CSE260 Fall 2022

Homework 1 - Matrix Multiplication

Aayush Gupta (A59018415) and Sourav Ravikumar (A59012195)

University of California San Diego

October 17, 2022

Abstract

In this assignment, we aim to develop an optimized matrix multiplication routine that performs faster than naive matrix multiplication. We explore efficient cache utilization, structuring of data and utilization of the underlying machine architecture and their effects on performance. We present our results and a study on why our implementation works much faster than naive matrix multiplication.

1 Introduction

Matrix multiplication is by far the most studied algorithm in high performance computing. The problem is known for its high "parallelizability" because of the way the data is structured and the arithmetic that happens. It is an important kernel in many real-world problems and can quickly become a bottleneck if not implemented optimally. In this report, we demonstrate why the naive multiplication algorithm has much room for optimization. We first show the results of our implementation and then describe how it works in detail.

2 Scope for optimizations in the Naive Algorithm

Consider the simple matrix multiplcation problem of multiplying two matrices A and B, to obtain the result C. The naive matrix multiplication algorithm loads different columns of B over and over for every row of A. This adds to the memory overhead and reduces computational efficiency. The majority of the execution time of matrix multiplication does not come from the number of multiplications and additions, but rather from the uncached memory accesses. Reading a number from the main memory can be 100 times more expensive than a multiplication operation. Therefore, rearranging the order of operations, such that the portion of the matrix currently being used sits in faster cache memory, can itself lead to an improvement. An efficient way of "packing" the matrices so that cache misses are minimized, is described later in the report.

Another optimization that is possible is parallelization: The latency of a multiplication operation cannot really be reduced, however, the throughput - the number of multiplication operations per unit time, can be drastically increased by computing them parallely. SIMD instructions can be used to perform multiple floating point operations just as fast as a single floating point operation.

In this assignment, we explored all of the above optimzation techniques and more. The final optimized implementation employs better cache utilization, SIMD instructions and register tiling. We present our findings, compare the different algorithms and describe the optimizations in more detail.

3 A summary of the GotoBLAS method

The Goto approach first divides matrices A and B into smaller blocks which can fit in different levels of cache. If blocks are in cache, they can be accessed faster and it improves the performance of matrix multiplication. For this, different parameters are defined namely M_c, K_c, N_c, M_r, N_r and they determine how matrices A and B will be divided.

Matrix A is broken into strips of height M_C so that each strip has size M_c X k. Each such strip is then broken further into panels of width K_c such that each panel has size M_c X K_c. At the same time matrix B is also broken into panels of height K_c resulting into panels of size K_c X n.

Then the panels in A are packed into sub-panels of size M_r X K_c. Packing is done in a way that mimics how the micro-kernel accesses A for computing the outer product. So, for any element in panel of A, the next element that the micro-kernel will access, comes sequentially after the current element in the packed sub-panels. This reduces the loading time as nearby elements can be found in the cache (spatial locality).

Panels in matrix B of height K_c are divided into columns of width N_c and each of them is further packed into smaller sub-panels of size K_c X N_r. Just like A, elements of B are packed such that the order in which the micro-kernel accesses elements of B are placed in a sequential manner.

Blocking is structured in such a way that the blocks and panels of matrices A and B can fit in different levels of cache. For instance, a block of A of size M_c X K_c should fit in L3 cache, a block of B of size K_c X N_c should fit in L2 cache, a panel of A of size M_r X K_c should fit in L1 cache and so on.

Then the approach iterates over sub-panels of A (of size M_r X K_c) and sub-panels of B (of size K_c X N_r) and calls a highly optimized micro-kernel to multiply these sub-panels. If data is present sequentially in these sub-panels, elements required for multiplication are in cache most of the time. Accessing these elements becomes faster and memory overhead is reduced, resulting in better performance.

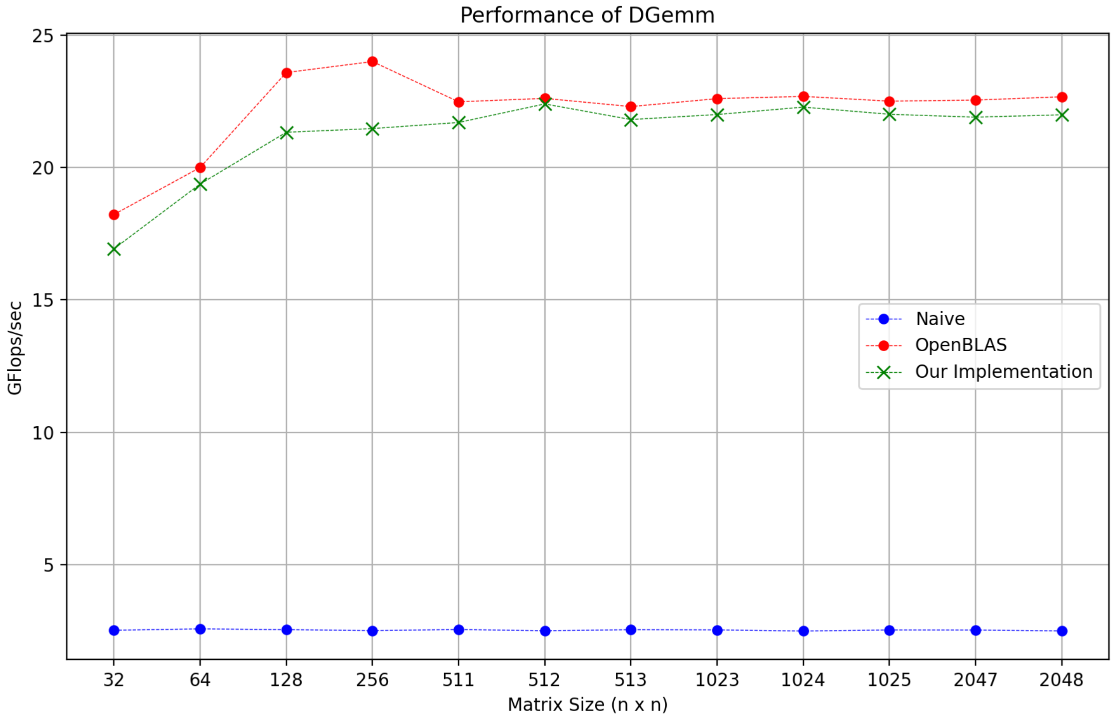
4 Results (Q1)

The results of our implementation are summarized below. We compared the most optimized version of our implementation with OpenBLAS and a naive DGeMM kernel for a variety of square matrix sizes, as shown in Table 1. To summarize, our implementation achieved an average of 21.26 GFlops/second. This is only 4% slower compared to OpenBLAS' average performance of 22.1 GFlops/second. Note that these results have been obtained on a Grav3 AWS instance (c7g.medium).

Table 1: Performance Comparison

N	Peak GF (Our code)	Peak GF (OpenBLAS)	Peak GF (Naive)
32	16.93	18.225	2.540
64	19.375	20	2.535
128	21.33	23.585	2.551
256	21.47	24	2.569
511	21.7	22.48	2.51
512	22.4	22.61	2.53
513	21.81	22.3	2.571
1023	22	22.6	2.495
1024	22.28	22.685	2.55
1025	22.01	22.505	2.532
2047	21.90	22.545	2.504
2048	21.99	22.67	2.52

Figure 1 shows this comparison as a graph. It can be seen that the naive matrix multiplication algorithm hits a plateau very soon. The other two algorithms, though faster, hit a plateau for square matrices of size 500 and bigger. For smaller matrix sizes, the two algorithms are relatively slower but still leaps ahead of the naive implementation.



Performance Comparison of Final Implementation

5 Analysis (Q2)

5.1 How does the program work? (Q2.a)

The program takes two matrices A and B as input. Then these matrices are divided into smaller blocks (defined by M_c, K_c and N_c) and sub-divided into panels and sub-panels (defined by M_r and N_r) as described in Section 3 (summary of GotoBLAS method).

Finally, the micro-kernel performs an outer product of the two sub-panels of A (M_r X K_c) and B (K_c X N_r) and computes an M_r X N_r sub-matrix of C. This computation is done using SIMD instructions provided by SVE for ARM architecture.

Using SIMD, one row of sub-panel of B is loaded into an SVE register. In our case, the $VLEN$ or vector length of the machine was 4. This meant that one can store upto 4 doubles in one register. Then an element of sub-panel of A is loaded and broadcasted to fill one whole SVE register and then both of them are multiplied to obtain a partial result which is also stored in an SVE register. This whole process is done K_c times for different elements of A and B and final partial result is stored back in C. By using SIMD instructions, we can heavily parallelize the floating point operations.

Calling the micro-kernel for all sub-panels creates matrix C - the result of matrix multiplication between A and B.

5.2 Development Process (Q2.b)

We now describe the development process with incremental optimizations. The first two versions only focus on cache friendly data organization which reduces the time the CPU spends on fetching data from main memory. The later versions focus on actual parallelization where multiple operations happen in parallel at the same time.

5.2.1 Version 1 - Packing

The first optimization that was made was packing. By "packing" blocks of matrices in a cache friendly way, the program would spend much less time fetching data from the slower main memory. Implementing packing was rather straightforward, as it is only re-ordering a 2D block of data into a flattened array. However, the order of flattening is different for the two matrices A and B. In A, we flatten by going "down the column", and in B, we flatten by going "across the row". The validity of matrix multiplication was checked after implementing packing - though it was working for matrices of sizes which are multiples of the packing parameters, it resulted in incorrect multiplications for other cases.

When the matrix size, n, is not a multiple of packing parameters, some panels and sub-panels have height and width smaller than the packing parameters. Such panels are not packed properly. To handle these cases, padding was implemented. Panels with height and width less than the packing parameters were padded with zeros to make their height and width equal to the packing parameters. This way the matrices, whose sizes are not a multiple of packing parameters, were also packed correctly and resulted in correct matrix multiplication.

This version can be found against the commit SHA $a89e394$ in the github repository.

5.2.2 Version 2 - Loop unroll and registers

Just implementing packing did not cause any significant performance increase. This was surprising because packing, in theory, should reduce the number of loads and stores from the main memory. Even with some other optimizations like loop unrolling in the packing routines, direct addressing in pointers, performance did not improve by more than 1 GFlops/sec. This was because even though packing was implemented, the micro-kernel had not been changed to take advantage of packing.

Therefore, the values of M_r and N_r were fixed to 4 each, and the inner two loops of the microkernel were unrolled. The matrix multiplication arithmetic for a 4x4 matrix was laid out in sequential instructions. Another major change that was implemented was using registers to store the intermediate values of $C[i, j]$. Since we are performing an outer product and then adding the partial sums, accessing $C[i, j]$ from memory can become expensive. These modifications resulted in a significant performance boost as seen in Figiure 2.

This version can be found against the commit SHA $6b70a3$ in the github repository.

5.2.3 Version 3 - SVE instructions

Now that the memory aspect of matrix multiplication was optimized, actual parallelization was added. The micro-kernel was optimized using ARM SVE instructions which allows register tiling and vectorization. In essence, it allows multiple floating point operations to happen in a single shot by vectorizing the registers. This way multiple elements were loaded into registers at a time and operations were performed on all loaded data in one go. By doing so a big performance jump was observed with an average value of 16 GFlops/sec.

In order for the algorithm to be compatible with any matrix size n, predicates were generated using the ARM SVE library function which allows acccurate vector operations even when the entire vector is not filled out. Doing so handled the fringe cases.

This version can be found against the commit SHA $4d9b1b3$ in the github repository.

5.2.4 Version 4 - Parameter tuning

The final version of our implementation optimized the blocking parameters to achieve a high GFlops/sec rate. The inner loop of micro-kernel was completely unrolled to a bigger M_r value to utilize the maximum number of SVE registers possible. Along with this, the parameters M_c, N_c, K_c, M_r, N_r were tuned to get better performance. The cache sizes of the machine were kept in mind while tuning these parameters. A rudimentary grid search was written which finds optimal values of the blocking parameters. Table 2 shows the final optimal parameters that the grid search landed upon. The final optimized version saw a significant bump up to a peak of 22.4 GFlops/second.

Table 2: Optimal blocking parameters

Parameter	Optimal Value
K_c	256
M_c	256
N_c	512
M_r	16
N_r	4

This version can be found against the commit SHA 56429$d7$ in the github repository.

The result of each of the intermediate versions is summarized in Figure 2. The first version, which implemented packing, obtained a peak GF rate of 3.4; the second version with register usage and unrolled microkernel obtained a peak of 9.6; the third version with SVE instructions and vectorization obtained a peak of 17.5. The final version stands at a peak of 22.4 GFlops/second.

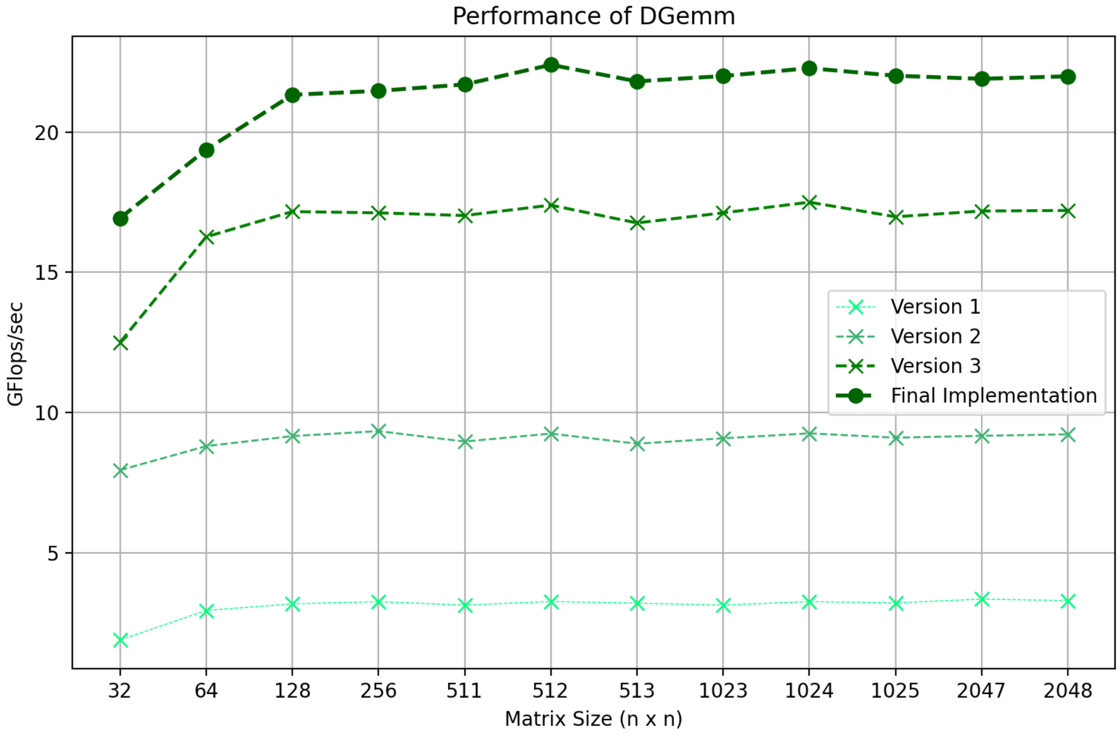


Figure 2: Performance of Intermediate versions

5.3 Irregularities in data (Q2.c)

According to the graph in previous section, performance keeps increasing up to n=512 and then decreases a bit for n=513. Same behaviour can be seen for n=1024 and n=1025.

When n=512 or 1024, packing perfectly divides the matrices into panels and sub-panels because n is a multiple of the blocking parameters $(M_c, K_c, N_c, M_r, N_r)$ and so, registers and caches are properly utilized.

On the other hand, when n=513 or 1025, packing creates some extra panels and sub-panels which are only partially filled with useful data and rest is padded with zeros. Due to this, some resource goes into storing these padded zeros but this does not add to any computation thereby lowering the number of GFlops/sec.

5.4 Supporting Data (Q2.d)

In our Amazon EC2 GRav3 instance, L1 cache was 64KiB, L2 cache was 1MiB and L3 cache was 32MiB.

The search space for blocking parameters was such that the blocks, panels and sub-panels of A and B can easily fit into different levels of cache keeping in mind that other processes on the machine are also using these caches. A grid search was done in this restricted space to arrive at the right set of parameters that give the best performance.

The machine level statistics were also captured during the run of our algorithm. The $perf$ command was used to achieve this. Table 3 shows this. Our implementation uses less than half of the CPU cycles that the naive algorithm uses. The overall number of branch-misses is also almost 10 times less in our implentation.

Table 3: Output of perf command

	Naive	Our code
task-clock	130533.52 msec	56531.00 msec
context-switches	5704	3590
cycles	337800506806	145809249028
instructions	2439215215621	351168465786
branch-misses	170701546	19612668
time-elapsed	133.246388284 seconds	59.042698511 seconds

On a different note, the skeleton code here was organized in a different way than the BLISlab tutorial because in our implementation, A is stored as a row-major matrix; whereas in BLISLab matrix A is stored in column-major format. When A is column major, the loop over A should be the inner loop because a column of A will need a particular row of B over and over while computing the outer product. Since the column is contiguous in memory it can be loaded in a cache friendly manner.

On the other hand when A is row-major, having the loop over A inside has no benefit, like in previous case, because no column of A is contiguous in memory. That is why, here the loop over A is the outermost loop and loop over B is inside. Since rows of B are contiguous in memory they can be accessed faster.

5.5 Future work (Q2.e)

- Experiment with the 4X4 rank-1 Update i.e. Butterfly permutation, to check if that speeds up the program.
- Parallelize the program further by using OpenMP.
- Write the same algorithm using AVX2 to support this program on x86 architecture.

5.6 Additional insights (Q2.f)

While we measured the performance of the various implementations using the metric of GFlops per second, it does not practically capture the performance in a human-understandable way. Therefore, an additional study was performed with the metric of time. The time taken to perform matrix multiplication in each implementation was measured and averaged over multiple iterations.

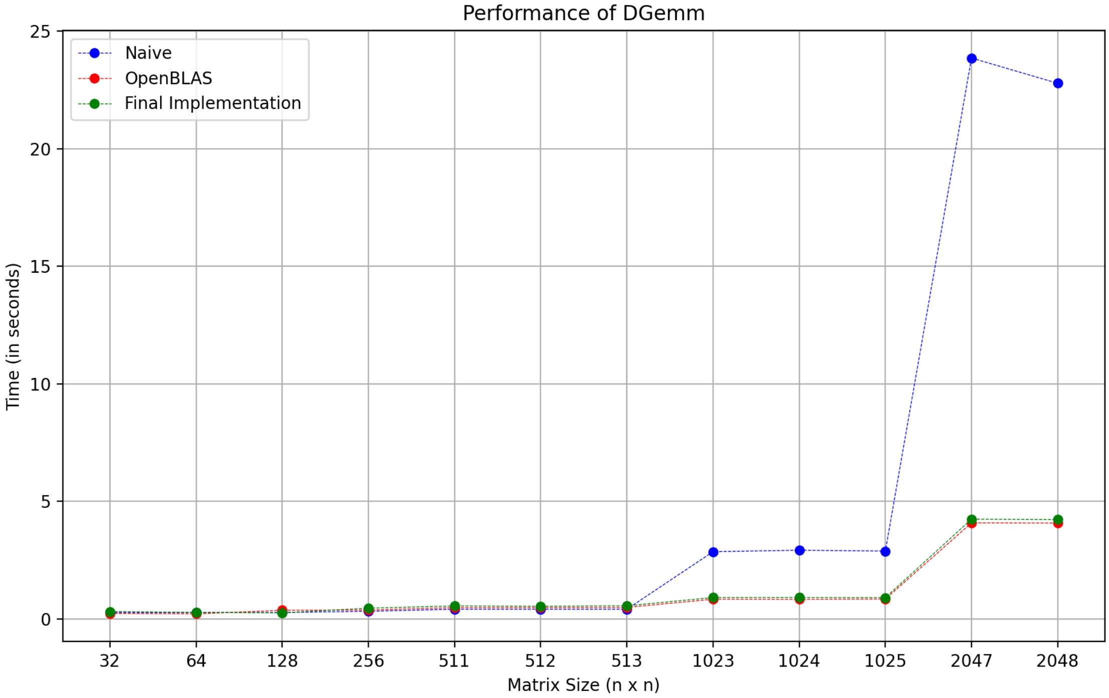


Figure 3: Performance in Time Taken

An interesting observation was made - though the naive code only has a peak GFlop rate of 2.5 and our final implementation has 22.4, the amount of time taken for matrix multiplication is more or less similar for matrices of size upto 512. This is shown in Figure 3. It is only for $n > 512$ that we can see a real reduction in time taken.

The reason for this might be as follows: for smaller matrices, the entire matrix A and B might be able to fit in the cache of modern machines. Therefore, naive multiplication performs as fast as our implementation with packing. However, for larger matrices, they might not fit directly in the cache and the naive algorithm faces a high cache miss rate. This is where packing and SVE instructions kick in and provide faster matrix multiplication results.

6 References (Q3)

1. Jianyu Huang and Robert A Van de Geijn. 2016. BLISlab: A Sandbox for Optimizing GEMM. FLAME Working Note #80, TR-16-13. The University of Texas at Austin

2. Anatomy of high-performance matrix multiplication. Kazushige Goto, Robert A. van de Geijn. ACM Transactions on Mathematical Software (TOMS), 2008.

3. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. Field G. Van Zee, Robert A. van de Geijn. ACM Transactions on Mathematical Software (TOMS), 2015.

4. OpenBLAS, an optimized BLAS library. http://www.openblas.net.

5. T Low, F Igual, T Smith, E Quintana-Orti, “Analytical Modeling is Enough for High-Performance BLIS”, ACM Transactions On Mathematical Software, Vol 43, No. 2, August 2016