Lecture 2
CSE 260 – Parallel Computation
(Fall 2015)
Scott B. Baden

Memory hierarchies
Memory locality optimizations
Announcements

• Wait list
• Can you login to Bang and Moodle?
• Did you submit the XSEDE form?
• Office hours
• Programming Lab #1
Summary from last time

- Parallelism is inevitable owing to technological factors limiting clock speeds
- Increased parallelism on chip, computation is becoming cheaper
- But data motion is becoming more costly relative to computation
- Resultant disruption has a huge impact on how we write high quality applications
I increased performance – so what’s the catch?

• A well behaved single processor algorithm may behave poorly on a parallel computer, and may need to be reformulated numerically.

• There currently exists no tool that can convert a serial program into an efficient parallel program … for all applications … all of the time… on all hardware.

• Many Performance programming issues:
  ‣ The new code may look dramatically different from the original.
Application-specific knowledge is important

- The **more we know** about the application...
  
  ... specific problem ... math/physics ... initial data ...  
  ... context for analyzing the output... 
  ... **the more we can** improve performance

- Particularly challenging for **irregular problems**
- Domain-specific knowledge is important in optimizing performance, especially locality
- Parallelism introduces many new tradeoffs
  - Redesign the software
  - Rethink the problem solving technique
Today’s Lecture

• Memory hierarchies
• Managing memory locality
• HW #1
The processor-memory gap

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

http://www.extremetech.com
(see http://tinyurl.com/ohmbo7y)
An important principle: locality

- Memory accesses generally come from nearby addresses:
  - In space (spatial locality)
  - In Time (temporal locality)
- Often involves loops
- Opportunities for reuse
- What are examples of non-localized accesses?

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

- Ruse data (and instructions) using 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.

Scott B. Baden / CSE 260, UCSD / Fall '15
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 `processor`: 7
• Detailed information at
  `/sys/devices/system/cpu/cpu/*/cache/index/*/`
Stampede’s Sandy Bridge Memory Hierarchy

- `/proc/cpuinfo` summarizes the processor
  - `vendor_id` : GenuineIntel
  - `model name` : Intel®Xeon®CPU E5-2680 0 @ 2.70GHz
  - `cache size` : 20480 KB
  - `cpu cores` : 8
- `processor` : 0 through `processor` : 16
- Detailed information at
  `/sys/devices/system/cpu/cpu*/cache/index*/`

```
<table>
<thead>
<tr>
<th>Level</th>
<th>Size</th>
<th>Latency</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>32 KB</td>
<td>4 Cycles</td>
<td>40-100 GB/s</td>
</tr>
<tr>
<td>L1 Data</td>
<td>256 KB</td>
<td>12 Cycles</td>
<td>30-60 GB/s</td>
</tr>
<tr>
<td>L2</td>
<td></td>
<td>26-31 Cycles</td>
<td>20-40 GB/s</td>
</tr>
<tr>
<td>L3 shared</td>
<td></td>
<td>~175-350 Cycles</td>
<td>4-10 GB/s</td>
</tr>
<tr>
<td>External Memory</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

www.cac.cornell.edu/Stampede/CodeOptimization/multicore.aspx
The 3 C’s of cache misses

- Cold Start
- Capacity
- Conflict
Today’s Lecture

- Memory hierarchies
- Managing memory locality
- HW #1
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
Performance of Matrix Multiply

\[ R_\infty = 4 \times 2.33 = 9.32 \text{ Gflops} \]

\( \approx 87\% \) of peak
Unblocked Matrix Multiplication

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

\[
C(i,j) = \sum_k A(i,k) \times B(k,j)
\]
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 = n^3/L, L=cache line size    B[:,,:]
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
Computational Intensity

• Performance is limited by the ratio of computational done and the volume of data moved between processor and main memory

• We call this ratio the *computational intensity*, or $q$; it is intrinsic to the application

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

$\approx 2$ as $n \to \infty$
Blocking for cache: motivation

- To increase performance, we need to move less data
- Assume a 2 level memory hierarchy, fast and slow
- Minimum running time $= f \times t_f$ when all data is in fastest memory, $f = \#$ arithmetic ops, $t_f = \text{time} / \text{flop}$
- Actual time
  \[
  f \times t_f + m \times t_m = f \times t_f \times (1 + \frac{1}{(t_f/t_m)} \times \frac{1}{q})
  \]
  \[
  m = \# \text{ words transferred}
  \]
  \[
  t_m = \text{slow memory access time}
  \]
  \[
  q = f/m = \text{computational intensity}
  \]
  \[
  t_f/t_m \quad \text{Establishes machine balance}
  \]
  \[
  \text{“Hardware computational intensity”}
  \]
**Blocked Matrix Multiply**

- Divide $A$, $B$, $C$ into $N \times N$ sub-blocks ($b \times b$)
- All 3 blocks must fit into cache
- Assume we have a good quality library to perform matrix multiplication on subblocks

\[ \Theta(M^{1/2}) \]
Blocked Matrix Multiplication

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
            // = 2N^3 \times (n/N)^2 = 2Nn^2
            // write each block C[i,j] once : $n^2$

Total: $(2*N+2)*n^2$
Lower Bound on Performance

- If $M_{\text{fast}} = \text{size of fast memory}$
  
  $$3b^2 \leq M_{\text{fast}} \Rightarrow b \leq \left( \frac{M_{\text{fast}}}{3} \right)^{1/2}$$

- To run at half peak speed
  
  $$M_{\text{fast}} \geq 3b^2 = 3(t_m/t_f)^2$$

- We change the order of arithmetic, so slightly different answers due to roundoff

- **Lower bound Theorem (Hong & Kung, 1981):**
  Any reorganization of this algorithm (that uses only associativity) is limited to
  
  $$O\left( \left( M_{\text{fast}} \right)^{1/2} \right)$$

  \#words moved between fast and slow memory $= \Omega \left( \frac{n^3}{\left( M_{\text{fast}} \right)^{1/2}} \right)$

- From previous slide: $(2N+2) \cdot n^2$, $N = n/b$
  
  $$\approx \frac{2N^3}{b}$$
Computational Intensity

• Blocking raises the computational intensity by reducing the number of reads from main memory

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

\[ \approx \frac{n}{N} = b \]

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
The roofline model

- As we’ve seen, the time spent in matrix multiplication includes both data motion time as well as the time spent computing.
- In a modern processor, we may be able to perform both at the same time.
- Thus, the total running time is
- We can vary q (and the computation)

\[
= \max \begin{cases} 
\text{Computation } (f \times t_f) \\
q^{-1} \times \text{Main memory bandwidth } (BW = t_m^{-1})
\end{cases}
\]
Roofline model

- Peak performance intrinsic to the processor, theoretical limit based on clock speed
- Lower rooflines correspond to successively degraded performance as we remove optimizations
- If our hardware as a multiplier and an adder, but algorithm can use only adders, then we can do no better than $\frac{1}{2}$ peak
- Similarly with SIMD parallelism and ILP
Roofline model – impact of locality

- In matrix multiply, we only count compulsory misses
- But conflict misses can occur as well

![Graph showing Roofline model](image)

- Column of matrix is stored in red cache lines

Sam Williams

Scott B. Baden / CSE 260, UCSD / Fall '15
Avoiding conflict misses

• Copying and cache friendly layouts

Row major

<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>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
</tbody>
</table>

Reorganized into 2x2 blocks

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

Column of matrix is stored in red cache lines

Larry Carter
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
  ♦ Hierarchical blocking (at least 2 levels of cache, TLB, too)
  ♦ Cache friendly layouts
  ♦ SSE (vectorization, next time)
  ♦ Compiler optimizations
  ♦ Other hand optimizations may be needed

• See www.cs.berkeley.edu/~demmel/cs267_Spr12/Lectures/lecture02_memhier_jwd12.ppt