Last Time, Lecture 16: GPUs

- Data-Level Parallelism the least flexible but cheapest form of machine parallelism, and matches application demands
- Graphics processing units have developed general-purpose processing capability for use outside of traditional graphics functionality (GP-GPUs)
- SIMT model presents programmer with illusion of many independent threads, but executes them in SIMD style on a vector-like multilane engine.
- Complex control flow handled with hardware to turn branches into mask vectors and stack to remember µthreads on alternate path
- No scalar processor, so µthreads do redundant work, unit-stride loads and stores recovered via hardware memory coalescing
Uniprocessor Performance (SPECint)

- VAX: 25%/year 1978 to 1986
- RISC + x86: 52%/year 1986 to 2002
- RISC + x86: ??%/year 2002 to present


Parallel Processing: Déjà vu all over again?

- “... today’s processors ... are nearing an impasse as technologies approach the speed of light...”
- Transputer had bad timing (Uniprocessor performance↑)
  ⇒ Procrastination rewarded: 2X seq. perf. / 1.5 years
- “We are dedicating all of our future product development to multicore designs. ... This is a sea change in computing”
  - Paul Otellini, President, Intel (2005)
- All microprocessor companies switch to MP (2+ CPUs/2 yrs)
  ⇒ Procrastination penalized: 2X sequential perf. / 5 yrs
- Even handheld systems moved to multicore
  - Nintendo 3DS, iPhone4S, iPad 3 have two cores each (plus additional specialized cores)
  - Playstation Portable Vita has four cores
Symmetric Multiprocessors

- All memory is equally far away from all processors
- Any processor can do any I/O (set up a DMA transfer)

Synchronization

The need for synchronization arises whenever there are concurrent processes in a system *(even in a uniprocessor system)*

Two classes of synchronization:

*Producer-Consumer:* A consumer process must wait until the producer process has produced data

*Mutual Exclusion:* Ensure that only one process uses a resource at a given time
### A Producer-Consumer Example

Producer posting Item x:
- Load $R_{\text{tail}}$ (tail)
- Store ($R_{\text{tail}}$), x
- $R_{\text{tail}} = R_{\text{tail}} + 1$
- Store (tail), $R_{\text{tail}}$

Consumer:
- Load $R_{\text{head}}$ (head)
- spin:
  - Load $R_{\text{head}}$ (tail)
  - if $R_{\text{head}} == R_{\text{tail}}$ goto spin
  - Load $R_{\text{head}}$ (head)
  - $R_{\text{head}} = R_{\text{head}} + 1$
  - Store (head), $R_{\text{head}}$
  - process($R$)

The program is written assuming instructions are executed in order.

---

### A Producer-Consumer Example continued

Can the tail pointer get updated before the item x is stored?

Programmer assumes that if 3 happens after 2, then 4 happens after 1.

Problem sequences are:
- 2, 3, 4, 1
- 4, 1, 2, 3
Sequential Consistency

**A Memory Model**

“**A system is sequentially consistent** if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in the order specified by the program”

Leslie Lamport

Sequential Consistency =

arbitrary *order-preserving interleaving*

of memory references of sequential programs

Sequential concurrent tasks: T1, T2
Shared variables: X, Y (initially X = 0, Y = 10)

\[
\begin{align*}
\text{T1:} & \quad & \text{T2:} \\
\text{Store (X), 1 (X = 1)} & & \text{Load R}_1, (Y) \\
\text{Store (Y), 11 (Y = 11)} & & \text{Store (Y'), R}_3 (Y' = Y) \\
\text{Load R}_2, (X) & & \text{Load R}_2, (X') \\
\text{Store (X'), R}_2 (X' = X) & & \text{Store (X'), R}_2 (X' = X)
\end{align*}
\]

what are the legitimate answers for X’ and Y’?

\[(X', Y') \in \{(1,11), (0,10), (1,10), (0,11)\} \quad ?
\]

*If y is 11 then x cannot be 0*
Sequential Consistency

Sequential consistency imposes more memory ordering constraints than those imposed by uniprocessor program dependencies (→)

**What are these in our example?**

T1:
- Store (X), 1 (X = 1)
- Store (Y), 11 (Y = 11)

T2:
- Load R₁, (Y)
- Store (Y'), R₁ (Y' = Y)
- Load R₂, (X)
- Store (X'), R₂ (X' = X)

additional SC requirements

Does (can) a system with caches or out-of-order execution capability provide a sequentially consistent view of the memory?

*more on this later*

---

Issues in Implementing Sequential Consistency

Implementation of SC is complicated by two issues

- **Out-of-order execution capability**
  
  - Load(a); Load(b) yes
  - Load(a); Store(b) yes if a ≠ b
  - Store(a); Load(b) yes if a ≠ b
  - Store(a); Store(b) yes if a ≠ b

- **Caches**
  
  Caches can prevent the effect of a store from being seen by other processors

*No common commercial architecture has a sequentially consistent memory model!*
Memory Fences
Instructions to sequentialize memory accesses

Processors with relaxed or weak memory models (i.e., permit Loads and Stores to different addresses to be reordered) need to provide memory fence instructions to force the serialization of memory accesses.

Examples of processors with relaxed memory models:
- Sparc V8 (TSO, PSO): Membar
- Sparc V9 (RMO):
  - Membar #LoadLoad, Membar #LoadStore
  - Membar #StoreLoad, Membar #StoreStore
- PowerPC (WO): Sync, EIEIO
- ARM: DMB (Data Memory Barrier)
- X86/64: mfence (Global Memory Barrier)

Memory fences are expensive operations, however, one pays the cost of serialization only when it is required.

Using Memory Fences

Producer posting Item x:
Load \( R_{\text{tail}} \), (tail)
Store \((R_{\text{tail}})\), x
Membar_{SS}\
\( R_{\text{tail}}=R_{\text{tail}}+1 \)
Store (tail), \( R_{\text{tail}} \)

Consumer:
spin:
Load \( R_{\text{head}} \),\ (head)
if \( R_{\text{head}} == R_{\text{tail}} \) goto spin
Membar_{LL}
Load \( R_{\text{head}} \), \( (R_{\text{head}}) \)
\( R_{\text{head}}=R_{\text{head}}+1 \)
Store (head), \( R_{\text{head}} \)
process(R)

ensures that tail ptr is not updated before x has been stored
ensures that R is not loaded before x has been stored
CS152 Administrivia

- Quiz 4, Tuesday April 9
  - Lectures 13-16, Lab 4, PS 4
  - VLIW, Multithreading, Vector, GPU

Multiple Consumer Example

Producer posting Item x:
- Load $R_{\text{tail}}$, (tail)
- Store ($R_{\text{tail}}$, x)
- $R_{\text{tail}} = R_{\text{tail}} + 1$
- Store (tail), $R_{\text{tail}}$

Critical section:
Needs to be executed atomically by one consumer

Consumer:
- spin:
  - Load $R_{\text{head}}$, (head)
  - if $R_{\text{head}} == R_{\text{tail}}$ goto spin
  - Load $R_{\text{head}}$, (head)
  - $R_{\text{head}} = R_{\text{head}} + 1$
  - Store (head), $R_{\text{head}}$
  - process(R)

What is wrong with this code?
Mutual Exclusion Using Load/Store

A protocol based on two shared variables $c_1$ and $c_2$. Initially, both $c_1$ and $c_2$ are 0 (not busy)

Process 1

```
...  
c1=1;
L: if c2=1 then go to L  < critical section>
c1=0;
```

Process 2

```
...  
c2=1;
L: if c1=1 then go to L  < critical section>
c2=0;
```

What is wrong?  **Deadlock!**

Mutual Exclusion: *second attempt*

To avoid *deadlock*, let a process give up the reservation (i.e. Process 1 sets $c_1$ to 0) while waiting.

Process 1

```
L:  c1=1;
    if c2=1 then  
      { c1=0; go to L}  < critical section>
c1=0
```

Process 2

```
L:  c2=1;
    if c1=1 then  
      { c2=0; go to L}  < critical section>
c2=0
```

- Deadlock is not possible but with a low probability a *livelock* may occur.
- An unlucky process may never get to enter the critical section $\Rightarrow$ *starvation*
**A Protocol for Mutual Exclusion**

*T. Dekker, 1966*

A protocol based on 3 shared variables $c_1$, $c_2$ and turn. Initially, both $c_1$ and $c_2$ are 0 (*not busy*).

**Process 1**

```plaintext
... 
c1=1; 
turn = 1; 
L: if c2=1 & turn=1 
    then go to L 
    < critical section>
    c1=0;
```

**Process 2**

```plaintext
... 
c2=1; 
turn = 2; 
L: if c1=1 & turn=2 
    then go to L 
    < critical section>
    c2=0;
```

- $\text{turn} = i$ ensures that only process $i$ can wait
- Variables $c_1$ and $c_2$ ensure mutual exclusion

*Solution for $n$ processes was given by Dijkstra and is quite tricky!*

**Analysis of Dekker’s Algorithm**

**Scenario 1**

**Process 1**

```plaintext
... 
c1=1; 
turn = 1; 
L: if c2=1 & turn=1 
    then go to L 
    < critical section>
    c1=0;
```

**Process 2**

```plaintext
... 
c2=1; 
turn = 2; 
L: if c1=1 & turn=2 
    then go to L 
    < critical section>
    c2=0;
```

**Scenario 2**

**Process 1**

```plaintext
... 
c1=1; 
turn = 1; 
L: if c2=1 & turn=1 
    then go to L 
    < critical section>
    c1=0;
```

**Process 2**

```plaintext
... 
c2=1; 
turn = 2; 
L: if c1=1 & turn=2 
    then go to L 
    < critical section>
    c2=0;
```
N-process Mutual Exclusion  
Lamport’s Bakery Algorithm

Process $i$

Initially $\text{num}[j] = 0$, for all $j$

Entry Code

$\text{choosing}[i] = 1$;
$\text{num}[i] = \max(\text{num}[0], \ldots, \text{num}[N-1]) + 1$;
$\text{choosing}[i] = 0$;

for ($j = 0; j < N; j++$) {
    while ($\text{choosing}[j]$);
    while ($\text{num}[j] && ($ (\text{num}[j] < \text{num}[i]) || ($\text{num}[j] == \text{num}[i] && j < i))$);
}

$\text{num}[i] = 0$;

Exit Code

Locks or Semaphores
E. W. Dijkstra, 1965

A semaphore is a non-negative integer, with the following operations:

$P(s)$: if $s>0$, decrement $s$ by 1, otherwise wait

$V(s)$: increment $s$ by 1 and wake up one of the waiting processes

$P$'s and $V$'s must be executed atomically, i.e., without

• interruptions or
• interleaved accesses to $s$ by other processors

Initial value of $s$ determines the maximum no. of processes in the critical section
Implementation of Semaphores

Semaphores (mutual exclusion) can be implemented using ordinary Load and Store instructions in the Sequential Consistency memory model. However, protocols for mutual exclusion are difficult to design...

Simpler solution:

*atomic read-modify-write instructions*

Examples: *m* is a memory location, *R* is a register

```
Test&Set (m), R:
R ← M[m];
if R==0 then
    M[m] ← 1;
```

```
Fetch&Add (m), R, Rv, R:
R ← M[m];
M[m] ← R + Rv;
```

```
Swap (m), R:
Rt ← M[m];
M[m] ← R;
R ← Rt;
```

Multiple Consumers Example

*using the Test&Set Instruction*

```
P:
    Test&Set (mutex), R_
     if (R_
     goto P
    spin:
    Load R
     if R
     Load R
     spin:
    Load R, (R
    R
    Store (head), R
V:
    Store (mutex), 0
    process(R)
```

Other atomic read-modify-write instructions (Swap, Fetch&Add, etc.) can also implement *P*'s and *V*'s

*What if the process stops or is swapped out while in the critical section?*
Nonblocking Synchronization

\[
\text{Compare\&Swap}(m), \ R_t, \ R_s:
\begin{align*}
\text{if} \ (R_t &= M[m]) \\
\text{then} \ M[m] &= R_s; \\
R_s &= R_t; \\
\text{status} &\leftarrow \text{success}; \\
\text{else} \ &\text{status} \leftarrow \text{fail};
\end{align*}
\]

\[
\begin{align*}
\text{try:} \\
\text{spin:}
\end{align*}
\]

\[
\begin{align*}
\text{try:} \\
\text{spin:}
\end{align*}
\]

Load-reserve & Store-conditional

Special register(s) to hold reservation flag and address, and the outcome of store-conditional

\[
\begin{align*}
\text{try:} \\
\text{spin:}
\end{align*}
\]

\[
\begin{align*}
\text{try:} \\
\text{spin:}
\end{align*}
\]
**Performance of Locks**

Blocking atomic read-modify-write instructions  
e.g., *Test&Set, Fetch&Add, Swap*  
vs  
Non-blocking atomic read-modify-write instructions  
e.g., *Compare&Swap, Load-reserve/Store-conditional*  
vs  
Protocols based on ordinary Loads and Stores

*Performance depends on several interacting factors:*  
degree of contention,  
caches,  
out-of-order execution of Loads and Stores  

later ...

---

**Acknowledgements**

- These slides contain material developed and copyright by:  
  - Arvind (MIT)  
  - Krste Asanovic (MIT/UCB)  
  - Joel Emer (Intel/MIT)  
  - James Hoe (CMU)  
  - John Kubiatowicz (UCB)  
  - David Patterson (UCB)

- MIT material derived from course 6.823  
- UCB material derived from course CS252