Reagan Leonard

Matrix Multiplication Optimization Techniques

1. *Abstract:*

This document is a summary of the techniques and methods that I used to optimize the performance of a simple matrix multiplication program called “matmul_naive.c” that was given to me for this project. My goal with this project was to manually apply common loop optimization techniques (and other manual compiler optimizations) to attempt to speed up the process of the “naïve” matrix multiplication implementation. The techniques that I used to do this were 1) loop interchanging, 2) register variables, 3) loop unrolling, and 4) blocking (with block sizes of 16 and 36). I applied these optimizations one-by-one and measured the time taken to perform the matrix multiplication in microseconds for various sizes of matrices (25x25, 50x50,…400x400). I then calculated the speedup of each level of optimization (labeled T1-T4) for each matrix size by dividing the time taken by the naïve code by the time taken by the optimized code ($Time_{naive}/Time_{opt}$). The greatest speedups that I found for each level of optimization are as follows: T1 = 272.94, T2 = 391.1, T3 = 4161.57, T4 (blk16) = 20.89, and T4 (blk36) = 28.95.

2. *Related Work:*

As stated above, I utilized optimization techniques such as loop interchanging, register variables, loop unrolling, and blocking, none of which are my own invention. These code optimization techniques are well known to experienced programmers in the real world. I simply implemented the code necessary to achieve these optimizations for this specific program. I will now briefly describe what each of them are. Loop interchanging is the process of reordering nested loops until the greatest or desired speedup is achieved. Executing loops in different orders can often improve latency by taking advantage of spatial or temporal locality (a common theme for many

loop optimization techniques) [4]. “Register variables” is the term given to the practice of signaling to a compiler that any variable marked by the keyword “register” (typically the most commonly used variables in a program) may be stored in a register which is more quickly accessed than memory [5]. The compiler will decide whether to actually put it into a register, but this practice is an easy one to potentially help speed up a program. Loop unrolling is the process of breaking a loop down into “a repeated sequence of similar independent statements” [3]. So rather than having a one-line loop body repeated 100 times, one could have a 10-line loop body repeated just 10 times. This helps by reducing branch penalties and latencies. Finally, blocking is the process of iterating through a multi-dimensional array, for instance, block-by-block [6]. In this case, rather than iterating through an entire row of a 100x100 matrix (100 elements), the program iterates through a 4x4 block and then increments to the next 4x4 block until the entire array has been indexed.

3. *Methodology:*

3a. *Loop Interchanging:* The first technique that I implemented was loop interchanging. In the naïve code, there were 3 nested loops (j, l, and i). I tried every possible combination of these 3 loops and ran the code at least 10 times per combination. I found that the loop order that resulted in the greatest speedup for me was the order i-l-j.

3b. *Register Variables:* I then moved on to the next optimization technique: register variables. This involved simply adding the keyword register to the beginning of each declaration of all the most commonly used variables in this program. I decided to apply this only to the variables directly used in the 3 loops mentioned above (i, j, k, l, m, and n) as well as any other variables that I created after this step that were directly used inside these loops (rem, rounds, and a).

3c. *Loop Unrolling:* The next step was to implement loop unrolling. This was the first step that required a real dramatic change of code. I decided to unroll my innermost loop by a factor of 4,

meaning that instead of j incrementing by 1 each iteration, j would increment by 4 each time through the loop. In order for this to work, I had to change the program to accommodate this. I turned my one-line loop body into 4 lines where C[i][j] = C[i][j] + B[l][j]*A[i][l]; was followed by 3 lines that looked like C[i+1][j] = C[i+1][j] + B[l][j]*A[i+1][l]; (I being added to 1, 2, and 3, respectively). This is the basics of loop unrolling. However, once I did this and began testing, I began to get segmentation faults over and over. Eventually I realized that this is because unless the dimensions of the square matrix were perfectly divisibly by 4, then this code would try to index to rows in the matrix that did not exist. To fix this, I created a variable called rem (set equal to the dimension of the input matrix % 4) which represents the remaining rows that need to be covered after the largest multiple of 4 (that's still less than the dimension number) has been iterated through. For example, for a 10x10 matrix, rem = 10 % 4 = 2. So the program would run normally for 2 iterations, incrementing j by 4, then it would go into a switch statement based on rem and--in this case—would iterate over the final 2 rows in the matrix.

3d. ***Blocking:*** The final optimization technique that I applied was blocking. I had to do this is such a way that it intertwined with the loop unrolling that I had just done. To do this, I made another loop within the j loop that is controlled by a new variable, a, which will iterate from 0-3 indexing through the rows. This is so that I can now change i to also be incremented by 4 each loop (which will be whenever we need to shift right 4 units to the next "column" of blocks) because the rows will now be controlled by a instead of i [2]. This change means that my blocks will be of size 16 (4x4). I also made another variable called "rounds" just to keep track of whether the program is iterating over a "regular block" (that is, 4x4 or 6x6 depending) or a "remainder block" (which could be something like 2x4 or 2x2 or 5x6), in which case there is code to handle those edge cases. The final step was to change the code to work in blocks of size 36 (or 6x6). To do this, all I had to do was change i and j to be incremented by 6 each loop and

add 2 additional lines to each instance of unrolling. For more details on this, see my code file.

4. *Results:*

I ran the naïve implementation and my optimized version of this program on Clemson's School of Computing machines via ssh. Specifically I used the Ada12 machine (OS = Ubuntu 16.04, Architecture = x86_64 i5, CPU Cores = 4 x 3.40GHz, Memory = 16GB). I chose to ssh into this machine because its resources are more abundant than my own laptop's. Therefore, I thought, the timing would be more consistent, as my computer could be stressed at any given moment due to the number of other programs/processes that are running on it. I edited my program in Atom, saved the file to my computer, used FileZilla to transfer it to my Clemson account on access.computing.clemson.edu, and used MobaXterm to ssh. For timing, I created my own function called get_time() which uses the timeval struct and gettimeofday() so that I can extract the time_in_micros using the following line of code [1]: unsigned long time_in_micros = 1000000 * tv.tv_sec + tv.tv_usec; . This made it easy to simply call get_time() before and after the matrix multiplication, subtract the two, and print the output in microseconds. Below is the entirety of my timing results in a table. Each time in microseconds (denoted us) was obtained by running that level of optimization on that size matrix *at least* 10 times, but quite often many more. Again, T1=loop interchanging, T2=register variables, T3=loop unrolling, and T4=blocking. The greatest speedup for each level of optimization is highlighted.

Technique / Matrix Size	**25**	**50**	**100**	**200**	**400**
Naïve + (-O0)	243 us	1943 us	7866 us	45623 us	337087 us
Transformed code with T1 + (-O0)	94 us	36 us	137 us	669 us	1235 us
$Time_{naive}/Time_{opt}$	2.59	53.97	57.42	68.20	272.94
Transformed code with T1 & T2 + (-O0)	90 us	24 us	100 us	487 us	862 us
$Time_{naive}/Time_{opt}$	2.7	80.96	78.66	93.68	391.1
Transformed code with T1 & T2 & T3 + (-O0)	26 us	16 us	12 us	246 us	81 us

$Time_{naive}/Time_{opt}$	9.35	121.44	655.5	185.50	4161.57
Transformed code with T1 & T2 & T3 & T4 (blk=16) +	14 us	93 us	885 us	4302 us	17408 us
$Time_{naive}/Time_{opt}$	17.36	20.89	8.88	10.61	19.36
Transformed code with T1 & T2 & T3 & T4 (blk=36) +	11 us	98 us	528 us	3428 us	11643 us
$Time_{naive}/Time_{opt}$	22.09	19.83	14.89	13.31	28.95

Clearly the greatest speedup time overall was achieved by T3 (interchanging + register variables + unrolling) on the largest matrix I tested (400x400). The absolute smallest multiplication time, however, was with T4 (blk=36) with the smallest matrix (25x25). A general trend that is easy to see here is that the largest matrix (400x400) typically had the highest speedup per level of optimization, despite also typically having the longest execution time per level. I think this is because the naïve implementation gets exponentially worse as the size of the matrix increases. So, while the execution time for T4(blk=36), matrix400x400 was 3.4 times longer than the time for 200x200 on the same level, the equivalent size matrix times on the naïve level were separated by a factor of 7.4! In other words, the naïve implementation is so bad once the matrices start getting large that the speedup for big matrices was bound to be the greatest.

4a. *A few timing trends that don't make sense:* 1) Some rows of the table above did not increase linearly with the size of the matrices (T1,T2, and T3…particularly with the 50x50 matrix). I think the only way to explain this is a possible increase/inconsistency in stress on and usage of the physical machine's resources to complete the computation. So for instance, at the times when I ran my 10 tests of the 25x25 matrix at both level T1 and T2, there were other programs running on Ada12 that were vying for resources and this caused the execution times to be unexpectedly large relative to the 50x50 matrix on those same levels. 2) Some columns did not *decrease* linearly with the number of optimization techniques applied (all columns except the 25x25 column). Every size matrix except the 25x25 matrix had a spike in execution time once it got to

the T4 level. I believe this was due to the corresponding spike in the complexity of my code in order to implement the T4 level. Other than these 2 anomalies, the overall timing trend was this: times increased the larger the matrix was and decreased the more optimization techniques were used (again, with the exception of T4).

References

[1] "Time in Milliseconds in C," *stackoverflow.com*, [Online]. Available: https://stackoverflow.com/questions/10192903/time-in-milliseconds-in-c [Accessed April 17, 2020].

[2] "Multi-dimensional Arrays in C," *tutorialspoint.com*, [Online]. Available: https://www.tutorialspoint.com/cprogramming/c_multi_dimensional_arrays.htm [Accessed April 17, 2020]

[3] "Loop Unrolling," *Wikipedia.com*, [Online]. Available: https://en.wikipedia.org/wiki/Loop_unrolling [Accessed April 17, 2020]

[4] "Loop Interchange," *Wikipedia.com*, [Online]. Available: https://en.wikipedia.org/wiki/Loop_interchange [Accessed April 17, 2020]

[5] "Understanding Register Keyword in C," *geeksforgeeks.org*, [Online]. Available: https://www.geeksforgeeks.org/understanding-register-keyword/ [Accessed April 17, 2020]

[6] "Optimizing Matrix Multiplication," *medium.com*, [Online]. Available: https://medium.com/binary-maths/optimizing-matrix-multiplication-35751b19da66 [Accessed April 17, 2020]