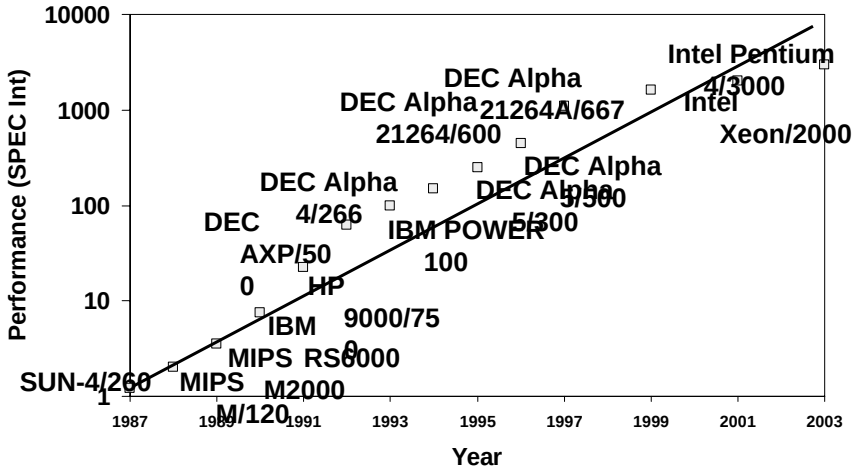
CS 316: Multicore/GPUs-II

Kavita Bala
Fall 2007
Computer Science
Cornell University

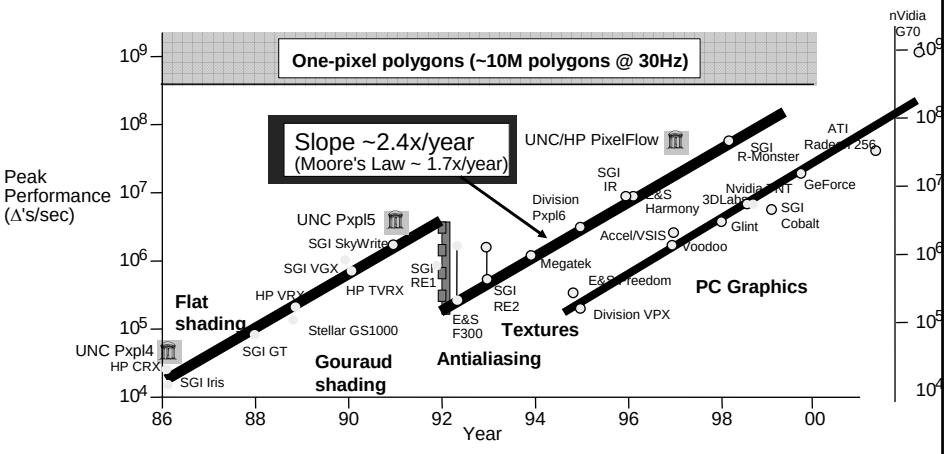
Announcements

- PA 4 graded
- Corewars due next Tuesday: Nov 27
  - Corewars party next Friday: Nov 30
- Prelim 2: Thursday Nov 29 (7-9:30)
  - Prelim review: Wed Nov 28
- PA 6 will be out next week
  - Due Dec 13: Demos from 2 onwards

Processor Performance Increase

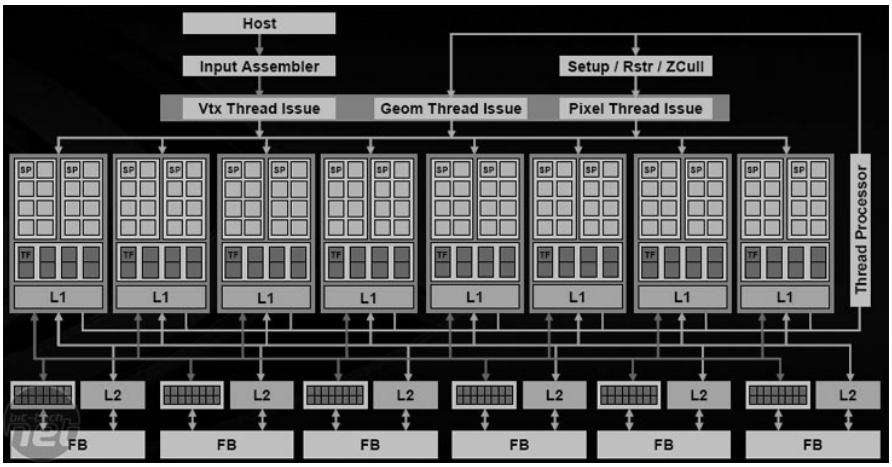


GPU Performance Growth



Graph courtesy of Professor John Poulton (from Eric Haines)

G80



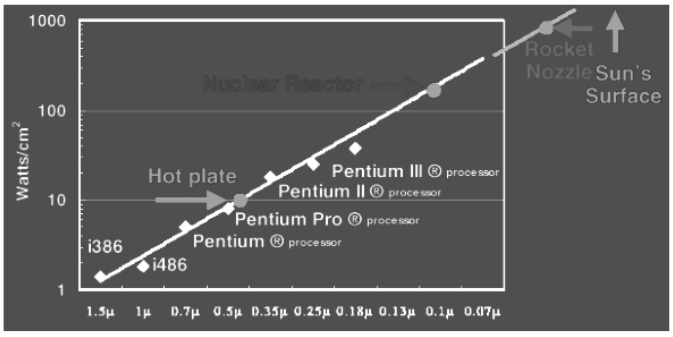
GPGPUs

- Can we use these machines for general computation?

- Scientific Computing
  - MATLAB codes
- Convex hulls
- Molecular Dynamics
- Etc.

- The case for parallelism….

Power Limits Performance



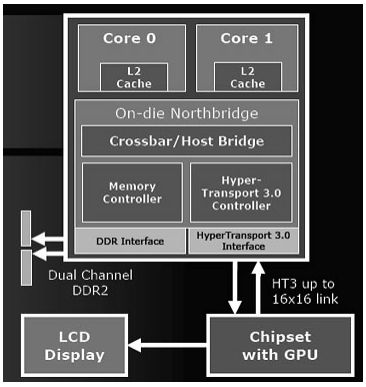
Why Multicore?

- Moore's law
  - A law about transistors
  - Smaller means faster transistors
- Power consumption growing with transistors: $P = CfV^2$, f prop V, P prop V^3
- Many lower speed cores: use same power

Nutshell: the argument for multicore

- Can't keep going faster
- Too much power
- Instead go slower, but have more cores

AMDs Hybrid CPU/GPU



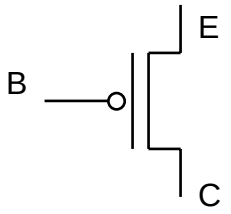
Intel's argument

- Multicore
  - 8, 32, 128
- Manycore
  - 1000s of cores

Do you believe?

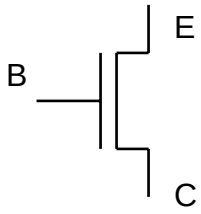
P and N Transistors

- PNP Transistor



- Connect E to C when base = 0

- NPN Transistor



- Connect E to C when base = 1

Amdahl's Law

- Task: serial part, parallel part
- As number of processors increases,
  - time to execute parallel part goes to zero
  - time to execute serial part remains the same
- *Serial part eventually dominates*
- Must parallelize ALL parts of task

$$\text{Speedup}(E) = \frac{\text{Execution Time without } E}{\text{Execution Time with } E}$$

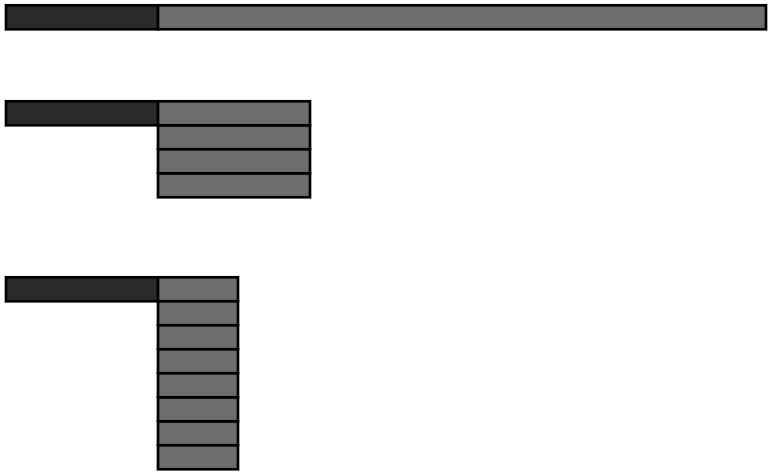
Amdahl's Law

- Consider an improvement E
- F of the execution time is affected
- S is the speedup

$$\text{Execution time (with } E) = ((1 - F) + F/S) \cdot \text{Execution time (without } E)$$

$$\text{Speedup (with } E) = \frac{1}{(1 - F) + F/S}$$

Amdahl's Law

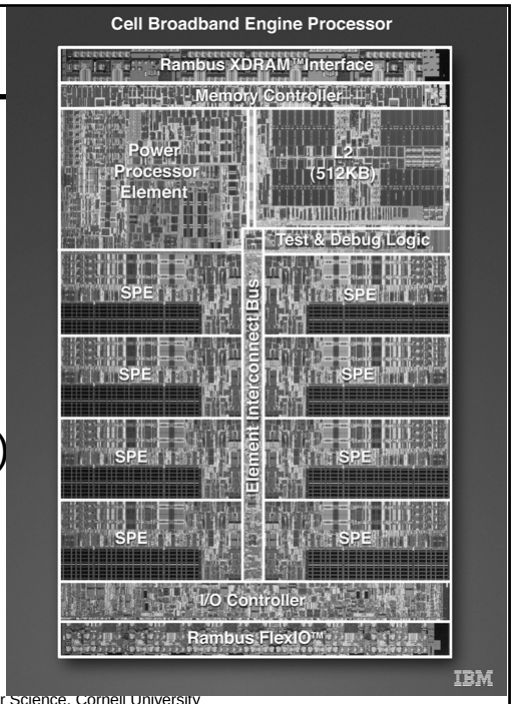


Argument for Heterogeneous Cores

- Main processor(s)
  - high speed
- Cores: lower speed
  - For parallel part of application

Cell

- IBM/Sony/Toshiba
- Sony Playstation 3
- PPE
- SPEs (synergestic)

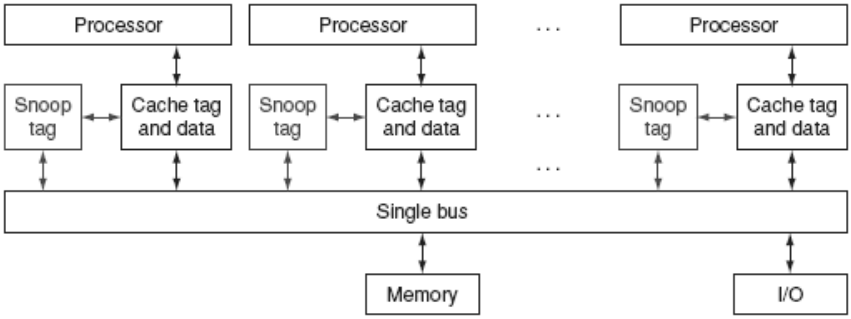


Implications of multicore

- Shared memory architectures
- Cache coherence is one big problem

Snooping Caches

- Read: respond if you have data
- Write: invalidate or update your data



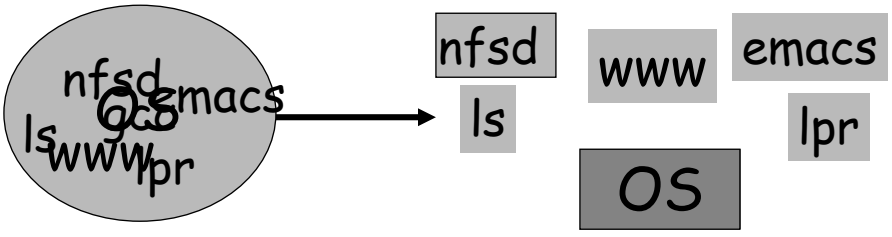
Writing

- Write-back policies for bandwidth
- Write-invalidate coherence policy
  - First invalidate all other copies of data
  - Then write it in cache line
  - Anybody else can read it
- Permits one writer, multiple readers

- In reality: many coherence protocols
  - Snooping doesn't scale

Processes

- Hundreds of things going on in the system



- How to make things simple?
  - Decompose computation into separate *processes*
- How to make things reliable?
  - Isolate processes to protect from each others' faults
- How to speed up?
  - Overlap I/O bursts of one process with CPU bursts of another

What is a process?

- A program being executed
  - Sequential, one instruction at a time
- OS abstraction: a thread of execution running in a restricted virtual environment
  - a virtual CPU and virtual memory environment, interfacing with the OS via system calls.
  - The unit of execution
  - The unit of scheduling
  - Thread of execution + address space

The same as “job” or “task” or “sequential process”. Closely related to “thread”

Context Switch

- Context Switch
  - Process of switching CPU from one process to another
- State of a running process must be saved and restored:
  - Program Counter, Stack Pointer, General Purpose Registers
- Suspending a process: OS saves state
  - Saves register values
- To execute another process, OS restores state
  - Loads register values

Details of Context Switching

- Very tricky to implement
  - OS must save state without changing state
  - Must run without changing any user program registers
    - CISC: single instruction saves all state
    - RISC: reserve registers for kernel
      - Or way to save a register and then continue
- Overheads: CPU is idle during a context switch
  - Cost of loading/storing registers to/from main memory
  - Cost of flushing useful caches (cache, TLB, etc.)
  - Waiting for pipeline to drain in pipelined processors

Cooperating Processes

- Processes can work cooperatively
  - Cooperating processes exploit parallelism
- Cooperating processes can be used for:
  - speedup (spread computation over multiple cores)
  - better interactivity: one process works while others are waiting for I/O
  - better structuring of an application into separate concerns
    - E.g., a pipeline of processes processing data
- But: cooperating processes need ways to
  - Communicate information
  - Coordinate (synchronize) activities

Shared memory

- Default: processes w/ disjoint physical memory
  - complete isolation prevents communication
- Set up shared segment of memory
  - Process creates shared mem as part of its memory
  - Other processes attach this to their memory space
- Allows high-bandwidth communication between processes by just writing into memory

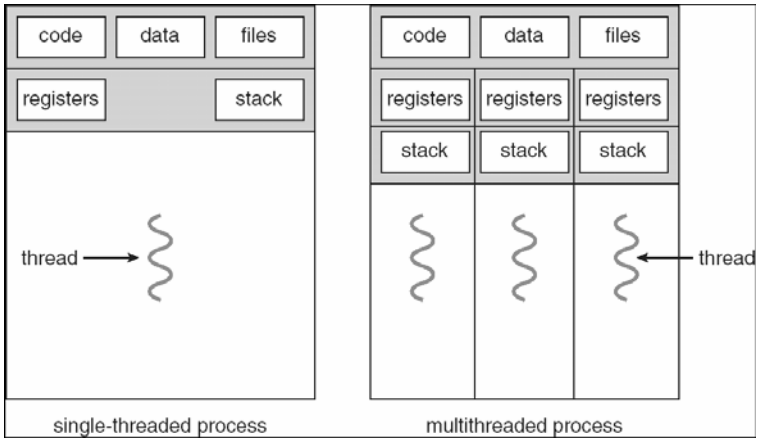
Processes are heavyweight

- Parallel programming with processes share:
  - same code
  - data in shared memory
  - same privileges
- What don't they share?
  - Each has its own PC, registers, and stack
- Idea: separate the idea of process (address space, accounting, etc.) from minimal "thread of control" (PC, SP, registers)

Threads vs. processes

- Most OSes support two entities:
  - Process: defines the address space and general process attributes
  - Thread: defines a sequential execution stream within a process
- For each process: there may be many threads
- Each thread associated with one process
  - Threads are the unit of scheduling
- Processes are *containers* in which threads execute

Multithreaded Processes



Two threads, one counter

Web servers use concurrency:

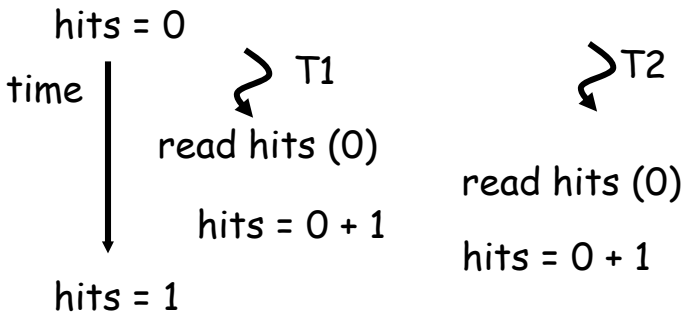
- Multiple threads handle client requests in parallel.
- Some shared state, e.g. hit counts:
  - each thread increments a shared counter to track number of hits

```
…
hits = hits + 1;
…
```

- What happens when two threads execute concurrently?

Shared counters

- Possible result: lost update!



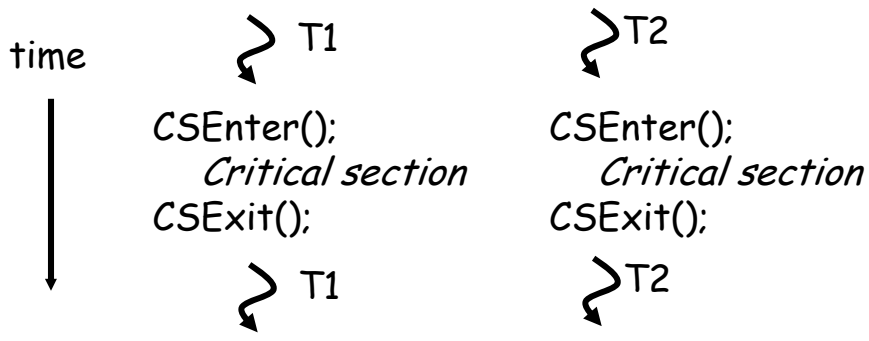
- Occasional timing-dependent failure ⇒ race condition
  - Difficult to debug

Race conditions

- Def: a timing dependent error involving shared state
  - Whether it happens depends on how threads scheduled: who wins "races" to instructions that update state
  - Races are intermittent, may occur rarely
    - Timing dependent = small changes can hide bug
  - A program is correct *only* if *all possible* schedules are safe
    - Number of possible schedule permutations is huge
    - Need to imagine an adversary who switches contexts at the worst possible time

Critical Sections

- Basic way to eliminate races: use *critical sections* that only one thread can be in
  - Contending threads must wait to enter



Mutexes

- Critical sections typically associated with mutual exclusion locks (*mutexes*)
- Only one thread can hold a given mutex at a time
- Acquire (lock) mutex on entry to critical section
  - Or block if another thread already holds it
- Release (unlock) mutex on exit
  - Allow one waiting thread (if any) to acquire & proceed

```
                pthread_mutex_init(m);
pthread_mutex_lock(m);      pthread_mutex_lock(m);
   hits = hits+1;              hits = hits+1;
pthread_mutex_unlock(m);    pthread_mutex_unlock(m);
```

T1 T2

Using atomic hardware primitives

- Mutex implementations usually rely on special hardware instructions that atomically do a read and a write.
- Requires special memory system support on multiprocessors

Mutex init: lock = false;

```
while (test_and_set(&lock));
```

Critical Section

```
lock = false;
```

test_and_set uses a special hardware instruction that sets the lock and returns the OLD value (true: locked; false: unlocked)
- Alternative instruction: compare & swap

Test-and-set

```
Boolean TestandSet (boolean lock) {
   boolean old = lock;
   lock = true;
   return old;
}
```

Except this is fully atomic

Using test-and-set for mutual exclusion

```
boolean lock = false;

 function Critical(){
    while TestAndSet(lock) skip
    //spin until lock is acquired critical section only
    //one process can be in this section at a time


   lock = false ;
   //release lock when finished with the
   // critical section
}
```

Spin waiting

- Example is a spinlock
  - Also called busy waiting or spin waiting
- Fine when have to wait for short time
- Wasteful when waiting for a long time
- Heuristic: spin for a bit, and then switch to some other thread

Happy Thanksgiving!