CSE260 Assignment 1 Report

Isamu Poy and Roshan Karande

Q1.

a) Performance of optimized code (based on running benchmark)

MC = 2016, KC = 224, NC = 64, MR = 12, NR = 4

N	Peak GFlops
32	5.4345
64	12.095
128	18.42
256	19.705
511	21.275
512	21.35
513	21.23
1023	20.8
1024	21.34
1025	21.355
2047	20.8
2048	21.025

Performance study:
Based on the 12 given performance value, it shows that the peak GFlops spike to larger values starting at N = 64. Interestingly, the performance stabilizes near 20 after 511 GFlops and stays there for larger values of N. The average GeoMean achieved for N >= 511 was 21.06 GFlops based on the benchmark.

b) Plotted performance of the three versions of code

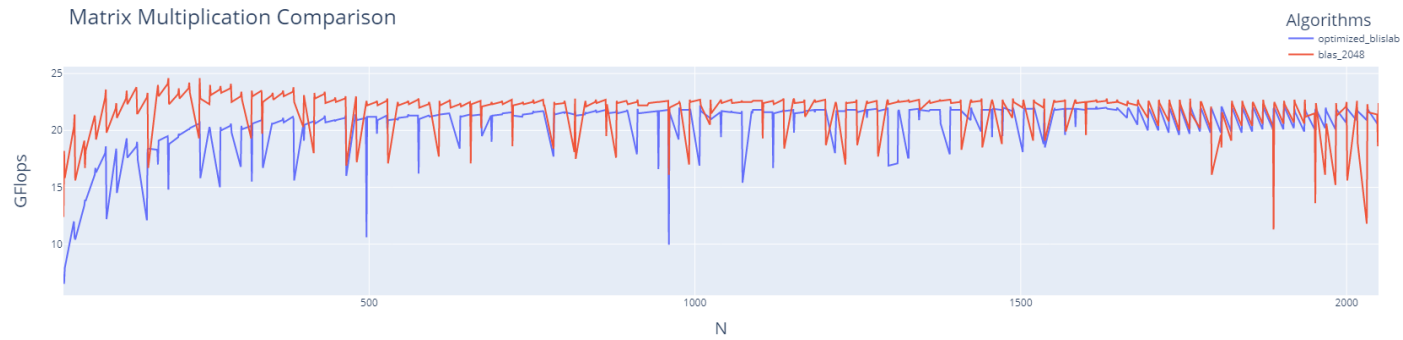

Figure 1: Performance of our optimized code and OpenBLAS

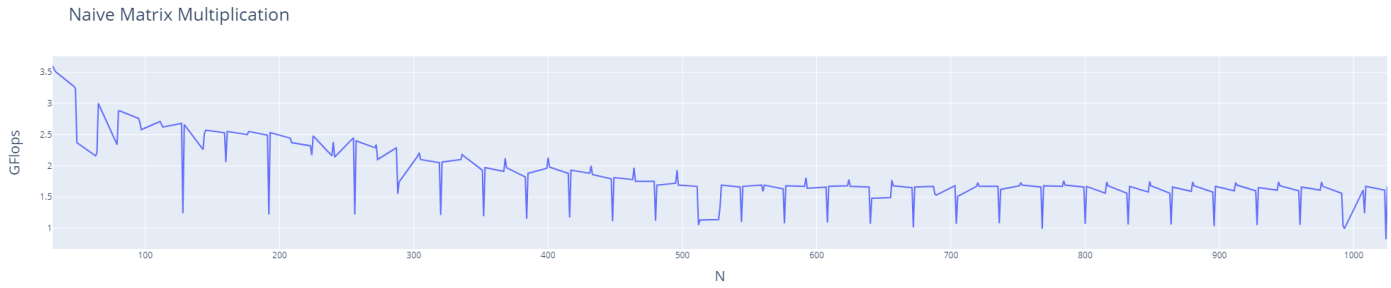

Figure 2: Performance of provided naive code

Q2.

a) How the program works via pseudo code

Figure 3 shows how the program works on a high level (each box representing a nested loop):

- At the beginning, matrix C and A are partitioned into blocks of m_c x n.
- A is partitioned into panels of m_r x k_c and B is partitioned into panels of k_c x n.
    - Each panel of A is packed into subpanels of m_r x k_c.
- Panels of B are packed into k_c x n_r subpanels and C is partitioned into m_c x n_c panels.
- Finally, within the last 3 nested loops, the outer product is applied on each packed subpanel of A and B to produce an m_r x n_r block of C which will append to itself each iteration to eventually form the final C matrix.

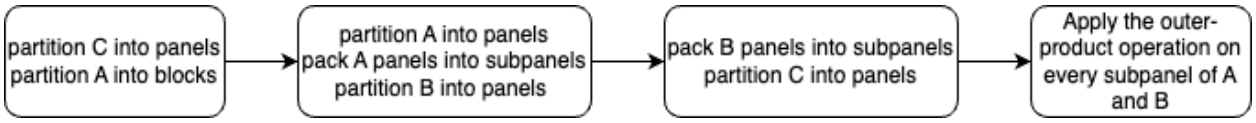

Figure 3: Order of operations for matrix multiplication with packing [2]

We have programmed two main components to BLISLab:

1. methods for packing matrices A and B into subpanels
2. microkernel optimization to make computation faster.

Packing Matrices

Packing has been implemented for matrices A and B. They have been separated into separate functions due to their differences in packing order, where matrix A packs horizontal subpanels and matrix B packs vertical subpanels. Additionally, we implement zero padding to consider cases where submatrices of A or B do not have divisible dimensions.

When the submatrix is passed into the packing function, the matrix pointer points to the index which is the first element to be packed in a subpanel (handled by the skeleton code). The following text represents the pseudocode for each algorithm:

Packing A

For matrix A, we consider a horizontal subpanel that is packed vertically, left-to-right order for a submatrix of A shown in Figure 4. In the following pseudocode, for-loops will be bounded by the dimensions ldXA (the leading index of matrix A), k_c, and m_r which are passed into the function parameters. [3]

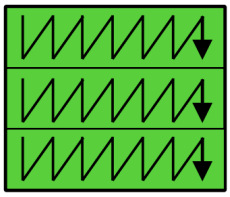

Figure 4: packing of A

- First, instantiate a vector which will contain the vectors containing pointers to the addresses for the first column of the first subpanel (called vector **aptr** for this explanation). To access a column we index submatrix A by the leading index ldXA.
    - The inputted value of **m** is used from the skeleton code to be the width of **aptr**. The value of **m** (the height of the subpanel) is the minimum of either m_r or the remainder after dividing N by m_r if it is not divisible.
- Next, we go through two nested loops. The outer loop iterates through the width of the submatrix of A, and the inner loop iterates through the height of each subpanel (m_r).
- The logic of packing is then implemented within the inner loop.
    - First, consider cases in which the submatrix dimensions were not divisible either horizontally (by k_c) or vertically (by m_r).
    - If the index is greater than the divided bounds of the horizontal or vertical dimensions of the submatrix, insert 0s to the pack vector and increment the pack pointer's index.
    - Otherwise, If the index is within divisible bounds, store the value of the pointed element into the pack vector, increment the pack index pointer's index, and then increment the value of the address being pointed to in vector **aptr.**
    - Since the matrix is stored in memory via row-major order, incrementing the address value will shift the pointer to the right. Thus, after every element of the

subpanel is done being packed, **aptr** will all have updated pointers pointing to the address of the adjacent column to the right.

- **Note:** We believed that this would be the most optimal way of addressing through a subpanel as it is relatively easy to understand, and avoids 2-dimensional indexing.

Packing B

For matrix B, we consider a vertical subpanel that is packed horizontally, top-to-bottom order in a submatrix of B shown in Figure 5. In the following pseudocode, the for-loops will be bounded by the dimensions ldXB (the leading index of matrix B), k_c, and n_r which are passed into the function parameters. [3]

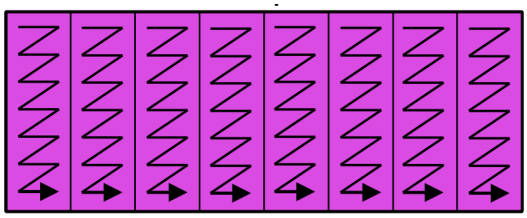

Figure 5: packing of B

- First, instantiate a vector which will contain the vectors containing pointers to the addresses for the first row of the target subpanel (called vector **bptr**).
    - Use the inputted value of **n** from the skeleton code to be the width of the vector. The value of **n** is the minimum of either n_r or the remainder after dividing N by n_r if it is not divisible.
- Next, we go through two nested loops. The outer loop iterates through the height of the submatrix of B, and the inner loop iterates through the width of each subpanel (n_R).
- The logic of packing is then implemented within the inner loop.
    - First, consider cases in which the submatrix dimensions were not divisible either vertically (by k_c) or horizontally (by n_r).
    - If the index is greater than the divided bounds of the horizontal or vertical dimensions of the submatrix, insert 0s to the pack vector and increment the pack pointer's index.
    - Otherwise, If the index is within divisible bounds, store the value of the pointed element into the pack vector, increment the pack pointer's index, and then increment the value of the address being pointed to in vector **bptr** by ldXb.
    - Since we want to pack the matrix vertically, we must add by the leading index of matrix B in order to access the memory address below the current address. After every element of the subpanel is done being packed, vector **bptr** will all have updated pointers pointing to the address of the adjacent row below.

Microkernel

For the microkernel, we eventually settled on using SIMD ARM SVE code to apply matrix multiplication via outer product. It works as follows:

- First instantiate SVE registers for matrices A, B, and C.
- For every index between 1 and k_c compute the accumulated row values of C

    - Every row of C is equal to C plus the outer product
- After the loop is done, store the register values back into the rows of matrix C
- The width of the microkernel matrix is limited to 4 because SVE vectors have a maximum length of 4. We use 4-width vectors to maximize register use.

b) Development Process

Change #1: After examining the skeleton code, different loop nestings were tested for naive matrix multiplication and an arrangement resembling an outer product produced the best performance shown in Figure 6. Thus using this implementation, packing was implemented on the panels of matrix A and B using the paper by Goto et al. as a reference. [2]
Commit #1:
https://github.com/cse260-fa22/pa1-ipoy-rskarande/commit/cc7a89324214bc968677c1be9911c3a786fd3eed)

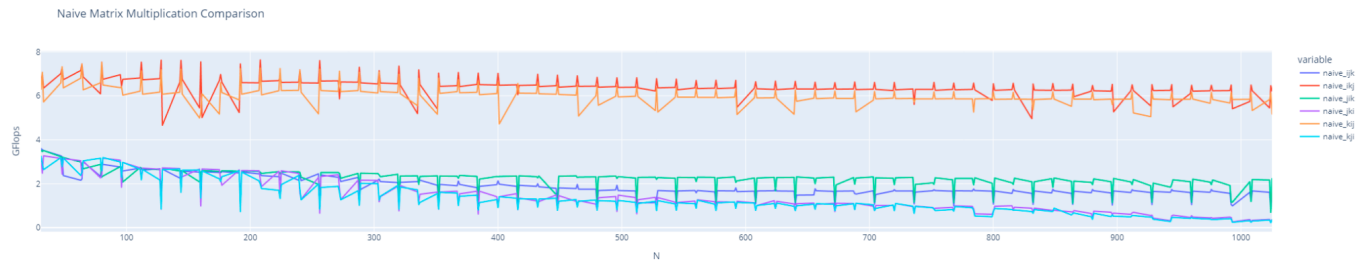

Figure 6: performance differences between different naive configurations

Change #2: zero padding was added to the packing algorithm. Based on Figure 7, the zero padded implementation (red) performed marginally better than the unpadded version (blue). This is likely due to the fact that padding ensures that only one pack is present per cache line.
Commit #2:
(https://github.com/cse260-fa22/pa1-ipoy-rskarande/commit/39131c22809a9b94dbc1719512fe472a31efde93)

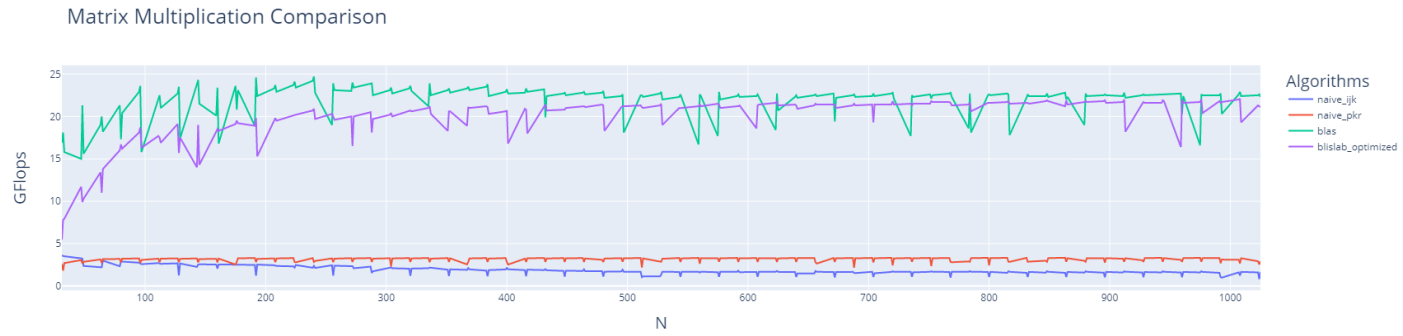

Figure 7: comparison of naive, naive with packing, blas and optimized blislab

Change #3: Zero padding was only implemented horizontally, and did not vertically per pack. This was fixed by adding another nested condition within the packing loop. Performance was marginally different, likely because most padding was done by the previous commit.
Commit #3:
(https://github.com/cse260-fa22/pa1-ipoy-rskarande/commit/39131c22809a9b94dbc1719512fe472a31efde93)

Change #4: SVE vectors were implemented to optimize the microkernel for $m_r = 4$ and $n_r = 4$. While it generated a significant boost in performance as shown in Figure 8 (for $m_r = 4$), the code did not work for values of n_c other than 4. This resulted in a deep dive in debugging the microkernel to see if there were any bounds issues.
Commit #4:
(https://github.com/cse260-fa22/pa1-ipoy-rskarande/commit/8f932e907b0ec57a1a2e05ed7c27831c3cf477e2)

Change #5: A bug was found in the SVE vector implementation where the bounds of each pack were accidentally set to DGEMM_KC rather than just the passed parameter of k. This allowed a better understanding of the difference between the passed parameter k and DGEMM_KC.
Commit #5:
(https://github.com/cse260-fa22/pa1-ipoy-rskarande/commit/0f1cb33f9669334d9bb0779146b1a18ceb17d8b2)

Commit #6: A non-square microkernel matrix (8x4) was attempted to see whether there would be boosts in performance. It resulted in a boost in performance as shown in Figure 8 (light blue).
Commit #6:
(https://github.com/cse260-fa22/pa1-ipoy-rskarande/commit/720e1e9be57ded472bfd08ca1e16988fc3aeac0e)

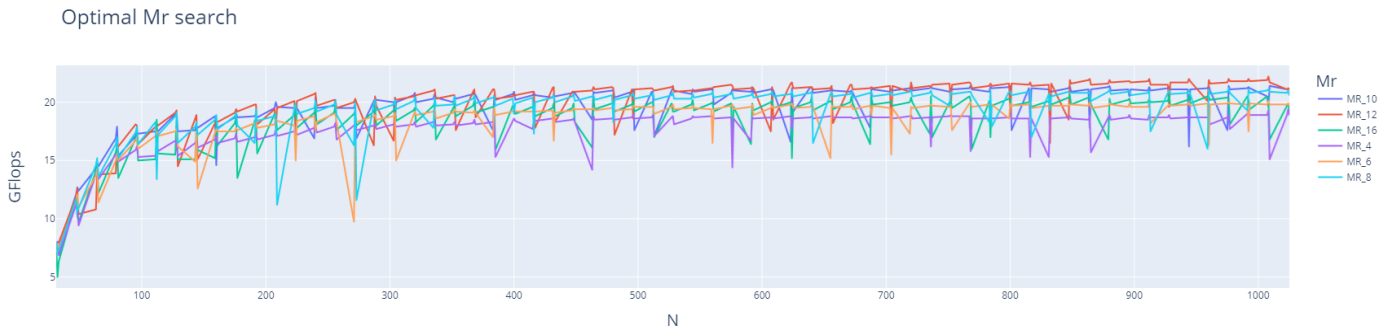

Figure 8: comparison of performance for different values of M_r parameter

Change #7: After trial/error and calculations, 12x4 microkernels performed the best. Any dimension greater than 12 resulted in a drop in performance as shown in Figure 8. This is likely because 12x4 provides the best tradeoff between register use and cache line memory.
Commit #7:
(https://github.com/cse260-fa22/pa1-ipoy-rskarande/commit/bbfbab0eb80418c8965da6b4fd616b734cdd7313)

Change #8: Loop unrolling was attempted for microkernel size of 8x4 to see any improvement. However, it hurt the performance instead resulting in significant drops in performance (Figure 9). This may be due to the ARM processor already loop unrolling in the backend.
Commit #8:
(https://github.com/cse260-fa22/pa1-ipoy-rskarande/commit/437994765a8b95c2e91693066219e93f2be80122)

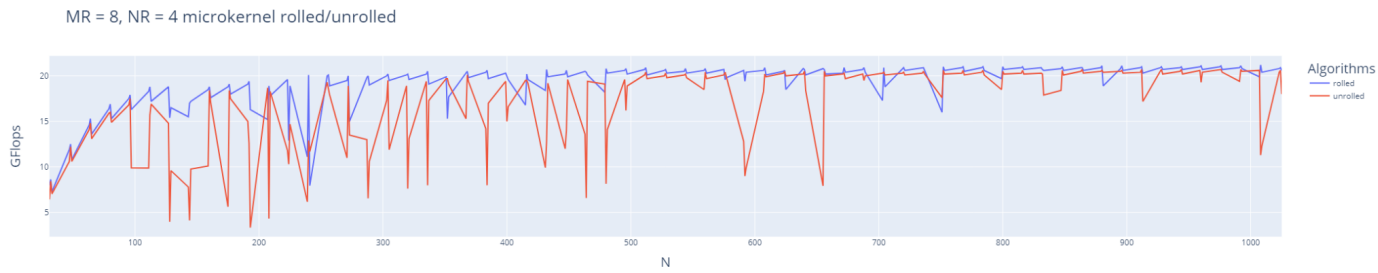

Figure 9: loop unrolled vs unchanged comparison for $m_r = 8$ and $n_r = 4$

Change #9: In order to find the best possible parameters, a script was created to generate a heat map of highest performance values shown in Figure 10. Since Low et al. mention that the value of n_c barely affects performance, the heatmap generated was a function of m_c and k_c. The results show that the best performance was concentrated near $k_c = 224$ and $m_c = 2016$ which were close to the computed values based on formulas by Low et al. [1]
Commit #9:
(https://github.com/cse260-fa22/pa1-ipoy-rskarande/commit/71d37ddc9e52985e47c4a894c98d788f19f29ab2)

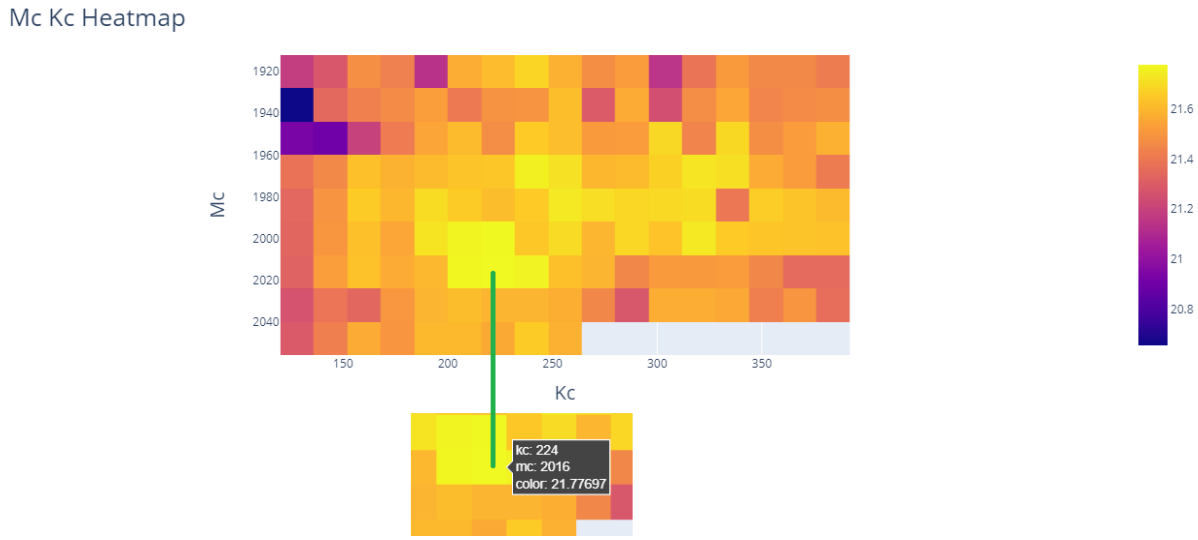

Figure 10: heat map of performance based on m_c and k_c

Change #10: for further optimization, restricted tags were added to the pointers being passed into the microkernel. As a result, the performance was marginally improved on average for select values of N. However, figure 11 showed that with small increments of N, the performance is actually, since it resulted in larger cache miss penalties. Thus, restricted tags were removed from the optimization. Perhaps there was unseen overhead in the restrict pointers tag.
Commit #10:
(https://github.com/cse260-fa22/pa1-ipoy-rskarande/commit/baa8aa3debaace214d53bb28c3507bdeb3afbaed)

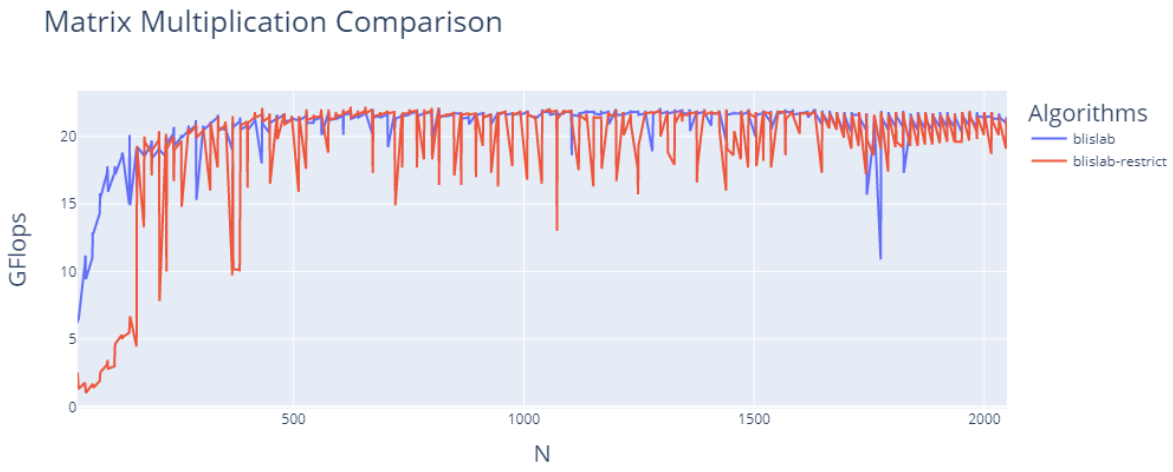

Figure 11: Comparison between restrict and no restrict pointers

c) High level irregularities in the data using graph from Q1b

A common irregularity among all graphs in Q1b were such that the performance was relatively low for small values of N <= 124. As N approaches higher values, the performance went up exponentially showing nonlinear behavior. This is likely due to the fact that the panels were set to sizes typically greater than 64, so size of N smaller than the panel sizes would result in less effective use of the cache lines.

Additionally, every graph also contained a series of drops in performance. These were likely caused by cache misses, where at some point there is a cache miss during matrix multiplication. The worse implementations contained larger valleys which hurt the average performance.

Figure 1 shows that openBLAS suffers with cache misses as N becomes larger, where the implemented optimization remains relatively good performing. Figure 2 shows that the naive implementation suffers as N starts to increase, most likely due to many cache misses.

d) Supporting data

<u>Finding parameters with “Analytical Modeling Is Enough for High-Performance BLIS”</u>

Using the formulas given by the paper, we initially found the “optimal” setting to be $m_r = 8$ and $n_r = 4$ based on a research paper . However, it was found that the configuration of $m_r = 12$ and $n_r = 4$ performed better on average. This discrepancy was likely due to the fact that the paper was based on x86 architecture, and not ARM Neoverse V1. Since our ARM processor involves a predicate computation for variable length vectors, this likely introduced overhead. [1]

<u>Parametric searches</u>

Through trial and error we found the following:

1. Parameter n_c has negligible impact on overall performance as described in “Analytical Modeling Is Enough for High-Performance BLIS” [1].

2. Increasing the number of SVE registers being used by increasing m_r improved performance to some extent. After $m_r = 12$, performance began to drop, likely due to oversized packs unable to fit in a cache line.
3. $m_r = 12$ was shown to have the best performing parameter based on Figure 8.
4. n_r was fixed to size 4 since it was the maximum size of SVE vectors.
5. Given $m_r = 12$ and $n_r = 4$ we computed estimated values of k_c and m_c (following the formulae described in the paper). n_c was fixed to 64 since it had negligible impact and was given as a default value. [1]
6. Computed m_c and k_c values ($m_c = 2048$ and $k_c = 170$) were used as a starting point on Figure 10 in order to locate the parameters of neighboring, highest heat values.

Analysis of cache behavior

In Figure 1, the optimized microkernel performed much worse for lower N values. Peticularly, they start at single digits for GFlop performance, and eventually go up to two digits as N increases. This is most likely due to the fact that for smaller values of N we are not using all of the space per cache line, which lessens the benefit of locality.

Figure 1 also showed that for higher values of N, openBLAS starts to suffer in performance due to cache misses (shown by sharp drops), where the optimized code did not suffer as much. Figure 2 shows that the naive code keeps dropping in performance with consistent sharp drops in performance, likely to resemble cache line misses.

e) Future work

In future work we can parallelize our microkernel even further by using both vectorization and threads. This would allow several microkernel blocks to be generated simultaneously while each SVE vector parallelism the computation of each microkernel. This can potentially be done using a pthread library if it is supported on ARM processors.

Additionally, one may deep dive into using OpenMP with SVE vectors. This would create another avenue of parallelism where nested loops loops can be parallelized via pragmas and vectorization can parallelize individual add/multiply operations. It is very likely that this would result in even better performance than without OpenMP threads.

f) Additional insight on the optimizations and how it affected performance

Loop Unrolling

Loop unrolling was attempted to improve the microkernel; For example, to see if it was worth implementing for higher microkernel dimensions, we unrolled our code by 1 loop for $m_r = 8$ and $n_r = 4$ dimensions. However, it was noticed that loop unrolling actually hurt our performance as shown in Figure 9. There are many drops in performance observed in the unrolled graph, which are likely caused due to the processor over-optimizing. Additionally, it is possible that this causes the instruction cache to overflow, resulting in cache line misses.

SVE vectors

Using SVE vectors to implement outer product matrix multiplication significantly boosted the overall performance. SVE vectors essentially parallelize the outer product computation of the resulting row vector. As a result, this saves resources from extra for loops.

Tradeoff between taller microkernel and more registers

Since using SVE commands boosted performance, it was initially thought that using more registers would result in more GFlops/s. However, it was found that after a certain point, using more registers actually hurt performance. It can be thought that there is tension between the cache line size and the number of registers being used, where the cache line (size 64 bytes) will limit how many registers can be effectively used for optimization. [4]

Restricted pointers

Using restricted pointers prevents aliasing between passed pointers, in which they update the same location. By restricting pointers the implemented SVE kernel underwent a boost in performance by approximately 1 GFlop/s for select values of N. However, for fine increments of N, it was observed that the performance suffered after enabling restricted pointers.

Q3. References

[1] Low, T. M., Igual, F. D., Smith, T. M., & Quintana-Orti, E. S. (2016). Analytical modeling is enough for high-performance blis. *ACM Transactions on Mathematical Software*, *43*(2), 1–18. https://doi.org/10.1145/2925987

[2] Goto, K., & Geijn, R. A. (2008). Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software*, *34*(3), 1–25. https://doi.org/10.1145/1356052.1356053

[3] Flame. (n.d.). *Flame/Blislab: BLISlab: A sandbox for optimizing GEMM*. GitHub. Retrieved October 16, 2022, from https://github.com/flame/blislab

Q4. Extra Credit

a) OpenMP optimization

We tried doing OpenMP optimization for naive implementation and were able to see significant improvements as we increased the number of threads. However, these pragma directives did not improve the performance around microkernel / macrokernel loops. Perhaps, the code has to be looked at further. Repo location - **/bonus/openmp**

b) AVX2 optimization

We have implemented a 4x4 microkernel for AVX2 and have tested it separately. However, integrating it with the code needed changes in the makefile etc. As a result, we have created a separated folder **/bonus/avx** in the repository where our implementation resides. We also test it for a few matrices in it and the results are correct.