CS4473/CS5473 Lab03 - Group 8

Group 8: Jacob Baber, Mark Cunningham, Rachelle Phipps

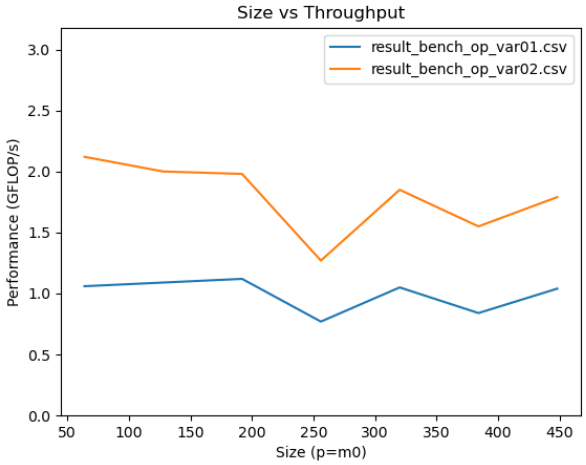

Fig. 1: Performance plot for If-Statement vs No-If-Statement

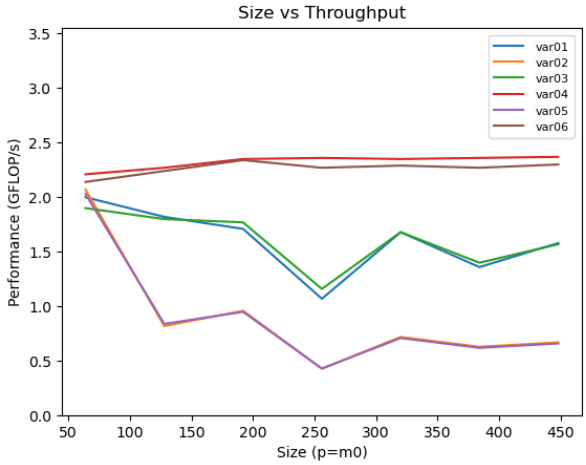

Fig. 2: Performance plot for Different Loop Orderings: IJP (var01), IPJ (var02), JIP (var03), JPI (var04), PIJ (var05), PJI (var06)

***Abstract*—This lab report explores the use of parallelism in Triangular Matrix Times Matrix Multiplication (TRMM). Our results find that reducing condition checking within the innermost loops along with tools like openMP and SIMD instructions yield a formidable return on organizing and processing data. Through creating multiple variants that incorporate several of these requirements, we are able to compare and optimize the performance of different operations.**

I. Loop Transformations

A. Removing the If-Statement

The if-statement contained in the baseline implementation is located in the innermost loop, causing a condition check that occurs very frequently. This if-statement can be removed by instead altering the bounds at which the second loop (i0) iterates through. Instead of i0 looping from 0 to m0, it only needs to loop until j0. Not only does this reduce the number of total iterations that the loops perform, it removes the if statement from a problematic location.

B. Loop Orderings

Rearranging the order that the loops were executed in proved to be beneficial. Figure 2 shows three major trends where each trend represents two loop orderings. JPI & PJI loop orderings performed the best out of all six, nearly quadrupling the performance of loops IPJ & PIJ when approaching larger matrix sizes. Both of these loops were noticeably more consistent than the other four loop orderings. IPJ & PIJ loop orderings performed the worst, while IJP & JIP performed in between the two other trends.

A simple explanation for the poor performance of IPJ and PIJ orderings is that the J-loop is the inner-most loop for these variants. Since j0 is multiplied by the row stride of the B array (m0) when accessing values from the B array, any variation of j0 can lead to accessing non-consecutive elements in memory, which in tern leads to poor cache locality. In the case of these two variants, j0 is changed at every iteration in the innermost loop, leading to poor results.

The IJP and JIP variants have a similar problem to the IPJ and PIJ variants in that they also exhibit sub-optimal cache locality. For the IJP loop, instead of the value of j0 changing at every iteration of the inner-most loop, it is changed every time the P-loop is completed, which happens every m0 iterations. The JIP variant has a similar problem but instead the value of p0 changes at every iteration.

The JPI and PJI variants performed the best because they display the best cache locality of the six loop orderings. Both loops have the I-loop as their innermost loop, which means that the j0 and p0 values are changed as infrequently as possible, as well as providing a more consistent memory access pattern when compared to the other four loop variants.

C. Blocking/Loop Tiling Part I

We chose to use the JIP variant for out IJ, IP, and PJ loop blocking. The results from these variants would likely be quite different if we were to choose another variant such as JPI. For each blocking variant, we chose to use sizes of 8, 64, and 128. The IJ, IP, and PJ blocking variants are performed similarly,

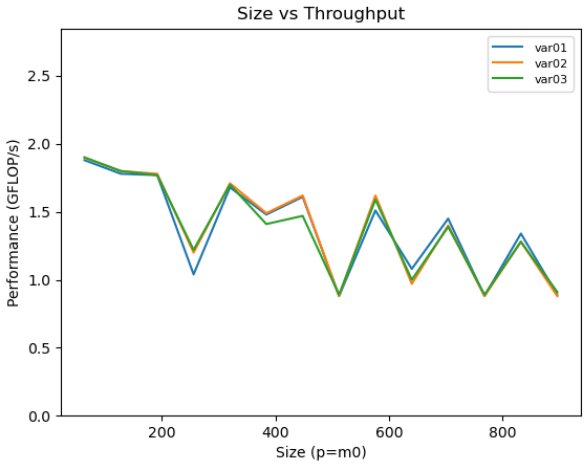

Fig. 3: Performance plot for (I,J) loops blocked at 8 (var01), 64 (var02), and 128 (var03) block sizes.

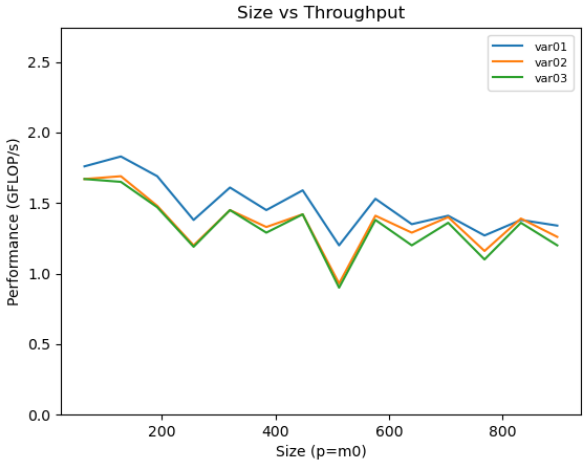

Fig. 4: Performance plot for (I,P) loops blocked at 8 (var01), 64 (var02), and 128 (var03) block sizes.

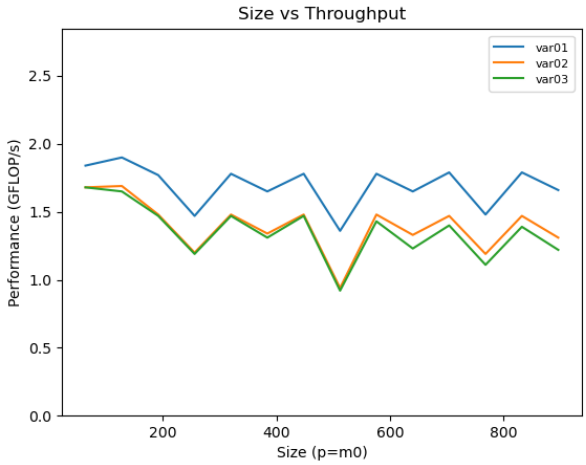

Fig. 5: Performance plot for (P,J) loops blocked at 8 (var01), 64 (var02), and 128 (var03) block sizes.

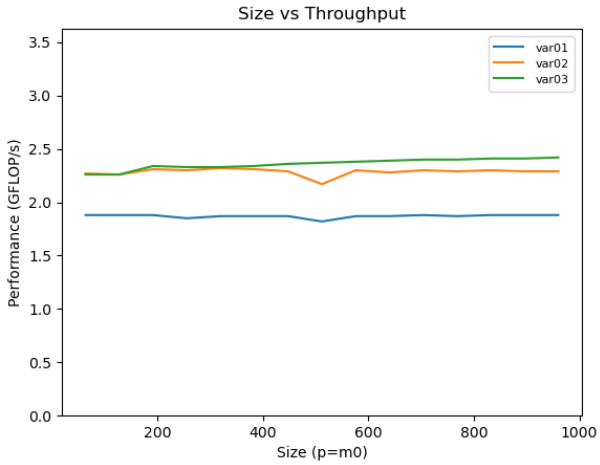

Fig. 6: Performance plot for all 3 (I,P,J) loops blocked at 8 (var01), 64 (var02), and 128 (var03) block sizes.

although IP was slightly better than IJ, and PJ was slightly better than IP. The processor that ran our tests had an L2 cache size of 256KB. After doing some math using the fact that a float occupies 4 bytes in memory, we figured that a square block size of 256 could fit in the cache. However, since there are two matrices, a block size of 128 would allow two blocks from two separate arrays to fit in the cache.

The IJ variant was the only blocking variant that showed virtually no change in performance between block sizes. This can be attributed to the fact that changing the block size does not change the memory access pattern in a meaningful way, since p0 will continue to be increment at every step of the innermost loop.

The IP variant showed a modest decrease in performance when increasing the block size that eventually tapers away at larger array sizes. When the block size is smaller for this variant, more I-loop iterations are performed over the same block of P-loop p0 values. This increases cache locality when compared to higher block sizes, where the value of p0 varies to a larger extent.

The PJ variant also showed a modest decrease in performance when increasing the block size that but does not taper away at larger array sizes. It has a similar memory access pattern to the IP variant, but keeps the same block of P-loop p0 values for a longer extent of time since they are also being blocked. For this reason, the performance does not decay as the array size increases.

D. Blocking/Loop Tiling Part II

When all three loops are blocked, the results are much more consistent across various array sizes, and the performance increases as the block size also increases. However, it doesn't take long before diminishing returns take effect. If the block

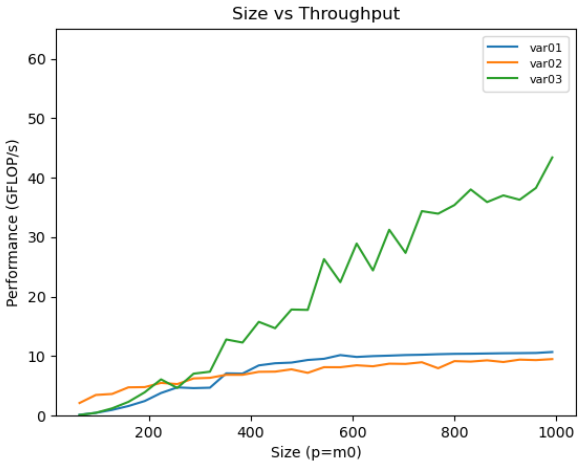

Fig. 7: Performance plot openMP, SIMD, and openMP+SIMD combined.

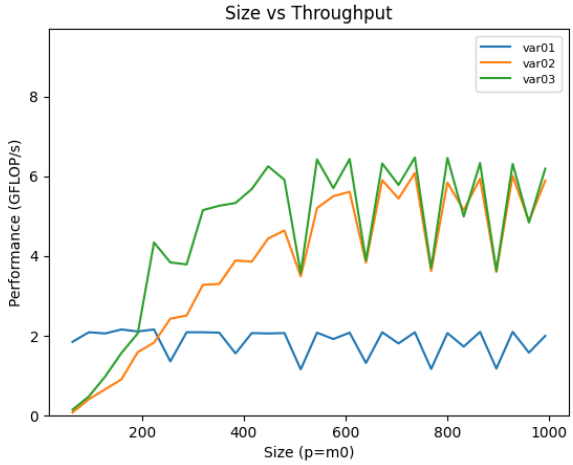

Fig. 8: Performance plot for different Mutex Techniques: Critical Section (var01), Locking (var02), and Reduction (var03).

size begins to approach one of the matrix dimensions, it basically defeats the purpose of blocking. This is why for array sizes less than 128, the 64 variant and the 128 variant have the same performance.

II. Parallelism in Isolation

Figure 7 shows our best variants including openMP with 8 threads, SIMD, and a combination of the two. We learned that openMP threads scale much better when used in combination with SIMD, but all are notable upgrades over the baseline.

A. Instruction Level Parallelism

Instruction level parallelism already exists within the code through the use of pipelining. Multiple instructions/operations are performed in a single cycle as the processor already implicitly and automatically leverages pipeline stages (Fetch, Decode, Execute, Memory access, and Writeback).

B. SIMD

To implement SIMD, our group used the blocked JPI loop ordering variant. The most difficult part of implementing it was determining where and how to implement the values along the diagonal vs not along the diagonal. We decided on placing an if-statement check the farthest from the inner-most loop as possible to determine whether or not range of I-loop values would be large enough to allow for SIMD to be used (8 needed for AVX2). Although an if-statement this far into the loop structure is undesirable, we were unable to implement it without it. We would have preferred breaking it into two separate for-loops, but did not succeed.

C. Shared Memory

1) Mutex

The critical section variant performed the worst and was the least scalable of the three techniques employed. Locking had the most consistent performance, but did not perform quite as

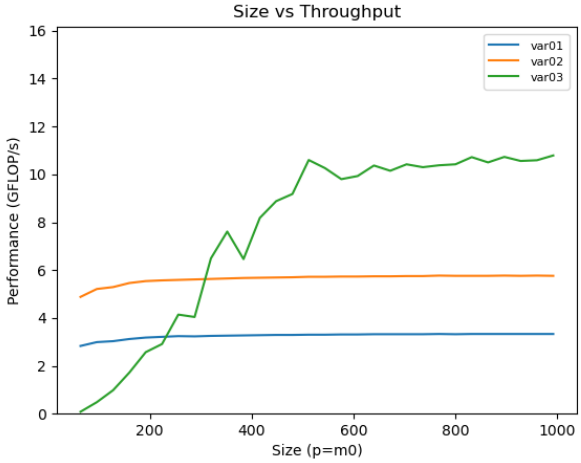

Fig. 9: Performance plot for 1, 8, and 16 threads using openMP parallel for num_threads

well as reduction for smaller array sizes. Reduction proved to perform the best, although it had the most variance of the three. Both locking and reduction appeared somewhat scalable.

2) Scalability

Since the number of tasks per node was set at 8 inside the parallel batch file, we set this as the upper limit to the number of threads to test. We tested using 2, 4, and 8 threads. The 2-threaded variant drops off quite quickly, and flatlines at around 3 GFLOPS/s. The 4-threaded variantly drops off slightly less quickly, and flatlines at around 6 GFLOPS/s. The best performing variant was the 8-threaded variant, which doesn't start to drop off until around size 512 or greater. It stabilizes at a much higher 12 GFLOPS/s. For all variants, it looks like once each thread is given enough work to fully occupy it, performance stabilizes at its peak.

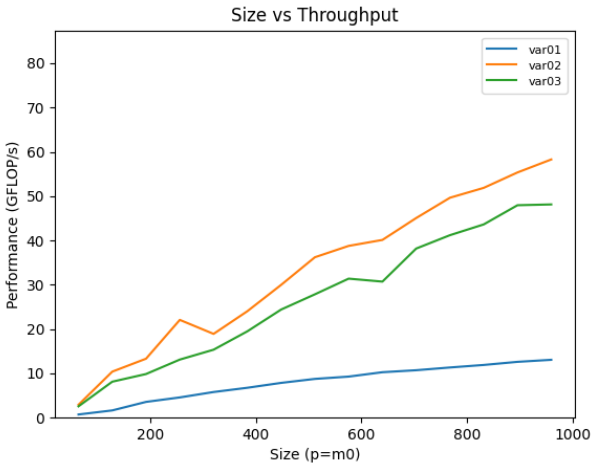

Fig. 10: Performance plot for Cuda GPU: 1 (var01), 8 (var02), and 16 (var03) threads per block and m0*n0 blocks per grid

III. GPUs

To parallelize the code using the CUDA framework, threads and blocks can be created to divide the work between multiprocessors of the GPU. Each thread within each block uses its thread ID to find which section of the code it should run. The thread finds its ID using the global variables blockDim, blockIdx, and gridDim. We used the code from our no-if statement JPI loop ordering. We only altered the J-loop, where a thread starts at j0 = id and is incremented by the block dimension multiplied by the grid dimension. This ensures that different threads process different elements, as long as the dimensions are large enough. For each of our variants, we used a constant number of blocks per grid equal to the size of the output array, and varied the number of the threads per block.

We found that a relatively low number of threads per block performed best for TRMM. We deduced that this is likely because threads within a warp perform best when they access contiguous memory locations, and our relatively simple JPI probably benefits from serialization within a block. Typically, more threads would be used to take advantage of the GPU, but in this scenario it was not necessary. A large number of blocks per grid improved performance the most. We believe this is due to each block of threads being able to work on different portions of the matrix independently, more effectively ”saturating” the GPU with work. Our GPU code is pretty basic and has many avenues for imporvement, but we were satisfied with performance demonstrated in Figure 10.

IV. Conclusion

TRMM requires careful consideration when implementing parallel techniques. The best performing CPU techniques were the combination of openMP and SIMD, while the best performing GPU techniques we tested were with 16 threads per block and the number of blocks per grid equal to the array size.