Lecture 9: More on Caches

William Gropp

www.cs.illinois.edu/~wgropp
Improving the Architectural Model

• We’ve made a number of simplifying assumptions
  ◆ Each data arrays starts at the beginning of a cacheline
  ◆ The entire cache memory is available for data

• Lets look briefly at both of these
Inefficient Use of Cache Line

- Note all elements of each cacheline may not be used in the small blocks.
- If the number of cachelines in the cache are large enough, if the recursive subdivision stops with a large enough submatrix, the inefficiency is at most a factor of 2 (a cache line is read or written twice, for the blocks either above or below in the same column).
- We assume column-major (Fortran) storage order; for row-major, the cache lines are aligned by rows.
How Much of the Cache is Available?

- We’ve assumed that when a cache line is loaded, it can go anywhere in the cache (the full cache is available for any line loaded from memory)
- This is rarely true
- Computing the location of an address in cache must be very fast, thus it is impractical to search through a large cache to find where a memory line was placed.
- Typically, a cache is divided into sets; an address is mapped to a set, and the choice of which set is made based on the cache replacement policy.
- For example, a 32KB cache might have 64 sets each with 4 entries (each entry is a cache line of 128 Bytes).

<table>
<thead>
<tr>
<th>Rest of address</th>
<th>Set #</th>
<th>Location in cache line</th>
</tr>
</thead>
<tbody>
<tr>
<td>19-51 bits</td>
<td>6 bits</td>
<td>7 bits</td>
</tr>
</tbody>
</table>
Example of Cache Sets

All memory items in a column get mapped to the set – the cache appears to have size two in this case!
Why Do We Care?

- Note the performance variation for the simple algorithm when the matrix size changes by a one row/column.
- Successive elements from the same row are being mapped to the same set, reducing the effective cache size to the number of entries in the set.

<table>
<thead>
<tr>
<th>Size</th>
<th>Naïve</th>
<th>Cache Oblivious</th>
</tr>
</thead>
<tbody>
<tr>
<td>2048</td>
<td>180</td>
<td>1100</td>
</tr>
<tr>
<td>2049</td>
<td>780</td>
<td>1100</td>
</tr>
<tr>
<td>4096</td>
<td>150</td>
<td>980</td>
</tr>
<tr>
<td>4097</td>
<td>560</td>
<td>1100</td>
</tr>
</tbody>
</table>
Reducing the Impact of Cache Sets

- Powers of two are bad for strides through memory – likely to map to the same location in the available sets
  - Solution: Pad data structures to change the access stride
  - Cost: Some extra memory
- Modern systems may have sets of size 8 or more
  - Be careful about the number of different variables
    - Large arrays may be allocated on power-of-two boundaries (simplifying memory allocation) but at the risk of mapping to the same location in the available sets (as if the collection of variables was really a single variable with strides that are a power of two).
When Copying is Not Bad

- When the successive accesses to data will be more efficient (or predictable) if the data is in a special form
- For example, rearranging the data to avoid mapping to the same cache set.
- Dense matrix-matrix multiply codes may do this (copy to a temporary, block-oriented structure) to improve performance for subsequent accesses
  - Note there are \( n^2 \) data items but \( n^3 \) operations – each data item is used \( n \) times, so rearranging for efficient access is worthwhile in this case
Message

• Performance of modern processors can vary significantly with small changes in program or data structure

• Performance modeling can suggest where to look (by giving a performance expectation) but not a guarantee of a specific level of performance

• Understanding cache sets necessary to diagnose performance problems but rarely needed as part of performance model
  ♦ Can be used as a hypothesis when performance is poor