Lecture 7: Matrix Transpose

William Gropp

www.cs.illinois.edu/~wgropp
Simple Example: Matrix Transpose

- do $j=1,n$
  - do $i=1,n$
    - $b(i,j) = a(j,i)$
  - enddo
- enddo

- No temporal locality (data used once)
- Spatial locality only if $(\text{words/cacheline}) \times n$ fits in cache
  - Otherwise, each column of $a$ may be read $(\text{words/cacheline})$ times
- Transpose is *semilocal* at best
Performance Models

• What is the performance model for transpose?
  ♦ $N^2$ loads and $N^2$ stores
  ♦ Simple model predicts STREAM performance
  • It's just a copy, after all
Example Results

<table>
<thead>
<tr>
<th>Matrix Size</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>100x100</td>
<td>4700 MB/s</td>
</tr>
<tr>
<td>400x400</td>
<td>1200 MB/s</td>
</tr>
<tr>
<td>2000x2000</td>
<td>705 MB/s</td>
</tr>
<tr>
<td>8000x8000</td>
<td>*did not complete</td>
</tr>
</tbody>
</table>

- Why is the performance so low?
  - Compiler fails to manage spatial locality in the large matrix cases
  - Why does performance collapse at 8000x8000 matrix
    - May seem large, but requires 1GB of memory
    - Should fit into main memory

- What might be done to improve performance?
Question

- Model the performance of a transpose with this simple model:
  - Assume that the size of the cache is just a few cachelines. Then
    - Access to consecutive elements in memory will read from the cacheline (spatial locality)
    - Access to nonconsecutive elements in memory (the b array in our example) will not be in the available cachelines, forcing a full cacheline to be accessed for every store. Assume a cacheline stores 64 bytes.
  - What is the time cost of a transpose with this model? Use the STREAM performance data as the sustained memory performance in moving data to or from memory to cache
A Simple Performance Model for Transpose

- If source and destination matrices fit in cache, then
  - $T = n^2(r_c + w_c)$
- If the source and destination matrices do not fit in cache
  - $T = n^2(r + Lw)$
  - Where L is the number of elements per cacheline.
- Note that these are not sharp predictions but (approximate) bounds
Let's Look at One Case

- My Laptop
- STREAM performance in Fortran, for 20,000,000 element array
  - 11,580 MB/sec
- Simple Fortran transpose test
  - gfortran –o trans –O1 trans.f
  - Low optimization to avoid “smart compiler” issues with this demonstration
- Performance bound (model):
  - Assume r = w = 1/11,580e6
  - T=n²(r+8w) = n²(9r)
  - Rate = n²/T = 1/9r
Transpose Performance
Observations

- Cache effect is obvious
  - Performance plummets after \( n=1000 \)
  - Need to hold at least one row of target matrix to get spatial locality
    - \( N \times L \) bytes (64k for \( N=1000, L=64 \) bytes)

- STREAM estimate gives reasonable but not tight bound

- Achievable performance for the operation (transpose) is much higher (effectively COPY)
Yes Another Complication

• How many loads and stores from memory are required by \( a=b \)?
  - Natural answer is
    - One load (b), one store (a)
  - For cache-based systems, the answer may be
    - Two loads: Cacheline containing b and cacheline containing a
    - One store: Cacheline containing a
    - Sometimes called write allocate
And Another Complication

- When do writes from cache back to memory occur
  - When the store happens (i.e., immediately)
    - This is called “write through”
    - Simplifies issues with multicore designs
    - Increases amount of data written to memory
  - When the cache line is needed
    - This is called “write back”
    - Reduces amount of data written to memory
    - Complicates hardware in multicore designs

- “Server” systems tend to have write-back; lower performance systems have write-through
Loop Transformations

• Reorder the operations so that spatial locality is preserved

• Break loops into blocks
  • Strip mining
  • Loop reordering
Strip Mining

- Break a loop up into blocks of consecutive elements
- Do $k=1,n$
  
  $a(k) = f(k)$

  enddo

- Becomes
  
  do $kk=1, n, \text{stride}$
    
    do $k=kk, \min(n, kk+\text{stride}-1)$
      
      $a(k) = f(k)$

    enddo

  enddo

- For C programmers, do $k=1,n,\text{stride}$ is like
  
  for($k=1; k<n; k+=\text{stride}$)
Strip Mining

- Applied to both loops in the transpose code,
- \( \text{do } j=1,n \)
  \( \text{do } i=1,n \)
  Becomes
  \( \text{do } jj=1,n, \text{stride} \)
  \( \text{do } j=jj, \text{min}(n,jj+\text{stride}-1) \)
  \( \text{do } ii=1,n, \text{stride} \)
  \( \text{do } i=ii, \text{min}(n,ii+\text{stride}-1) \)
- Still the same access pattern, so we need another step ...
Loop Reordering

• Move the loop over j inside the ii loop:
  do jj=1,n,stride
  do ii=1,n,stride
    do j=jj,min(n,jj+stride-1)
      do i=ii,min(n,ii+stride-1)
        b(i,j) = a(j,i)
  
• Value of stride chosen to fit in cache
  ♦ Repeat the process for each level of cache that is smaller than the matrices
    • Even a 1000 x 1000 matrix is 8 MB, = 16MB for both A and B. Typical commodity processor L2 is 2MB or smaller, so even modest matrices need to be blocked for both L1 and L2
Multiple levels of Cache

• Blocking is not free
  ♦ There is overhead with each extra loop, and with each block
    • Implies that blocks should be as large as possible and still ensure spatial locality

• Moving data between each level of cache is not free
  ♦ Blocking for each level of cache may be valuable
  ♦ Block sizes must be selected for each level
Example Times for Matrix Transpose

<table>
<thead>
<tr>
<th>5000x5000 transpose (a very large matrix)</th>
<th>Unblocked</th>
<th>L1 Blocked</th>
<th>L1/L2 Blocked</th>
</tr>
</thead>
<tbody>
<tr>
<td>(20,100,g77)</td>
<td>2.6</td>
<td>0.55</td>
<td>0.46</td>
</tr>
<tr>
<td>(32,256,g77)</td>
<td>2.6</td>
<td>0.46</td>
<td>0.42</td>
</tr>
<tr>
<td>(32,256,pgf77,main)</td>
<td>0.58</td>
<td>0.48</td>
<td>0.55</td>
</tr>
<tr>
<td>Same, within a subroutine</td>
<td>2.8</td>
<td>0.55</td>
<td>0.48</td>
</tr>
</tbody>
</table>
Observations

• Blocking for fast (L1) cache provides significant benefit

• Smart compilers can make this transformations
  ♦ See pgf77 results

• But only if they have enough information about the data
  ♦ When the array passed into a routine instead of everything in the main program, results no better than g77

• Parameters are many and models are (often) not accurate enough to select parameters
Why Won’t The Compiler Do This?

- Blocking adds overhead
  - More operations required
- Best parameter values (stride) not always easy to select
  - May need a different stride for the I and the J loop
- Thus
  - Best code depends on problem size, for small problems, simplest code is best
- Notes some compilers support annotations to perform particular transformations, such as loop unrolling, or to provide input on loop sizes (the “n”)
Why Don’t Programmers Do This?

- Same reason compilers often don’t – not easy, not always beneficial
- But you have an advantage
  - You can form a performance expectation and compare it to what you find in the code
    - Measure!
  - You often know more about the loop ranges (n in the transpose)
- This is still hard. Is there a better way?
  - Sort of. We’ll cover that in the next lecture.
Questions

• Develop a performance bound for this operation
  ♦ do i=1,n
    a(i*stride) = b(i)
  enddo
  ♦ How does your model depend on stride?
  ♦ What changes in your model if the cache uses a write-allocate strategy?
  ♦ What changes if the copy is
    do i=1,n
      a(i) = b(i+stride)
    enddo

• Note: such a “strided copy” is not uncommon and may be optimized by the hardware
  ♦ This model does not take that into account
Question

• In blocking the transpose, we used the same block size for the rows and column. Is this necessary? Why might a different value make sense?