Lecture 2

Memory locality optimizations
Address space organization
Announcements

• Programming Lab #1 will be posted on Friday at noon
• Be sure you can login to Bang and Moodle
• Submit NERSC form
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
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

for t=0 to T-1
  for i = 1 to N-2
    \[ u[i] = \frac{u[i-1] + u[i+1]}{2} \]
Memory hierarchies

• Exploit reuse through a hierarchy of smaller but faster memories
• Put things in faster memory if we reuse them frequently
Bang’s Memory Hierarchy

- Intel “Clovertown” processor
- Intel Xeon E5355 (Introduced: 2006)
- Two “Woodcrest” dies (Core2) on a multichip module
- Two “sockets”
- *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Tab 2.16

Access latency, throughput (clocks)

- 3, 1
- 14*, 2

* Software-visible latency will vary depending on access patterns and other factors

**Line Size = 64B (L1 and L2)**

**Associativity**

- 8
- 16

**Write update policy:**

- Writeback

Sam Williams et al.

©2014 Scott B. Baden /CSE 260/ Winter '14
Examining Bang’s Memory Hierarchy

- `/proc/cpuinfo` summarizes the processor
  - `vendor_id`: GenuineIntel
  - `model name`: Intel(R) Xeon(R) CPU E5345 @2.33GHz
  - `cache size`: 4096 KB
  - `cpu cores`: 4
- `processor`: 0 through 7
- Detailed information at `/sys/devices/system/cpu/cpu*/cache/index*/`
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

\[
\begin{align*}
&\text{for (i=0; i<N; i++)} & \text{for (j=0; j<N; j++)} \\
&\text{for (j=0; j<N; j++)} & \text{for (i=0; i<N; i++)} \\
&\quad a[i][j] += b[i][j]; & \quad a[i][j] += b[i][j]; \\
\end{align*}
\]

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
4 & 5 & 6 & 7 \\
8 & 9 & 10 & 11 \\
12 & 13 & 14 & 15 \\
\end{array}
\]

The 3 C’s
Cold Start
Capacity
Conflict

©2014 Scott B. Baden / CSE 260/ Winter '14
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>

©2014 Scott B. Baden / CSE 260/ Winter '14
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

[Website Link](http://www.cs.berkeley.edu/~demmel/cs267_Spr99/Lectures/Lect02.html)
Unblocked Matrix Multiplication

\[
\begin{align*}
\text{for } i & := 0 \text{ to } n-1 \\
\text{for } j & := 0 \text{ to } n-1 \\
\text{for } k & := 0 \text{ to } n-1 \\
C[i,j] & += A[i,k] \times B[k,j]
\end{align*}
\]
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 × n^2 / L loads = n^3/L, L=cache line size
  B[:,i]

for j = 0 to n-1
  // n^2 / L loads = n^2/L
  A[i,:]

  // n^2 / L loads + n^2 / L stores = 2n^2 / L
  C[i,j]

for k = 0 to n-1
  C[i,j] += A[i,k] * B[k,j]

Total: (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 \( N \times N \) subblocks
• 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 \( b \)?

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

Matrix Multiplication Performance

8.14 GFlops

$R_{\infty} = 4 \times 2.33 = 9.32$ Gflops

$\sim 87\%$ of peak
Programming Assignment #1

• Implement high performance matrix multiplication

• Provided code
  ♦ Single level blocked matrix multiplication code
  ♦ Reference: calls highly optimized matrix multiply,, provided by ATLAS

• Further optimization required
  ♦ 2 level blocking
  ♦ SSE (vectorization, next week)
  ♦ Compiler optimizations
  ♦ Other hand optimizations may be needed

• Will post the results
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
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

```plaintext
forall i = 0 : n-1
  if ( x[i] < 0)  then
    y[i] = x[i]
  else
    y[i] = √x[i]
  end if
end forall

forall i = 0 : n-1
  x[i] = y[i] + z [ K[i] ]
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
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
- Reduce number of accesses to shared state
- False sharing occurs a small fixed number of times

```java
static int counts[];
for (int k = 0; k<reps; k++)
    for (int r = first; r <= last; ++ r)
        if ((values[r] % 2) == 1)
            counts[TID]++;
```

```java
int _count = 0;
for (int k = 0; k<reps; k++){
    for (int r = first; r <= last; ++ r)
        if ((values[r] % 2) == 1)
            _count++;
    counts[TID] = _count;
}
```

4.7s, 6.3s, 7.9s, 10.4 [NT=1,2,4,8] 3.4s, 1.7s, 0.83, 0.43 [NT=1,2,4,8]
• Put each counter in its own cache line

**Spreading**

```java
static int counts[];
for (int k = 0; k<reps; k++)
    for (int r = first; r <= last; ++ r)
        if ((values[r] % 2) == 1)
            counts[TID]++;

static int counts[][LINE_SIZE];
for (int k = 0; k<reps; k++)
    for (int r = first; r <= last; ++ r)
        if ((values[r] % 2) == 1)
            counts[TID][0]++;
```
Cache performance bottlenecks in nearest neighbor computations

- Recall the image smoothing algorithm

\[
\text{for } (i,j) \text{ in } 0: N-1 \times 0: N-1 \\
I^{\text{new}} [i,j] = \left( I[i-1,j] + I[i+1,j] + I[i,j-1] + I[i, j+1] \right) / 4
\]
Memory access pattern

- Some nearest neighbors in space are far apart in memory
- Stride = N along the vertical dimension

\[
\text{for } (i,j) \text{ in } 0:N-1 \times 0:N-1 \\
I_{\text{new}}[i,j] = \frac{I[i-1,j] + I[i+1,j] + I[i,j-1] + I[i,j+1]}{4}
\]
False sharing and conflict misses

- False sharing involves internal boundaries, poor spatial locality, cache line internally fragmented
- Large memory access strides: conflict misses, poor cache locality
- Even worse in 3D: large strides of $N^2$
- Contiguous access on a single processor

On a single processor

On multiple processors

Cache block straddles boundary

Parallel Computer Architecture, Culler, Singh, & Gupta

©2014 Scott B. Baden /CSE 260/ Winter '14