Lecture 26: Multiprocessors

• Today’s topics:
  - Shared memory vs message-passing
  - Simultaneous multi-threading (SMT)
  - GPUs
Shared-Memory Vs. Message-Passing

Shared-memory:
- Well-understood programming model
- Communication is implicit and hardware handles protection
- Hardware-controlled caching

Message-passing:
- No cache coherence $\rightarrow$ simpler hardware
- Explicit communication $\rightarrow$ easier for the programmer to restructure code
- Software-controlled caching
- Sender can initiate data transfer
Procedure Solve(A)
begin
    diff = done = 0;
    while (!done) do
        diff = 0;
        for i ← 1 to n do
            for j ← 1 to n do
                temp = A[i,j];
                A[i,j] ← 0.2 * (A[i,j] + neighbors);
                diff += abs(A[i,j] – temp);
            end for
        end for
        if (diff < TOL) then done = 1;
    end while
end procedure
Shared Address Space Model

```c
int n, nprocs;
float **A, diff;
LOCKDEC(diff_lock);
BARDEC(bar1);

main()
begin
read(n); read(nprocs);
A ← G_MALLOC();
initialize (A);
CREATE (nprocs,Solve,A);
WAIT_FOR_END (nprocs);
end main

procedure Solve(A)
int i, j, pid, done=0;
float temp, mydiff=0;
int mymin = 1 + (pid * n/procs);
int mymax = mymin + n/nprocs -1;
while (!done) do
  mydiff = diff = 0;
  BARRIER(bar1,nprocs);
  for i ← mymin to mymax
    for j ← 1 to n do
      ...
    endfor
  endfor
  LOCK(diff_lock);
  diff += mydiff;
  UNLOCK(diff_lock);
  BARRIER (bar1, nprocs);
  if (diff < TOL) then done = 1;
  BARRIER (bar1, nprocs);
endwhile
```
main()
    read(n); read(nprocs);
    CREATE (nprocs-1, Solve);
    Solve();
    WAIT_FOR_END (nprocs-1);

procedure Solve()
    int i, j, pid, nn = n/nprocs, done=0;
    float temp, tempdiff, mydiff = 0;
    myA \leftarrow \text{malloc}(\ldots)
    initialize(myA);
    while (!done) do
        mydiff = 0;
        if (pid != 0)
            SEND(&myA[1,0], n, pid-1, ROW);
        if (pid != nprocs-1)
            SEND(&myA[nn,0], n, pid+1, ROW);
        if (pid != 0)
            RECEIVE(&myA[0,0], n, pid-1, ROW);
        if (pid != nprocs-1)
            RECEIVE(&myA[nn+1,0], n, pid+1, ROW);
        for i \leftarrow 1 to nn do
            for j \leftarrow 1 to n do
                \ldots
            endfor
        endfor
        if (pid != 0)
            SEND(mydiff, 1, 0, DIFF);
        RECEIVE(done, 1, 0, DONE);
        else
            for i \leftarrow 1 to nprocs-1 do
                RECEIVE(tempdiff, 1, *, DIFF);
                mydiff += tempdiff;
            endfor
            if (mydiff < TOL) done = 1;
        for i \leftarrow 1 to nprocs-1 do
            SEND(done, 1, I, DONE);
        endfor
        endif
    endwhile
Multithreading Within a Processor

• Until now, we have executed multiple threads of an application on different processors – can multiple threads execute concurrently on the same processor?

• Why is this desirable?
  ➢ inexpensive – one CPU, no external interconnects
  ➢ no remote or coherence misses (more capacity misses)

• Why does this make sense?
  ➢ most processors can’t find enough work – peak IPC is 6, average IPC is 1.5!
  ➢ threads can share resources → we can increase threads without a corresponding linear increase in area
How are Resources Shared?

Each box represents an issue slot for a functional unit. Peak thruput is 4 IPC.

<table>
<thead>
<tr>
<th>Cycles</th>
<th>Superscalar</th>
<th>Fine-Grained Multithreading</th>
<th>Simultaneous Multithreading</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 1</td>
<td>Red</td>
<td>Red</td>
<td>Red</td>
</tr>
<tr>
<td>Thread 2</td>
<td>Red</td>
<td>Red</td>
<td>Red</td>
</tr>
<tr>
<td>Thread 3</td>
<td>Red</td>
<td>Red</td>
<td>Red</td>
</tr>
<tr>
<td>Thread 4</td>
<td>Red</td>
<td>Red</td>
<td>Red</td>
</tr>
<tr>
<td>Idle</td>
<td>Yellow</td>
<td>Grey</td>
<td>Grey</td>
</tr>
</tbody>
</table>

- Superscalar processor has high under-utilization – not enough work every cycle, especially when there is a cache miss
- Fine-grained multithreading can only issue instructions from a single thread in a cycle – can not find max work every cycle, but cache misses can be tolerated
- Simultaneous multithreading can issue instructions from any thread every cycle – has the highest probability of finding work for every issue slot
Performance Implications of SMT

• Single thread performance is likely to go down (caches, branch predictors, registers, etc. are shared) – this effect can be mitigated by trying to prioritize one thread.

• With eight threads in a processor with many resources, SMT yields throughput improvements of roughly 2-4.
SIMD Processors

• Single instruction, multiple data

• Such processors offer energy efficiency because a single instruction fetch can trigger many data operations

• Such data parallelism may be useful for many image/sound and numerical applications
GPUs

- Initially developed as graphics accelerators; now viewed as one of the densest compute engines available

- Many on-going efforts to run non-graphics workloads on GPUs, i.e., use them as general-purpose GPUs or GPGPUs

- C/C++ based programming platforms enable wider use of GPGPUs – CUDA from NVidia and OpenCL from an industry consortium

- A heterogeneous system has a regular host CPU and a GPU that handles (say) CUDA code (they can both be on the same chip)
The GPU Architecture

- **SIMT** – single instruction, multiple thread; a GPU has many SIMT cores

- A large data-parallel operation is partitioned into many thread blocks (one per SIMT core); a thread block is partitioned into many warps (one warp running at a time in the SIMT core); a warp is partitioned across many in-order pipelines (each is called a SIMD lane)

- A SIMT core can have multiple active warps at a time, i.e., the SIMT core stores the registers for each warp; warps can be context-switched at low cost; a warp scheduler keeps track of runnable warps and schedules a new warp if the currently running warp stalls
The GPU Architecture
Architecture Features

- Simple in-order pipelines that rely on thread-level parallelism to hide long latencies

- Many registers (~1K) per in-order pipeline (lane) to support many active warps

- When a branch is encountered, some of the lanes proceed along the “then” case depending on their data values; later, the other lanes evaluate the “else” case; a branch cuts the data-level parallelism by half (branch divergence)

- When a load/store is encountered, the requests from all lanes are coalesced into a few 128B cache line requests; each request may return at a different time (mem divergence)
GPU Memory Hierarchy

- Each SIMT core has a private L1 cache (shared by the warps on that core)
- A large L2 is shared by all SIMT cores; each L2 bank services a subset of all addresses
- Each L2 partition is connected to its own memory controller and memory channel
- The GDDR5 memory system runs at higher frequencies, and uses chips with more banks, wide IO, and better power delivery networks
- A portion of GDDR5 memory is private to the GPU and the rest is accessible to the host CPU (the GPU performs copies)
Title

• Bullet