Report of CSE 260 PA 1: Matrix Multiplications

Zihao Kong, Kehan Long
University of California, San-Diego
zikong@ucsd.edu, k3long@ucsd.edu

I. Introduction

In this project, we optimized square matrix (size $N \times N$) multiplications and ensured the correctness of results with respect to different matrix sizes. The experiment was done on an Amazon EC2 t2.micro instance, which has a single CPU and runs with the x86_64 architecture. Our goal is to reach about $55 - 65\%$ of performance of OpenBLAS, and the performance is evaluated based on GFLOPS. As matrix multiplication always requires $\mathcal{O}(n^3)$ operations for any implementations, our focus is to reduce the data-access time during the multiplication. In other words, we have a rule in our implementation: data that needs more frequent access should be stored in the faster caches. The provided OPENBLAS benchmark has a mean performance result of 35.2 GFLOPS for matrix size $N = 2048$, and our implementation successfully reached the mean performance of > 28 GFLOPS for the same matrix size.

II. Results

A. Q2A: Show speedup over naive code for 6 interesting points that you select

We select different matrix sizes ($N = 64, 128, 256, 512, 1024, 2048$) and compare the GF performance by Naive implementation and our implementation. The results are shown in Table I. From the table, we see that our implementation improves the performance for any matrix sizes. However, the improvement is more significant for larger matrix sizes ($N \geq 256$).

TABLE I: Speedup Over Naive Code

Matrix Size N	Naive GF	Our GF
64	2.16	3.305
128	1.72	9.217
256	1.62	17.26
512	1.56	21.4
1024	1.31	28.5
2048	1.08	28.43

B. Q2b: Point out and explain at a high level irregularities in the data (Places where performance scales in a non-linear way)

For the matrix dimension N, from Table. I, we can see that the performance of our implementation over the naive implementation improves in a non-linear way. For example, when N is relatively small, such as $N = 64$, our implementation only achieves a little better performance over the naive implementation. However, when N is relatively large, such as $N = 1024, 2048$, our implementation achieves a much better performance over the naive implementation. There are two reasons behind that:

- For large matrix sizes, since the naive matrix multiply calculate one element of C by fetching a whole row of A and a whole column of B. So it doesn't fully utilize cache locality, which waste a lot of CPU cycle fetching data into cache and overwrite cache again. Our implementation uses cache blocking and reduces the cache misses. It's obvious that small matrix will have smaller number of cache misses for naive implementation, which explains that the performance improvement on small matrix is not as significant as the improvement on large matrix.
- As we are using a microkernel of size 4×12, we need to pad our matrix with zeros to deal with edge cases, which may lower our performance especially for small matrix whose heights and lengths are less than out packed matrix's size ($m_c \times k_c$ and $k_c \times n_c$).

TABLE II: Peak Performance for different sizes of Matrix

Matrix Size N	Peak GF
32	1.19
64	6.75
128	19.3
256	25.5
511	27.6
512	28.5
513	26.2
1023	28.6
1024	29
1025	26.5
2047	28.325
2048	28.9

C. Q2C: Show performance for the following numbers

In this section, We show our peak performance for different sizes of matrices. We ran our implementation several times and picked the peak performance, as shown in Table. II.

III. Analysis

A. Q3a: How does the program work

Previous work [1] studied how to optimize the matrix multiplication and our algorithm is built on this work. Generally, the program starts by writing two packing algorithms that divides two matrices of size $m_c \times k_c$ and $k_c \times n_c$ respectively. The first matrix of size $m_c \times k_c$ is divided into subpanels of size $m_r \times k_c$, the second matrix of size $k_c \times n_c$ is divided into subpanels of size $k_c \times n_r$. We call the most inner matrix

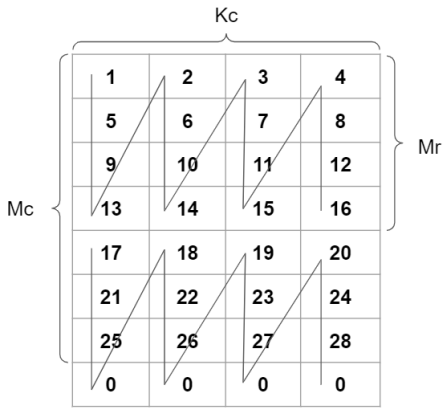

Fig. 1: Example of $m_c = 4$, $k_c = 4$, $m_r = 3$

of size $m_r \times n_r$ as microkernel. To complete the packing process, we need to carefully deal with the edge cases, such as when m_c cannot be evenly divided by m_r and/or n_c cannot be evenly divided by n_r. In our implementation, we padded the matrix with zeros to the multiple of m_r and n_r to avoid dealing with the different sizes of the inner microkernel. Fig. 1 is an example of padding algorithm we implemented.

If $m_c = 7, k_c = 4, m_r = 3$, in this case we can't divide evenly between m_c and m_r, so we allocate memory that can hold 32 doubles, which has 4 more memory slots than the 7×4 array itself. We then pack our array along the m_c column with a zig-zag pattern of 4, when we finish one $m_r \times k_c$ block, we go to the left 3×4 block, padding zeros as we pack.

After writing packing algorithm, the remaining parts of the algorithm are 5 loops that divides the big matrix multiplications $\boldsymbol{A} \times \boldsymbol{B} = \boldsymbol{C}$ into subsections $m_c \times k_c$, $k_c \times n_c$ for matrix $\boldsymbol{A}, \boldsymbol{B}$, here is an overview of the algorithm:

- First loop (outer most loop) index is "ic", it strides with a step of m_c. In this loop, It determines the height of the matrix of size $m_c \times k_c$, which is "ib", when "ic" goes to almost the edge of the matrix $\boldsymbol{A}$ (where the remaining is less than m_c), the height of matrix m_c, which is variable "ib", will be the remaining length.
- Second loop index is "pc", with a stride step of k_c, which determines the length of matrix $m_c \times k_c$, using variable "pb". Similarly, when "pb" is less than k_c, "pb" will be the remaining distance along B as well. In this loop, it divides matrix A into subsections $m_c \times k_c$, then put this subsection into the packing algorithm, copying values from matrix A into array "packA", which is stored in column major ordering.
- The third loop index is "jc", with a stride step of n_c. Since matrix multiplication requires the first matrix's length equals to the second matrix's height, we only need to determine the third dimension n_c. In our code, we used "jb" index to specify n_c's actual length. According to the way of striding of former two loops, "jb" will be the minimum between the remaining length of stride and n_c. After deciding the dimension of $k_c \times n_c$, we pack it into array packB, with row major ordering.
- When we have two small matrix of size $m_c \times k_c$ and $k_c \times n_c$ respectively, we send them to a macrokernel for further division. The fourth loop partitions $m_c \times k_c$ matrix into $m_r \times k_c$. Based on our packing algorithm mentioned before, there will be no edges cases because we filled "packA" and "packB" with zeros to get rid of these edge cases.
- The fifth loop further partitions $k_c \times n_c$ into $k_c \times n_r$. Then these two matrices are sent to a microkernel of size $m_r \times n_r$. Finally, we can save the multiplication results of the microkernel to the matrix $\boldsymbol{C}$.

TABLE III: Kernel Layout

c00-c03	c04-c07	c08-c0B
c10-c13	c14-c17	c18-c1B
c20-c23	c24-c27	c28-c2B
c30-c33	c34-c37	c38-c3B

The microkernel is implemented with the help of AVX-2 and FMA instruction extensions. And we chose $m_r = 4$ and $n_r = 12$. We first declared 12 variables on YMM registers, each variable represents 4 doubles in a 4×12 grid. Then we declared a_0 on YMM register. It takes one double from $m_r \times k_c$ and broadcast the value across entire register. We also declared another 3 variables denoted by b_0, b_1, b_2 on YMM registers as well. b_0 takes first four doubles in a row of $k_c \times n_r$ matrix, b_1 represents later 4 doubles and b_2 represents the last 4 doubles.

Inside the microkernel, it had a loop with number k_c times, In this loop, we did loop unrolling by calculating rows of matrix $m_r \times n_r$ manually by four times, instead of using one loop to do so. This increases some performance as well.

B. Q3b: Development process

We approached this problem by writing the packing algorithm first, which involves calculating global indices verses local packed array indices. We wrote several example matrices on scratch paper and did some hand calculations, and finally came up with a mathematical formula for indices conversion. Then we tested on a self written driver, which would print each packed array when it is finished. After that, we compared the printed result with the results we calculated on the scratch paper, and thus finalized the implementation.

Our next step is to implement the microkernel. We looked up some intrinsic documentations on [2] and learned how to use these instructions. We also looked at class slides [3] as well, learning the standard way to use a YMM register to calculate four doubles at a time.

We tried using a 4×4 microkernel in the beginning for testing, which used 8 registers. However, this implementation produced around 15 GFlops for matrix size $N = 2048$. But it did not achieve the required performance, but at least we learned that we were on the right track. From then, we plan to implement different sizes of microkernels in order to achieve better performance results. We believed that we were able to

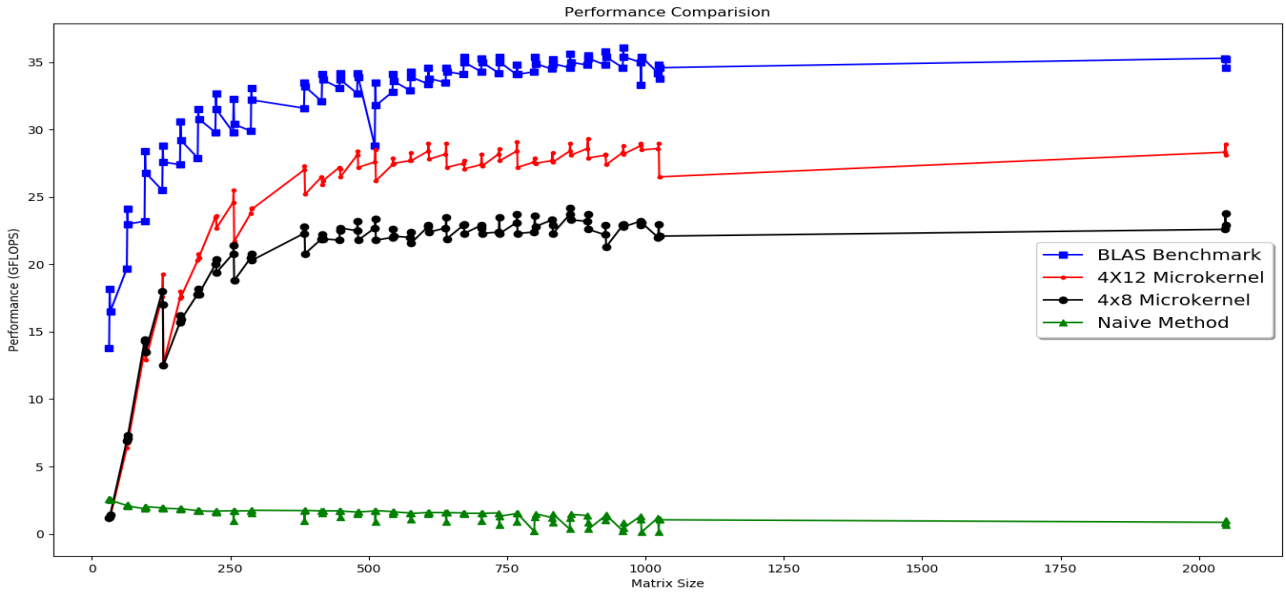

Fig. 2: Overall Performance Comparison

achieve better results because the 4×4 microkernel only used 8 of the 16 AVX .

Therefore, we decided to implement 4×12 and 4×8 kernels simultaneously and pick the one with the better performance results. Following the same procedures, which were loading registers, calculating, storing registers. After several modifications and tests, for relative bigger matrix sizes ($N \geq 256$), we got > 23 GFlops using the microkernel of size 4×8 and > 25 GFlops using the microkernel of size 4×12.

The part where it got tricky was that we need to consider some edge cases such as packing around matrix edges. The first time we didn't pad "packA" and "packB" buffer with zeros, which causes the microkernels to load wrong values from pack array. We came up with several ways of solving this problem. One was to use a naive kernel to process these edge cases, but this approach is intrinsically slow. The other approach was to pad zeros on packA and packB buffer when there were not enough rows or columns they need to pack. This approach won't involve the use of any naive kernels, and even though it increases the number of multiplication with zeros, microkernel's high performance outweigh this downgrade.

The overall results that comparing the 4 implementations (naive, 4×8 microkernel, 4×12 microkernel and BLAS) are shown in Fig. 2

C. Q3c: Supporting Data

Since the L1 cache on our machine is 8-way associative, with each cacheline of 64 bytes, and a total of 32768 bytes. So there are totally 8 sets with $32768/8/64 = 64$ cache lines per set. Each cacheline holds 8 doubles. so for one entire set, we can store $8 * 64 = 512$ doubles. By making $k_c = 128, m_r = 4$, we have a matrix that has 512 doubles, thus filling $\frac{1}{8}$ of L1 cache.

By making $k_c = 128, n_r = 12$, each sub-matrix from matrix B has 1536 doubles. That fits $\frac{3}{8}$ of L1 cache. Later on, all of the data in L1 cache is read into YMM register for high speed microkernel computations, thus eliminating a lot of time for CPU to fetch from low level cache. After processing 1 microkernel of $m_c \times n_c$, another 1536 doubles were fetched from L2 cache again, filling the rest $\frac{3}{8}$ of L1 cache. Without replacing any data in existing L1 cache, the microkernel can access data without conflict misses. Thus our 4*12 kernel design enable high speed.

The skeleton code in Blislab tutorial were written differently from us because the arrays there were stored in memory with column major order. But we used array of row major order. During packing, this led to a difference in code.

In Blislab. They made use of OpenMP as well, parallelizing inner loops with multiple threads because the algorithm of dividing matrix into sub matrices doesn't have data dependence. They can take use of multiple cores in a processor. However, since we are reproducing this algorithm on ti.micro, which only has 1 vCPU, we are not able to use multiple OpenMP threads.

D. Q3d: Future Work

In this experiment, we are using the t2.micro instance in Amazon ec2, which only contains a single CPU. We look to implement the matrix multiplications on multiple CPUs and optimize the performance even more, as we may able to implement the algorithm parallelly on multiple CPUs. Besides, we currently choose the microkernel size (4×8 or 4×12) based on the performance (GFLOPS) results because either design looks good based on our cache sizes. We further plan to theoretically understand how the microkernel size will affect the performance (how 4×12 works better than 4×8).

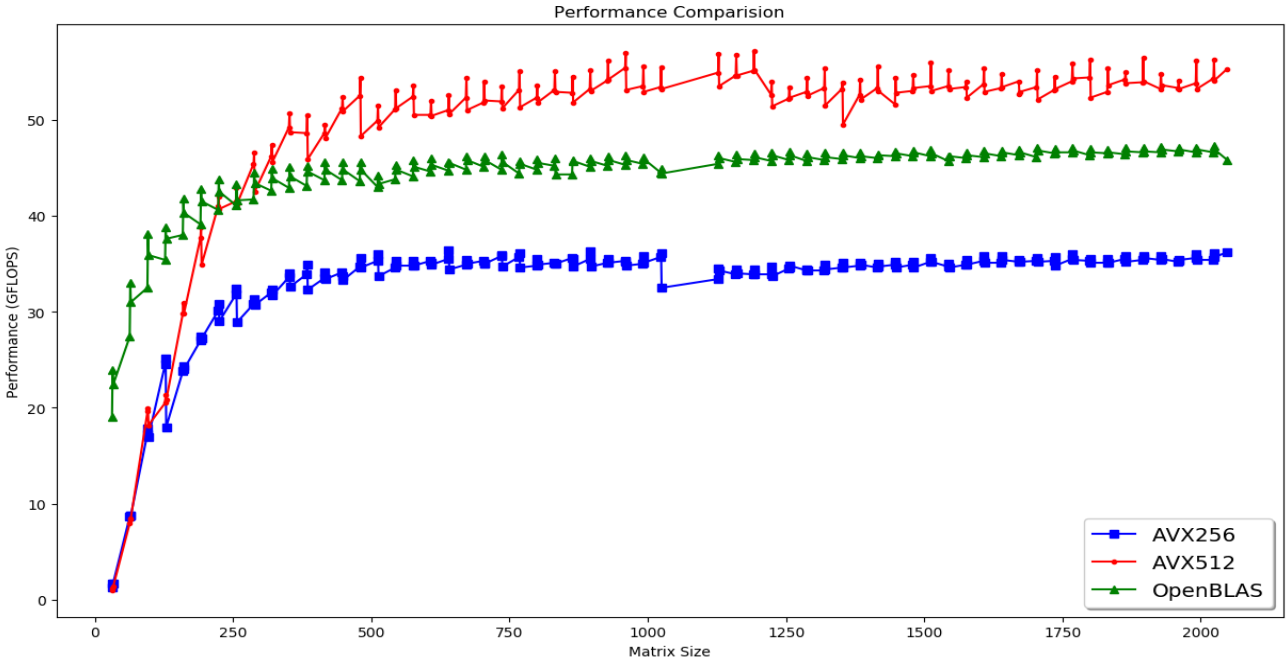

Fig. 3: AVX512 Performance Comparison

IV. Extra Credit

As it's possible to achieve the performance goal of $>$ 40GFLOPS on c5.largemachine, we tried to run it on c5.large instance with AVX256 and AVX512, which is not supported on t2.micro instance. The CPU on the c5.large instance is Intel(R) Xeon(R) Platinum 8124M 3GHz with 2 cores and 2 threads. In our implementation, we used microkernel of size 8×24 as we can use more registers in AVX512. Finally, we achieved the performance of $>$ 50GFLOPS using AVX512 for large size matrices, which is even better than the OpenBLAS benchmark! The performance comparison between our implemented AVX256, AVX512 and the OpenBLAS benchmark is shown in Fig. 3.

References

[1] J. Huang and R. A. van de Geijn, "BLISlab: A sandbox for optimizing GEMM," FLAME Working Note #80, TR-16-13, The University of Texas at Austin, Department of Computer Science, 2016.

[2] "Intel intrinsics guide," 2020. https://software.intel.com/sites/landingpage/IntrinsicsGuide/.

[3] B. Chin, "Cse260: Parallel computation. lecture2-4: Memory organization and optimization & multicomputer organization & simd shared memory computing," FA 2020.