Austin Herring

CS 540 Homework 3, Part 1

October 14, 2015

1. The “write allocate” means that every time something which is not in the cache gets written to, the entire block containing it is brought into the cache, and then the value is written to the cache. Generally, with write allocate, the write back to main memory is only done when the block is removed from the cache. Though this can be beneficial if many local writes occur at once, this can have very bad effects if the writes are strided.

Considering the diagram, contained in file problem1a.jpg and labeled *A*, the first write will be to A(1, 1). Because this is the first data access, A(1, 1) will not be in the cache, so because of write allocate, A(1, 1), and its associated block, will be brought into the cache. However, the next memory access will be at A(1, 2), which, because of the way Fortran stores arrays, will not be local to A(1,1), so, assuming N is large enough, A(1, 2) will not be in the same block as A, and write allocate will force another reading from main memory into the cache. This will continue across the entire matrix: for each column, all the way up to A(1, N), there will be a cache miss, and the block will have to be read into the cache.

Eventually, j will reach N, and the outer loop will progress to the next i, bringing the diagram back to the first column, where A(2, 1) will now have to be written. However, all of the columns in A would have been brought into the cache as j progressed, so, assuming N is large enough, A(2, 1) will no longer be in the cache. First, note that this means that A(1,1), and its block, must have been written back to memory at one point. Next, note that this also means that, now, A(2, 1) and its block must be brought back into the cache, because of write allocate, and then written to the cache. Again, this happens all the way across the matrix, for all 1 <= j <= N: the entry will no longer be in the cache, so the line must be read into the cache. Also note that this will repeat for every row in the matrix. That is, for every row, in every column, the entry will no longer be in the cache, meaning that it was already

written back once and that it must now be read into the cache again.

Bringing this information all together, this essentially means that there are two main memory accesses, one read and one write, for every entry in the matrix, because there will be a cache miss on every access, forcing a read, and there will be a removal from the cache and a write back before the next row can access the cache. This is in contrast to a strided read: here, although when B(2) is reached after finishing with row 1 (or any element B(k) is accessed after finishing with row k – 1) and it will no longer be in the cache, only one memory access is required, the one to read it into the cache. Though this will repeat for every column and over every row, as with write, there will only be one access per element, as opposed to two for the write. Additionally, if write is not strided, a cache miss will only happen once for every block which the columns take up, and there will be only one write for each of these blocks. In other words, there are two memory access per *block* for a non-strided write rather than two per individual entry, which could provide an fair improvement in performance.

Therefore, to take advantage of the fact that strided read is more efficient than strided write, the loops should be rewritten as follows:

```
do j=1,N
      do i=1,N
            A(i, j) = B(j, i);
      enddo
enddo
```

The visualization of these reads and writes will now look like A' and B' in problem1a.jpg and problem1b.jpg. Note that A is no longer being accessed in a strided fashion, so entire blocks can be written at once, and note that B is strided now, meaning that a read, but only that one memory access, will need to happen for every element.

2. First, it can be seen that, for every iteration of the outer for loop, the inner for loop executes N times, for j from 1 to N: using a simple sum, N – 1 + 1 = N. The outer loop similarly, for i from 1 to N,

executes exactly N times: using the same sum, N – 1 + 1 = N.

Now, note that exactly one element of A is needed for every iteration of the outer loop, A(i). By the reasoning above for number of iterations of the outer loop, this means there will be N loads from A(i). or N loads of words or, in other words, N words worth of traffic. Similarly, note that there is one access of B for every iteration of the inner loop, meaning N accesses and loads of words for every iteration of the inner loop. However, this inner loop executes once for every iteration of the outer loop, giving a total of N * N = N^2 memory accesses, and, in turn, N loads of words and N^2 words worth of traffic. Combining theses calculations for accesses to A and B gives $N^2 + N = N(N + 1)$ words of traffic that must be loaded.

3. First, note that the outer loop runs N / b times: jj runs from 1 to N in steps of b, so, letting k be the number of iterations, 1 + bk >= N + 1 → k >= N / b. Therefore, the number of iterations of the outer loop is N / b.

Now a sum for the number of memory accesses by A can be constructed. Within the middle i loop, only one element of A is accessed, A(i), over all iterations of that middle loop. Therefore, a sum for the memory traffic generated by A for the whole computation becomes:

$$\sum_{jjj=1}^{N/b} \sum_{i=1}^{N} 1 = \sum_{jjj=1}^{N/b} (N-1+1) = \sum_{jjj=1}^{N/b} N = N \sum_{jjj=1}^{N/b} 1 = N(N/b-1+1) = N^2/b$$

where:

$$jjj = (jj-1)/b+1$$

and represents the current loop *iteration* number, from 1 to N / b, as argued in the first paragraph.

Similarly, for B, only one element is accessed for each iteration of the innermost j loop, so a sum for the memory traffic that the innermost loop generates becomes:

$$\sum_{j=jj}^{jj+b-1} 1 = (jj+b-1)-jj+1 = b$$

Now, assume that b is small enough that the cache can maintain all b elements of B over all iterations

of the middle i loop. Then, for each iteration of the jj loop, the iterations of the i loop can be ignored, and the final sum becomes:

$$\sum_{jjj=1}^{N/b} b = b \sum_{jjj=1}^{N/b} 1 = b(N/b - 1 + 1) = b(N/b) = N$$

where jjj is defined as before.

Combining the calculations of memory traffic for the two arrays and factoring out an N yields the final answer:

$$N^2/b + N = N(N/b + 1)$$

This proves that there is this much traffic flow of words required for this blocking scheme, assuming that b is a small enough size to fit within the cache.