Stanford | ENGINEERING
Electrical Engineering

STANFORD DAWN

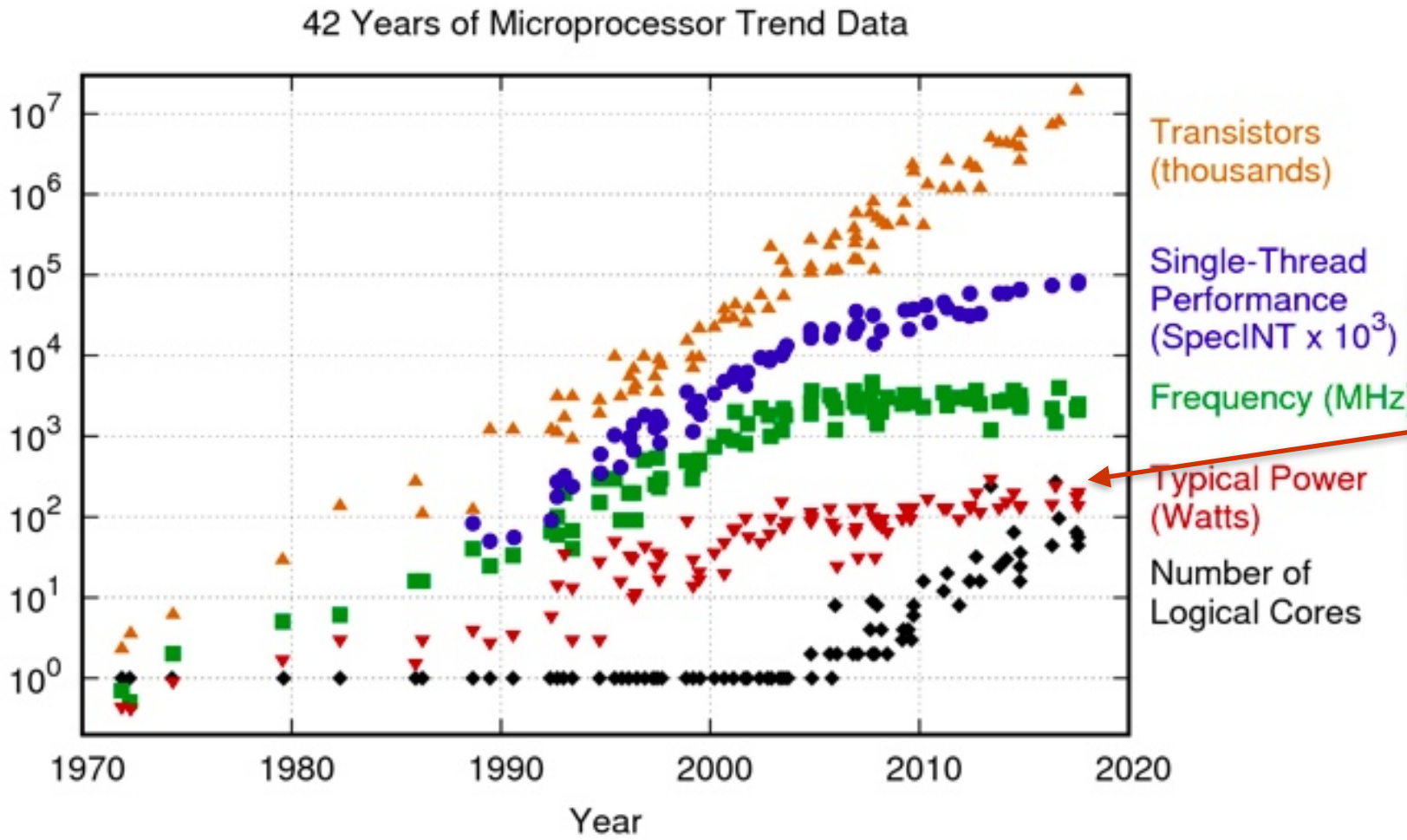
Spatial Language and Compiler

Matthew Feldman, Luigi Nardi, Artur Souza, Kunle Olukotun

Introduction

The transition from instruction-based architectures to custom hardware

The CPU Power Wall



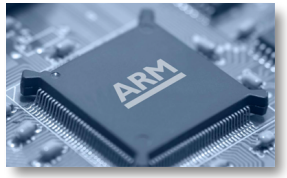
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

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

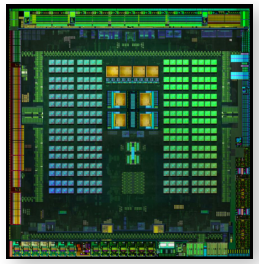
Packing more transistors onto a chip will no longer work because of limits on power delivery and heat dissipation.

Compute Devices at a Glance

Energy-optimized CPU



Throughput-Oriented Device GPU



Programmable DSP

HEXAGON

Programmable Logic FPGA

Application-Specific Integrated Circuit (ASIC)

Least efficient
Easiest to program
Least expensive

Specialization is the key to achieving good performance

Hardest to program

Most efficient
Not programmable
Most expensive

Spatial is focused here

Credit: Stanford CS149

FPGA Crash Course

- Field-programmable gate array
  - Reconfigurable logic device consisting of
    - On-chip Memory (BRAMs) - ~10s Mb
    - Logic Cells (LUTs + FFs) - ~1M
    - Processing blocks (DSPs) - ~1000s

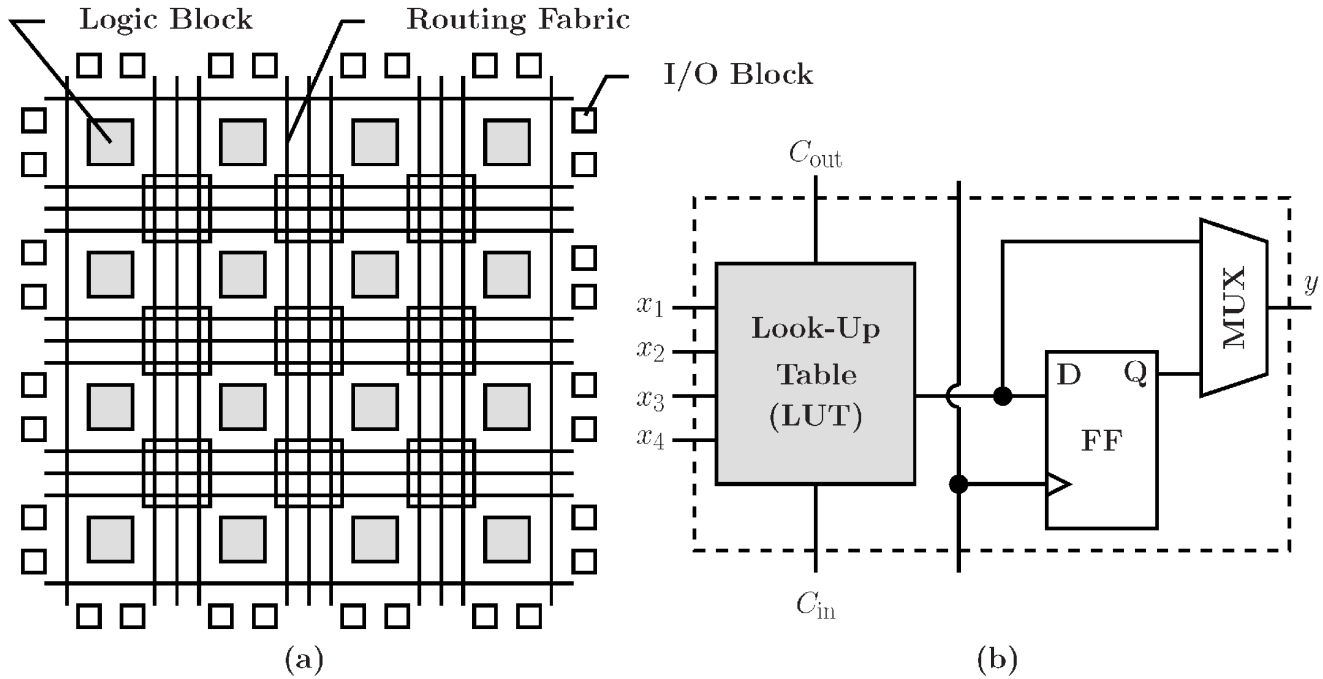


Image credit: Bai et al. 2014

Languages for Programming FPGAs

At a glance [1]

[1] N. Kapre et al, "Survey of domain-specific languages for FPGA computing," FPL 2016

Register-Transfer Level

- Traditional hardware description languages are **Verilog** or **VHDL**
- There are newer, user-friendly alternatives, like **Chisel**[1], **PyMTL**[2], **Bluespec**[3], **MaxJ**[4], **SystemVerilog**, etc.

[1] J. Bachrach et al. "Chisel: Constructing hardware in a Scala embedded language" DAC 2012
[2] D. Lockhart et al. "PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research" MICRO 2014
[3] https://www.ece.ucsb.edu/its/bluespec/index.html
[4] https://www.maxeler.com/products/software/maxcompiler/

Domain Specific

- Languages rooted in a particular application domain include **Aetherling**[5], **Halide**[6], **LeFlow**[7], **DNNWeaver**[8], **Spiral**[9], **SNORT**[10], **ASV**[11], etc.

[5] D. Durst et al. "Type-Directed Scheduling of Streaming Accelerators" PLDI 2020
[6] J. Ragan-Kelley et al. "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines" PLDI 2013
[7] D. Noronha et al. "LeFlow: Enabling Flexible FPGA High-Level Synthesis of Tensorflow Deep Neural Networks" FSP 2018
[8] H. Sharma et al. "From high-level deep neural models to FPGAs" MICRO 2016
[9] J. Moura et al. "SPIRAL: Automatic Implementation of Signal Processing Algorithms" HPEC 2000
[10] A. Mitra et al. "Compiling PCRE to FPGA for accelerating SNORT IDS" ANCS 2007
[11] Y. Feng et al. "ASV: Accelerated Stereo Vision System" MICRO 2019

High Level Synthesis

- C+pragmas approach: **OpenCL**[12], **Vivado HLS**[13], **SDAccel**[14], **LegUp**[15], **Merlin**[21], **SOFF**[16], etc.
- JVM-based hardware DSL approach: **Liquid Metal (Lime)**[18], **Spatial**[19], etc.

[12] https://www.khronos.org/opencl/
[13] https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html
[14] https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html
[15] A. Canis et al. "LegUp: An Open Source High-Level Synthesis Tool for FPGA-Based Processor/Accelerator Systems" ECS 2012
[16] G. Jo et al. "SOFF: An OpenCL High-Level Synthesis Framework for FPGAs" ISCA 2020
[18] S. Huang et al. "Liquid Metal: Object-Oriented Programming Across the Hardware/Software Boundary" ECOOP 2008
[19] D. Koeplinger et al. "Spatial: a language and compiler for application accelerators" PLDI 2018
[21] https://www.falconcomputing.com/merlin-fpga-compiler/

Hardware Design At a Glance

- A good accelerator has **optimized computation and memory accesses** and **keeps all parts of the circuit active at all times**
- In order to do this, the designer must make decisions about
  - **Parallelism** - Run operations concurrently
  - **Data Locality** - Manually manage on-chip scratchpads
  - **Control Flow** - Orchestrate how loops execute relative to each other
- **Spatial** is an MIT License open source language that exposes these knobs, which leads to massive design spaces
- **HyperMapper** is the key to exploring the large design spaces automatically

Introduction to Spatial

Understanding **loops** and the **memory hierarchy**

Spatial: Control And Design Parameters

Explicit parallelization factors
(optional, but can be explicitly declared)

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

Implicit/Explicit control schemes
(also optional, but can be used to override compiler)

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

Explicit size parameters for loop step size and buffer sizes
(informs compiler it can tune this value)

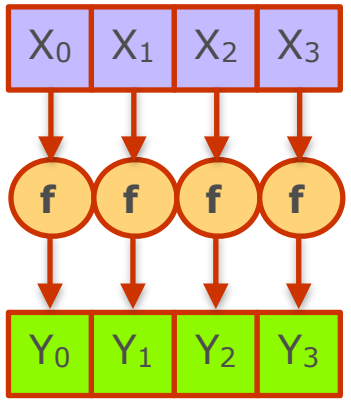
```
val B = 64 (64 → 1024)
val buffer = SRAM[Float](B)
Foreach(N by B){i =>
  …
}
```

Implicit memory banking and buffering schemes for parallelized access

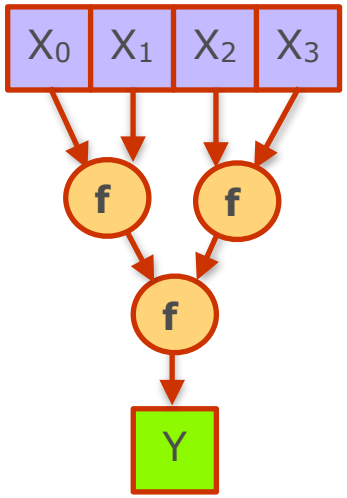
```
Foreach(64 par 16){i =>
  buffer(i) // Parallel read
}
```

Parallel Patterns

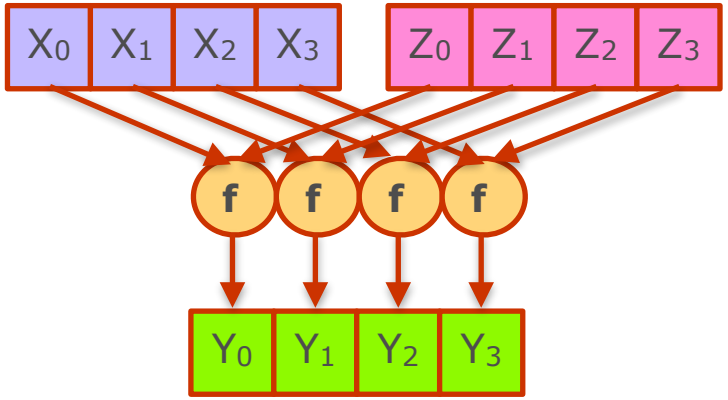
- Parallel patterns are loop abstractions with implicit information about parallelism and access patterns



Map
Element-wise function *f*



Reduce
Combine elements with (associative) function *f*



Zip
Element-wise combine function *f*

- Spatial is an *imperative language* that is designed to easily capture **parallel patterns**[20]

[20] R. Prabhakar et al. "Generating Configurable Hardware from Parallel Patterns." ASPLOS 2016

Spatial: Loops in Hardware

- A software "loop" is **counter chain + controller**
  - **Counter chain** - Collection of iterators that are chained together
  - **Controller** - A container for a data path or other controllers

- Controllers are nested:
  - **Inner -** contains datapaths of *only* primitive nodes
  - **Outer -** contains *only* other controllers (called "children")

```
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
}
```

Spatial: Inner Loop Execution

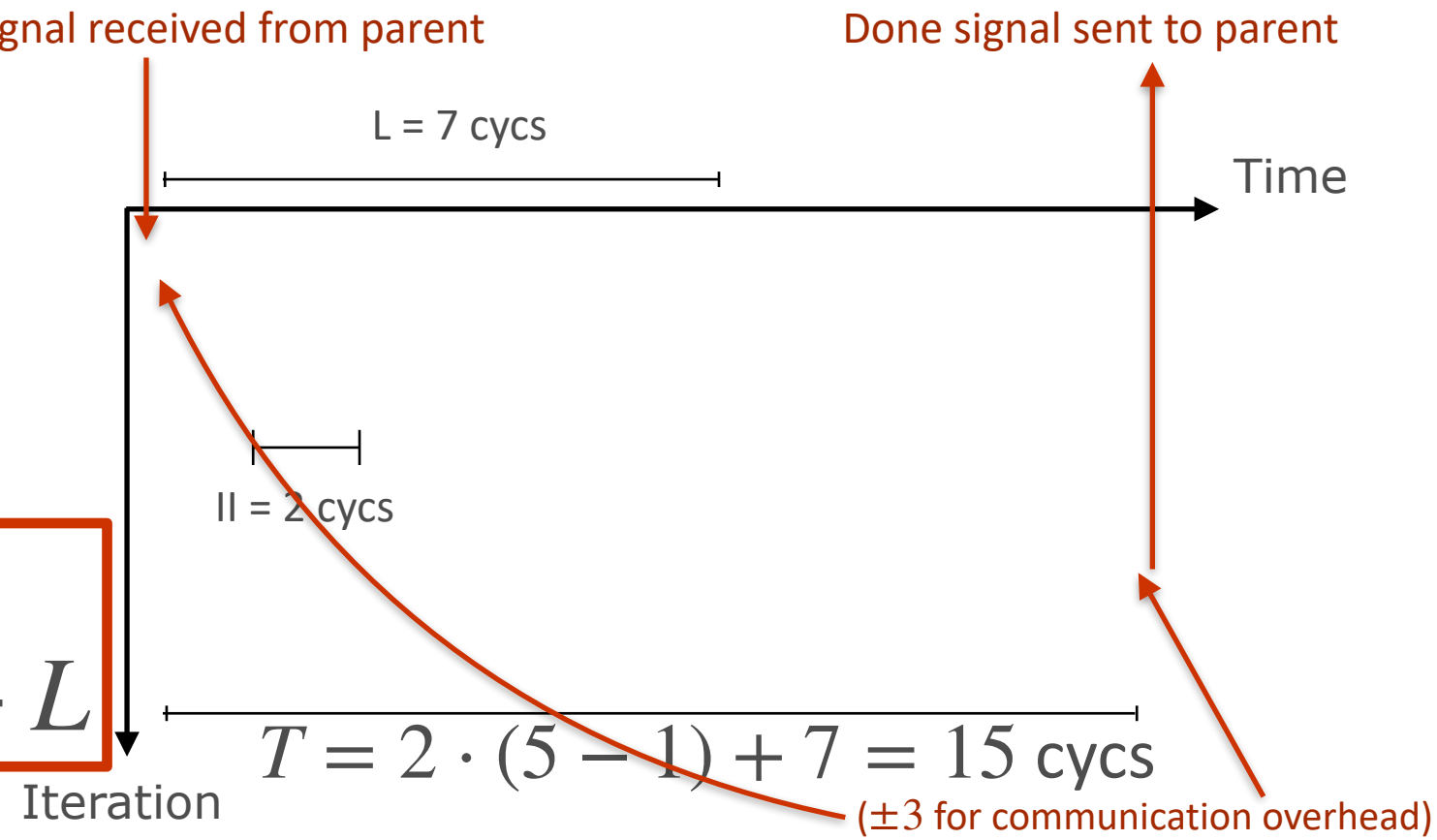
- The runtime of a controller (**T**) depends on its latency (**L**), initiation interval (**II**), and number of iterations (**iters**)

```
Foreach(5 by 1) {i =>
  … // L = 7, II = 2
}
```

Abstract Example

Key Equation:

$$T = II \cdot (iters - 1) + L$$

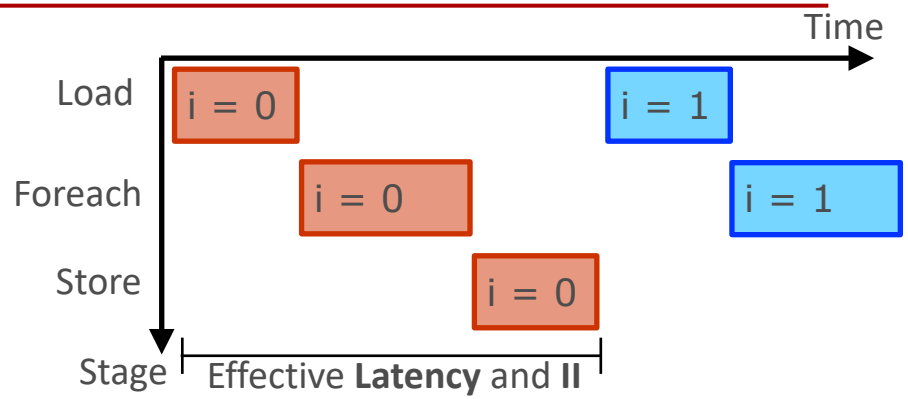


Spatial: Outer Controller Schedules

- **Outer controller** must take a schedule to describe how their children execute relative to each other
- Schedules include:
  - **Sequential** - No overlapping of child controllers
  - **Pipelined** - Coarse-grained overlapping of child controllers
  - **Stream** - Data-driven execution of child controllers
- **Sequential** and **Pipelined** are interchangeable without code rewrites

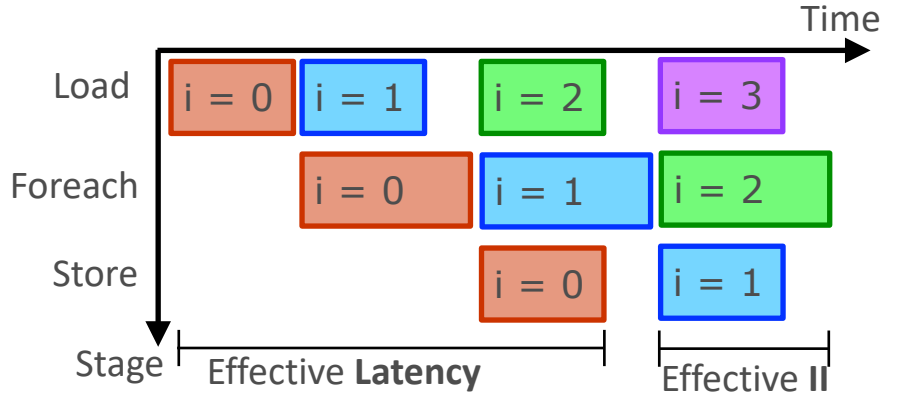
A Closer Look at Schedules

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

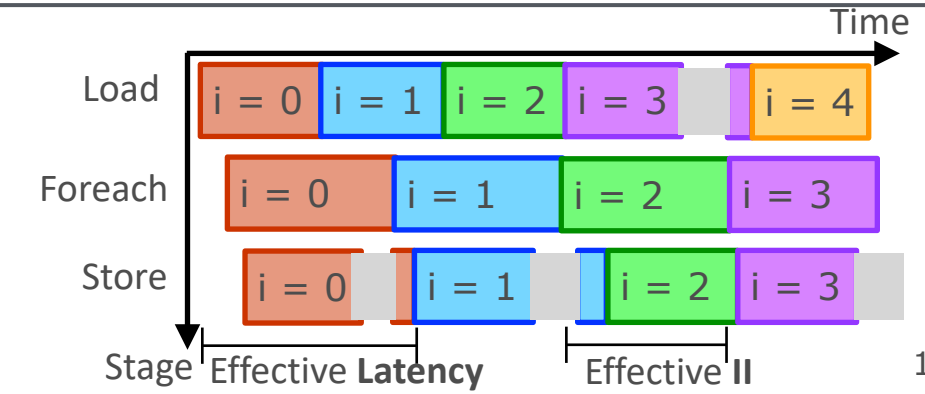


Note: **Foreach** with no annotation is implicitly "Pipelined"

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

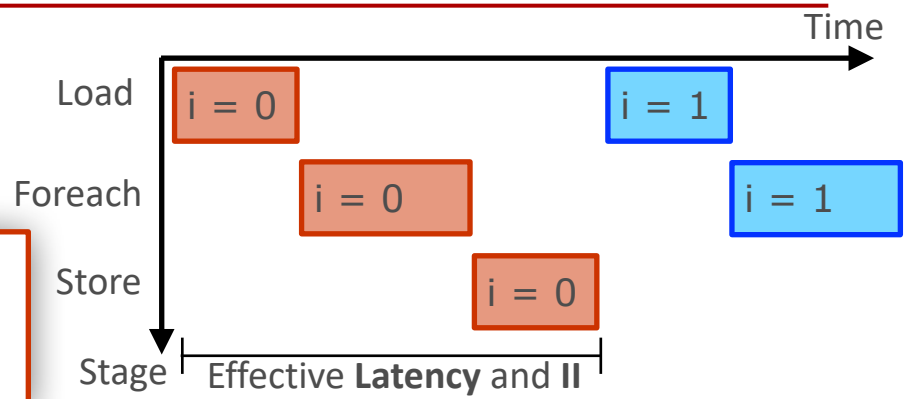



```
Stream.Foreach(…){i =>
  fifoIn load dram
  Foreach(M by 1){ j => fifoOut.enq(fifoIn.deq() * j) }
  dram2 store fifoOut
}
```



A Closer Look at Schedules

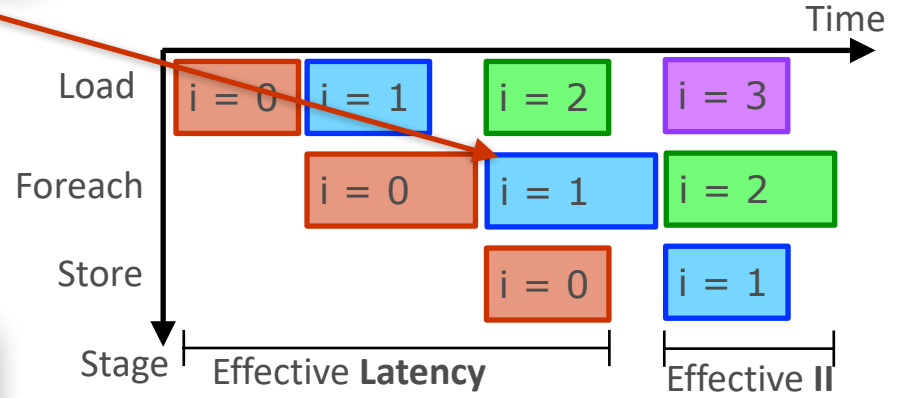
```
Sequential.Foreach(…){i =>
  sram load dram
  Foreach(M by 1){ j => sram2(j) = sram(j) * j }
  dram store sram2
}
```



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

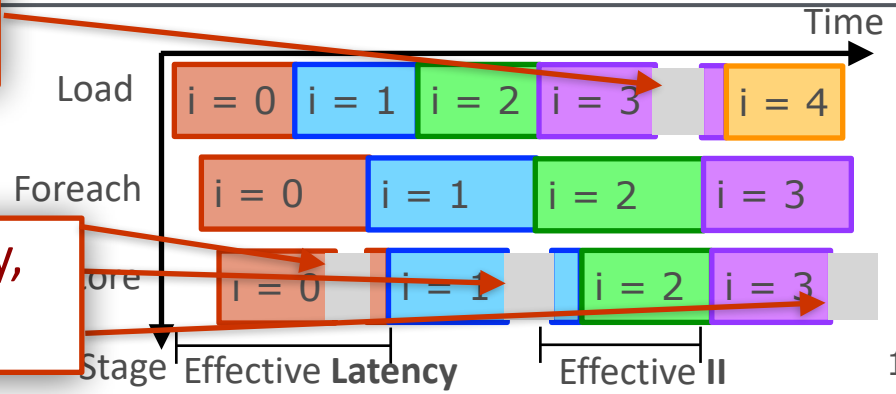
Note: **Foreach** with no annotation is implicitly "Pipelined"

```
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
}
```

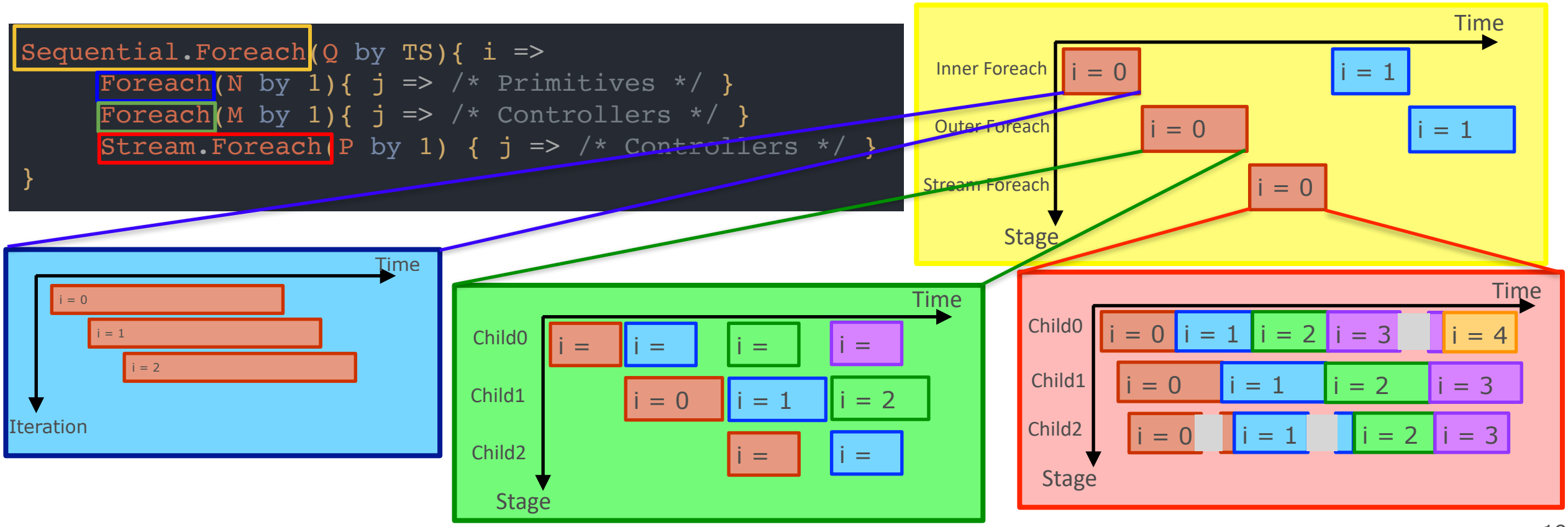


When an intermediate FIFO is empty, the consumer stage is **starved**.

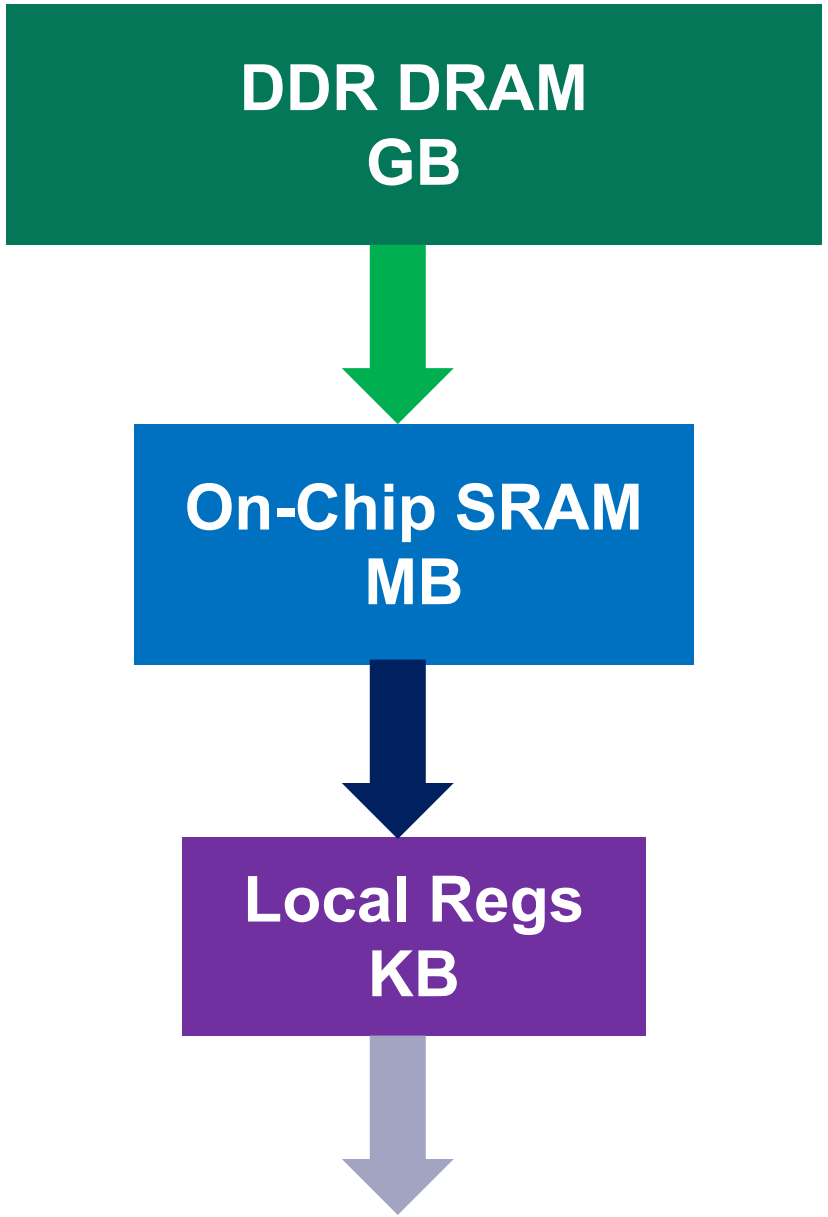
Spatial: Piecing the Hierarchy Together

- Consider the slice of a loop nesting with a parent Sequential Controller and three children.

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



Spatial: Memory Hierarchy




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

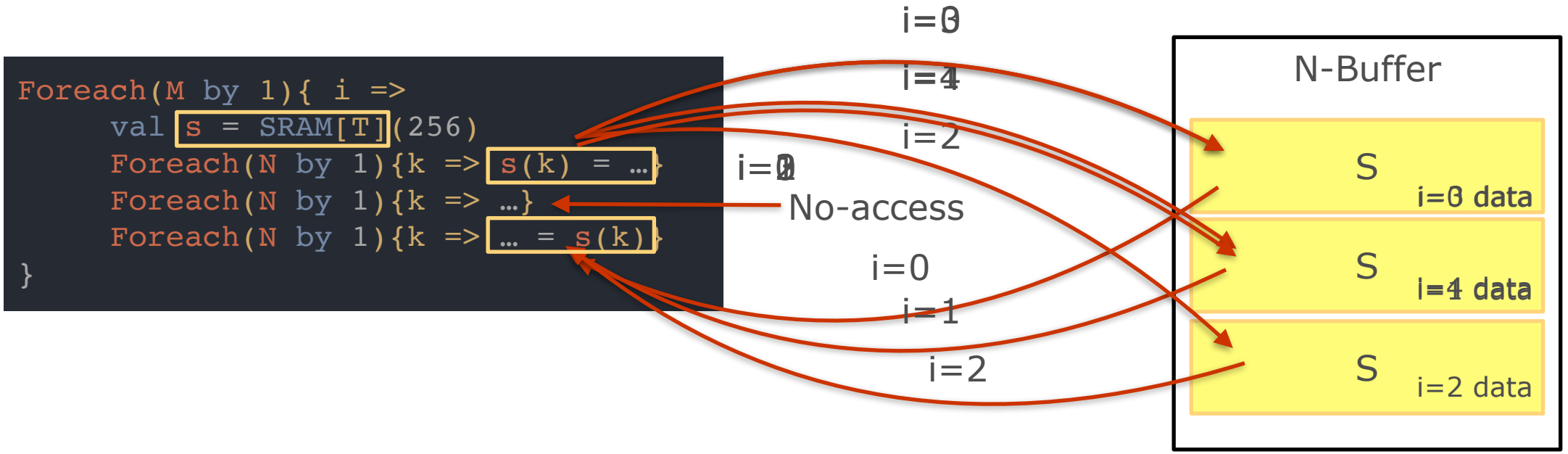
```
buffer load   image(i, j::j+C) // dense
buffer gather image(a)         // sparse
```

```
val buffer = SRAM[UInt8](C)
val fifo   = FIFO[Float](D)
val lbuf   = LineBuffer[Int]
(R,C)
```

```
val accum  = Reg[Double]
val pixels = RegFile[UInt8](R,C)
```

Spatial: Memory Buffering

- **Buffering** is implicit duplication of a memory to protect accesses from each other in a **Pipelined** controller



- The compiler computes buffering automatically, which can explode the resource utilization

Summary

- There is always a trade-off between resource utilization and performance
- The trade-offs are complex, but HyperMapper can help!