CSE 260A PA1 Report

Chao Wang Ruicheng Li

1. Result

1.1. Q1.a. Show performance of your optimized code for the following numbers

We test our code in a set of matrices of different matrix size N, $N = \{32, 64, 128, 256, 511, 512, 513, 1023, 1024, 1025, 2047, 2048\}$. The performance is shown in table 1.

Matrix Size	Peak GF
32	0.108
64	0.84
128	5.78
256	15.9
511	21.3
512	21.9
513	19.7
1023	23.5
1024	21.7
1025	20.4
2047	22.5
2048	22.6

Table 1. Peak performance for matrices of different sizes

1.2. Q1.b. Make a plot of the performance of the three versions of code

We compare our code with naive implementation and open-blas benchmark, as is shown in figure 1.

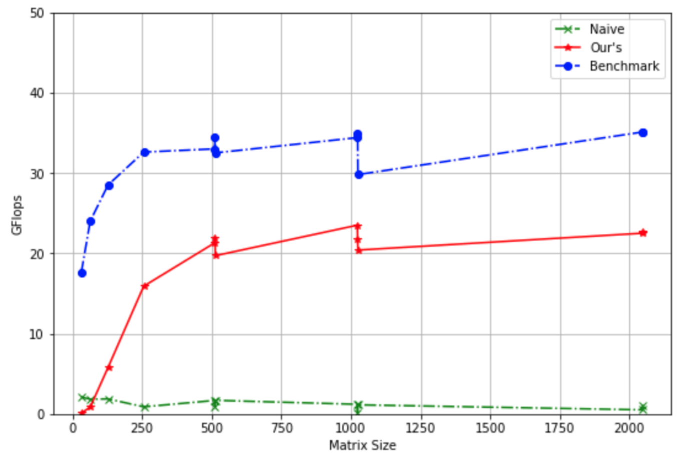

Figure 1. Compare the performance of our implementation with naive implementation and blas benchmark.

2. Analysis

2.1. Q2.a How does the program work

2.1.1. Three-step packing

Given A of size $M \times K$, B of size $K \times N$, C of size $M \times N$, we pack matrices in a three-step fashion.

1. We pack A into Ac of size $Mc \times Kc$ so that Ac can fit in L3 cache, pack B into Bc of size $Kc \times Nc$, pack C into Cc of size $Mc \times Nc$ so that Bc can fit in L2 cache.
2. **Macro-kernel packing.** We pack Ac into subpanel Ap of size $Mr \times Kc$ and Bc into subpanel Bp of size $Kc \times Nr$ so that Ap and Bp can fit in L1 cache. During packing, we reorder the matrices' elements in a zig-zag fashion so that they can be accessed in a cache-friendly way.
3. **Micro-kernel packing.** Finally, we pack Cc into Cr of size $Mr \times Nr$ so that Cr can fit into registers and loop over k from 0 to Kc to calculate:

$$Cr[i, j] = Ap[k, i] * Bp[k, j] + Cr[i, j]$$

2.1.2. Padding to solve fringe case

In the cases where Mc is not divisible by Mr or Nc is not divisible by Nr, we pad the matrix with zeros so that the padded matrix's length is divisor of Mr or Nr. Specifically, we pad

$$\begin{aligned} Ac &: Mc \times Kc \to (Mc/Mr + 1)Mr \times Kc \\ Bc &: Kc \times Nc \to Kc \times (Nc/Nr + 1)Nr \\ Cc &: Mc \times Nc \to (Mc/Mr + 1)Mr \times (Nc/Nr + 1)Nr \end{aligned}$$

2.1.3. Speedup micro-kernel with AVX and unrolling

We use AVX instructions to speed up mirco-kernel calculations. In our $4 * 8$ kernel, we declare 8 registers for Cr with each register holding 4 doubles. We declare one register for Ap which takes one double from Ap and broadcast the value through the whole register. We also declare two registers for Bp which each taking 4 doubles from Bp.

We also unroll the loop of mircro-kernel by a factor of 4 to speed up computation.

2.2. Q2.b Development Process

We started by implementing packing. Our packing worked successfully with matrix of size 64 but failed at size 65. We then realized that we have to perform padding to deal with irregular matrices. Our initial solution for padding is to pad matrices at the outer loop, that is, we make the length of Ac divisible by Mc and the width of Bc divisible by Nc. But then we realized this method could lead to significant overheads when Mc, Nc are large. Meanwhile, we found that it's sufficient to make the length of Ap divisible by Mr and the width of Bp divisible by Nr. Thus we change our padding method and apply it in the inner loop. With this method, we were able to get correct results for all matrices while avoiding huge padding overheads.

Second, we try to implement a simple 4 * 4 kernel according to the knowledge we learned during the class. It is just the beginning of the optimization problem because this kernel only used 8 registers. This implementation reaches 10.04 Gflops so we need to continue to optimize the kernel.

Next, we used 4 * 8 kernel to use more registers and we encountered problems during this part. First, we found a segmentation fault error during running the problem and we found out it was because the matrix C is unaligned, which means we already packed matrix A and B but forgot to pack matrix C. Therefore, we packed matrix C to solve the problem. After that, we encountered the correctness problem of calculating the matrix multiplication. We utilized debug functions like printMat to visualize the matrix and found out there is some incompatibility between the packing matrix and kernel development. The problem is we did not store packC back to matrix C after processing the packing part. After solving the problem, we got a better performance with 4 * 8 kernel.

In the end, after running 4 * 8 kernel, we realized that we did not fully utilize the spaces of the cache, which leading us to try a larger kernel like 4 * 12 and 8 * 8. According to the figure 2, we found out 4 * 12 have a better performance than 4 * 8, which already achieved our requirements for performance.

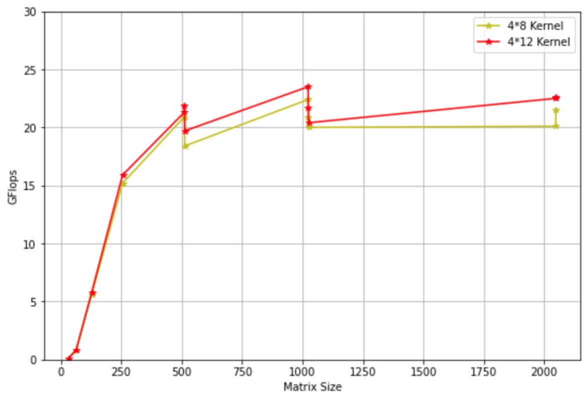

Figure 2. Compare the performance of our 4*8 with 4*12 implementation.

2.3. Q2.c Point out and explain at a high level irregularities in the data

We could see the performance does not increase linearly with the size of the matrix.

Irregularities of small matrices We could notice that when size = 32, the performance is 0.106. We believe this is because there are some overheads for realizing the process of packing, for example, memcpy packA, packB, unnecessary calculation for padding unit, load and store from matrix to packed matrix. All of those would have a high cost when the operations of computation of the original matrix are not high, i.e, the size of the matrix is not large. Actually, the purpose of packing is to place the data as much as possible into the cache. If the size of the matrix is very small and everything is already in the cache, it would be costly and unnecessary to do the packing.

Irregularities of large unaligned matrices We could find out the performance of size 1024 is better than size 1025, this is because 1025 needs another block of padding and it produces many unnecessary padding 0 to participate in the calculation of matrix multiplication.

2.4. Q2.d. Supporting data

2.4.1. Cache behavior

We used Cachegrind to analyze the cache misses for naive code and our implementation. As we know, our code has a better performance than naive code. According to table

	Naive	Our's
I1 miss rate	0.00%	0.02%
D1 miss rate	4.1%	2.2%

Table 2. Comparison of instruction and data miss rate between naive code and our code

2, after running the Cachegrind tool on sizes 32 and 64, we found out the cache misses for naive code is 4.1% while the misses is 2.2% for our code. This result proves that higher cache misses would lead to a lower performance because we need to visit some data from a slower memory instead of faster cache.

2.4.2. Parameteric searches

Parameter	Ap	Bp1	Bp2	Total
Mr=4, Nr=8	$\frac{1}{8}$ L1	$\frac{2}{8}$ L1	$\frac{2}{8}$ L1	$\frac{5}{8}$ L1
Mr=4, Nr=12	$\frac{1}{8}$ L1	$\frac{3}{8}$ L1	$\frac{3}{8}$ L1	$\frac{7}{8}$ L1

Table 3. Usage of L1 cache for different paramter setups.

We analyze the size of cache by calculating how many numbers could we fit into L1, L2 cache. First, it is easy to discover the cache parameter of this AWS instance. For example, L1 cache has 8 associative way with 64 bytes cacheline and 32678 bytes. The cache lines of this cache is $32678/(8*64) = 64$ for each set.By knowing the above information, the cache could theoretically store 40896 doubles.

Now we perform an anlysis on the cache for $Mr = 4, Kc = 128, Nr = 8$. Ap of size $Mr \times Kc$ takes up $4 * 128 = 516$ doubles, which constitute $\frac{1}{8}$ of L1 cache. We take two Bp blocks Bp_1, Bp_2 into L1 cache to avoid conflict miss. Each Bp takes up $8 * 128 = 1024$ doubles, constitute $\frac{2}{8}$ of L1 cache. And two Bp blocks takes up $\frac{4}{8}$ L1 cache. This parameter setup is fairly good as it allows us to do multiple computations at one step with almost zero conflict misses. Yet we still have $1 - \frac{1}{8} - \frac{4}{8} = \frac{3}{8}$ L1 cache unused. We can make a full use of L1 cache by increasing Nr to 12. Under this setup, by applying the same calculations as above, we can make use of $\frac{7}{8}$ L1 cache, with only $\frac{1}{8}$ L1 cache unused (table 3). Theoretically, the setup$Mr = 4, Kc = 128, Nr = 12$ should lead to the best performance. This result is in line with our experiment result as shown in figure 2

2.4.3. Row major vs. column major

Our skeleton code is based on row-major while the BLISlab tutorial is on column-major. The reason for BLISlab using column-major storage of arrays because their interface was for Fortran users first. Thus, we need to process the packing in a different way.

3. References

1. https://github.com/flame/blislab - Jianyu Huang, Robert A. van de Guijn, BLISlab: A Sandbox for Optimizing GEMM, August 31, 2016

2. https://github.com/flame/how-to-optimize-gemm/wiki . How to Optimize GEMM WikiHow to Optimize GEMM Wiki