Report

Students: David Morales, Sam Tsoi

Phase 0:

For this phase we followed the instructions that were detailed in the prompt. Copied mmm.c, matrix_checksum.c to our directory. We made the Makefile and added the advance attributes just in case if we need to add more files.

Phase 1:

In this phase, we used the information that was given from the slides (https://cs61.seas.harvard.edu/wiki/images/0/0f/Lec14-Cache_measurement.pdf) to help us code the matrix multiplication. We declared three pointers that represents our matrix. From the main function, we did some error checking on the user input, as outlined in the prompt. We malloc memory for each of the matrices and initialize matrix A and B with the given specs by doing a double for loop to fill in the values. These matrices are 1D matrices and we filled them in by i * N + j with a_{ij} = i + j and b_{ij} = i + j * 2 as our values (all defined in the prompt). From there the program leads into a series of 6 conditional statements, checking which permutation the user typed.

After checking which one, we jump to one of the six function to compute the matrix mutliplication. There are six functions that compute each ijk combination. They all have the similiar code structure except when it comes to the indexes of the loop. Here we used the slides to help us code the three for loops for each matrix combination.

Initially we tried to have one triple nested for loop and tried to define the order of the loops by detecting the order of user input. However, we found it way easier and more clear if we just wrote loops for each of the 6 approaches, being separated into 6 different functions. We ran into some issues when running each ijk pair againt his reference program. For the ijk combination, our program did not compute matrix C correctly. We fixed it by changing the data-type of our sum variable (the variable that hold the result of each C_ij value) from float to double.

At the top of the c file, we added the signature for matrix_checksum.

Phase 2:

For this phase, we had trouble getting the time from nanoseconds to seconds. We googled the conversion which also showed us how to use clock_gettime(). When we ran the program, the output still resulted in nanoseconds. We googled again to find another conversion scheme as we found that nsec determines the nanosecond that the current second is on. So the actual time is {sec}.{nsec}. We decided to declare a function that takes in the start and end variable (declared as struct timespec) to calculate the difference in time.

The function simply checks if the difference between end and start is negative. If so, use a

temporary variable to hold the difference between end and start and subtract -1(in seconds). Then for the same variable (but in nanoseconds) find the difference between end and start and add 1e9 to convert to seconds. The function then prints the time in seconds.

We referenced http://www.guyrutenberg.com/2007/09/22/profiling-code-using-clock_gettime/ to find the time.

Timing for Memory Pattern Access

For ijk

50	200	500	1000	2000
0.13188	.14859	.1998	.98230	46.6972
0.13181	.12596	.11579	.92910	46.494
0.403680	.14077	.11523	.93812	46.628

For ikj

50	200	500	1000	2000
0.00051	.01506	.15234	.15888	.27747
0.00075	.01682	.15503	.14624	.26124
0.00083	.01546	.15148	.15246	.29416

For jik

50	200	500	1000	2000
0.17231	.98311	.95871	.79126	45.119
0.45845	.11011	.96371	.79270	45.134
0.45842	.17465	.94204	.79369	46.510

For jki

50	200	500	1000	2000
0.74013	.17549	.24082	11.485	113.68
0.28913	.19889	.24508	11.561	108.544
0.16665	.19194	.24429	11.793	108.626

For kij

50	200	500	1000	2000
0.50957	.99774	.15249	1.1636	113.687
0.38975	.14289	.14626	11.561	108.544
0.50976	.50976	.15174	11.793	108.626

For kji

50	200	500	1000	2000
0.28871	.28871	.24476	11.103	108.564
0.60283	.60283	.23768	11.631	110.216
0.41453	.41453	.24569	11.435	109.167

Explanation for memory access pattern.

We found that ikj combination outperformed the rest and this is due to cache performance. Since arrays in C are allocated in arranaged in row-major, we need to take advantage of getting cache lines in order to achieve spatial locality. By having i as the outer loop, the index at matrix A will stay constant. Then having k then j respectively, k and j will stay fix on the same row this achieving spatial locality because the cpu doesn't have to go to memory because the values have been brought in by the cache line. The other combinations will have cache miss because they don't follow spacial locality. They data is being accessed in a column like manner thus when the loop moves to the iteration, a cache miss is definte because it is not in the cache line.

Phase 3:

In this phase, we implemented pthread on ikj as we found that this loop order was the fastest, especially as n increases. We decided to implement 8 different threads, and in order to do this, we created two data structures. One data structure is the TCB, which is the thread control block of all the threads. There will be 8 TCBs that we will initialize, and each thread will have its unique ID (TID) within the data structure. We also created a data structure for the argument that is going to be passed into matrixMultiplythread(), the function that the threads are being passed into when they are created. We call this data structure threadArgs and it contains matrix a, matrix b, matrix c, n, the thread number (so it will be {0,1,2,...,7} for our 8 threads}, and the size of each thread, which is n/8. We initialized the values of the threadArgs data structure with the said parameters and malloc space for the data structure. In a loop, we created thread one by one and passing it into a self-defined function called matrixMultiplypthread. Note that the thread number changes for each thread we pass into matrixMultiplypthread, but the rest (matrix A, B, C, n, etc.) stay the same. In matrixMultiplypthread(), we will have these 8 threads execute simulataneously on 8 partzs of the

matrix independently. So, what we did is that each thread will work on their own part of the matrix. Which part of the matrix is dependent on the thread number of the thread. For example, if n=2000, thread 0 will work on i=0 to i=249, thread 1 will work on i=250 to i=499, thread 3 will work on i=500 to i=749, and so on. How these values are calculated is that the starting i will be (thread number) * (size of thread), so for thread 0, it will be 0*250 = 0. The ending i will be (thread number +1)*(size of thread), which will be 250 for thread 0. In this case each thread can work on their own parts. We use the same arithmetic as we did in the matrixMultiplyikj() function with 2 loads and 1 store. When these threads are finished with their respective parts, we do pthread_join() on each thread.

Timing for Memory Pattern Access

For pthread (PHASE 3)

50	200	500	1000	2000
0.251289	.60225	.26800	.16754	1.66725
0.30052	.54047	.29488	.18369	1.58212
0.20059	.52864	.25212	.17868	1.57365

Just like the other phases, we got the time it took for this approach and we found that the pthread approach is much faster as n gets larger. However, it would be better to stick with the traditional ikj approach for when n is smaller. This is because making, intiializing multiple threads and then joining them using another for loop actually ends up taking longer thus is made unnecessary when the matrices are small. Threads will only be useful when n is large as these threads will be computing its respective parts parallely.

We found a lot of challenges in this phase. We could not find out why our resulting C matrix was hashed an incorrect function with matrix_checksum. We made sure that the threads are working on their separate parts. After a day of being stuck on this, we realized that it was because we didn't initialize each thread every time the threads were made. We only initialized the first thread initially.

Phase 4:

For phase 4, we decided to improve temporal locality with the blocking method. This method incorporates dividing the matrices into smaller subblocks and computing these smaller subblocks separatedly. Before starting this phase, we made sure to do research on how to do blocking, and we referenced http://www.cs.rochester.edu/~sandhya/csc252/lectures/lecture-memopt.pdf. We created a function called bijk(), so the user can use the blocking method by typing ./mmm bijk n. We decided to do blocking on the ijk approach of multiplication. We initialized B = 8, which will be used as the size of the sub block. We initialize the sub block of matrix C to be 0 first by going through each width of the each of the sub blocks and then traversing from i to n. For each sub block, which is indicated by the innermost loops, we found each element of matrix C in that sub block but summing

the row sliver of matrix A with the same block of matrix B. We will traverse through that sub block of the matrix A until it updates the respective elements of matrix C until we move onto the next part of matrix B. Note that for each row for matrix A, we use the same section of B to find C in order to perform the matrix multiplication in that spot for matrix C. Thus, we will use this block n times as we traverse through matrix A to compute those elements of its sub block of matrix C. We defined a MIN function at the top of the page to use in this loop just in case the the number of sub blocks is the divisible by our B. Each sub block should work on its own sub blocks of matrices from the beginning of that sub block (in our example, jj for width of matrix B, kk for the width of matrix A) until the end of that sub block (which is i+B). However, we needed to implement the MIN function, because n is not divisible by B, we would need the sub block to iterate from the beginning of that last sub block until n. (So if it is divisble, i+B will be equivalent to n for the last sub block). The outer loop does all of this computation for each of the sub blocks. As you can see, there are many nested for loops, but this still increases efficiency as each multiplication is done on different parts of the matrices since we are treating the sub blocks as scalars and then adding them all up. We do all of this n times for each subblock!

Timing for Memory Pattern Access

For bijk (PHASE 4)

50	200	500	1000	2000
0.51701	.13890	.10803	.92709	8.1237
0.19999	.13519	.10650	.90613	7.6492
0.41926	.14182	.10646	.94083	7.9133

Just like the other phases, we got the time it took for the blocking approach and we found that blocking for ijk is much faster than the ijk approach, especially as n increases. For n = 2000, the average time it took for the ijk approach is 46.6064 seconds whereas the average time it took for the bijk approach is 7.8954 seconds. At smaller n's, like n=50, the difference is much smaller, and in our case, the original ijk approach is faster. The ijk approach took an average of .22246 seconds and the blocking approach took an average of .37875 seconds. This is because when n is smaller, the need for splitting the matrices decreases as well since the many nested for loops would be computationally expensive when it can just traverse the matrices as a whole when the matrices are small. For when n is larger, ijk approach is multiplying the matrices as a whole, traversing each row and column of matrix A and matrix B, up to n, to calculate matrix C. However, we were able to fasten this process by implementing blocking, which means we split up the matrices into smaller sub blocks. We are able to do this as we took advantage of the how matrices multiply, and we treated these sub blocks like scalars to fill up the C matrix. Even though this compromises readability of the quote with so many nested for loops, it was able to improve the performance.