Lecture 2

Memory locality optimizations
Address space organization
Announcements

• Office hours in EBU3B Room 3244
  ✷ Mondays @ 4.30 pm (but next week only: 5.30pm)
  ✷ Thursdays at 4pm

• Programming Lab #1
  ✷ Uses bang
Today’s Lecture

• Memory hierarchies
• Address space organization
  ◆ Shared memory
  ◆ Distributed memory
• Control mechanisms
• Shared memory hierarchy
  ◆ Cache coherence
  ◆ False sharing
• HW #1
The processor-memory gap

- The result of technological trends
- Difference in processing and memory speeds growing exponentially over time
The processor-memory gap

- The result of technological trends
- Difference in processing and memory speeds growing exponentially over time

Moore’s Law

Processor-Memory Performance Gap:
(grows 50% / year)

- CPU: 60%/yr.
- DRAM: 7%/yr.
An important principle: locality

- Programs generally exhibit two forms of locality in accessing memory
  - Temporal locality (time)
  - Spatial locality (space)
- Often involves loops
- Opportunities for reuse

\[
\begin{align*}
&\text{for } t=0 \text{ to } T-1 \\
&\quad \text{for } i = 1 \text{ to } N-2 \\
&\quad \quad u[i] = (u[i-1] + u[i+1]) / 2
\end{align*}
\]
Memory hierarchies

- Exploit reuse through a hierarchy of smaller but faster memories
- Put things in faster memory if we reuse them frequently

![Diagram of memory hierarchy]

- CPU: 1CP (1 word)
- L1: 2-3 CP (10 to 100 B)
- L2: O(10) CP (10 - 100 B)
- DRAM: O(100) CP
- Disk: O(10^6) CP

- Memory sizes:
  - CPU: Many GB or TB
  - DRAM: 256KB to 4 MB
  - L2: 32 to 64 KB
  - L1: 2-3 CP (10 to 100 B)
  - CPU: 1CP (1 word)
Nehalem’s Memory Hierarchy

- **Source:** *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Table 2.7

<table>
<thead>
<tr>
<th></th>
<th>Latency (cycles)</th>
<th>Associativity</th>
<th>Line size (bytes)</th>
<th>Write update policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-inclusive 4</td>
<td>4</td>
<td>8</td>
<td>64</td>
<td>Writeback</td>
</tr>
<tr>
<td>Non-inclusive 10</td>
<td>10</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inclusive 35+</td>
<td></td>
<td>16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4MB for Gainestown

---

**realworldtech.com**
The 3 C’s of cache misses

- Cold Start
- Capacity
- Conflict
Managing locality with loop interchange

• The success of caching depends on the ability to **re-use** previously cached data
• Data access order affects re-use
• Assume a cache with 2 entries, each 2 words wide

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        a[i][j] += b[i][j];
```

```
for (j=0; j<N; j++)
    for (i=0; i<N; i++)
        a[i][j] += b[i][j];
```

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

**The 3 C’s**
- Cold Start
- Capacity
- Conflict
Testbed

- 2.7GHz Power PC G5 (970fx)
- Caches: 128 Byte line size
  - 512KB L2 (8-way, 12 CP hit time)
  - 32K L1 (2-way, 2 CP hit time)
- TLB: 1024 entries, 4-way
- gcc version 4.0.1 (Apple Computer, Inc. build 5370), -O2 optimization
- Single precision floating point
The results

for (i=0; i<N; i++)
for (j=0; j<N; j++)
    a[i][j] += b[i][j];

for (j=0; j<N; j++)
for (i=0; i<N; i++)
    a[i][j] += b[i][j];

<table>
<thead>
<tr>
<th>N</th>
<th>IJ (ms)</th>
<th>JI (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>0.007</td>
<td>0.007</td>
</tr>
<tr>
<td>128</td>
<td>0.027</td>
<td>0.083</td>
</tr>
<tr>
<td>512</td>
<td>1.1</td>
<td>37</td>
</tr>
<tr>
<td>1024</td>
<td>4.9</td>
<td>284</td>
</tr>
<tr>
<td>2048</td>
<td>18</td>
<td>2,090</td>
</tr>
</tbody>
</table>
Blocking for Cache
Matrix Multiplication

• An important core operation in many numerical algorithms
• Given two *conforming* matrices $A$ and $B$, form the matrix product $A \times B$
  - $A$ is $m \times n$
  - $B$ is $n \times p$
• Operation count: $O(n^3)$ multiply-adds for an $n \times n$ square matrix
• Discussion follows from Demmel
  [www.cs.berkeley.edu/~demmel/cs267_Spr99/Lectures/Lect02.html](http://www.cs.berkeley.edu/~demmel/cs267_Spr99/Lectures/Lect02.html)
Unblocked Matrix Multiplication

\[
\text{for } i := 0 \text{ to } n-1 \\
\quad \text{for } j := 0 \text{ to } n-1 \\
\quad \quad \text{for } k := 0 \text{ to } n-1 \\
\quad \quad \quad C[i,j] += A[i,k] \times B[k,j]
\]
Analysis of performance

for i = 0 to n-1
  // for each iteration i, load all of B into cache
  for j = 0 to n-1
    // for each iteration (i,j), load A[i,:] into cache
    // for each iteration (i,j), load and store C[i,j]
    for k = 0 to n-1
      C[i,j] += A[i,k] * B[k,j]
Analysis of performance

for $i = 0$ to $n-1$
    
    // $n \times n^2 / L$ loads $= \frac{n^3}{L}$, $L=$cache line size
    \[ B[:,:] \]

for $j = 0$ to $n-1$
    
    // $n^2 / L$ loads $= \frac{n^2}{L}$
    \[ A[i,:], C[i,j] \]

    // $n^2 / L$ loads + $n^2 / L$ stores $= \frac{2n^2}{L}$

for $k = 0$ to $n-1$
    
    $C[i,j] += A[i,k] \times B[k,j]$ \[ \text{Total:} (\frac{n^3 + 3n^2}{L}) \]
Flops to memory ratio

Let $q = \# \text{ flops} / \text{ main memory reference}$

$$q = \frac{2n^3}{n^3 + 3n^2}$$

$\approx 2$ as $n \to \infty$
Blocked Matrix Multiply

- Divide A, B, C into $N \times N$ sub blocks
- Assume we have a good quality library to perform matrix multiplication on subblocks
- Each sub block is $b \times b$
  - $b=n/N$ is called the block size
  - How do we establish $b$?

$$C[i,j] = C[i,j] + A[i,k] \times B[k,j]$$
Blocked Matrix Multiplication

for i = 0 to N-1
    for j = 0 to N-1
        // load each block C[i,j] into cache, once :
        // b = n/N = cache line size
        for k = 0 to N-1
            // load each block A[i,k] and B[k,j] N^3 times
            // = 2N^3 \times (n/N)^2 = 2Nn^2
            // write each block C[i,j] once :
        Total: (2*N+2)*n^2
Flops to memory ratio

Let \( q = \# \text{ flops} / \text{ main memory reference} \)

\[
q = \frac{2n^3}{(2N + 2)n^2} = \frac{n}{N + 1}
\]

\( \approx \frac{n}{N} = b \) as \( n \to \infty \)
The results

<table>
<thead>
<tr>
<th>N,B</th>
<th>Unblocked Time</th>
<th>Blocked Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>256, 64</td>
<td>0.6</td>
<td>0.002</td>
</tr>
<tr>
<td>512,128</td>
<td>15</td>
<td>0.24</td>
</tr>
</tbody>
</table>

Amortize memory accesses by increasing memory reuse
More on blocked algorithms

- Data in the sub-blocks are contiguous within rows only
- We may incur conflict cache misses
- Idea: since re-use is so high... let’s copy the subblocks into contiguous memory before passing to our matrix multiply routine

“The Cache Performance and Optimizations of Blocked Algorithms,”
M. Lam et al., ASPLOS IV, 1991

http://www-suif.stanford.edu/papers/lam91.ps
Programming Assignment #1

• Implement blocked matrix multiplication
• Compare with provided code
  - Naïve unblocked variant
  - Calls highly optimized matrix multiply, DGEMM, provided by ATLAS
  - 10% bonus if you can beat the ATLAS’ DGEMM
Address space organization
Control
Address Space Organization

• We classify the address space organization of a parallel computer according to whether or not it provides global memory

• When there is a global memory we have a “shared memory” or “shared address space” architecture
  - multiprocessor vs partitioned global address space

• Where there is no global memory, we have a “shared nothing” architecture, also known as a multicomputer
Multiprocessor organization

• The address space is global to all processors
• Hardware automatically performs the global to local mapping using address translation mechanisms
• Two types, according to the uniformity of memory access times

• **UMA**: Uniform Memory Access time
  - In the absence of contention all processors observe the same memory access time
  - Also called *Symmetric Multiprocessors*
  - Usually bus based
NUMA

• Non-Uniform Memory Access time
  - Processors see distant-dependent access times to memory
  - Implies physically distributed memory

• We often call these
  distributed shared memory architectures
  - Commercial example: SGI Origin Altix, up to 512 cores
  - Elaborate interconnect with a directory structure to monitor sharers
Architectures without shared memory

- Each processor has direct access to local memory only
- Send and receive messages to obtain copies of data from other processors
- We call this a *shared nothing* architecture, or a *multicomputer*
Hybrid organizations

- Multi-tier organizations are hierarchically organized
- Each node is a multiprocessor, usually and SMP
- Nodes communicate by passing messages, processors within a node communicate via shared memory
- All clusters and high end systems today

![Diagram of hybrid organization structure]
Today’s Lecture

- Memory hierarchies
- Address space organization
  - Shared memory
  - Distributed memory
- Control mechanisms
- Shared memory hierarchy
  - Cache coherence
  - False sharing
Control Mechanism

Flynn’s classification (1966)
How do the processors issue instructions?

**SIMD:** Single Instruction, Multiple Data
Execute a global instruction stream in lock-step

**MIMD:** Multiple Instruction, Multiple Data
Clusters and servers processors execute instruction streams independently
SIMD (Single Instruction Multiple Data)

- Operate on regular arrays of data
- Two landmark SIMD designs
  - ILIAC IV (1960s)
  - Connection Machine 1 and 2 (1980s)
- Vector computer: Cray-1 (1976)
- Intel and others support SIMD for multimedia and graphics
  - SSE
    - Streaming SIMD extensions, Altivec
  - Operations defined on vectors
- GPUs, Cell Broadband Engine
- Reduced performance on data dependent or irregular computations

\[
\begin{array}{c}
2 \\
4 \\
8 \\
7 \\
\end{array}
= \begin{array}{c}
1 \\
2 \\
3 \\
5 \\
\end{array}
+ \begin{array}{c}
1 \\
2 \\
5 \\
2 \\
\end{array}
\]

forall \( i = 0 : n-1 \)
\[
x[i] = y[i] + z[K[i]]
\]
end forall

forall \( i = 0 : n-1 \)
\[
\text{if } (x[i] < 0) \text{ then}
\[
y[i] = x[i]
\]
else
\[
y[i] = \sqrt{x[i]}
\]
end if
end forall
Today’s Lecture

• Memory hierarchies
• Address space organization
  ◆ Shared memory
  ◆ Distributed memory
• Control mechanisms
• Shared memory hierarchy
  ◆ Cache coherence
  ◆ False sharing
Cache Coherence

- A central design issue in shared memory architectures
- Processors may read and write the same cached memory location
- If one processor writes to the location, *all* others must eventually see the write

\[
X := 1
\]

Memory
Cache Coherence

- P1 & P2 load X from main memory into cache
- P1 stores 2 into X
- The memory system doesn’t have a coherent value for X
Cache Coherence Protocols

• Ensure that all processors *eventually* see the same value

• Two policies
  - Update-on-write (implies a write-through cache)
  - Invalidate-on-write

![Diagram of cache coherence protocols]

X:=2

P1

X:=2

Memory

P2

X:=2

X:=2

X:=2

37 ©2012 Scott B. Baden /CSE 260/ Fall 2012
SMP architectures

- Employ a *snooping protocol* to ensure coherence
- Processors listen to bus activity
Memory consistency and correctness

• Cache coherence tells us that memory will eventually be consistent
• The memory consistency policy tells us when this will happen
• Even if memory is consistent, changes don’t propagate instantaneously
• These give rise to correctness issues involving program behavior
Memory consistency model

• The memory consistency model determines when a written value will be seen by a reader

• **Sequential Consistency** maintains a linear execution on a parallel architecture that is consistent with the sequential execution of some interleaved arrangement of the separate concurrent instruction streams

• Expensive to implement

• **Relaxed consistency**
  ◆ Enforce consistency only at well defined times
  ◆ Useful in handling false sharing
False sharing

- Consider two processors that write to different locations mapping to different parts of the same cache line.
False sharing

- P0 writes a location
- Assuming we have a write-through cache, memory is updated
False sharing

- P1 reads the location written by P0
- P1 then writes a different location in the same block of memory
False sharing

- P1’s write updates main memory
- Snooping protocol invalidates the corresponding block in P0’s cache
False sharing

Successive writes by P0 and P1 cause the processors to uselessly invalidate one another’s cache
Eliminating false sharing

• Cleanly separate locations updated by different processors
  ♦ Manually assign scalars to a pre-allocated region of memory using pointers
  ♦ Spread out the values to coincide with a cache line boundaries

```c
#pragma omp parallel for
for (i=0; i<N; i++)
  sum[i]++;
```