Part 1: Matrix-Matrix Multiplication

** Note: All experiments are done on a 4k by 4k matrix for part 1 + 2

Tiling:

Intuitively, it makes sense that the best results would come from tiling all three dimensions. This approach allows for the reuse of data in the cache for matrices A,B,C which minimizes cache misses and prevents cache overflows for all three matrixes. So when I decided to optimize the tile size, I ran the algorithm with all three dimensions blocked. In 1D tiling, it is most beneficial to optimize for the L1 cache to minimize cache misses. I benchmarked the GEMM for various tile sizes; the results can be seen in the graphs below. It is important to note that the x axis is in the log scale so its easier to interpret the data.

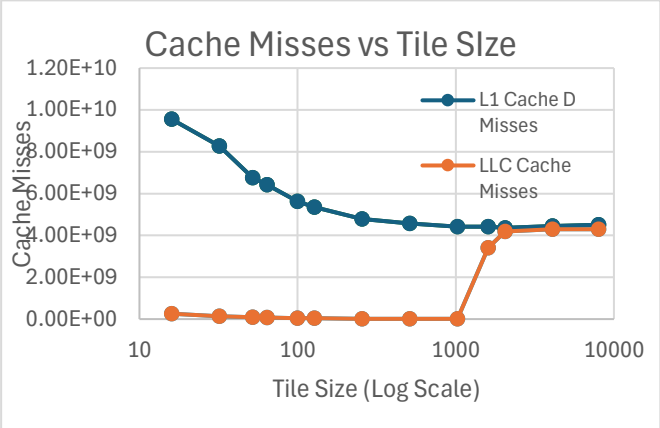

Figure 1 Tile Size vs Cache Misses for gemmT1

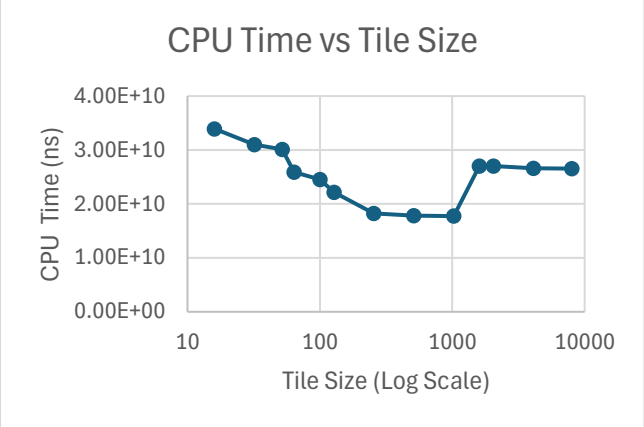

Figure 2 CPU Time vs Tile Size for gemmT1

In the graphs, increasing the tile size initially leads to a reduction in CPU time and L1 cache misses. During this phase, the tile size remains small enough to fit within the L1 cache (no cache misses), and there are no LLC cache misses. However, once the tile size reaches around 1000, the trend shifts. The number of LLC cache misses starts to increase, while L1 cache misses plateau. At this point, the CPU time also begins to rise. This behavior occurs because the tile size has grown too large to fit within the L1 cache, leading to overflow into the lower-level caches, such as the LLC. LLC cache misses are costly since they require data retrieval from main memory, resulting in a significant increase in CPU time. Thus, the sharp rise in CPU time is a direct consequence of LLC cache overflow as we increase the tile size past 1000.

From this, I know that a tile size of 1024 will result in the best performance. Analytical this makes sense as well. Based on the system's specifications, I know that the L1 instruction and data cache is 32 KiB per core. I know that the matrix elements are floats, which are 4 bytes each. To determine the maximum tile size that can fit into the L1 cache without overflow, I calculated:

$$\frac{32KiB}{4\ Bytes\ x\ 3\ Matrixes} = 2700\ tile\ size$$

The analytical value of 2700 and the observed **optimal tile size of 1024** are within a similar range. They are slightly different due to the tag overhead etc. With that in mind, for 2d level tiling, I kept the optimized loop order from gemT1 as it exploits the spatial locality of the matrixes. In 2-level cache tiling, the general algorithm fetches a block of the matrix into the L2 cache (using outer loops), and that block stays in L2 while smaller pieces of it are fetched into L1 cache (inner loops) for computation.

I know the L2 tile size needs to be larger than the L1 tile size since L2 cache is larger and is used in the outer loop. I kept the L1 tile size fixed at the optimized size from gemmT1 (1024) and increased the L2 cache tile size in powers of 2, starting from 1024. The ideal L2 tile size fluctuated between 2048 and 4096, with the smallest CPU times observed in this range due to run-to-run variation. Beyond 4096, L1 cache misses increased, even though LLC cache misses remained constant. This increase in L1 misses resulted in longer CPU times. This can be seen in the graphs below:

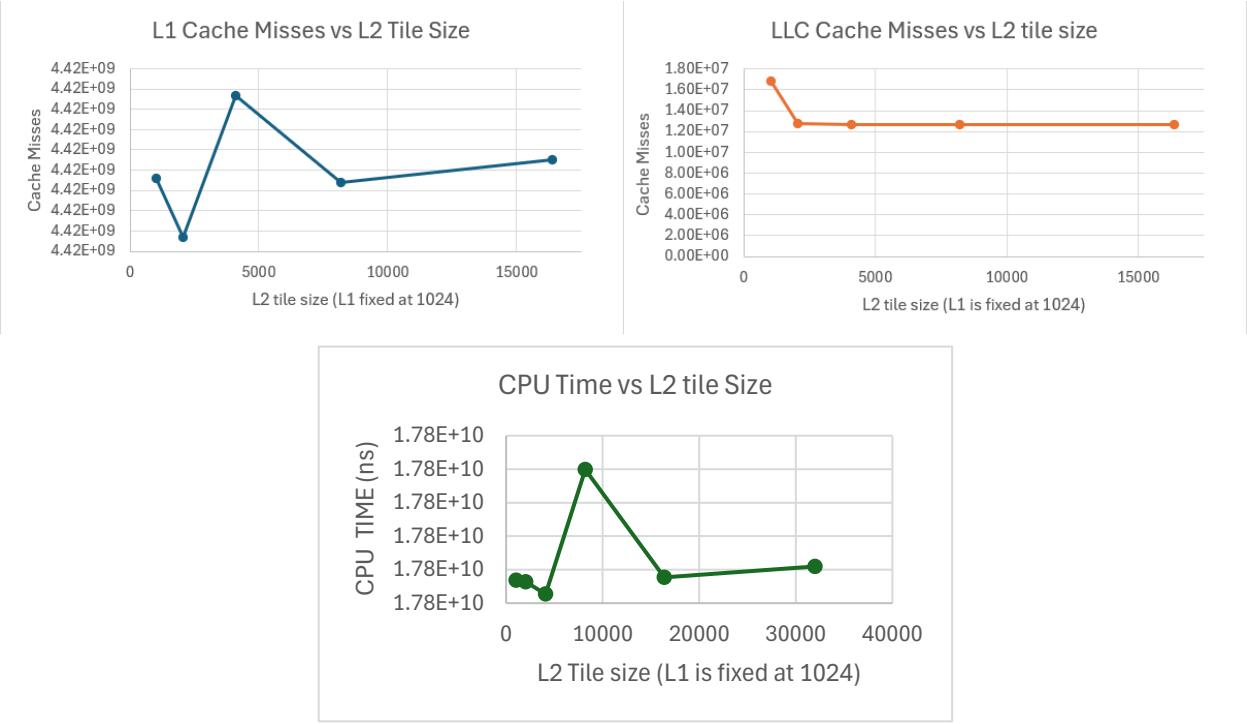

Figure 3 Run time Information about gemmT2

The reason L1 misses increase, even though the L1 tile size is constant, is that as the L2 tile size grows, more data is processed in each outer loop. Thus, this will increase the amount of data the inner loops (L1 cache needs to process), placing more pressure on the L1 cache. This leads to more frequent evictions and reloads in L1, causing cache misses. With this in mind, the L2 tile size should not be significantly greater than the L1 tile size.

To address this, I experimented with increasing the L1 tile size to better handle the larger L2 tile size. However, I found that as I increase the L1 tile size, the L1 cache misses were still higher than when the L1 tile size was 1024. This is because if the L1 tile size is too large and exceeds the cache capacity, it leads to more frequent evictions and reloads. Thus, the optimal tile size for gemmT2 is **L1 tile size = 1024 and L2 tile size = 2048 – 4096**. There was minimal improvement in CPU time from T1 to T2 as the

L1 cache is already optimized so adding a second level of tiling for the L2 cache doesn't provide further benefits, especially since L1 cache misses account for the majority of total cache misses. This is further proven as the L1 and LLC cache misses are almost identical for both implementations.

Vectorization:

To further improve the program performance, vectorization is used to take advantage of SIMD capabilities in CPU and process multiple pieces of data simultaneously. The AVX instructions are used to vectorize the program and there are 16 YMM Registers available.

From previous experiments we found that the best tile size is 1024 for T1 and the cpu time performance between T1 and T2 is small, so we decide to vectorize the T1 code.

In matrix multiplication, we learn that the interchange loops (from row-col to row-row) have a relative speed up of 6.50(from Matmul slide) due to the spatial locality, so the idea of our vectorization is to still maintain the good usage of spatial locality. To preserve this spatial locality, the row loop was unrolled by 4 within each iteration. This technique allows the AVX registers to compute 8 numbers per row, increasing the total number of floating-point operations per loop iteration from 1 to 32 (4 rows × 8 columns). The use of AVX registers minimizes cache misses as it improves the spatial locality and reduces the frequency of the memory access by processing multiple data points at once.

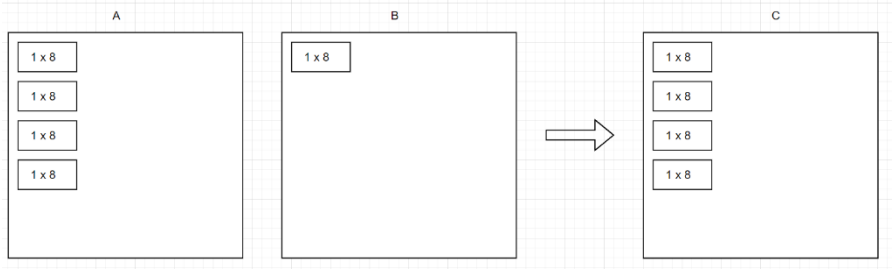

*Figure 4 A:4*8 Vectorization Blocks / B:18 To multiply Block / C:4*8 Block to store mutiplication results*

In the bar graph below, the optimized findings for GEMM are summarized. It can be seen that tiling significantly decreases CPU time compared to the baseline GEMM implementation. However, there is no significant change in CPU time between gemmt1 and gemmt2. The gem Vectorized algorithm produced the lowest CPU times with the least number of caches misses.

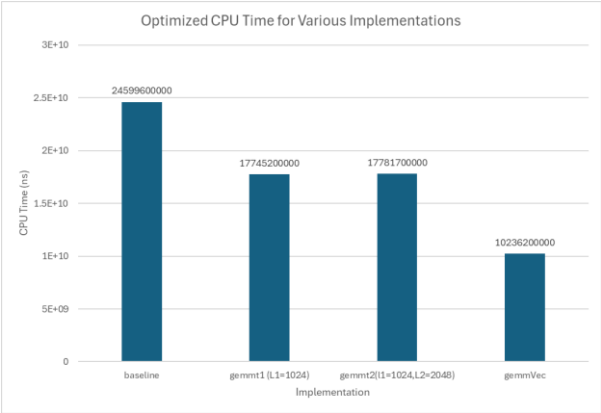

Figure 5 gemm execution time for all implementations

Part 2: Matrix-Vector Multiplication

Tiling:

The purpose of tiling is to divide data into smaller chunks that can fit into the cache. This allows for more efficient cache reuse and reduces the time spent accessing data from the main memory. Once the data is loaded into the cache, it is used for as many computations as possible before being evicted.

In GEMM, each element of the result matrix is the dot product of a row from matrix A and a column from matrix B. Thus, each row of A and each column of B needs to be reused multiple times. For example, when computing C(2,3) and C(2,5), both require row 2 of matrix A. The algorithm of GEMM makes tiling extremely effective, as it allows you to maximize cache reuse by keeping chunks of A and B in the cache for as long as possible to be used for multiple computations.

However, in GEMV, each element in the result vector is the dot product of a row from matrix A and the vector x. Unlike GEMM, where rows and columns are reused, each row of A is only used once. The vector x, however, is reused across all rows of A, m times. Given this, there is less data reuse in GEMV compared to GEMM, which makes tiling less beneficial. Since each row of A is only accessed once, the m dimension does not need to be tiled. Instead, when choosing the tile size for GEMV, the main focus should be to make sure that vector x is tiled so that blocks of x can remains in the cache for reuse during the computation.

With this in mind, the GEMV algorithm was only blocked in the n dimension. Starting from 16, the code was sweeped with the tiles_n increasing in powers of 2 until 4096. From the baseline code, there was minimal change in the running time for all block sizes. The cache misses remained relatively the same.

Upon further thought, this makes sense as the additional for loops required for blocking will increase loop overhead. The time to load and store the additional registers also increases cpu time and will potentially mask any improvement we get from tiling. In addition, the code was tested where the vector x has 4096 elements. The L1 cache is already large enough to store this vector and a row of A, thus tiling doesn't really have any impact. From the experiment, we see that larger tile sizes give us a slightly smaller run time because it results in a decrease in the loop iterations which will decrease the loop overhead.

Vectorization:

From gemmVec program, the total AVX registers we use are 13 (4 for A_row, 1 for B_row, 4 for C_row and 4 for intermediate products). Although this does not fully utilize the available 16 registers, it provides a good memory alignment and cache usage. Therefore, we would still use the same vectorization blocking size (32) to improve the performance.

But unlike the gemmVec (matrix-matrix multiplication), we found that it will be difficult to unroll the loop by 4 rows because the y dimension is a column vector. So we couldn't store the temporary vectorization results (8 float values) in row-wise as it would cause dependency issues. A possible solution is to compute the sum of each register and store it into y every time but this method would be expensive.

Therefore, instead of unrolling the rows, we decided to unroll the columns which we separate the column size into 4*1*8 small blocking size that is used for vectorization. For each iteration, the multiply results would store these 32 products into 4 AVX registers and then add them to a pre-defined AVX sum register. The sum AVX register only sets to zero when a new row start, and when it finishes one row, it would calculate the total sum of all the 8 temporary floating-point values inside this register and store the result to the correct index of y.

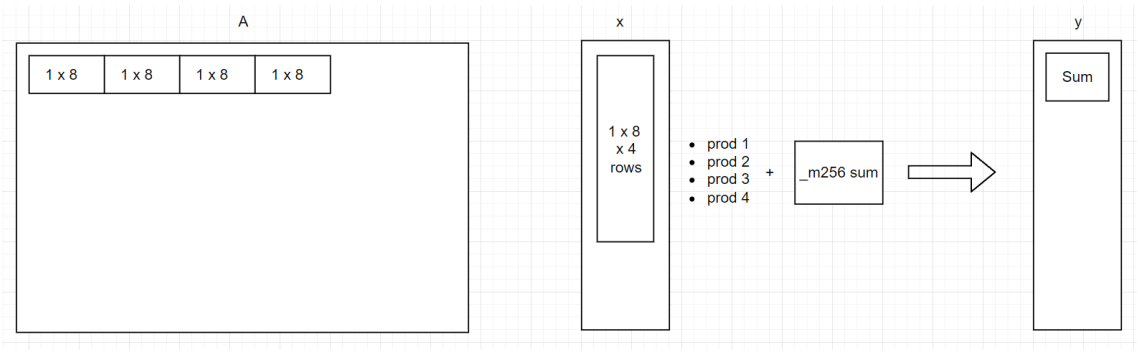

Figure 6 gemvVec vectorization flow

The final results of gemv can be summarized below in the bar graph. It can be seen that gemv vectorized has significantly reduced the number of cache misses and run time. This is because vectorization allows multiple elements to be processed simultaneously using AVX registers. This will improve the data throughput and reduce the need for frequent memory accesses. Since the AVX registers hold multiple values, it allows for fewer cache lines to be fetched from memory, which leads to fewer cache misses. In addition, vectorization improves spatial locality as it allows consecutive elements in memory to be loaded/used. This results in better utilization of the cache and will reduce the cache misses + run time.

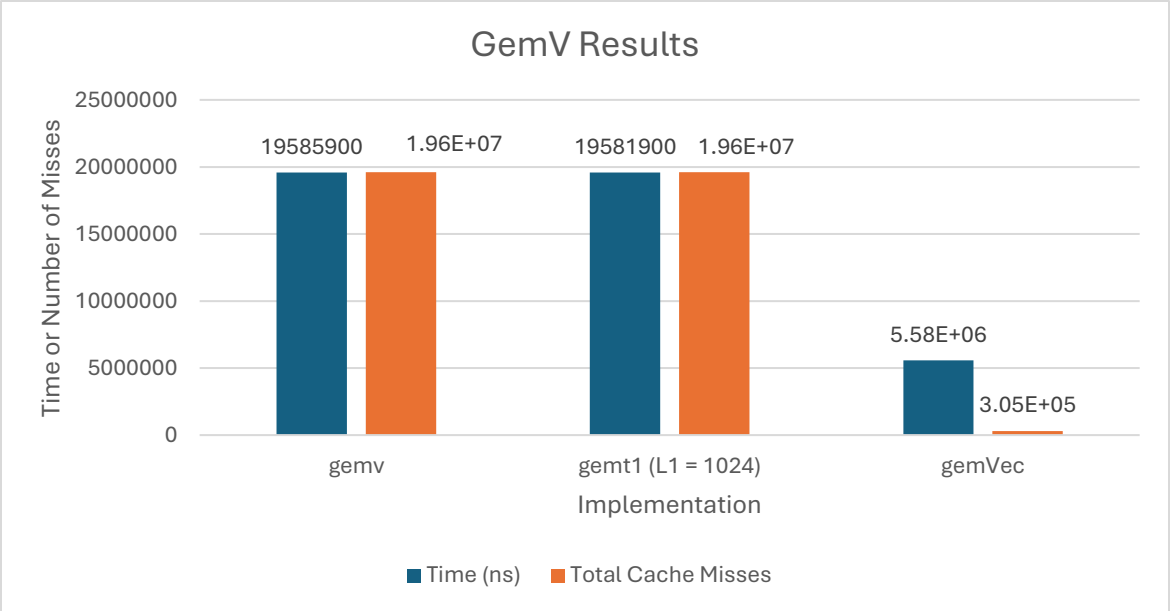

Figure 7 Run time info for gemv

Part 3: Dense Neural Network

The dense neural network was implemented with both gemm and gemv based on the equations in the lab document. The gemv dense nn code was hard coded to test only the first 10 features of the dataset. Thus when running the gemv code to get the correct accuracy calculation, the total number of samples must be set to 10. The accuracy was around 90% (gemv) and 92.11% (gemm) due to floating point rounding errors.

The profiling results can be seen in the table below for a gemm neural network:

Implementation of NN	**CPU Time (ns)**
Gemm	6851770000
GemmT1 (Tile size = 1024)	6894310000
GemmT1 (Tile size = 512)	6810090000
GemmVec (Tile Size = 512)	4016030000

Table 1Dense NN implemented using GEMM running time results

The code was first ran with the optimal tile size for the 4k by 4k matrix found in part 1. This tile size increased the cpu time from the baseline. This is because the matrix multiplication in the dense NN does not have matrixes with dimensions that large. As a result, the tile size was too big result and resulted in cache overflow. A much smaller tile size like 512 produced much better results, even beating the baseline.

With the most optimized code (gemmVec), it can be seen that the execution time decreases by 2835740000 ns from the baseline. This is almost a **42%** improvement.

The profiling results for gemv neural network can be seen below (for the first 10 feature predictions only):

Implementation of NN	**CPU Time (ns)**
Gemv	4554450
Gemv T1 (Tile size = 1024)	4543510
GemvVec (Tile Size = 512)	1395850

Table 2 Dense NN implemented using GEMV running time results

The gemvT1 implementation has no significant impact on CPU time as seen in part 2. With the most optimized code (gemvVec), it can be seen that the execution time decreases by 3155950 ns from the baseline. This is almost a **69%** improvement.