Lecture 17:
Hardware Specialization and Spatial Programming

Parallel Computing
Stanford CS149, Fall 2019
Energy-constrained computing
Performance and Power

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

FIXED

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

What is the magnitude of improvement from specialization?
Pursuing highly efficient processing…
(specializing hardware beyond just parallel CPUs and GPUs)
Efficiency benefits of compute specialization

- Rules of thumb: compared to high-quality C code on CPU...
- Throughput-maximized processor architectures: e.g., GPU cores
  - Approximately 10x improvement in perf / watt
  - Assuming code maps well to wide data-parallel execution and is compute bound

- Fixed-function ASIC (“application-specific integrated circuit”)
  - Can approach 100-1000x or greater improvement in perf/watt
  - Assuming code is compute bound

[Source: Chung et al. 2010, Dally 08]
Fast Fourier transform (FFT): throughput and energy benefits of specialization

ASIC delivers same performance as one CPU core with ~ 1/1000th the chip area.

GPU cores: ~ 5-7 times more area efficient than CPU cores.

ASIC delivers same performance as one CPU core using only ~ 1/100th the power
Why is a “general-purpose processor” so inefficient?

Wait... this entire class we’ve been talking about making efficient use out of multi-core CPUs and GPUs... and now you’re telling me these platforms are “inefficient”?
Consider the complexity of executing an instruction on a modern processor...

- Read instruction
- Address translation, communicate with icache, access icache, etc.
- Decode instruction
- Translate op to uops, access uop cache, etc.
- Check for dependencies/pipeline hazards
- Identify available execution resource
- Use decoded operands to control register file SRAM (retrieve data)
- Move data from register file to selected execution resource
- Perform arithmetic operation
- Move data from execution resource to register file
- Use decoded operands to control write to register file SRAM

Review question:
How does SIMD execution reduce overhead of certain types of computations?
What properties must these computations have?
Digital signal processors (DSPs)

Programmable processors, but simpler instruction stream control paths

Complex instructions (e.g., SIMD/VLIW): perform many operations per instruction (amortize cost of control)

Example: Qualcomm Hexagon DSP
Used for modem, audio, and (increasingly) image processing on Qualcomm Snapdragon SoC processors

VLIW: “very-long instruction word”
Single instruction specifies multiple different operations to do at once (contrast to SIMD)

Below: innermost loop of FFT
Hexagon DSP performs 29 “RISC” ops per cycle

64-bit Load and 64-bit Store with post-update addressing

Variable sized instruction packets (1 to 4 instructions per packet)

Vector 4x16-bit Add

Complex multiply with round and saturation

Zero-overhead loops
- Dec count
- Compare
- Jump top

Below: Hexagon DSP
- Dual 64-bit execution units
- Standard 8/16/32/64bit data types
- SIMD vectorized MPY / ALU / SHIFT, Pmulh, BItOps
- Up to 8 16b MAC/cycle
- 2 SP FMA/cycle

Unified 32x32bit General Register File is best for compiler.
- No separate Address or Accum Registers
- Per-Thread

Hexagon DSP is in Google Pixel phone

Stanford CS149, Fall 2019
Let’s crack open a modern smartphone

Google Pixel 2 Phone: Qualcomm Snapdragon 835 SoC + Google Visual Pixel Core

Visual Pixel Core
Programmable image processor and DNN accelerator

“Hexagon” Programmable DSP
data-parallel multi-media processing

Image Signal Processor
ASIC for processing camera sensor pixels

Multi-core GPU
(3D graphics, OpenCL data-parallel compute)

Video encode/decode ASIC

Display engine
(compresses pixels for transfer to high-res screen)

Multi-core ARM CPU
4 “big cores” + 4 “little cores”
Specialized processors for evaluating deep networks

Example: Google’s Tensor Processing Unit (TPU) Accelerates deep learning operations

Countless recent papers at top computer architecture research conferences on the topic of ASICs or accelerators for deep learning or evaluating deep networks...

- EIE: Efficient Inference Engine on Compressed Deep Neural Network, Han et al. ISCA 2016
- Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators, Reagen et al. ISCA 2016
- vDNN: Virtualized Deep Neural Networks for Scalable, Memory-Efficient Neural Network Design, Shu et al. MICRO 2015
- Fused-Layer CNN Architectures, Alwani et al. MICRO 2016
- PRIME: A Novel Processing-in-memory Architecture for Neural Network Computation in ReRAM-based Main Memory, Chi et al. ISCA 2016
- DNN Weaver: From High-Level Deep Network Models to FPGA Acceleration, Sharma et al. MICRO 2016

Intel Lake Crest ML accelerator (formerly Nervana)
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 architecture]

- Programmable lookup table (LUT)
- Flip flop (a register)

*Image credit: Bai et al. 2014*
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

<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]
Modern FPGAs

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

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

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

[Putnam et al. ISCA 2014]
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
Summary: choosing the right tool for the job

<table>
<thead>
<tr>
<th></th>
<th>Energy-optimized CPU</th>
<th>Throughput-oriented processor (GPU)</th>
<th>Programmable DSP</th>
<th>FPGA/reconfigurable logic</th>
<th>ASIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Efficiency</td>
<td>~10X more efficient</td>
<td>~100X?? (jury still out)</td>
<td>~100-1000X more efficient</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Programmability</td>
<td>Easiest to program</td>
<td>Difficult to program (making it easier is an active area of research)</td>
<td>Not programmable + costs 10-100's millions of dollars to design / verify / create</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 accelerators that was built with performance-oriented programmers in mind
  - Parallelism
  - Locality
What is Spatial?

Spatial is a high-level language embedded in Scala that puts control and design parameters in the hands of the programmer.

Explicit parallelization factors
Execute loops faster

Explicit transfers across memory hierarchy
Easily access off-chip and on-chip data

Explicit tiling factors
Control over how to use on-chip memories

Implicit memory banking/buffering schemes
Memory optimizations done automatically to guarantee correctness

```scala
Foreach(N by 1 par 4){ i =>
  /* ... */
}
sram load dram(0::T)
argOut := a + b

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

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

Let’s build an accelerator to see how Spatial works

Code

Sketch of generated hardware
Inner Product in C

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
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)
}
// 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

```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 → intra-tile accumulate → 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
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
### 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.

```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
  Pipeline.Reduce(output)(N by tileSize) { 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 2) { i =>
      tile1(i) * tile2(i)
    }{ _ + _ }
  }{ _ + _ }
}
```
Controllers

- Every “loop” in Spatial is a controller

- **Controller** - A hardware counter chain whose values control **datapaths** or other controllers

- Controllers are either:
  - **Inner Controller** - Datapath: consisting of *only* primitive nodes
    - arithmetic, if-then/mux, memory-access, etc.)
  - **Outer Controller** - Other controllers

```c
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 runtime of a single controller is:

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

- \( T \) = Cycles per execution
- \( II \) = Initiation interval
- \( \text{iters} \) = 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)
Controllers - Inner

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

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

Foreach \((N \times 1, M \times 1, P \times 1, Q \times 1)\)\{(i, j, p, q) =>

val \(\text{sum} = i + j + p + q\)

val \(\text{next} = \text{reg}.\text{value} \^ \text{sum}\)

\(\text{reg} := \text{mux}(q == 0, \text{reg}.\text{value}, \text{next})\)\}

Stanford CS149, Fall 2019

Diagram showing the execution flow and registers.
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 => ... }
Controllers - Outer

Initiation interval and latency for outer controllers depends on their “schedule,” which we will introduce next.

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

\[ T = II_{eff} \times (iters - 1) + L_{eff} \]
Controllers - Outer

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

**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
}
ForkJoin

\[
T = \frac{I}{I_{\text{eff}}} \cdot (\text{iters} - 1) + L_{\text{eff}}
\]

Fork-Join

Stream.Foreach(*){i =>
    fifoIn load dram
    Foreach(M by 1){ j => fifo.enq(fifoIn.deq()) * j }}
    dram2 store fifo
}
The runtime equation and schedules are important, but understanding the controller hierarchy and how these pieces fit together is the key to designing good accelerators

Let’s talk about performance debugging
Performance Debugging

Performance debugging is a process applied to one parent-child slice of the hierarchy at a time.

Consider this example parent with three children:

```plaintext
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 runtime of the Parent Sequential Controller

The waveform is nested like this:
Performance Debugging

We want to minimize the runtime of the **Parent Sequential Controller**

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

We want to minimize the runtime of the **Parent Sequential Controller**

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

**iters** = \( \frac{Q}{TS} \)

**II}_{\text{eff}} = L_{\text{eff}} = \sum T_{\text{child}}
Performance Debugging

The controller hierarchy is a more convenient way to visualize performance
Spatial **automatically** generates these trees for your application

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

**Sequential (i)**
[line:1]

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

**Pipelined (j)**
[line:3]

**Stream (j)**
[line:4]

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

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

Compressed controller hierarchy is a more convenient way to visualize performance
Spatial **automatically** generates these trees for your application

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

How do you use T and iteration counts effectively?

Sequential Foreach(0 by TS){ i =>
  Foreach(N by 1){ j => /* Primitives */ }
  Pipe.Foreach(M by 1){ j => /* Controllers */ }
  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

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

Parallelization under 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
Optimization Example

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

- Outer controller schedule (Reduce)
- PO
- PI

```scala
// Inner product accelerator
Accel {
  val tile1 = SRAM[Int](tileSize)
  val tile2 = SRAM[Int](tileSize)
  // Outer reduce
  Reduce(outputs(N by tileSize by PO)){ 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 PI){ i =>
      tile1(i) * tile2(i)
    }{a, b => a + b}
  }{a, b => a + b}
}
```
Optimization Example

The baseline implementation is PI=1, PO=1, and schedule=Sequential

Our instrumented controller tree will look like this

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

- for a series of optimizations let’s track
  - logic utilization (arithmetic nodes)
  - memory utilization (bytes)
  - runtime (cycles)

```scala
// Inner product accelerator
Accel {
  val tile1 = SRAM[Int](tileSize)
  val tile2 = SRAM[Int](tileSize)
  // Outer reduce
  Reduce(output)(N by tileSize by P0){ 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 PI){ i =>
      tile1(i) * tile2(i)
    }{a, b => a + b}
  }{a, b => a + b}
}
```
Optimization Example

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

- 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.
- There is no logic increase since we are not changing the datapaths.

<table>
<thead>
<tr>
<th>Schedule</th>
<th>PI</th>
<th>PO</th>
<th>Logic (Normalized)</th>
<th>Memory (Normalized)</th>
<th>Runtime (Cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>20000</td>
</tr>
<tr>
<td>Pipe</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>15000</td>
</tr>
<tr>
<td>Pipe</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>10000</td>
</tr>
<tr>
<td>Pipe</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>5000</td>
</tr>
<tr>
<td>Pipe</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Pipe</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Pipe</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Pipe</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

**Optimization Example**

- 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.
- There is no logic increase since we are not changing the datapaths.
Optimization Example

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

\[
T = II_{eff} \times (iters - 1) + L_{eff}
\]
Optimization Example

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

- From PI=2 to PI=4, we consume more logic but did not see much speedup
  - Inner reduce is no longer the bottleneck
Optimization Example

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

\[
T = II_{eff} \times (iters - 1) + L_{eff}
\]
**Optimization Example**

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

- The best design has the shortest runtime and uses the fewest resources
- Scale back some parallelization factors to get a better design
Summary

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

- 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
  - When does application become memory-bound

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