Caches Part 2

CMPU 224 – Computer Organization
Jason Waterman
General Cache Organization (S, E, B)

- E = 2^e lines per set
- S = 2^s sets
- B = 2^b bytes per cache block (the data)

Cache size:
\[ C = S \times E \times B \text{ data bytes} \]
Cache Read

$E = 2^e$ lines per set

$S = 2^s$ sets

$B = 2^b$ bytes per cache block (the data)

1. Locate set
2. Check if any line in set has matching tag
3. Yes + line valid: hit
4. Locate data starting at offset

Address of word:

- $t$ bits (tag)
- $s$ bits (set index)
- $b$ bits (block offset)

valid bit

data begins at this offset
Cache Lookup Practice

- Memory is byte addressable
- Addresses are 12 bits wide, cache is 4-way set associative, with a 4 byte block size, and 8 total lines
- Address: 0xE34
  - Fill out the tables below

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>Valid</th>
<th>Byte0</th>
<th>Byte1</th>
<th>Byte2</th>
<th>Byte3</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>058</td>
<td>0</td>
<td>02</td>
<td>55</td>
<td>AD</td>
<td>87</td>
</tr>
<tr>
<td>S0</td>
<td>123</td>
<td>1</td>
<td>3E</td>
<td>98</td>
<td>47</td>
<td>51</td>
</tr>
<tr>
<td>S0</td>
<td>12B</td>
<td>0</td>
<td>6C</td>
<td>77</td>
<td>89</td>
<td>14</td>
</tr>
<tr>
<td>S0</td>
<td>0EF</td>
<td>1</td>
<td>B9</td>
<td>64</td>
<td>78</td>
<td>25</td>
</tr>
<tr>
<td>S1</td>
<td>069</td>
<td>0</td>
<td>00</td>
<td>FF</td>
<td>14</td>
<td>43</td>
</tr>
<tr>
<td>S1</td>
<td>12B</td>
<td>1</td>
<td>92</td>
<td>63</td>
<td>42</td>
<td>21</td>
</tr>
<tr>
<td>S1</td>
<td>075</td>
<td>0</td>
<td>33</td>
<td>BE</td>
<td>AF</td>
<td>31</td>
</tr>
<tr>
<td>S1</td>
<td>1C6</td>
<td>1</td>
<td>22</td>
<td>17</td>
<td>02</td>
<td>24</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block offset</td>
<td></td>
</tr>
<tr>
<td>Set Index</td>
<td></td>
</tr>
<tr>
<td>Cache Tag</td>
<td></td>
</tr>
<tr>
<td>Cache Hit? (Y/N)</td>
<td></td>
</tr>
<tr>
<td>Cache Byte returned</td>
<td></td>
</tr>
</tbody>
</table>
Cache Lookup Practice

- Memory is byte addressable
- Addresses are 12 bits wide, cache is 4-way set associative, with a 4 byte block size, and 8 total lines
- Address: 0xE34
  - Fill out the tables below

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>Valid</th>
<th>Byte0</th>
<th>Byte1</th>
<th>Byte2</th>
<th>Byte3</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_0</td>
<td>058</td>
<td>0</td>
<td>02</td>
<td>55</td>
<td>AD</td>
<td>87</td>
</tr>
<tr>
<td>S_0</td>
<td>123</td>
<td>1</td>
<td>3E</td>
<td>98</td>
<td>47</td>
<td>51</td>
</tr>
<tr>
<td>S_0</td>
<td>12B</td>
<td>0</td>
<td>6C</td>
<td>77</td>
<td>89</td>
<td>14</td>
</tr>
<tr>
<td>S_0</td>
<td>0EF</td>
<td>1</td>
<td>B9</td>
<td>64</td>
<td>78</td>
<td>25</td>
</tr>
<tr>
<td>S_1</td>
<td>069</td>
<td>0</td>
<td>00</td>
<td>FF</td>
<td>14</td>
<td>43</td>
</tr>
<tr>
<td>S_1</td>
<td>12B</td>
<td>1</td>
<td>92</td>
<td>63</td>
<td>42</td>
<td>21</td>
</tr>
<tr>
<td>S_1</td>
<td>075</td>
<td>0</td>
<td>33</td>
<td>BE</td>
<td>AF</td>
<td>31</td>
</tr>
<tr>
<td>S_1</td>
<td>1C6</td>
<td>1</td>
<td>22</td>
<td>17</td>
<td>02</td>
<td>24</td>
</tr>
</tbody>
</table>

```
  11  10  9  8  7  6  5  4  3  2  1  0
  |   |   | O | O | O |   |   | O |   |   |   | O |
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block offset</td>
<td></td>
</tr>
<tr>
<td>Set Index</td>
<td></td>
</tr>
<tr>
<td>Cache Tag</td>
<td></td>
</tr>
<tr>
<td>Cache Hit? (Y/N)</td>
<td></td>
</tr>
<tr>
<td>Cache Byte returned</td>
<td></td>
</tr>
</tbody>
</table>
Cache Lookup Practice

- Memory is byte addressable
- Addresses are 12 bits wide, cache is 4-way set associative, with a 4 byte block size, and 8 total lines
- Address: 0xE34
  - Fill out the tables below

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>Valid</th>
<th>Byte0</th>
<th>Byte1</th>
<th>Byte2</th>
<th>Byte3</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>058</td>
<td>0</td>
<td>02</td>
<td>55</td>
<td>AD</td>
<td>87</td>
</tr>
<tr>
<td>S0</td>
<td>123</td>
<td>1</td>
<td>3E</td>
<td>98</td>
<td>47</td>
<td>51</td>
</tr>
<tr>
<td>S0</td>
<td>12B</td>
<td>0</td>
<td>6C</td>
<td>77</td>
<td>89</td>
<td>14</td>
</tr>
<tr>
<td>S0</td>
<td>0EF</td>
<td>1</td>
<td>B9</td>
<td>64</td>
<td>78</td>
<td>25</td>
</tr>
<tr>
<td>S1</td>
<td>069</td>
<td>0</td>
<td>00</td>
<td>FF</td>
<td>14</td>
<td>43</td>
</tr>
<tr>
<td>S1</td>
<td>12B</td>
<td>1</td>
<td>92</td>
<td>63</td>
<td>42</td>
<td>21</td>
</tr>
<tr>
<td>S1</td>
<td>075</td>
<td>0</td>
<td>33</td>
<td>BE</td>
<td>AF</td>
<td>31</td>
</tr>
<tr>
<td>S1</td>
<td>1C6</td>
<td>1</td>
<td>22</td>
<td>17</td>
<td>02</td>
<td>24</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block offset</td>
<td></td>
</tr>
<tr>
<td>Set Index</td>
<td></td>
</tr>
<tr>
<td>Cache Tag</td>
<td></td>
</tr>
<tr>
<td>Cache Hit? (Y/N)</td>
<td></td>
</tr>
<tr>
<td>Cache Byte returned</td>
<td></td>
</tr>
</tbody>
</table>
Cache Lookup Practice

- Memory is byte addressable
- Addresses are 12 bits wide, cache is 4-way set associative, with a 4-byte block size, and 8 total lines
- Address: 0xE34
  - Fill out the tables below

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>Valid</th>
<th>Byte0</th>
<th>Byte1</th>
<th>Byte2</th>
<th>Byte3</th>
</tr>
</thead>
<tbody>
<tr>
<td>S₀</td>
<td>058</td>
<td>0</td>
<td>02</td>
<td>55</td>
<td>AD</td>
<td>87</td>
</tr>
<tr>
<td>S₀</td>
<td>123</td>
<td>1</td>
<td>3E</td>
<td>98</td>
<td>47</td>
<td>51</td>
</tr>
<tr>
<td>S₀</td>
<td>12B</td>
<td>0</td>
<td>6C</td>
<td>77</td>
<td>89</td>
<td>14</td>
</tr>
<tr>
<td>S₀</td>
<td>0EF</td>
<td>1</td>
<td>B9</td>
<td>64</td>
<td>78</td>
<td>25</td>
</tr>
<tr>
<td>S₁</td>
<td>069</td>
<td>0</td>
<td>00</td>
<td>FF</td>
<td>14</td>
<td>43</td>
</tr>
<tr>
<td>S₁</td>
<td>12B</td>
<td>1</td>
<td>92</td>
<td>63</td>
<td>42</td>
<td>21</td>
</tr>
<tr>
<td>S₁</td>
<td>075</td>
<td>0</td>
<td>33</td>
<td>BE</td>
<td>AF</td>
<td>31</td>
</tr>
<tr>
<td>S₁</td>
<td>0C6</td>
<td>1</td>
<td>22</td>
<td>17</td>
<td>02</td>
<td>24</td>
</tr>
</tbody>
</table>

Parameter | Value
---|---
Block offset |
Set Index |
Cache Tag |
Cache Hit? (Y/N) |
Cache Byte returned |
Cache Lookup Practice

- Memory is byte addressable
- Addresses are 12 bits wide, cache is 4-way set associative, with a 4 byte block size, and 8 total lines
- Address: 0xE34
  - Fill out the tables below

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>Valid</th>
<th>Byte0</th>
<th>Byte1</th>
<th>Byte2</th>
<th>Byte3</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>058</td>
<td>0</td>
<td>02</td>
<td>55</td>
<td>AD</td>
<td>87</td>
</tr>
<tr>
<td>S0</td>
<td>123</td>
<td>1</td>
<td>3E</td>
<td>98</td>
<td>47</td>
<td>51</td>
</tr>
<tr>
<td>S0</td>
<td>12B</td>
<td>0</td>
<td>6C</td>
<td>77</td>
<td>89</td>
<td>14</td>
</tr>
<tr>
<td>S0</td>
<td>0EF</td>
<td>1</td>
<td>B9</td>
<td>64</td>
<td>78</td>
<td>25</td>
</tr>
<tr>
<td>S1</td>
<td>069</td>
<td>0</td>
<td>00</td>
<td>FF</td>
<td>14</td>
<td>43</td>
</tr>
<tr>
<td>S1</td>
<td>12B</td>
<td>1</td>
<td>92</td>
<td>63</td>
<td>42</td>
<td>21</td>
</tr>
<tr>
<td>S1</td>
<td>075</td>
<td>0</td>
<td>33</td>
<td>BE</td>
<td>AF</td>
<td>31</td>
</tr>
<tr>
<td>S1</td>
<td>1C6</td>
<td>1</td>
<td>22</td>
<td>17</td>
<td>02</td>
<td>24</td>
</tr>
</tbody>
</table>

Parameter | Value
--- | ---
Block offset | 
Set Index | 
Cache Tag | 
Cache Hit? (Y/N) | 
Cache Byte returned |
Cache Lookup Practice

- Memory is byte addressable
- Addresses are 12 bits wide, cache is 4-way set associative, with a 4 byte block size, and 8 total lines
- Address: 0xE34
  - Fill out the tables below

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>Valid</th>
<th>Byte0</th>
<th>Byte1</th>
<th>Byte2</th>
<th>Byte3</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>058</td>
<td>0</td>
<td>02</td>
<td>55</td>
<td>AD</td>
<td>87</td>
</tr>
<tr>
<td>S0</td>
<td>123</td>
<td>1</td>
<td>3E</td>
<td>98</td>
<td>47</td>
<td>51</td>
</tr>
<tr>
<td>S0</td>
<td>12B</td>
<td>0</td>
<td>6C</td>
<td>77</td>
<td>89</td>
<td>14</td>
</tr>
<tr>
<td>S0</td>
<td>0EF</td>
<td>1</td>
<td>B9</td>
<td>64</td>
<td>78</td>
<td>25</td>
</tr>
<tr>
<td>S1</td>
<td>069</td>
<td>0</td>
<td>00</td>
<td>FF</td>
<td>14</td>
<td>43</td>
</tr>
<tr>
<td>S1</td>
<td>12B</td>
<td>1</td>
<td>92</td>
<td>63</td>
<td>42</td>
<td>21</td>
</tr>
<tr>
<td>S1</td>
<td>075</td>
<td>0</td>
<td>33</td>
<td>BE</td>
<td>AF</td>
<td>31</td>
</tr>
<tr>
<td>S1</td>
<td>1C6</td>
<td>1</td>
<td>22</td>
<td>17</td>
<td>02</td>
<td>24</td>
</tr>
</tbody>
</table>

Parameter | Value
--- | ---
Block offset | 0
Set Index | 1
Cache Tag | 0x1C6
Cache Hit? (Y/N) | Y
Cache Byte returned | 0x22
Cache Lookup Practice

- Memory is byte addressable
- Addresses are 12 bits wide, cache is 4-way set associative, with a 4 byte block size, and 8 total lines
- Address: 0x95B
  - Fill out the tables below

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>Valid</th>
<th>Byte0</th>
<th>Byte1</th>
<th>Byte2</th>
<th>Byte3</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>058</td>
<td>0</td>
<td>02</td>
<td>55</td>
<td>AD</td>
<td>87</td>
</tr>
<tr>
<td>S0</td>
<td>123</td>
<td>1</td>
<td>3E</td>
<td>98</td>
<td>47</td>
<td>51</td>
</tr>
<tr>
<td>S0</td>
<td>12B</td>
<td>0</td>
<td>6C</td>
<td>77</td>
<td>89</td>
<td>14</td>
</tr>
<tr>
<td>S0</td>
<td>0EF</td>
<td>1</td>
<td>B9</td>
<td>64</td>
<td>78</td>
<td>25</td>
</tr>
<tr>
<td>S1</td>
<td>069</td>
<td>0</td>
<td>00</td>
<td>FF</td>
<td>14</td>
<td>43</td>
</tr>
<tr>
<td>S1</td>
<td>12B</td>
<td>1</td>
<td>92</td>
<td>63</td>
<td>42</td>
<td>21</td>
</tr>
<tr>
<td>S1</td>
<td>075</td>
<td>0</td>
<td>33</td>
<td>BE</td>
<td>AF</td>
<td>31</td>
</tr>
<tr>
<td>S1</td>
<td>1C6</td>
<td>1</td>
<td>22</td>
<td>17</td>
<td>02</td>
<td>24</td>
</tr>
</tbody>
</table>

Parameter | Value
--- | ---
Block offset | 
Set Index | 
Cache Tag | 
Cache Hit? (Y/N) | 
Cache Byte returned |
Cache Lookup Practice

- Memory is byte addressable
- Addresses are 12 bits wide, cache is 4-way set associative, with a 4 byte block size, and 8 total lines
- Address: 0x95B
  - Fill out the tables below

### Set, Tag, Valid, Byte0, Byte1, Byte2, Byte3

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>Valid</th>
<th>Byte0</th>
<th>Byte1</th>
<th>Byte2</th>
<th>Byte3</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>058</td>
<td>0</td>
<td>02</td>
<td>55</td>
<td>AD</td>
<td>87</td>
</tr>
<tr>
<td>S0</td>
<td>123</td>
<td>1</td>
<td>3E</td>
<td>98</td>
<td>47</td>
<td>51</td>
</tr>
<tr>
<td>S0</td>
<td>12B</td>
<td>0</td>
<td>6C</td>
<td>77</td>
<td>89</td>
<td>14</td>
</tr>
<tr>
<td>S0</td>
<td>0EF</td>
<td>1</td>
<td>B9</td>
<td>64</td>
<td>78</td>
<td>25</td>
</tr>
<tr>
<td>S1</td>
<td>069</td>
<td>0</td>
<td>00</td>
<td>FF</td>
<td>14</td>
<td>43</td>
</tr>
<tr>
<td>S1</td>
<td>12B</td>
<td>1</td>
<td>92</td>
<td>63</td>
<td>42</td>
<td>21</td>
</tr>
<tr>
<td>S1</td>
<td>075</td>
<td>0</td>
<td>33</td>
<td>BE</td>
<td>AF</td>
<td>31</td>
</tr>
<tr>
<td>S1</td>
<td>1C6</td>
<td>1</td>
<td>22</td>
<td>17</td>
<td>02</td>
<td>24</td>
</tr>
</tbody>
</table>

### Parameter, Value

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block offset</td>
<td></td>
</tr>
<tr>
<td>Set Index</td>
<td></td>
</tr>
<tr>
<td>Cache Tag</td>
<td></td>
</tr>
<tr>
<td>Cache Hit? (Y/N)</td>
<td></td>
</tr>
<tr>
<td>Cache Byte returned</td>
<td></td>
</tr>
</tbody>
</table>
Cache Lookup Practice

- Memory is byte addressable
- Addresses are 12 bits wide, cache is 4-way set associative, with a 4 byte block size, and 8 total lines
- Address: 0x95B
  - Fill out the tables below

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>Valid</th>
<th>Byte0</th>
<th>Byte1</th>
<th>Byte2</th>
<th>Byte3</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>058</td>
<td>0</td>
<td>02</td>
<td>55</td>
<td>AD</td>
<td>87</td>
</tr>
<tr>
<td>S0</td>
<td>123</td>
<td>1</td>
<td>3E</td>
<td>98</td>
<td>47</td>
<td>51</td>
</tr>
<tr>
<td>S0</td>
<td>12B</td>
<td>0</td>
<td>6C</td>
<td>77</td>
<td>89</td>
<td>14</td>
</tr>
<tr>
<td>S0</td>
<td>0EF</td>
<td>1</td>
<td>B9</td>
<td>64</td>
<td>78</td>
<td>25</td>
</tr>
<tr>
<td>S1</td>
<td>069</td>
<td>0</td>
<td>00</td>
<td>FF</td>
<td>14</td>
<td>43</td>
</tr>
<tr>
<td>S1</td>
<td>12B</td>
<td>1</td>
<td>92</td>
<td>63</td>
<td>42</td>
<td>21</td>
</tr>
<tr>
<td>S1</td>
<td>075</td>
<td>0</td>
<td>33</td>
<td>BE</td>
<td>AF</td>
<td>31</td>
</tr>
<tr>
<td>S1</td>
<td>1C6</td>
<td>1</td>
<td>22</td>
<td>17</td>
<td>02</td>
<td>24</td>
</tr>
</tbody>
</table>

- Parameter | Value
  - Block offset | 3
  - Set Index | 0
  - Cache Tag | 0x12B
  - Cache Hit? (Y/N) | N
  - Cache Byte returned | }
Intel Core i7 Cache Hierarchy

Processor package

Core 0
- L1 d-cache
- L1 i-cache
- L2 unified cache

Core 3
- L1 d-cache
- L1 i-cache
- L2 unified cache

... (shared by all cores)

L3 unified cache

Main memory

L1 i-cache and d-cache:
- 32 KB, 8-way,
- Access: 4 cycles

L2 unified cache:
- 256 KB, 8-way,
- Access: 10 cycles

L3 unified cache:
- 8 MB, 16-way,
- Access: 40-75 cycles

Block size: 64 bytes for all caches.
Today

- Cache performance metrics
- The graph on the cover of your textbook explained
- Writing cache friendly code
What about writes?

- Multiple copies of the data exist:
  - Cache and Main Memory
- What to do on a write-hit?
  - Update cache block with new contents
  - **Write-through** (write immediately to memory)
  - **Write-back** (defer write to memory until line is evicted)
    - Need a dirty bit (whether line is different from memory or not)
- What to do on a write-miss?
  - **No-write-allocate** (writes straight to memory, does not load into cache)
  - **Write-allocate** (load into cache, update line in cache)
    - Good if more writes to the location follow
- Typical Pairings
  - Write-through + No-write-allocate
    - Simpler
  - **Write-back + Write-allocate**
    - Better performance
Types of Cache Misses

• Cold (compulsory) miss
  • Cold misses occur because the cache is empty

• Conflict miss
  • Conflict misses occur when the cache is large enough, but multiple data objects all map to the same set in the cache
    • E.g., referencing blocks 0, 8, 0, 8, 0, 8 in our direct-mapped example would miss every time
    • If the cache were fully associative, this access pattern wouldn’t be a miss

• Capacity miss
  • Occurs when the set of active cache blocks (working set) is larger than the cache
Cache Performance Metrics

• Miss Rate
  • Fraction of memory references not found in cache (misses / accesses) = 1 – hit rate
  • Typical numbers:
    • 3-10% for L1
    • can be quite small (e.g., < 1%) for L2, depending on size, etc.

• Hit Time
  • Time to deliver a line in the cache to the processor
    • includes time to determine whether the line is in the cache
  • Typical numbers:
    • 4 clock cycles for L1
    • 10 clock cycles for L2

• Miss Penalty
  • Additional time required because of a miss
    • typically 50-200 cycles for main memory (Trend: increasing!)
Let’s think about those numbers

• Huge difference between a hit and a miss
  • Could be 100x, if just L1 and main memory

• Would you believe 99% hits is twice as good as 97%?
  • Consider:
    cache hit time of 1 cycle
    miss penalty of 100 cycles

  • Average access time:
    97% hits: 1 cycle + 0.03 * 100 cycles = 4 cycles
    99% hits: 1 cycle + 0.01 * 100 cycles = 2 cycles

• This is why “miss rate” is used instead of “hit rate”
  • 3% versus 1%
Writing Cache Friendly Code

• Make the common case go fast
  • Focus on the inner loops of the core functions

• Minimize the misses in the inner loops
  • Repeated references to variables are good (temporal locality)
  • Stride-1 reference patterns are good (spatial locality)

Key idea: Our qualitative notion of locality is quantified through our understanding of cache memories
The Memory Mountain

• **Read throughput** (read bandwidth)
  • Number of bytes read from memory per second (MB/s)

• **Memory mountain**
  • Measured read throughput as a function of spatial and temporal locality
  • Compact way to characterize memory system performance
Memory Mountain Test Function

```c
long data[MAXELEMS]; /* Global array to traverse */

/* test - Iterate over first "elems" elements of array “data” */
/* with stride of "stride", using using 4x4 loop unrolling. */
int test(int elems, int stride) {
    long i, sx2=stride*2, sx3=stride*3, sx4=stride*4;
    long acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0;
    long length = elems, limit = length - sx4;

    /* Combine 4 elements at a time */
    for (i = 0; i < limit; i += sx4) {
        acc0 = acc0 + data[i];
        acc1 = acc1 + data[i+stride];
        acc2 = acc2 + data[i+sx2];
        acc3 = acc3 + data[i+sx3];
    }

    /* Finish any remaining elements */
    for (; i < length; i += stride) {
        acc0 = acc0 + data[i];
    }
    return ((acc0 + acc1) + (acc2 + acc3));
}
```

Call `test()` with many combinations of `elems` and `stride`.

For each `elems` and `stride`:

1. Call `test()` once to warm up the caches
2. Call `test()` again and measure the read throughput (MB/s)
Core i7 Haswell
2.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size

The Memory Mountain

Ridges of temporal locality

Slopes of spatial locality

Aggressive prefetching

Read throughput (MB/s)

Stride (x8 bytes)

Size (bytes)
Matrix Multiplication Example

• Description:
  • Multiply N x N matrices
  • Matrix elements are doubles (8 bytes)
  • $O(N^3)$ total operations
    • $2N$ reads per source element
    • $N^2$ elements
  • $N$ values summed per destination
    • But may be able to hold in register

```c
/* ijk */
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```

Variable `sum` held in register
Miss Rate Analysis for Matrix Multiply

• Assume:
  • Block size = 32B (big enough for four doubles)
  • Matrix dimension (N) is very large
  • Cache is not even big enough to hold multiple rows

• Analysis Method:
  • Look at access pattern of inner loop
Layout of C Arrays in Memory (review)

- C arrays allocated in row-major order
  - Each row in contiguous memory locations
- Stepping through columns in one row:
  - for (i = 0; i < N; i++)
    sum += a[0][i];
  - Accesses successive elements
  - If block size (B) > sizeof(a_{ij}) bytes, exploit spatial locality
    - miss rate = sizeof(a_{ij}) / B
- Stepping through rows in one column:
  - for (i = 0; i < n; i++)
    sum += a[i][0];
  - Accesses distant elements
  - No spatial locality!
    - Miss rate = 1 (i.e. 100%)
Matrix Multiplication (ijk)

```c
/* ijk */
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```

Inner loop:
- **Row-wise**
- **Column-wise**
- **Fixed**

Misses per inner loop iteration:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.25</td>
<td>1.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>
Matrix Multiplication (jik)

```
/* jik */
for (j=0; j<n; j++) {
    for (i=0; i<n; i++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```

**Inner loop:**

- **Row-wise** Misses:
  - A: 0.25
  - B: 1.0
  - C: 0.0

- **Column-wise** Misses:
  - A: 0.0
  - B: 1.0
  - C: 0.0

- **Fixed** Misses:
  - A: 0.0
  - B: 1.0
  - C: 0.0
Matrix Multiplication (kij)

```c
/* kij */
for (k=0; k<n; k++) {
    for (i=0; i<n; i++) {
        r = a[i][k];
        for (j=0; j<n; j++)
            c[i][j] += r * b[k][j];
    }
}
```

Inner loop:

- **(i,k)**
- **(k,*)**
- **(i,*)**

Misses per inner loop iteration:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.0</td>
<td>0.25</td>
<td>0.25</td>
</tr>
</tbody>
</table>
Matrix Multiplication (ikj)

```c
/* ikj */
for (i=0; i<n; i++) {
    for (k=0; k<n; k++) {
        r = a[i][k];
        for (j=0; j<n; j++)
            c[i][j] += r * b[k][j];
    }
}
```

Inner loop:
- **(i,k)**
- **(k,*)**
- **(i,*)**

Misses per inner loop iteration:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.0</td>
<td>0.25</td>
<td>0.25</td>
</tr>
</tbody>
</table>
Matrix Multiplication (jki)

```c
/* jki */
for (j=0; j<n; j++) {
    for (k=0; k<n; k++) {
        r = b[k][j];
        for (i=0; i<n; i++)
            c[i][j] += a[i][k] * r;
    }
}
```

Misses per inner loop iteration:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
Matrix Multiplication (kji)

```
/* kji */
for (k=0; k<n; k++) {
    for (j=0; j<n; j++) {
        r = b[k][j];
        for (i=0; i<n; i++)
            c[i][j] += a[i][k] * r;
    }
}
```

Misses per inner loop iteration:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Inner loop:

- Column-wise
- Fixed
- Column-wise

A

B

C
Summary of Matrix Multiplication

- **ijk (& jik):**
  - 2 loads, 0 stores
  - misses/iter = 1.25

- **kij (& ikj):**
  - 2 loads, 1 store
  - misses/iter = 0.5

- **jki (& kji):**
  - 2 loads, 1 store
  - misses/iter = 2.0

For $i=0..n$:

```c
for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
        sum += a[i][k] * b[k][j];
    c[i][j] = sum;
}
```

For $k=0..n$:

```c
for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
        c[i][j] += r * b[k][j];
}
```

For $j=0..n$:

```c
for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
        c[i][j] += a[i][k] * r;
}
```
Core i7 Matrix Multiply Performance

![Graph showing performance comparison for different array sizes and iteration orderings.]

- jki / kji
- ijk / jik
- kij / ikj
Cache Summary

• Cache memories can have significant performance impact

• You can write your programs to exploit this!
  • Focus on the inner loops, where bulk of computations and memory accesses occur
  • Try to maximize spatial locality by reading data objects with sequentially with stride 1
  • Try to maximize temporal locality by using a data object as often as possible once it’s read from memory