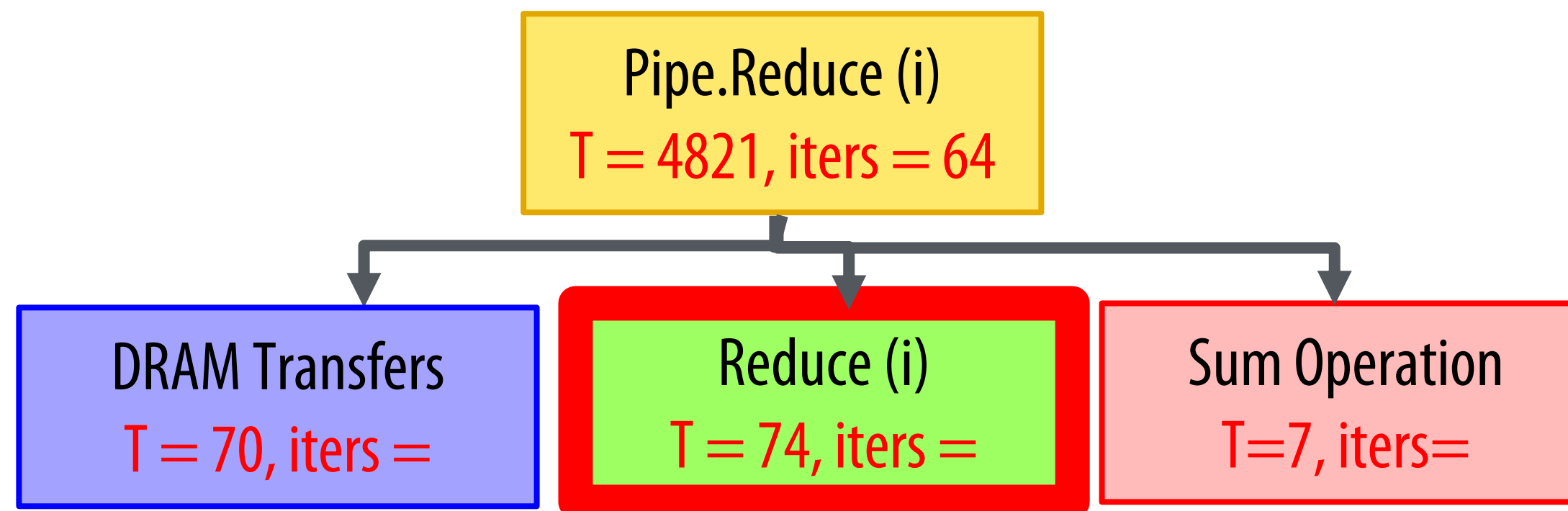


Parl	1	1	2	4	4	4	2
Par0	1	1	1	1	2	4	2
Sched	Seq	Pipe	Pipe	Pipe	Pipe	Pipe	Pipe

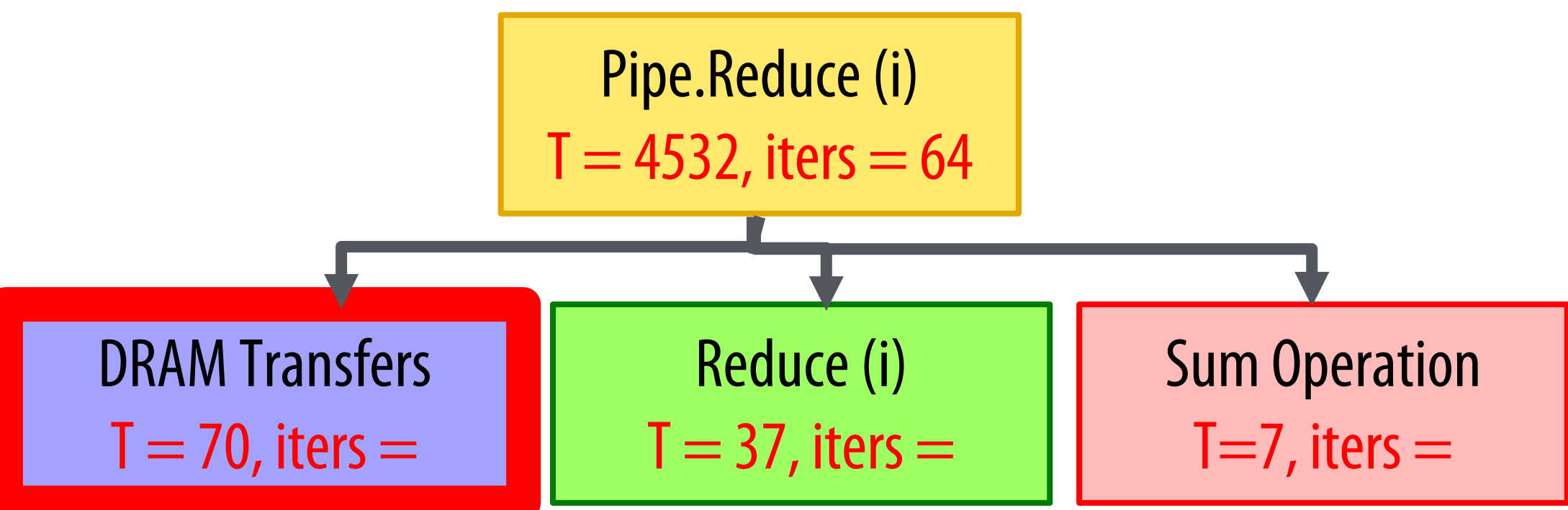
Inner Parallelization



$Par1 = 1$



$Par1 = 2$



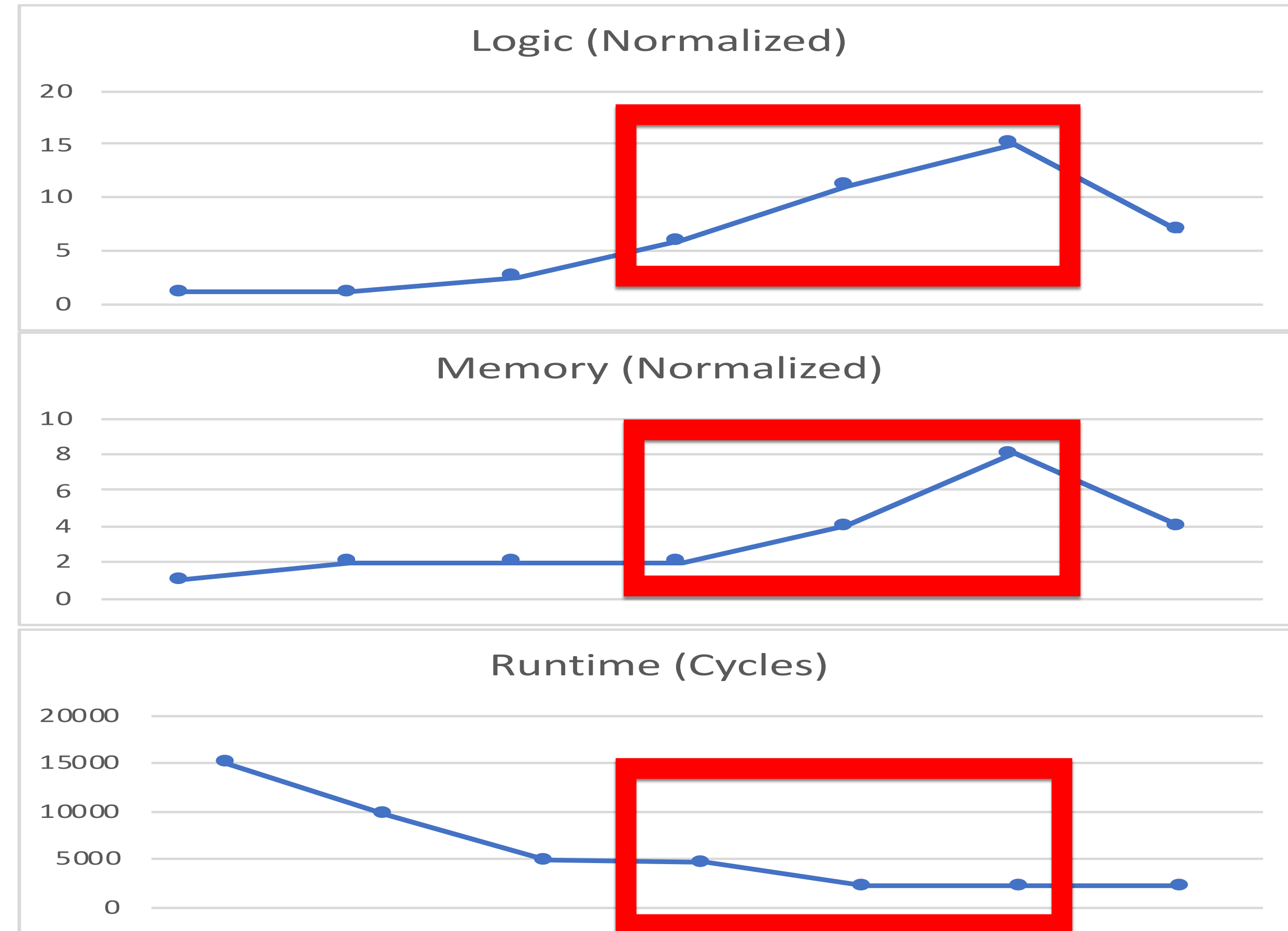
$Par1 = 4$

- The performance debugger explains what happened
- The bottleneck stage improves as a result of this parallelization
- There is still a *small* performance improvement for $Par1=4$ because $II_{eff} \approx \max(T_c)$ decreases a bit

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

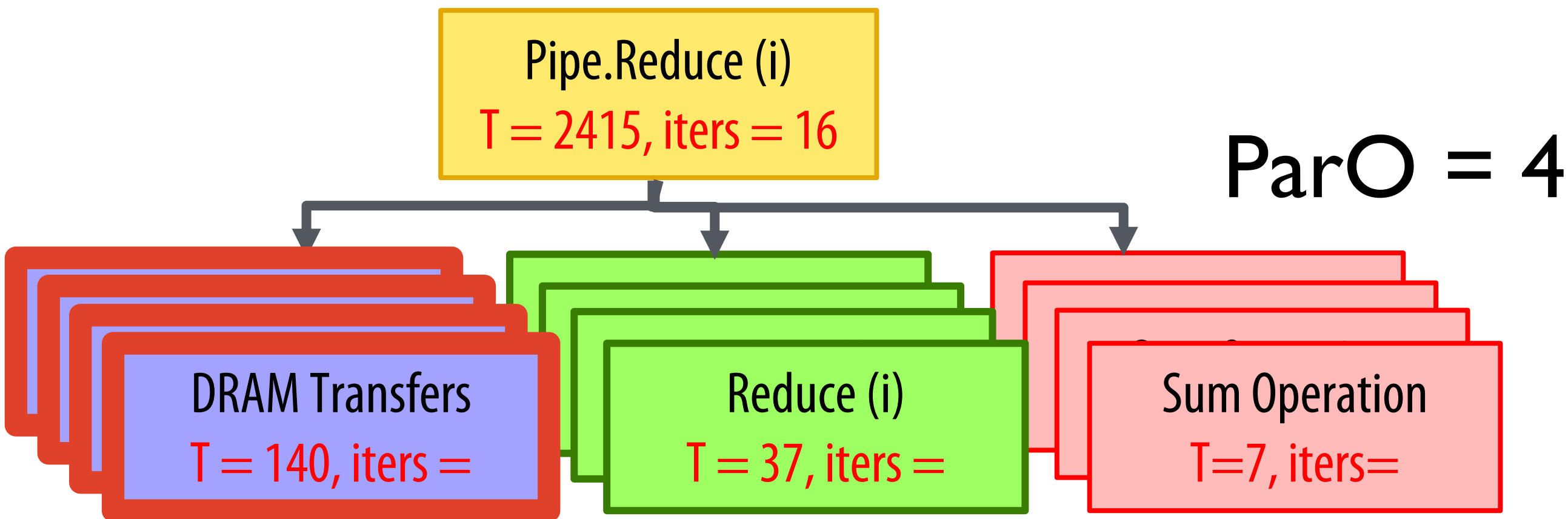
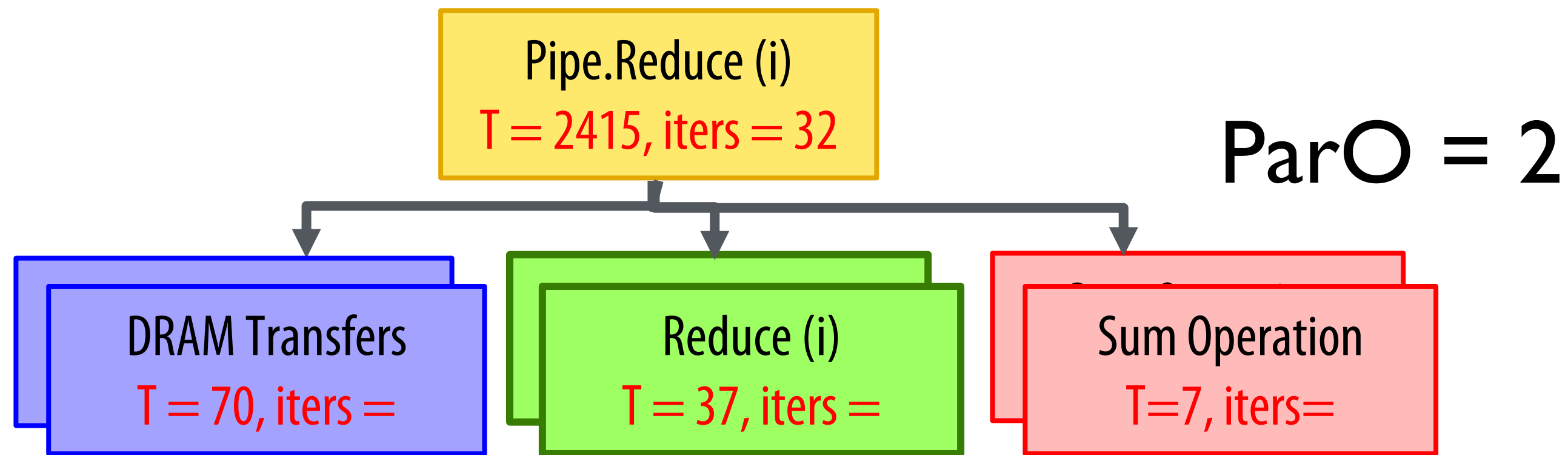
Outer Parallelization

- 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



Par1	1	1	2	4	4	4	2
Par0	1	1	1	1	2	4	2
Sched	Seq	Pipe	Pipe	Pipe	Pipe	Pipe	Pipe

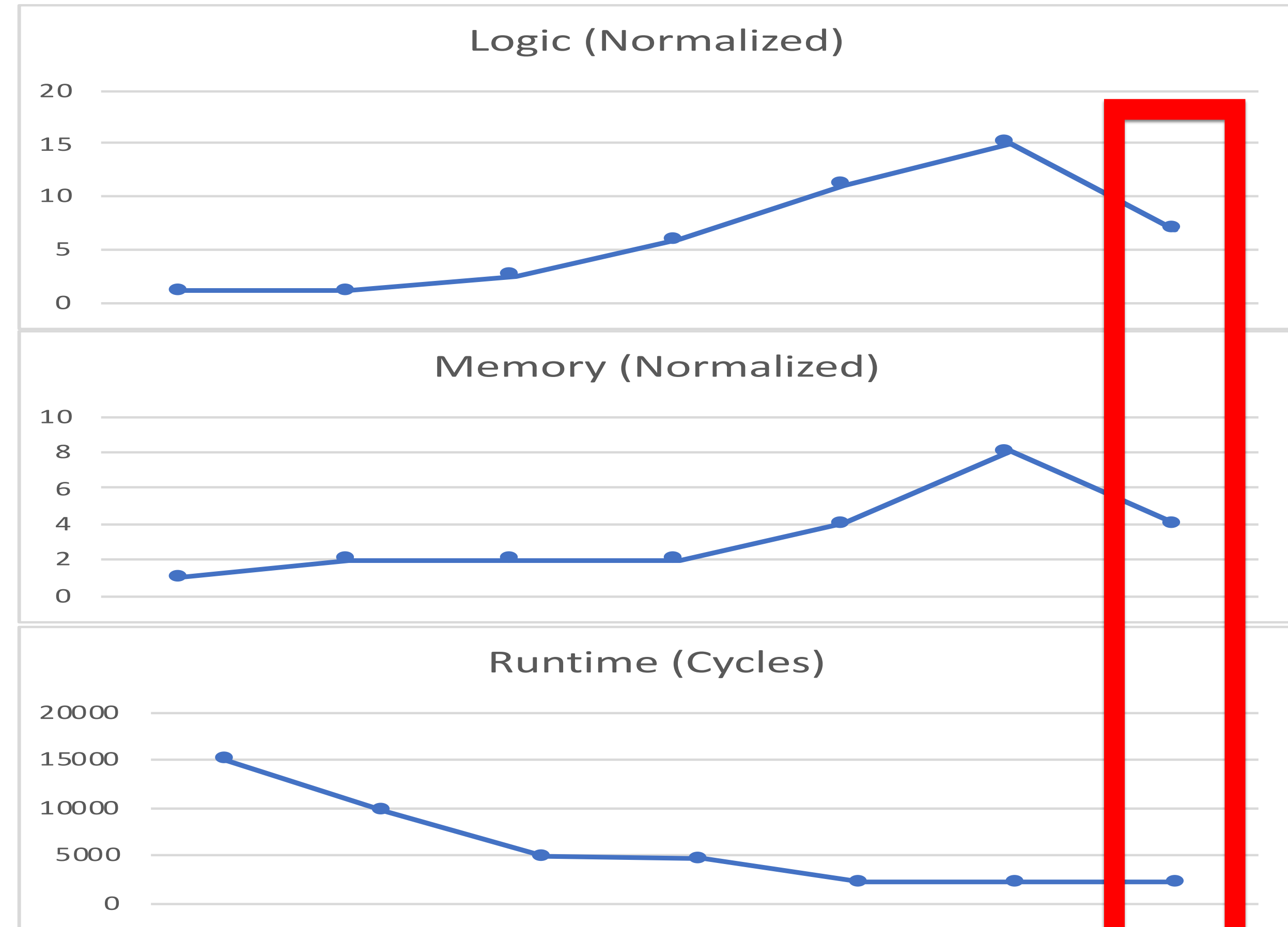
Outer Parallelization



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

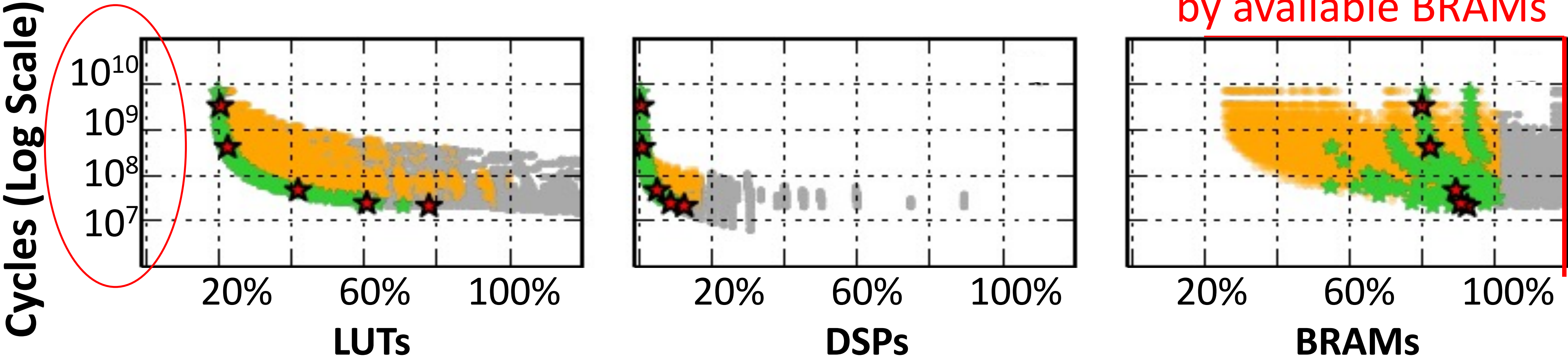


Par1	1	1	2	4	4	4	2
Par0	1	1	1	1	2	4	2
Sched	Seq	Pipe	Pipe	Pipe	Pipe	Pipe	Pipe

Spatial GDA Design Space Exploration

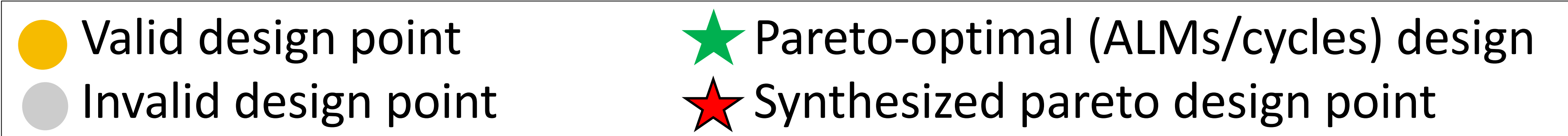
General Purpose Processor

Performance limited by available BRAMs



Space for GDA spans
four orders of magnitude

Resource Usage (% of maximum)



Summary

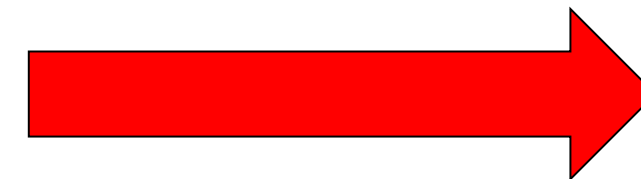
- **Significant energy efficiency improvements from specialized accelerators (100x–1000x)**
- **Designing an accelerator is a tradeoff between performance and resource utilization**
 - **Parallelism**
 - **Locality**
- **It requires the programmer to have insight into the application**
 - **Where is 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

