Lecture 18:
Hardware Specialization and Spatial Programming

Parallel Computing
Stanford CS149, Fall 2021
Energy-constrained computing
Performance and Power

\[
\text{Power} = \frac{\text{Ops}}{\text{second}} \times \frac{\text{Joules}}{\text{Op}}
\]

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

ASIC delivers same performance as one CPU core using only \( \sim 1/100 \text{th the power} \)

[Chung et al. MICRO 2010]
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

![Diagram of FPGA chip](Image credit: Bai et al. 2014)

- 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

<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>1</td>
</tr>
</tbody>
</table>

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

- **Energy-optimized CPU**
  - Throughput-oriented processor (GPU)
  - Programmable DSP
  - Domain Specific Accelerator
  - FPGA/reconfigurable logic
  - ASIC

- **Efficiency**
  - ~10X more efficient
  - ~20X
  - ~50X???
  - ~100-1000X more efficient

- **Programming Difficulty**
  - Easiest to program
  - Limited domain of programmability with DSLs (e.g. DNN)
  - Difficult to program (making it easier is an 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

- Shared Data Cache
- Core 0
  - Instruction selection
  - Fetch/Decode
  - ALU
  - Execution Context 0
  - Execution Context 1
  - Execution Context 2
  - Execution Context 3

- Core 1
  - Instruction selection
  - Fetch/Decode
  - ALU
  - Execution Context 4
  - Execution Context 5
  - Execution Context 6
  - Execution Context 7

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)

- Memory
  - DRAM
- Custom Memory 0
  - SRAM
- Custom Memory 1
  - SRAM

- Custom Registers
  - V0, V1, V2, V3, V4, V5, V6, V7

- ALU
  - (16-wide vector ALU)

- Custom Control
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
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 ↔ 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

\[
\begin{align*}
\text{val } \text{accum} &= \text{Reg[Double]} \\
\text{val } \text{fifo} &= \text{FIFO[Float]}(D) \\
\text{val } \text{lbuf} &= \text{LineBuffer[Int]}(R,C) \\
\text{val } \text{pixels} &= \text{ShiftReg[UInt8]}(R,C)
\end{align*}
\]

Explicit memory hierarchy

\[
\begin{align*}
\text{val } \text{buffer} &= \text{SRAM[UInt8]}(C) \\
\text{val } \text{image} &= \text{DRAM[UInt8]}(H,W)
\end{align*}
\]

Explicit transfers across memory hierarchy

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

Dense and sparse access

Streaming abstractions

\[
\begin{align*}
\text{val } \text{videoIn} &= \text{StreamIn[RGB]} \\
\text{val } \text{videoOut} &= \text{StreamOut[RGB]}
\end{align*}
\]
The Spatial Language: Control Templates

Blocking/non-blocking interaction with CPU

Arbitrary state machine / loop nesting with implicit control signals

```
Accel { ... }
Accel(*) { ... }
```

```
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

\[ \text{val } P = 16 \ (1 \rightarrow 32) \]

\[ \text{Reduce}(\emptyset)(N \ \text{by} \ 1 \ \text{par} \ P) \{ i \Rightarrow \text{data}(i) \} \{(a,b) \Rightarrow a + b\} \]

**Implicit/Explicit** control schemes

\[ \text{Stream.Foreach}(\emptyset \ \text{until} \ N) \{ i \Rightarrow \ldots \} \]

**Explicit** size parameters for stride and buffer sizes

\[ \text{val } B = 64 \ (64 \rightarrow 1024) \]

\[ \text{val buffer} = \text{SRAM[Float]}(B) \]

\[ \text{Foreach}(N \ \text{by} \ B) \{ i \Rightarrow \ldots \} \]

**Implicit** memory banking and buffering schemes for parallelized access

\[ \text{Foreach}(64 \ \text{par} \ 16) \{ i \Rightarrow \text{buffer}(i) \ \text{// Parallel read} \} \]
Inner Product

Let’s build an accelerator to see how Spatial works

Code

Sketch of generated hardware
Here is inner product written in C for a CPU

```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));
}
```
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
// 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)
Inner Product in Spatial

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

```scala
// 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 $\rightarrow$ intra-tile accumulate $\rightarrow$ full accumulate

Where is the parallelism?
Inner Product in Spatial

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

```scala
// 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)
    }{ _ + _ }
  }{ _ + _ }
}
```
Inner Product in Spatial

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

// 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)
    }{ _ + _ }
  }{ _ + _ }
}
Inner Product in Spatial

- 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

```plaintext
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:

\[ T = II \times (iters - 1) + L \]

- \( T \) = 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.

\[ T = II \times (\text{iters} - 1) + L \]

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)\}

Stanford CS149, Fall 2021
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

\[ T = II_{\text{eff}} \times (\text{iters} - 1) + L_{\text{eff}} \]
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.**
```plaintext
Foreach(...){
  i =>
  sram load dram
  Foreach(M by 1){ j => sram2(j) = sram(j) * j }
  dram store sram2
}
```

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

**Stream.**
```plaintext
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 Question

- Spatial programmer’s responsibility
  - Specifying algorithm as a hierarchy of controllers
  - Specifying memory hierarchy of algorithm
  - Explicit data movement
  - Picking tiling factors, parallelism (inner) and scheduling (outer)

- Spatial systems responsibility
  - Banking and buffering of memories to maximize perf and minimize resources
  - Hardware generation for target platform (FPGA, CGRA, ASIC)
  - Performance debugging feedback
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}
}

Spatial: ~30 lines

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

```java
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:
Performance Debugging

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

\[ T_{\text{parent}} = I I_{\text{eff}} \times (\text{iters} - 1) + L_{\text{eff}} \]
Performance Debugging

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

\[ T_{\text{parent}} = \Pi_{\text{eff}} \times (\text{iters} - 1) + L_{\text{eff}} \]

\[
\text{iters} = \frac{Q}{TS} \\
\Pi_{\text{eff}} = L_{\text{eff}} = \sum T_{\text{child}}
\]
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.

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 */ }
}

Sequential (i)
[line:1] T = #, iters = #

Inner (j)
[line:2] II = #, L = #
T = #, iters = #

Pipelined (j)
[line:3] T = #, iters=#

Stream (j)
[line:4] T=#, iters=#
Performance Debugging

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

\[ T = II \times (\text{iters} - 1) + L \]

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

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

- Outer controller schedule (`Reduce`)
- `ParO`
- `ParI`
Inner Product Controller Hierarchy

The baseline implementation is ParI=1, ParO=1, and schedule=Sequential

Our instrumented controller tree will look like this:

```scala
// 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}
}
```
Optimization Goal

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

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

<table>
<thead>
<tr>
<th>Sched</th>
<th>ParL</th>
<th>ParO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Seq</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Pipe</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Pipe</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Pipe</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Pipe</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Pipe</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Pipe</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

- Logic (Normalized)
- Memory (Normalized)
- Runtime (Cycles)

Stanford CS149, Fall 2021
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, \( I_{\text{eff}} \approx \sum T_c \).
  - For the Pipelined case, \( I_{\text{eff}} \approx \max(T_c) \).

\[
T = I_{\text{eff}} \times (\text{iters} - 1) + L_{\text{eff}}
\]
Inner Parallelization

- There was a performance improvement from ParI = 1 to ParI = 2
  - Improved bottleneck of the pipeline

- From ParI = 2 to ParI = 4, we consume more logic but did not see much speedup
  - Inner reduce is no longer the bottleneck
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 ParI=4 because $I_{eff} \approx \max(T_c)$ decreases a bit

ParI = 1
- DRAM Transfers $T = 70$, iters =
- Reduce (i) $T = 148$, iters =
- Sum Operation $T=7$, iters =

ParI = 2
- DRAM Transfers $T = 70$, iters =
- Reduce (i) $T = 74$, iters =
- Sum Operation $T=7$, iters =

ParI = 4
- DRAM Transfers $T = 70$, iters =
- Reduce (i) $T = 37$, iters =
- Sum Operation $T=7$, iters =

\[ T = I_{eff} \times (iters - 1) + L_{eff} \]
- There is a performance improvement from ParO=1 to ParO=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 ParO=2 to ParO=4, the app becomes memory-bound
  - Change increases resource utilization without improving performance
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 run for twice as long.
- This indicates that the DRAM has enough bandwidth to support ParO=2 but not ParO=4.
- 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
Spatial GDA Design Space Exploration

General Purpose Processor

Space for GDA spans four orders of magnitude

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

Performance limited by available BRAMs
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 => ... }
}