Problem 1 - Final

Saturday, April 11, 2015 5:36 PM

Explain why your improvement results in smaller number of cache misses.

Note: I did not see the cache size until I completed this whole project, so you may be wondering why I've been testing whether my array row/column can fit in the cache or not. I could have done a MUCH more accurate analysis of miss rate if I realized this earlier. Regardless, please enjoy my very long lab report.

```
--------------------------------------------------------------------------------
         Ir I1mr ILmr          Dr         D1mr         DLmr          Dw  D1mw DLmw  file:function
--------------------------------------------------------------------------------
848,625,104    3    3 524,188,383 1,035,255          626 74,660,026     1    0  /home/org/Documents/lab2/refcode.c:level_4
 63,063,181    3    3  37,831,300 2,637,694       38,295  2,113,669     2    2  /home/org/Documents/lab2/refcode.c:level_5
 17,832,989    3    3   7,343,110 1,048,573 1,048,573  1,049,606     2    2  /home/org/Documents/lab2/refcode.c:level_1
 15,207,947    3    3   6,817,283 1,041,408 1,041,408  1,573,380     2    2  /home/org/Documents/lab2/refcode.c:level_3
  7,353,385    5    5   2,102,282   262,210   262,178    263,179   514  513  /home/org/Documents/lab2/refcode.c:level_2
```

D1MR, D1MW

Level 1

Assume array "B" is stored at location B in memory. Each iteration accesses the array in the following manner:

- 1st iteration of the outer loop, i = 0: B[0][0], B[1][0], B[2][0],…., B[1024][0]
- 2nd iteration of the outer loop, i = 1: B[0][1], B[1][1], B[2][1],…, B[1024][1]
  …..
- 1024th iteration of the outer loop, i = 1023: B[0][1024], B[1][1024],…., B[1024][1024]

Accessing the array B using column-row major order, the formulated memory access pattern is:

- B + (j*1024 + i)*4

Therefore, each iteration accesses the memory (in decimal) in the following manner:

- 1st iteration of the outer loop, i = 0: B, B + 4096, B + 8192,…., B + 4194304
- 2nd iteration of the outer loop, i = 1: B+1024, B + 4112,….
  …..

The calculation shows that the program is not cache friendly because it is not exploiting spatial locality of the array B -- it is not accessing elements that are stored contiguously in the array. The program first accesses an element stored at address B then accesses an element that is 4096 bytes away instead of the next element in the array which is 4 bytes. By accessing an element that is 4096 bytes away, the program will likely generate another cache miss (assuming that my block size is smaller than 4096 bytes). For i = 1 and j =1 to j=n-1 (inner loop), this pattern of cache misses will continue for every element. Thus generating 1024 misses. However, depending on the cache size, there are two main scenarios for i = 2. Case 1, cache size is larger than the total size of the array. In this case, all of the blocks that were brought along will remain the cache and therefore future array accesses will only generate cache hits. Case 2, array size is bigger than my cache. Since array size is bigger than my cache size, the "first" (ie. set 0 for cache line in my cache contains B + largeNumber. Therefore, when I try to access B[0][1], I will get a cache miss because the block B that has B and its nearby element is not stored as the first element in cache anymore. This pattern will continue for every element, thus producing another 1024 cache misses. In summary, there will be 1024 misses per each i-th iteration and there are 1024 i-th iterations, so the total number of misses will be **1024 * 1024 = 1048576 (WOW)** which is approximately the same as the original cache miss showed above (this calculation has 3 bytes more than the above). From this we can conclude that it is case 2, where the cache size is smaller than the array size.

I reduced the number of cache misses by using a stride-1 reference pattern. I switched the i and j placements.

```
B[i][j] = 2*(B[i][j] + 2);
```

to

```
B[j][i] = 2*(B[j][i] + 2);
```

Thus, I am using row-column access pattern, which is the natural pattern of how elements are stored in memory. Therefore, stride-1 fixes the problem because the program is not bringing in a new block each time it access the next element defined by the iteration, it only transfers from memory into the cache when it finishes processing the last element of the block.

Essentially, this is what is happening in memory - the program generates a miss the first time it accesses a cold cache, then it brings in a block from memory, processes all the elements in the block, generate a miss because of the new empty

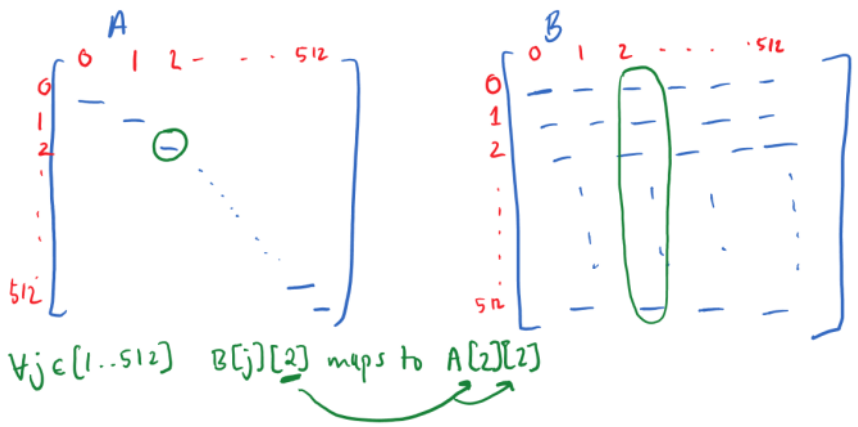
block (and then the process repeats).

Ratio = read miss of new program + write miss

(131073+2)/(1048573+2) = 0.125

Level 2

The main source of miss read is due to a similar problem as level 1 - the program is accessing the array using column-row (the math is pretty much the same as the above) order and thus does not take advantage of the spatial locality. The program brings a block from memory consisting of portion of a row in B, but only uses one of the elements, then it brings in the next row. From the original code, the total number of read misses is 262,177, which is the same as (512* 512)+33. Since we are making 512*512 accesses to B, this hints that every access to B is a miss. Therefore, since we are accessing every element in B, I switched to row major order (using stride-1 reference pattern) and this significantly reduced the number of read misses because it exploits spatial locality. Instead of bringing in a new block for each time I access B[j][i] (the same blocks are brought in from cache to RAM multiple times), I will only bring in a new block when every element of the block is processed (I will only bring each block into the cache one time). Because of the shift to row-column order I had to do a little bit of reformulation. First, I have to remap my values. We can clearly see that there is a bijective mapping from B's column number j --> to (i,i). The mapping is drawn below:



Using this mapping, I reformulated my code so that for each inner loop, I will write to all of the diagonal elements in A (as opposed to completing one diagonal at a time like the original). In addition, I had to create a separate loop to initialize A[i][i]. After initializations, blocks containing all of A[i][i] will be brought into cache. The new code is pasted below.

```
for (i = 0; i < DIM; i++){
    A[i][i] = 0;
}

for (i= 0; i < DIM; i++) {
    for (j = 0; j < DIM; j++){
        A[j][j] += B[i][j];
    }
}
```

This new code help reduced the ratio to 12.7%.
Ratio = (32833 + 514)/(262210+514) = 0.1269

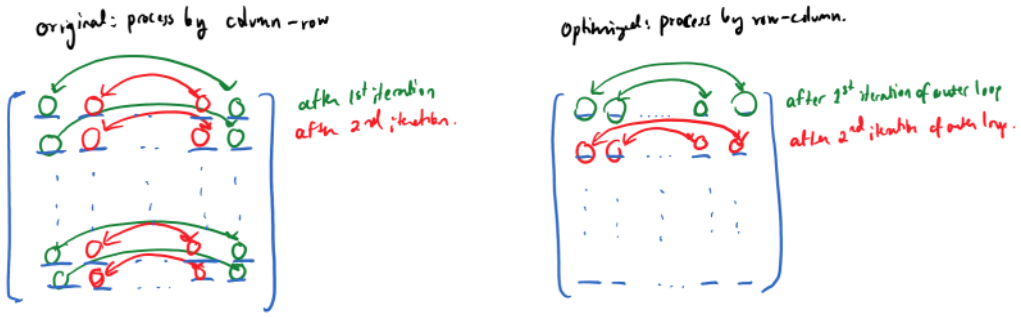
Level 3

What level 3 is doing is swapping elements from opposite edges starting from the far left most with the far right most. Then switch the next most element with the next right most (left + 1, right - 1). The pattern continues until the two middle elements C[j][511] is swapped with C[j][512]. Do this for all j's from 0 to 1023.

Level 3 has similar number of read misses as level 1, so using the same reasoning as level 1, it is very likely that every access to the array is a miss. However, even if the number of read misses are similar, they are generated differently. In level 1, I am generating 1 miss per each iteration of inner loop and I run the inner loop a total of 1024*1024 times. However, In level 3, I am generating 2 misses per each iteration of inner loop, but the total number of iteration of the

inner loop is 512 * 1024 (so multiply this by 2). Thus, the total comes out to about the same.

I improved this level using a similar approach as level 1 and 2. Instead of incrementing the row number and processing each element, which causes a miss each time so the program has to transfer in a new block, I process one row at a time using stride 1 reference thus exploiting spatial locality. Below is a pictorial representation of what my optimized code does:



I was able to reduce the read miss to 131,072 times.

ratio = (131072+2)/(1041408+2) = 0.1259

Level 4

I optimized the original bubble sort by coming "two" bubble sorts into one. The original code did one way bubble sort by organizing the largest numbers on the far right. So after first iteration of the outer loop, the largest element will be in its correct right-most location of the array. When the second outer loop iteration begins, the program has to transfer back to the beginning of the array an reload all of the blocks again. So even though this bubble sort takes advantage of spatial locality by comparing contiguous elements the array is large enough that it has to reload all the blocks over again when it starts a new outer loop iteration.

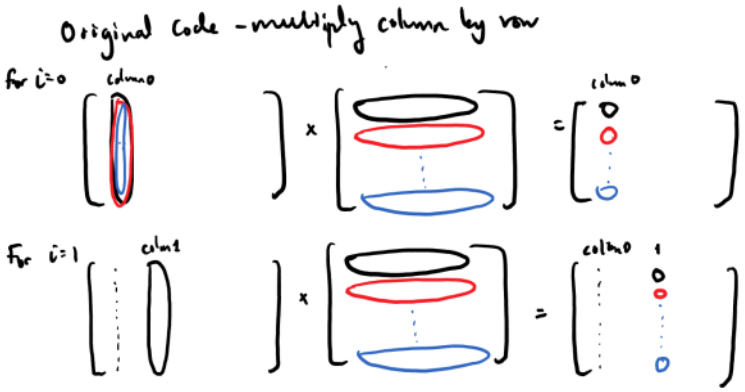
Per each iteration of the outer most loop, the optimized bubble sort will put the largest element on the farthest right in its correct location and the smallest element in the farthest left. This reduces the number of miss because instead starting from the farthest element, where the program has to load in a new block, the program iterates back from the farthest right towards the left, reusing the existing blocks that was recently loaded in.

Ratio:
214304/1035255 = 0.207

Level 5

How I approached this problem

- My first thought was to test whether the cache is big enough to hold a 128 by 128 array and with maybe some additional left over spaces. If it is big enough with leftoverAll I have to do is store the entire array A in the cache, generate only compulsory misses, and keep accessing every 128 elements like the original and I would not have any extra misses because the array A is already in cache. Then, I would change the way I access C from accessing every 128 elements to using stride-1 reference pattern. So now instead of brining in a new block each time, I will finish processing the entire block, then bring in a new one and never come back to the old block again. I would leave B the same (stride-1), and the read misses would be generated the same way as A. So I tested if C is large enough by deleting the access code to array B and A and just assigned the value C[i+j*n] to itself. It generated approximately 16,000 read misses, which the size of the array itself, thus indicating that every access was a miss so I can easily conclude that the cache is smaller than the array. So this will not work.
- Wow. I just realized after AN HOUR that this was matrix multiplication and not those fancy patterns we did for level 1 to 3. **Great trick professor** - A,B,C were declared as 1D array with DIMxDIM and not declared with traditional 2D array notation like Java and I just realized that in C 1D array is the same as 2D!!!!
- this not a regular matrix multiplication. It multiplies columns by row instead of row by column.



Miss rate analysis (approximation):

- Array A: The inner loop increments A's row each time, which generates a miss. There are 128 * 128 * 128 = 2,097,152 misses total.
- Array B: The program accesses elements in B using stride-1 reference, which is good. B only generates misses when it brings in a new block to the cache, which is done only when it finishes processing all of the elements in the cache line (block). There are 128 elements, of 4 bytes each per row, so that is a total of 512 bytes per row - a relatively small number of bytes. Therefore it is reasonable to assume that B takes two trips to the memory to bring in two blocks(which makes up a row), or in other words, when the middle loop gets incremented the program makes two trips to memory. The total number of times the middle loop gets incremented multiplied by two is an approximation of total number of misses from array B. so 2 * 128 * 128 = 32,768 misses.
- Array C: The program takes advantage of temporal locality for array C since each element is accessed consecutively 128 times. However, it is not taking advantage of spatial locality. There are 128*128 elements in the array therefore a total of 16,384 misses.
- Approximation of the total number of misses is around 2,220,000 times -- relatively close to the actual answer.
- Most importantly, this shows what I have to focus on - minimize the main source of misses, which comes from array A.

My improvement

- I followed the same pattern as level 1 to 4 by making the code spatial locality as friendly as possible by accessing contiguous elements in row A, which was the main source of misses. Furthermore, with the optimized code, array A only accesses each row one time, per one iteration of the outer loop. Therefore, array A will go through 128 rows 128 times in the life time of the program -- which is exactly the same as how array B was accessing its elements in the original code. I also saved array B's value before I enter the inner loop, so the value is saved in the register as opposed to the cache. This means array B may only generate 1 miss maximum per 128 iterations of the inner loop. Therefore the maximum possible number of misses generated by array B is 128*128 = 16,384 times. Array C also accesses elements using stride-1 reference pattern and it will only access every row but only once throughout the entire life time of the program.

The ratio:
532482/2637000 = 0.2019

```
--------------------------------------------------------------------------------
         Ir I1mr ILmr          Dr    D1mr   DLmr         Dw  D1mw DLmw  file:function
--------------------------------------------------------------------------------
848,685,117    6    6 524,233,388 211,554    626 74,660,030     1    0  /home/org/Documents/lab2/optimized.c:level_4
 52,790,413    4    4  31,654,532 532,482  6,129  2,162,821     3    2  /home/org/Documents/lab2/optimized.c:level_5
 17,832,989    3    3   7,343,110 131,073 65,537  1,049,606     2    2  /home/org/Documents/lab2/optimized.c:level_1
 15,211,531    3    3   6,818,819 131,072 65,536  1,573,892     2    2  /home/org/Documents/lab2/optimized.c:level_3
  7,354,925    6    6   2,103,307  32,833 16,417    263,180   514  513  /home/org/Documents/lab2/optimized.c:level_2
```

Problem 2 - Final

Saturday, April 18, 2015 6:08 PM

Problem 2

Cache	S	t	s	b
1	64	24	6	2
2	1	30	0	2
3	128	22	7	3
4	1	29	0	3
5	32	22	5	5
6	8	24	3	5

Problem 3 - Final

Saturday, April 18, 2015 5:55 PM

32 BYTES

Source array

	col 0	col 1	col 2	col 3
row 0	m	m	h	m
row 1	m	h	m	h
row 2	m	m	h	m
row 3	m	h	m	h

Destination array

	col 0	col 1	col 2	col 3
row 0	m	m	m	m
row 1	m	m	m	m
row 2	m	m	m	m
row 3	m	m	m	m

64 BYTES

Source array

	col 0	col 1	col 2	col 3
row 0	m	m	h	h
row 1	m	h	m	h
row 2	m	h	h	m
row 3	m	h	h	h

Destination array

	col 0	col 1	col 2	col 3
row 0	m	m	h	h
row 1	m	m	m	h
row 2	m	h	m	m
row 3	m	h	h	m