

Lecture 25: Board Notes: Threads and GPUs

Announcements:

- Reminder: HW 7 due today
 - Reminder: Submit project idea via (plain text) email by 11/24
-

Recap:

- **Slide 4:** Lecture 23: Introduction to Parallel Processing
 - o Types of multi-processors
 - o Flynn's taxonomy
 - o What's a superscalar machine?
 - o Introduction to MIMD machines
 - o How parallelization impacts performance
 - o What can impede performance?
 - **Slide 5:** Lecture 24: Introduction to Parallel Processing + Parallel Processing on Multi-core
 - o Simple, quantitative examples on how (i) reliability, (ii) communication overhead, and (iii) load balancing can impact the performance of parallel systems
 - o "Technology drive to multi-core computing"
-

Today (Lecture 25):

- **Slide 6:** Multi-core HW/SW models
 - o All issues covered in Lectures 23 and 24 apply to "on-chip" in addition to "room level" computing systems
 - o That said, 2 models for HW-SW driven "on-chip" parallelism deserve special discussion:
 - Threads
 - GPU-based computing
-

Threads:

Outline (**Slide 7**):

- What's a thread?
- Coupling to architecture
- Example → scheduling
- Programming models
- Why HW knowledge is important

#####

Processes vs. Threads (**Slide 8**):

Processes:

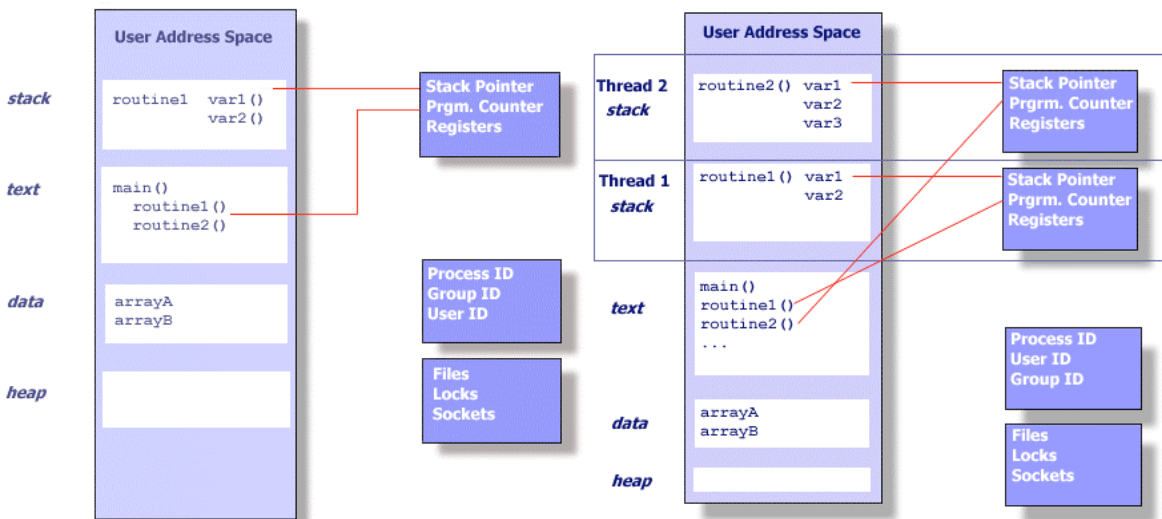
- Before discussing threads, first need to understand basic UNIX process
 - o Created by OS, typically a fair amount of "overhead" – encompassing program resources and execution state
 - o More specifically:
 - Process ID, process group ID, user ID
 - Working directory

- Program instructions
- Registers
- Stack space
- Heap
- File descriptors
- Shared libraries
- Inter-process communication tools: message queues, pipes, semaphores, shared memory, etc.

Threads:

- Threads can exist within processes and share process resources
 - o Duplicate bare essentials needed to execute code on-chip
 - Program counter
 - Stack pointer
 - Registers
 - Scheduling priority
 - Set of pending and blocked signals
 - Thread specific data

Threads vs. Processes in Pictures (Slide 9):



- **Part A:** Thus, in a UNIX environment a thread:
 - o Exists within a process, and uses process resources
 - o Has its own independent control flow (PC)
 - o Duplicates only resources it needs to be executed by processor datapath
 - o Shares process resources with other threads
 - Thus, if you close a file with one thread, all threads see file closed
 - o Dies if parent process dies

#####

Part B: (Some of the) advantages of threads:

- Can be much faster to schedule / run on HW
 - o Avoid overhead of a context switch

- Relatively speaking, process context switch requires considerable work
 - o 1000s of ns! Can execute instruction in just a few ns or less...
 - o Side note: Why?
 - Recall VA translation:
 - TLB → Cache → deliver data (in best case)
 - o Each process has its own Page Table
 - o TLB is fast cache for Page Table
 - o TLB has at least 64 entries
 - o TLB misses require main memory accesses...

- Other benefits:
 - Responsiveness:
 - o If one thread is blocked or writing, others can work
 - Resource Sharing:
 - o Application has several threads of activity in one 1 address space
 - Economy:
 - o In Solaris 2, crating a process 30X slower than thread, switching 5x slower
 - Utilization of multiprocessors:
 - o Each thread can run in parallel on a different processor core

#####

Hardware Models:

Part C: Generically, hardware multi-threading allows multiple threads to share functional units of single processor in an overlapping fashion

- Each processor has to duplicate independent state of each thread
 - o i.e. register file and PC
- Memory can be shared through virtual memory mechanisms which already could support multi-programming

Picture of hypothetical threaded datapath:

Ideally, HW should support relatively quick change to new thread very quickly
 - i.e. as opposed to process context switch

Slide 10 & Part D: 3 models of multi-threading to discuss:

- **Fine grained multi-threading:**

- o Might switch between threads EVERY instruction
- o Interleaving done in round robin fashion
 - Threads that are stalled could just be skipped
- o Practically, MUST be able to switch threads each CC
 - Therefore, would probably have separate register files – and would not do a register file content swap
- o Advantages:
 - Can high throughput losses from stalls...
 - i.e. could eliminate stall of lw followed by ALU in MIPS 5 stage pipe
- o Disadvantage
 - Slows down execution of individual (higher priority?) threads
 - i.e. each thread gets its turn

Sketch of datapath with multiple PC registers, register files...

Hypothetical threads and pipeline:

T1: a. lw \$5, 0(\$6) T2: c. sub \$10, \$11, \$12
 b. add \$7, \$5, \$9

Pipeline:

		1	2	3	4	5	6	7	8
a.	lw ...	F	D	E	M	W			
c.	sub...		F	D	E	M	W		
b.	add...			F	D	E	M	W	

- **Course grained multi-threading:**

- o Only switches threads on “expensive” stalls – i.e. L2 cache miss
- o Advantages:
 - Can simplify thread swap overhead; could take longer
 - Individual threads not slowed as much
- o Disadvantages:
 - Harder to overcome throughput losses from shorter stalls
 - i.e. on a stall, pipeline must be emptied or frozen and then refilled
 - Thus, pipeline drain/fill time should be small compared to stall time

- **Simultaneous multi-threading:**

- o A variation on HW multi-threading that uses the resources of superscalar machines
- o Exploit both instruction level parallelism (i.e. pipelining) and thread-level parallelism
- o Idea:
 - Multi-issue superscalar processors often have more functional unit parallelism available than 1 thread can use effectively
 - Moreover, with register renaming (mapping tables), dynamic scheduling, etc. multiple instructions can be issued from multiple threads without worrying about dependencies between them.

Remember register-renaming idea – i.e. have 1 large register file with remaps

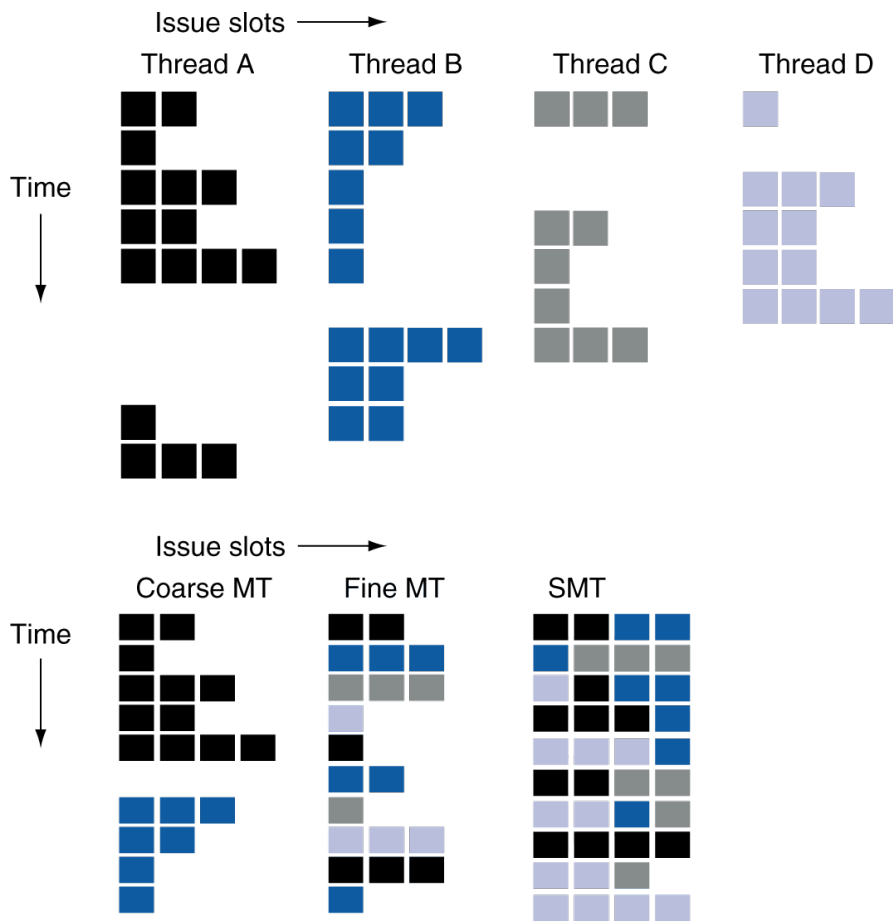
Assume:

	Map Table:		
	R1	R2	R3
	q7	q8	q9
Add R1, R2, R3 →	q10	q8	q9

Could say that q0 → q31 devoted to Thread 1, q31 → q62 devoted to Thread 2, etc.

Could also share data between threads this way.

Example 1 (Slide 11): Fine grained vs. Coarse grained vs. SMT:



#####

Part E: Example 2:

Thread 1:

- a. Add R1, R2, **R3**
- b. Div R8, R10, R12
- c. Add R17, R8, R19

Thread 2:

- d. Sub R7, R8, R9
- e. XOR R21, R22, R23
- f. AND R24, R24, R7

Thread 3:

- g. Load **R3**, 0(R2)
- h. Add R11, R3, R1
- i. Add R12, R3, R2

Consider the following:

- R3 referenced in T1 and R3 referenced in T3 could be 2 different physical registers
 - o Could also have 1 common register file and do re-mappings
- With fine grain model, each thread could get a turn in some round-robin fashion
 - o Having separate register files would significantly reduce context switches
 - o Instruction **a** could immediately be followed by instruction **d** in the pipeline
 - o Could skip T3 if waiting for page fault because of instruction **g**
 - o **Did example previously.**
- With coarse grain model, may do context switch if:
 - o Instruction **b** has high latency and need to wait on RAW hazard
 - o Instruction **g** has L2 cache miss and need to wait to process
- With SMT, Instructions **a**, **b**, **d**, and **g** may run simultaneously
 - o Realistic modern superscalar datapath might have a lw/sw unit, a few adders, 1 multiplier, 1 divider, etc.
 - o Draw picture of parallel execution paths.

Part F: Example 2:

Consider the following three CPU organizations:

- CPU SS:
 - o A 2-core superscalar microprocessor that provides out-of-order issue capabilities on two functional units. Only a single thread can run on each core at a time
- CPU MT:
 - o A fine-grained multi-threaded processor that allows instructions from two threads to be run concurrently (i.e. there are two functional units), though only instructions from a single thread can be issued on any cycle.
- CPU SMT:
 - o An SMT processor that allows instructions from two threads to be run concurrently (i.e. there are two functional units), and instructions from either or both threads can be run on any cycle.

Assume we have two threads X and Y to run on these CPUs that include the following operations:

Thread X	Thread Y
A1 – takes 2 cycles to execute	B1 – no dependencies
A2 – depends on the result of A1	B2 – conflicts for a functional unit with B1
A3 – conflicts for a functional unit with A2	B3 – no dependencies
A4 – depends on the result of A2	B4 – depends on the result of B2

Assume all instructions take a single cycle to execute unless noted otherwise OR they encounter a hazard.

Question 1:

Assume that you have one SS CPU. How many cycles will it take to execute these 2 threads? How many issue slots are wasted due to hazards?

Core 1	Core 2
A1, A3	B1, B3
A1	B2
A2	B4
A4	

9 of 16 slots used, 56% usage rate

Question 2:

Now assume that you have 1 MT CPU. How many cycles will it take to execute these 2 threads? How many issue slots are wasted due to hazards? Assume each thread gets 2 CCs

Functional Unit 1	Functional Unit 2
A1	A3
A1	
B1	B3
B2	
A2	
A4	
B4	

9 of 14 slots used for a 64% usage rate; could also argue 9 of 16 slots used for 54% usage rate.

Question 3:

Now assume that you have 1 MT CPU. How many cycles will it take to execute these 2 threads? How many issue slots are wasted due to hazards?

Functional Unit 1	Functional Unit 2
A1	B1
A1	B2
A2	B3
A3	B4
A4	

#####

Slides 12–15 Mixed Models:

- Threaded systems and multi-threaded programs are not specific to multi-core chips.
 - o In other words, could imagine a multi-threaded uni-processor too...
- However, could have an N-core chip where:
 - o ... N threads of a single process are run on N cores
 - o ... N processes run on N cores – and each core splits time between M threads
 - (later → real chip examples)

#####

Part G: Writing threaded programs for supporting HW:

- For UNIX systems, a standardized, C-language threads programming interface has been specified by the IEEE – POSIX threads (or Pthreads)
- For a program to take advantage of Pthreads...
 - o Should be capable of being organized into discrete, independent tasks which can execute concurrently
- More detailed discussion in Lecture 26...
- ...but generally, programs that have the following characteristics are well-suited for Pthreads:
 - o Work that can be executed OR data that can be operated on multiple tasks at the same time
 - o Have sections that will block and experience long I/O waits
 - i.e. while 1 thread is waiting for I/O system call to complete, CPU intensive work can be performed by other threads
 - o Use many CPU cycles in some places, but not others
 - o Must respond to asynchronous events
 - i.e. a web server can transfer data from previous requests and manage arrival of new requests
 - o Some work is more important than others (priority interrupts)
- SUNSparc T2 (Niagra T2) – for servers, circa October 2007
 - o Simple cores, fine-grained model
 - o Need to know so that if you have a high-priority thread, you can schedule accordingly

Slides 17–18 Technology Driven Machines:

GPUs:

Outline (Slide 19)

- Motivation for GPUs + necessary processing
- Example problem:
 - o Generic CPU pipeline
 - o GPU-based vs. Uni-processor Z-buffer problem
- What does a GPU architecture look like?
 - o Explain in context of SIMD
- Applicability to other computing problems

#####

Part H: Motivation for GPUs:

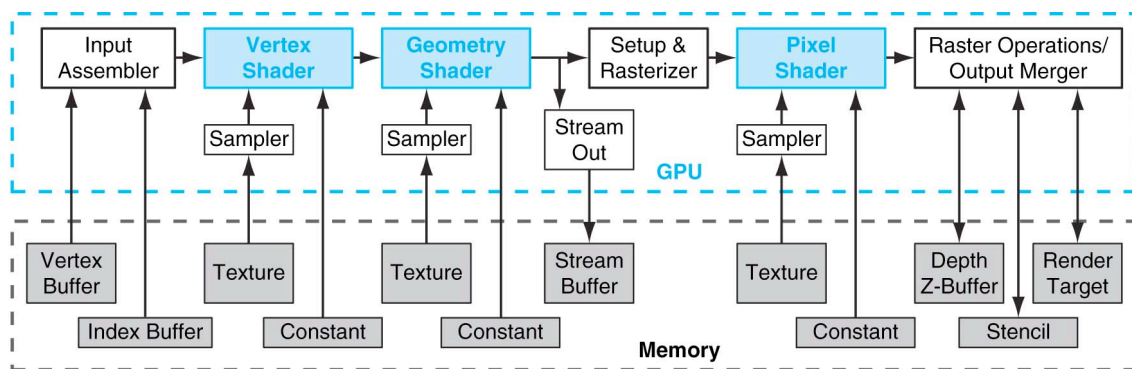
- From gaming, multi-media applications perspective, it is highly desirable to render complex 3D scenes, with high resolution, at interactive (real-time) frame rates
 - o Let's put some numbers to this...
 - Scene is rendered in 1080p format:
 - This means 1920 pixels by 1080 pixels = 2,073,600 pixels
 - Required frame rates usually 60 per second (or greater)
 - Thus, need to render 124,416,000 pixels per second
 - However, from standpoint of throughput, need to process information for frame buffer just once every ~16 ms (i.e. 1 frame / 60 s ~ 1 frame every 0.016 s)

#####

Slide 20 How is a frame rendered?

- Helpful to consider how the 2 standard graphics APIs – OpenGL and Direct 3D – work.
 - o These APIs define a logical graphics pipeline that is mapped onto GPU hardware and processors – along with programming models and languages for the programmable stages
 - o *In other words, API takes primitives like points, lines and polygons, and converts them into pixels*
- How does the graphics pipeline do this?
 - o First, important to note that “pipeline” does not mean the 5 stage pipeline we talked about earlier
 - o Pipeline describes sequence of steps to prepare image/scene for rendering

Slide 21 Example of Direct3D pipeline:



- **Part I:** More specifically:
 - o Input Assembler:
 - Collects vertices / primitives (points, lines, polygons)
 - o Vertex Shader:
 - Each object can be specified in its own logically defined coordinate system
 - Before rendering, need to transform to common coordinate system
 - Executes pre-vertex processing, including (i) transforming the vertex 3D position into a screen position and (ii) lighting the vertex to determine its color
 - o Geometry Shader:
 - Executes per primitive processing and can add or drop primitives based on scene development
 - o Setup and Rasterizer:
 - Each visible screen space triangle overlaps some pixel in display; determining pixels is called rasterization
 - Generates pixel fragments (fragments are potential contributions to pixels) that are covered by a geometric primitive
 - *Highly parallelizable*
 - o Pixel Shader:
 - Performs per-fragment processing including interpolating per fragment parameters, texturing and coloring

- Pixel shaders make extensive use of sampled and filtered lookups into texture arrays using interpolated floating point coordinates
 - Actual color can be taken from lighting calculation; but textures can give illusion of more detail
 - Textures stored in high speed memory
 - See **Slide 22**
- Raster Operations and Processing:
 - Performs Z-buffer depth testing and stencil testing
 - Will discuss shortly.

Result is output to frame buffer for display.

#####

Part J: Explore just 1 part of pipeline in more depth: Z-Buffer

- Z-buffering manages image depth coordinates in 3D graphics
- Is one solution to the “visibility problem”
 - i.e. rendering only parts of the scene that are visible – see **Slide 23**
- When object rendered by a graphics processor, depth of a generated pixel is stored in a buffer
 - (z is the “depth” coordinate)
- If another object of scene needs to be rendered in the same pixel, graphics processor can compare the 2 depths and chose the one closest to the observer
 - (i.e. a closer object will hide a farther one...)
 - Can be inefficient
 - Pixel’s visibility is determined after the pixel’s color value is finalized
 - Could do additional Z-check before texturing operations are performed
 - If pixel not visible, steps are skipped ... (i.e. pixel is culled)
- Algorithm:

Declare an array z buffer(x, y) with one entry for each pixel position.
Initialize the array to the maximum depth.

If have performed a perspective depth transformation then all z values are between 0.0 and 1.0

So initialize all values to 1.0.

```

for each polygon P {
  for each pixel (x, y) in P {
    compute z_depth at x, y
    if z_depth < z_buffer (x, y) {
      z_buffer (x, y) = z_depth
      color(x,y) = intensity of P at (x,y);
    }
  }
}

```

- Recall we need to render 124,416,000 per second for an HD display...
 - Assume a uni-processor with clock rate of 2 GHz
 - How many instructions could this loop be assuming 1 instruction finishes each CC?

$$\frac{\text{Instructions}}{\text{Iteration}} = \frac{1 \text{ second}}{124,416,000 \text{ iterations}} \times \frac{1 \text{ instruction}}{\text{CC}} \times \frac{1 \text{CC}}{0.5 \times 10^{-9} \text{seconds}} = 16.075 = 16$$

- This *may* be realistic...
- But this is also just 1 step of the graphics pipeline!!!
 - So, we need parallelism!

#####

Slides 24—26: GPU-based hardware

#####

Slides 27—28: Applicability to other problems