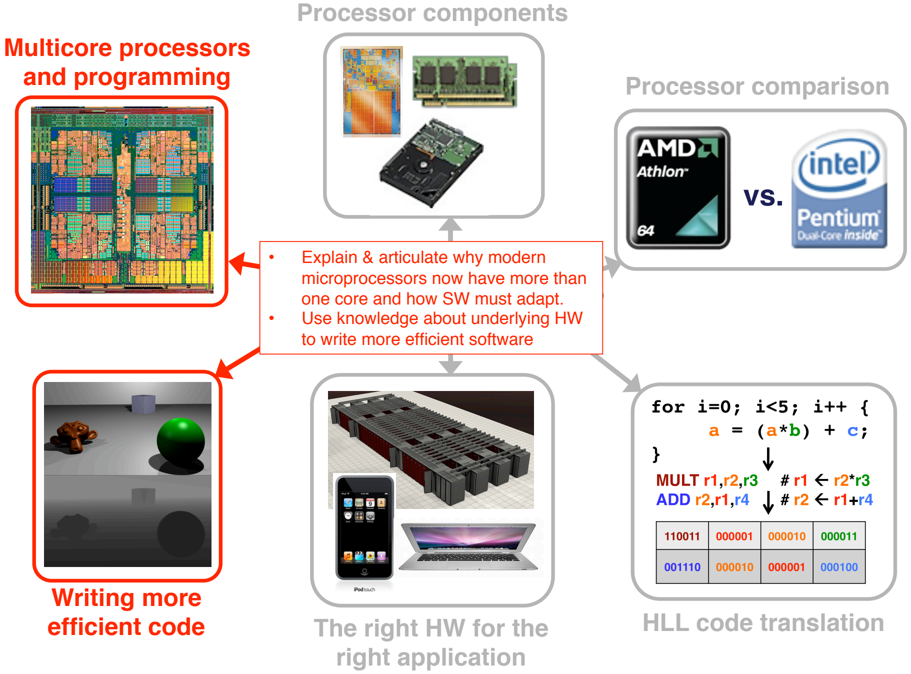
Lecture 25
Threads and GPUs

Suggested Readings

- Readings
  - H&P: Chapter 7 – especially 7.1-7.8
    - (Over next 2 weeks)
  - Introduction to Parallel Computing
    - https://computing.llnl.gov/tutorials/parallel_comp/
  - POSIX Threads Programming
    - https://computing.llnl.gov/tutorials/pthreads/
  - How GPUs Work
    - *www.cs.virginia.edu/~gfx/papers/pdfs/59_HowThingsWork.pdf*



Processor components

Multicore processors and programming

Processor comparison

vs.

- Explain & articulate why modern microprocessors now have more than one core and how SW must adapt.
- Use knowledge about underlying HW to write more efficient software

```
for i=0; i<5; i++ {
      a = (a*b) + c;
}
MULT r1,r2,r3    # r1 ← r2*r3
ADD r2,r1,r4     # r2 ← r1+r4
```

110011	000001	000010	000011
001110	000010	000001	000100

Writing more efficient code

The right HW for the right application

HLL code translation

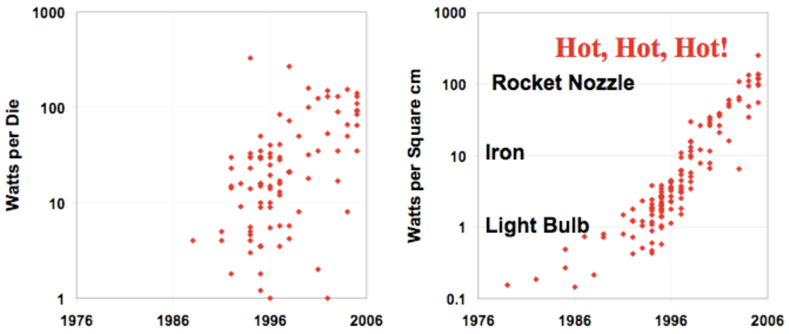
Recap: L23 – Intro to Parallel Processing

- Types of multi-processors
  - Room-level, on-chip
- Types of machines
  - SISD (uniprocessor, pipelined, superscalar)
  - SIMD (more today)
  - MISD (not really used)
  - MIMD (centralized & distributed shared memory)
- More detail about MIMD
- How parallelization impacts performance
- What can impede performance of parallel machines?
  - Coherency, latency, contention, reliability, languages, algorithms

Recap: L24 – Parallel Processing on MC

- Simple quantitative examples on how (i) reliability, (ii) communication overhead, and (iii) load balancing impact performance of parallel systems
- Technology drive to multi-core computing

Why the clock flattening? POWER!!!!



Today: L25 – Threads and GPUs

- All issues covered in L23 & L24 apply to "on-chip" computing systems as well as "room level" computing systems
- That said, 2 models that lend themselves well to on-chip parallelism deserve special discussion:
  - Threads
  - GPU-based computing

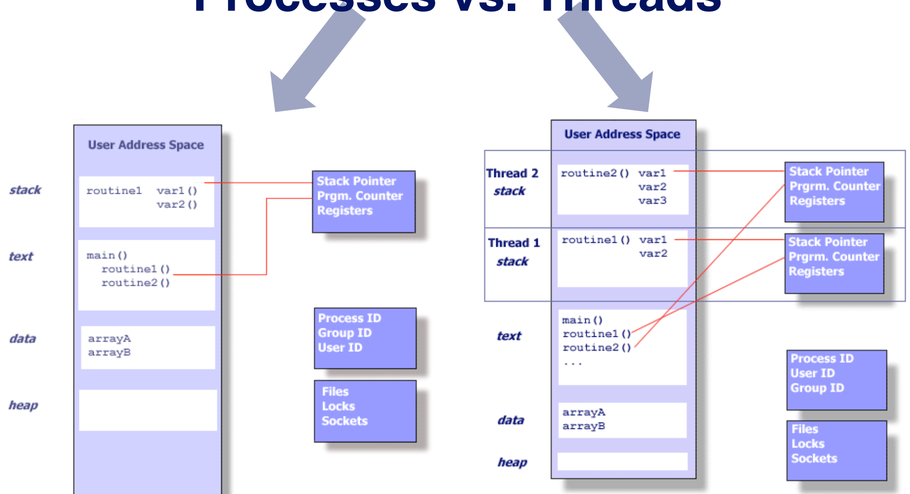
Threads First

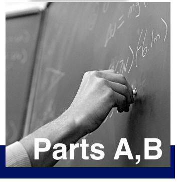
- Outline of Threads discussion:
  - What's a thread?
    - How many people have heard of / used threads before?
  - Coupling to architecture
  - Example: scheduling threads
    - Assume different architectural models
  - Programming models
  - Why intimate knowledge about HW is important

Processes vs. Threads

- Process
  - Created by OS
  - Much "overhead"
    - Process ID
    - Process group ID
    - User ID
    - Working directory
    - Program instructions
    - Registers
    - Stack space
    - Heap
    - File descriptors
    - Shared libraries
    - Shared memory
    - Semaphores, pipes, etc.
- Thread
  - Can exist within process
  - Shares process resources
  - Duplicate bare essentials to execute code on chip
    - Program counter
    - Stack pointer
    - Registers
    - Scheduling priority
    - Set of pending, blocked signals
    - Thread specific data

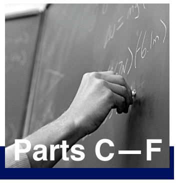
Processes vs. Threads



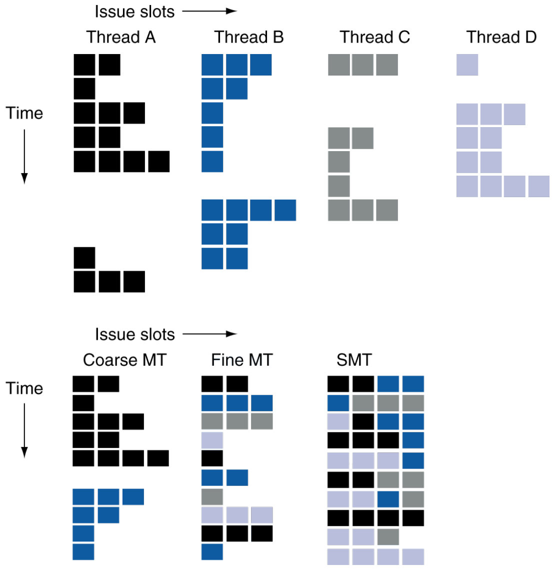


Multi-threading

- Idea:
  - Performing multiple threads of execution in parallel
    - Replicate registers, PC, etc.
  - Fast switching between threads
- Flavors:
  - Fine-grain multithreading
    - Switch threads after each cycle
    - Interleave instruction execution
    - If one thread stalls, others are executed
  - Coarse-grain multithreading
    - Only switch on long stall (e.g., L2-cache miss)
    - Simplifies hardware, but doesn't hide short stalls
      - (e.g., data hazards)
  - SMT (Simultaneous Multi-Threading)
    - Especially relevant for superscalar



Coarse MT vs. Fine MT vs. SMT



Mixed Models:

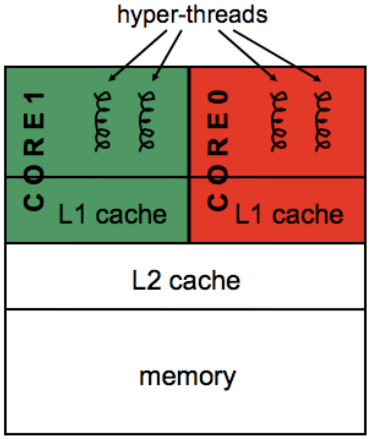
- Threaded systems and multi-threaded programs are not specific to multi-core chips.
  - In other words, could imagine a multi-threaded uni-processor too…
- However, could have an N-core chip where:
  - … N threads of a single process are run on N cores
  - … N processes run on N cores – and each core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

Comparison: multi-core vs SMT

- Multi-core:
  - Since there are several cores, each is smaller and not as powerful (but also easier to design and manufacture)
  - However, great with thread-level parallelism
- SMT
  - Can have one large and fast superscalar core
  - Great performance on a single thread
  - Mostly still only exploits instruction-level parallelism

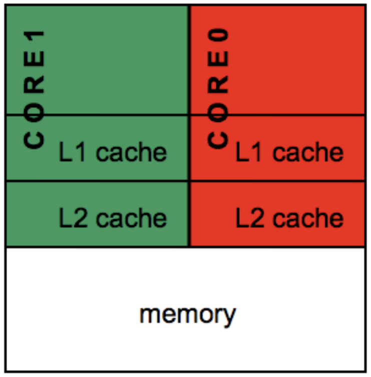
Or can do both…

- Dual-core Intel Xeon processors
- Each core is hyper-threaded
- Private L1 caches
- Shared L2 caches



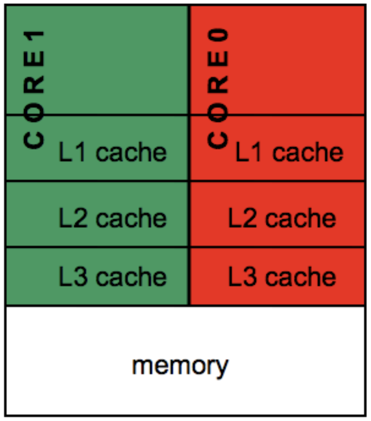
Real life examples…

Designs with private L2 caches



Both L1 and L2 are private

Examples: AMD Opteron, AMD Athlon, Intel Pentium D



A design with L3 caches

Example: Intel Itanium 2

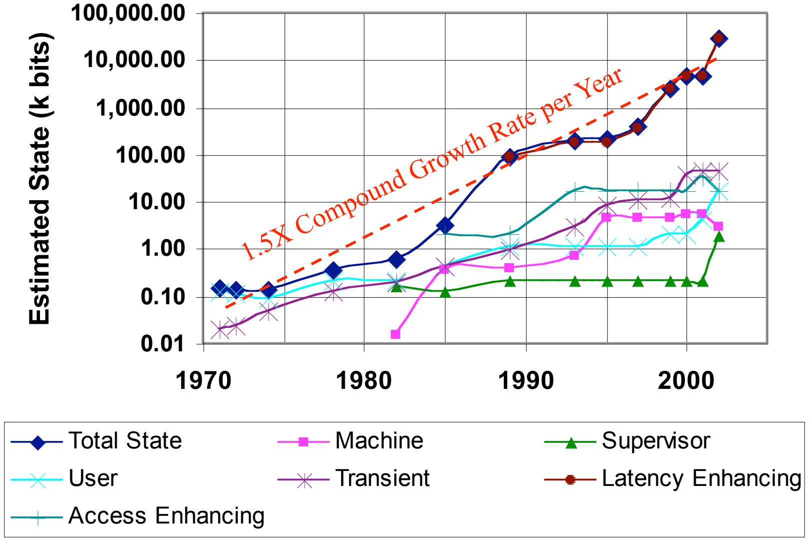
Writing threaded programs for supporting HW

Impact of modern processing principles (Lots of "state")

- **User:**
  - state used for application execution
- **Supervisor:**
  - state used to manage user state
- **Machine:**
  - state that configures the system
- **Transient:**
  - state used during instruction execution
- **Access-Enhancing:**
  - state used to simplify translation of other state names
- **Latency-Enhancing:**
  - state used to reduce latency to other state values

Impact of modern processing principles

(Total State vs. Time)



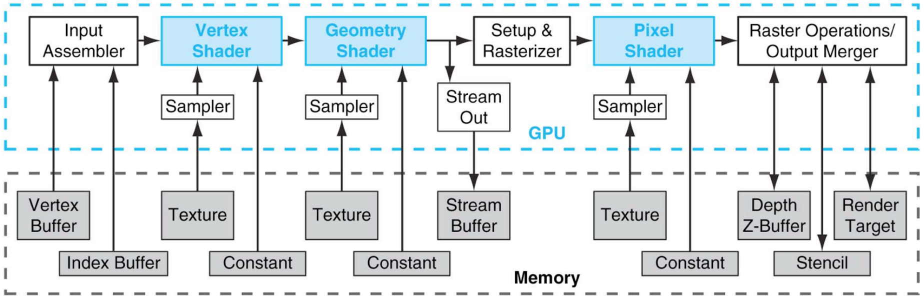
GPU discussion points

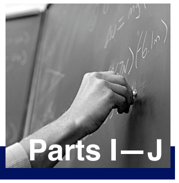
- Motivation for GPUs:
- Necessary processing
- Example problem:
  - Generic CPU pipeline
  - GPU-based vs. Uni-processor Z-buffer problem
- What does a GPU architecture look like?
  - Explain in context of SIMD
- Applicability to other computing problems

How is a frame rendered?

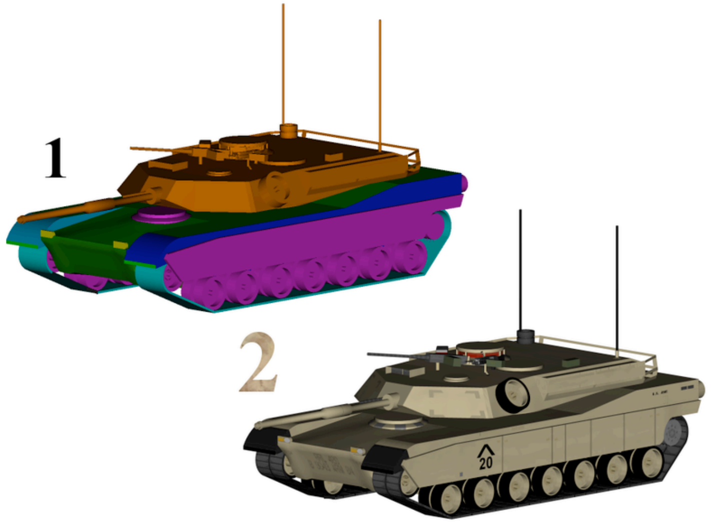
- Helpful to consider how the 2 standard graphics APIs – OpenGL and Direct 3D – work.
  - These APIs define a logical graphics pipeline that is mapped onto GPU hardware and processors – along with programming models and languages for the programmable stages
  - *In other words, API takes primitives like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

How is a frame rendered? (Direct3D pipeline)



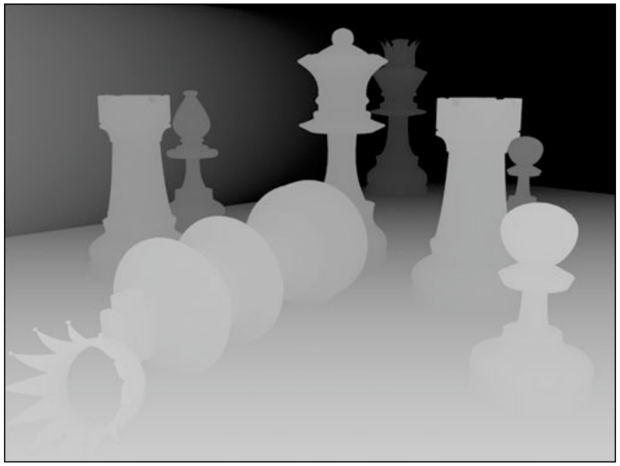


Example: Textures



http://en.wikipedia.org/wiki/File:Texturedm1a2.png

Example: Z-buffer



http://blog.yoz.sk/examples/pixelBenderDisplacement/zbuffer1Map.jpg

GPUs

- GPU = Graphics Processing Unit
  - Efficient at manipulating computer graphics
  - Graphics accelerator uses custom HW that makes mathematical operations for graphics operations fast/ efficient
    - Why SIMD? Do same thing to each pixel
- API language compilers target industry standard intermediate languages instead of machine instructions
  - GPU driver software generates optimized GPU-specific machine instructions

GPUs

- Also, SW support for GPU programming:
  - NVIDIA has graphics cards that support API extension to C – CUDA ("Computer Unified Device Architecture")
    - Allows specialized functions from a normal C program to run on GPU's stream processors
    - Allows C programs that can benefit from integrated GPU(s) to use where appropriate, but also leverage conventional CPU

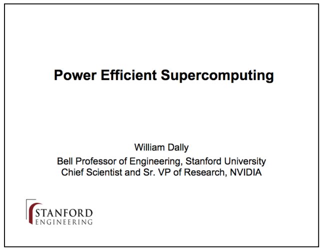
Modern GPU

- Often part of a *heterogeneous* system
  - GPUs don't do all things CPU does
  - Good at some specific things
    - i.e. matrix-vector operations
- GPU HW:
  - No multi-level caches
  - Hide memory latency with threads
    - To process all pixel data
  - GPU main memory oriented toward bandwidth

GPUs for other problems

- More recently:
  - GPUs found to be more efficient than general purpose CPUs for many complex algorithms
    - Often things with massive amount of vector ops
  - Example:
    - ATI, NVIDIA team with Stanford to do GPU-based computation for protein folding
    - Found to offer up to 40 X improvement over more conventional approach

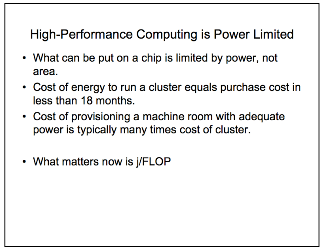
GPU Example



Power Efficient Supercomputing

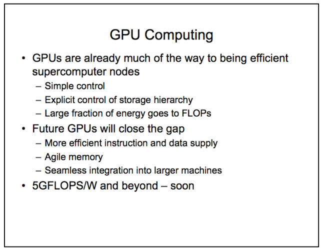
William Dally
Bell Professor of Engineering, Stanford University
Chief Scientist and Sr. VP of Research, NVIDIA

STANFORD ENGINEERING



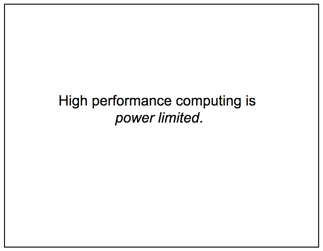
High-Performance Computing is Power Limited

- What can be put on a chip is limited by power, not area.
- Cost of energy to run a cluster equals purchase cost in less than 18 months.
- Cost of provisioning a machine room with adequate power is typically many times cost of cluster.
- What matters now is j/FLOP

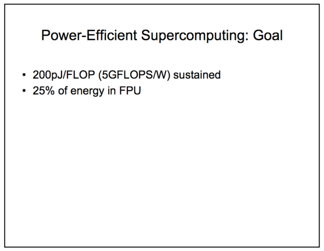


GPU Computing

- GPUs are already much of the way to being efficient supercomputer nodes
  - Simple control
  - Explicit control of storage hierarchy
  - Large fraction of energy goes to FLOPs
- Future GPUs will close the gap
  - More efficient instruction and data supply
  - Agile memory
  - Seamless integration into larger machines
- 5GFLOPS/W and beyond – soon



High performance computing is *power limited*.



Power-Efficient Supercomputing: Goal

- 200pJ/FLOP (5GFLOPS/W) sustained
- 25% of energy in FPU