CSE260 PA1 - General Matrix Multiply Acceleration by GOTO Blocking & SVE

Quan Luo QULUO@UCSD.EDU
Zhongrui Cao Z2CAO@UCSD.EDU

Abstract

General Matrix Multiply (GeMM) is a widely researched but yet interesting and fundamental problem in computing theory. In this report, we experiment GOTO method to do matrix blocking, which accelerates the process by cache locality. We also implement SVE SIMD instructions to further make the program faster.

1. Introduction

In this experiment, we aimed to optimize general matrix multiplication on an Amazon EC2 instance, which runs with ARM64 architecture. The CPU model is 1 core Neoverse V1 with 64KB L-1 and 1MB L-2 Cache [1]. For matrix multiplication, we use the brute force $\mathcal{O}(n^3)$ algorithm and optimize it by utilizing the CPU architecture & SIMD instructions to do parallel computing.

A typical thought to optimize GeMM is matrix blocking. GotoBLAS adopt the idea and try to fit the blocked matrices into caches, which reduces direct memory access and cache miss that further accelerates the program [2]. BLIS further breaks down the blocked matrices and cares only for a micro-kernel to calculate $C := AB + C$ [3]. We adopt the idea from BLIS and implement it on the platform described above. Experiments show us promising results compared to BLAS.

2. Q1 - Results

N	32	64	128	256	511	512	513	1023	1024	1025	2047	2048
Peak GF	8.45	18.7	20.1	20.5	20.4	20.2	19.9	20.1	20.1	19.6	20.1	20.8

Table 1: Numerical results in for different N

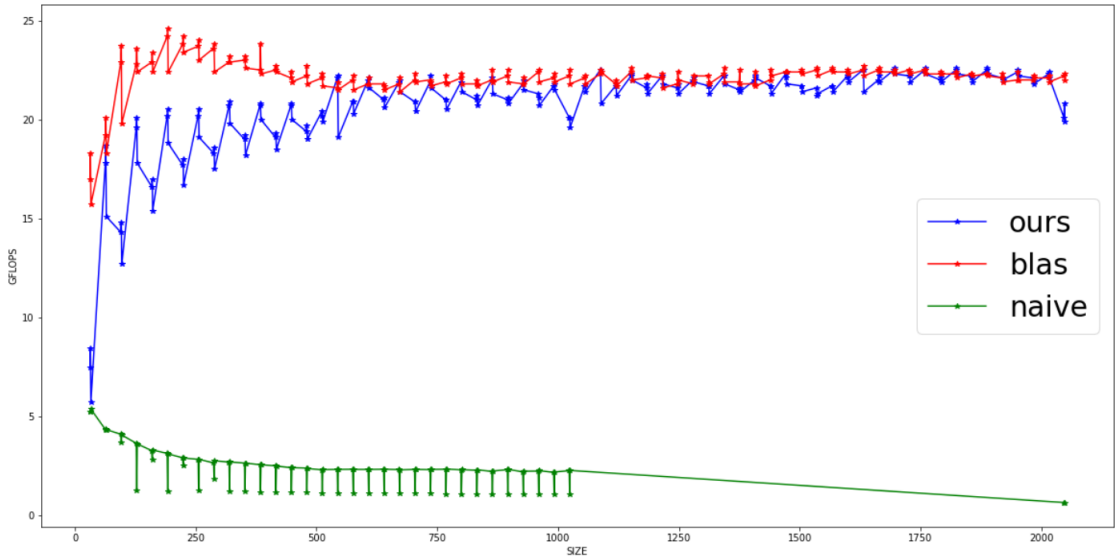


Figure 1: Performance of the three versions of code: the naive code, the OpenBLAS code, and our optimized code. *naive value between 1025 and 2048 was not tested

3. Q2 - Analysis

3.1. Q2a - How does the program work

Our algorithm comes from BLIS and basically as the following graph 2 shown

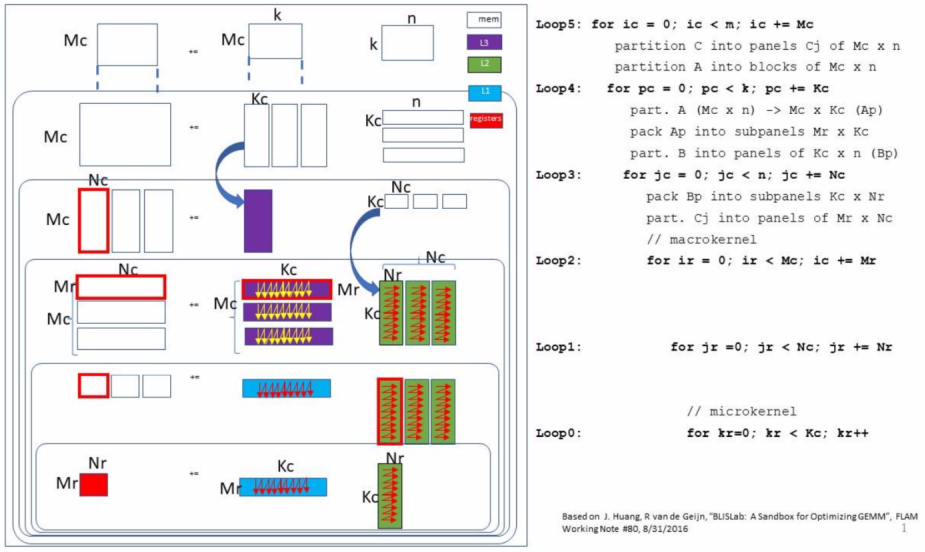


Figure 2: Algorithm Process with Pseudo-codes

Let's say there're three matrices $C \in \mathbb{R}^{m\times n}, A \in \mathbb{R}^{m\times k}, B \in \mathbb{R}^{k\times n}$, what we want to do is to calculate $A \times B$ and store it into C. As shown above, the first three loops (3/4/5) is slicing down the three matrices and takes out $A_s \in \mathbb{R}^{m_c\times k_c}, B_s \in \mathbb{R}^{k_c\times n_c}$. Note that we want the whole A_s and B_s to be stored in cache, so that the program will not have cache miss during the two small loops. Also the matrices are stored in certain order to ensure sequential memory access for micro-kernel.

Now having A_s and B_s we further cut them up into really small pieces $A_r \in \mathbb{R}^{m_r\times k_c}$ and $B_r \in \mathbb{R}^{k_c\times n_r}$. The corresponding C_r can be viewed as a rank-1 update of A_r and B_r. Our micro-kernel will be calculating the rank-1 update of those matrices and added up to corresponding C_r.

For the micro-kernel, we use SVE SIMD intrinsics to calculate the rank-1 update. In our processor Neoverse V1, we only have two 256bit SVE vector. We'll break the rank-1 update to many length 4 vector multiplications and addition. Thus we can use SVE to accelerate the micro-kernel.

3.2. Q2b - Development Process

In this subsection we mainly talk about the implementation process of our main program (packing / micro-kernel / parameters) and difficulties we met. More optimization and details about those optimization will be discussed in 3.6.

3.2.1. Packing Implementation

First we implemented packing for the matrices. There's not much need to say since we simply sliced the matrices as described in previous section. But need to mention that, since we are dealing with sub-matrices that may not be of satisfying sizes, we do padding during the packing procedure as indicated follow 3.

Suppose our micro-kernel only calculates 2×3 matrices while the bigger matrix is 6×6. We'll pad two extra columns and one extra row to the matrix. These 0's will not make any influence to the final result however, the kernel can be reused now for any kind of sizes matrices. Because of our

padding strategy, we'll have an extra padding cost here. However, keeping the sliced size m_r, n_r and k_c small, we won't need to worry much about it.

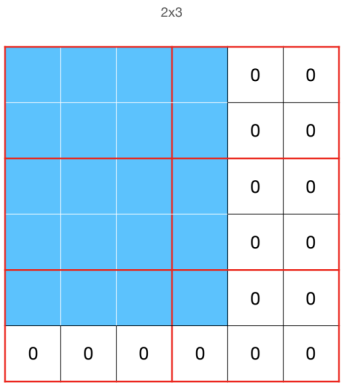


Figure 3: Padding Strategy in Packing

However, after packing, we don't see great difference in GFlops. It remains around 2-3 GFlops though the cache miss is reduced.

3.2.2. Kernel Implementation

For the kernel, we basically parallel computes the multiplication of a 4×1 vector with another 1×4 vector and add the result to corresponding place of the matrix as shown follow4.

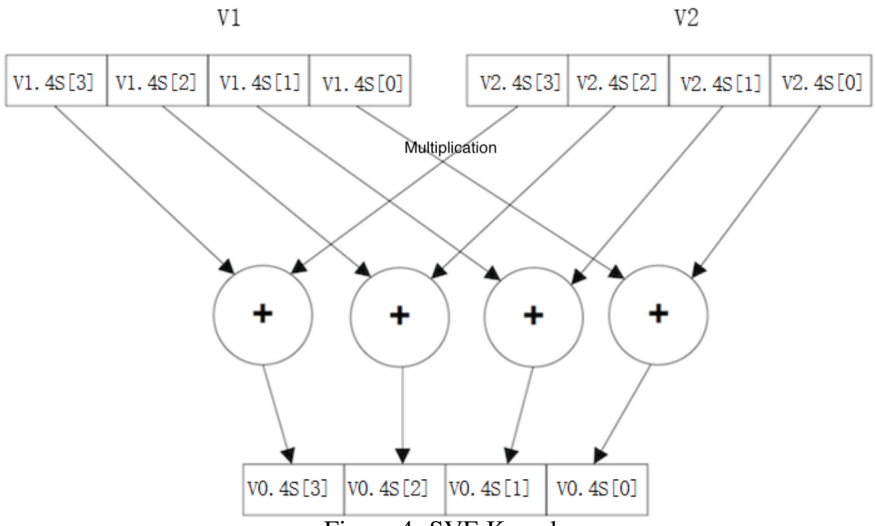


Figure 4: SVE Kernel

Note that to reduce function call, we try to enlarge the micro-kernel size. Since on Neoverse V1 we only have 256bits SVE and thus we choose the size to be multiplication of 4. Also notice that increasing m_r by x will actually only need x more SVE registers. However, increasing n_r by x will need $2x$ more. Thus finally we choose to increase m_r and fixed $m_r = 16$ and $n_r = 4$.

After the kernel is initially implemented, our performance can get to around 13 GFlops. By increasing the kernel sizes and further optimization introduced in 3.6, we can achieve 20 GFLops for large matrices.

3.2.3. TUNING PARAMETERS

Our tuning strategy is first using a brute force approach to do an exhaustive search using big steps of block sizes, then followed by a more detailed search based on the brute force results. For our brute force search, we searched the permutation of $[8, 16, 32, 64, 128, 256, 512, 1024, 2048]$ (with repetitions). We pruned the search space by the rule: $kc < 256, nc > 32, mc > 64$, so that extreme and unnecessary inputs are ruled out. Here are the results of the brute force search:

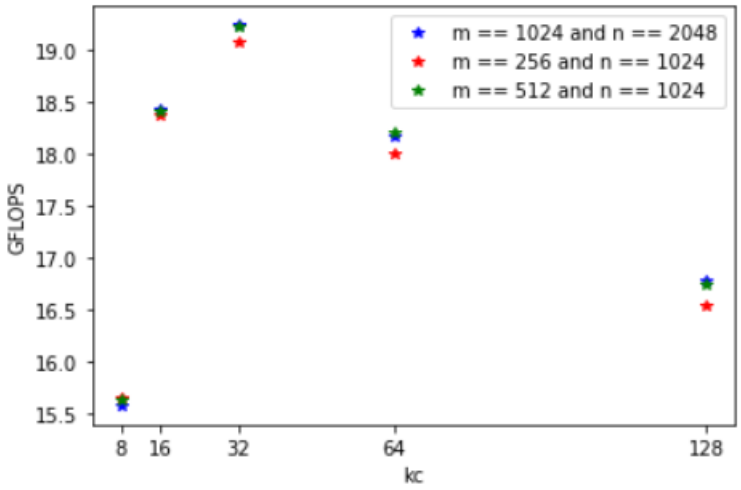
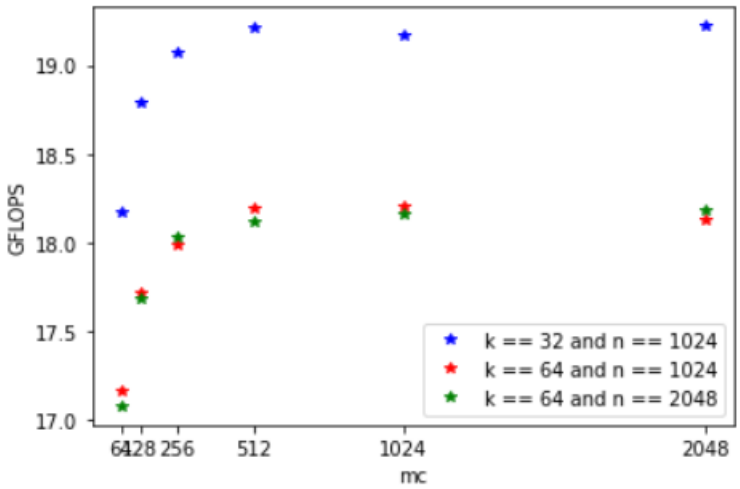


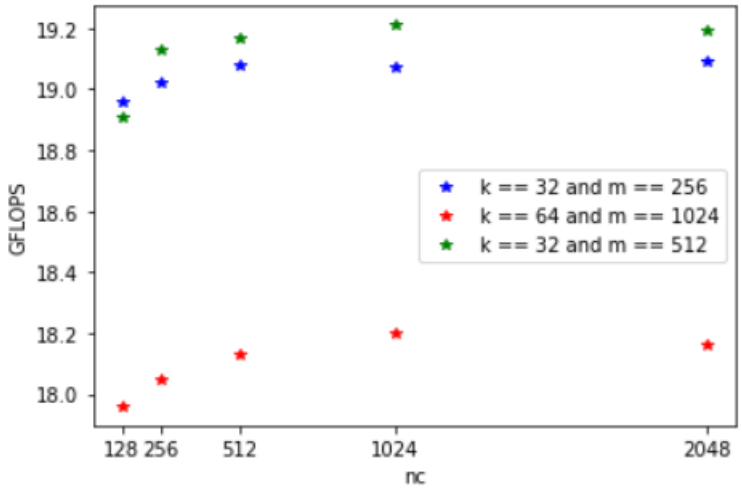
Figure 5: different kc with same mc and nc, GFLOPS is average of multiples of 32

As shown in the figure, the larger the block size the better the utilization of L1 is, however this surprisingly stops at kc = 32, and the performance starts to drop. Although the L1 of our machine is 64KB, the performance of kc = 32 is better than that of 64. This might be because we cannot use the whole of L1. However this might not be the case when our matrix size gets bigger, which we will discuss further in the following sections.



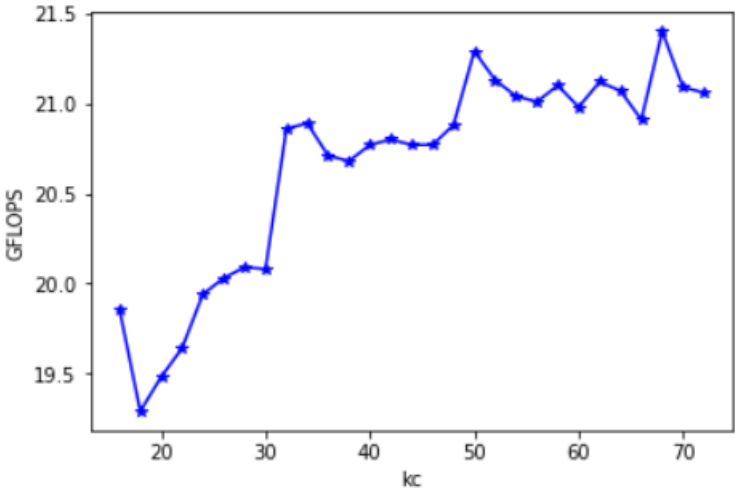
This figure shows the effect of different mc sizes on performance. As shown in the above figure, the larger the block size the better the utilization of L2 is, so we can see a increase in performance the bigger mc is. However performance stops increasing beyond mc = 512 and remains roughly the

same. Although different kc and nc value yield different results, the general pattern of increased performance with increased mc size persists. This result makes more sense than our research into kc, because the machine has L2 size of 512KB, so it makes sense for performance to plateau at mc = 512, which is the maximum utilization of L2.



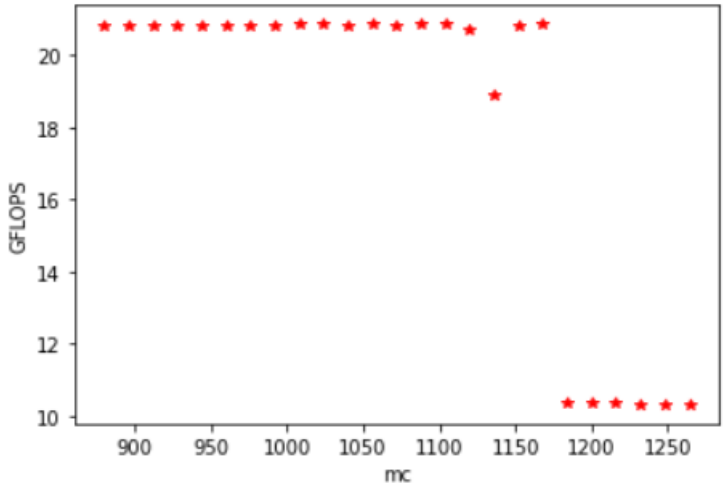
The above figure shows the effect of different nc sizes on performance. As shown, the performance generally gets better as the nc sizes gets bigger, but plateauing at nc = 1024. This effect can be explained by the abundance of memory in L3, such that it can hold as much as we give it. As the utilization of L3 if fully exploited, the increase in performance disappears.

After the brute force approach, we searched in detail around the optimal sizes we got in the previous step. We searched $16 <= kc <= 72$, $880 <= mc <= 1264$, $1920 <= nc <= 2320$. This time, the GFLOPS is calculated by averaging results from matrix sizes in benchmark.c: Multiples of 32 +-1, from 511 to 1025 only. (large matrix size only)

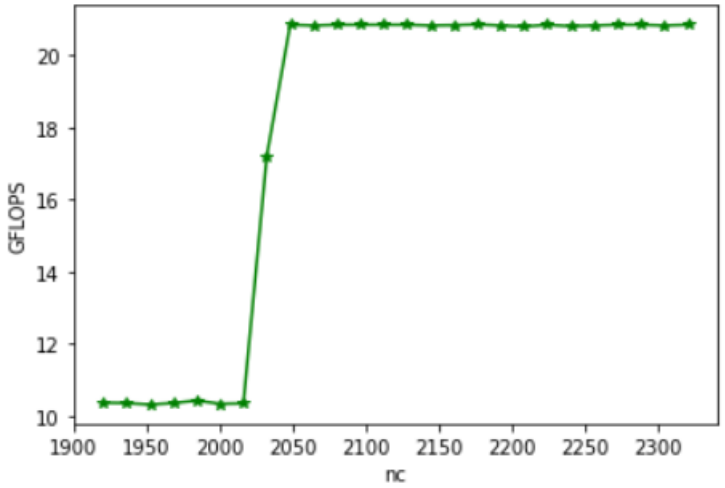


This figure shows the performance difference caused by difference kc values but same mc and nc values. There is an upward trend in increasing kc for more performance, which resolved the unexplained issue when we are brute forcing for the optimal kc. This time, performance plateaus at around kc = 64, which is appropriate for the fact that our L1 cache is 64KB large. The discrepancy

might be caused by the different tests run in the two passes. The first include small matrices and the second only include large ones. Since we focus on large matrix performance, we choose 68 as our optimal kc value.



This figure shows the performance difference caused by difference mc values but same kc and nc values. As the figure shows, there is a sudden performance drop when mc is bigger than around 1200. Since our L2 cache can be 1MB, this can be explained by not being able to fit the matrix into L2, thus creating a lot of memory misses, causing a lot of lose in performance. As a result, we chose the optimal mc to be 1088.



This figure shows the performance difference caused by difference nc values but same kc and mc values. There is a sudden increase in performance at around nc = 2048. As the utilization of L3 gets bigger, the performance will surly increase. However, performance plateaus at around 2100, so we chose the optimal to be 2176, which gave us the best average number.

3.3. Q2c - Reason of Irregularities

For our graph in Q1b, our result does not scale linearly because for smaller matrix sizes, the packing process will not be useful and create a lot of overhead. For example our implementation have roughly the same performance number as the naive implementation at the matrix size of 32.

However, as the matrix size increase, our implementation quickly catches up with blas, creating a near log(x)-like graph. Once the matrix size passes 500, our implementation yields similar performance numbers as the reference blas implementation. This is likely due to our parameter tuning process. We the test we used were exclusively matrices larger than 500, so the parameters was geared towards larger matrix performance.

3.4. Q2d - Supporting Data

```
GeoMean   = 2.54

 Performance counter stats for './benchmark-blislab':

       1564273572      cache-misses
       5822095512      dTLB-load-misses

     84.218935214 seconds time elapsed

     83.933194000 seconds user
      0.159918000 seconds sys
```

```
GeoMean   = 20.54

 Performance counter stats for './benchmark-blislab':

        613914732      cache-misses
        192652626      dTLB-load-misses

     27.591042715 seconds time elapsed

     27.396855000 seconds user
      0.131946000 seconds sys
```

```
GeoMean   = 21.42

 Performance counter stats for './benchmark-blislab':

        467362344      cache-misses
        162759698      dTLB-load-misses

     26.902994785 seconds time elapsed

     26.727684000 seconds user
      0.111898000 seconds sys
```

Figure 6: perf data showing the cache miss and TLB miss on three iterations of our code

The first figure shows the perf data on the original ”nopack” code. This version has a horrendous 1,564,273,572 cache misses and 5,822,095,512 TLB misses. The second figure shows the perf data after we implemented packing and microkernel, but have not tuned any parameters. Implementing packing reduced the cache misses an order of magnitude smaller and made TLB misses several times less. Cache misses is at 613,914,732 and TLB misses at 192,652,626. The last figure shows

the perf data after parameter tuning. As shown, the GeoMean of tuned version increased by almost 1 GFlop, which can be explained by the lower cache misses and TLB misses. Our final cache miss is at 467,362,344, which is 128,747,991 less than the untuned version. Final TLB miss is at 162,759,698, it is 29,892,928 less than before, which is a less pronounced decrease compared to cache misses. The analysis of our parametric search is also a big supporting data of our program, which is talked about in 3.2.3.

3.5. Q2e - Future Work

If we had more time, we would want to use openmp to parallelize our program, which would further increase our performance numbers. Also, we could have exhaust the entire search space of parameters (instead of pruning), to further increase our performance.

3.6. Q2f - Additional Insights & Optimization

3.6.1. Restrict Pointers & Inlining & Aligned Memory

Restrict pointers can increase the GFlops amazingly. This tells compiler to optimize the codes since there will not be any other pointers pointing to the same places. And in this lab, most of the pointers can be declared as restricted.

Also we inline almost all the short functions. Need to note if we inline one of the long function, we'll get worse performance. Probably due to TLB miss.

SIMD instructions will have better performance when loading from aligned memory, thus we pack the partial matrix into a not only contiguous but also aligned array.

3.6.2. Loop Unrolling

We try to unroll loops by ourselves. However the performance only turns worse. However if we turn off $O4$ optimization to $O0$, the performance can improve a little bit. Thus we can say that compiler has done most of the loop unrolling for us in $O4$ optimization. It's hard for us to make it better.

3.6.3. Avoid Conditions

There's almost no conditions in our codes except for some min. One typical optimization we've done is that we implement the function $b = max(0, a)$ by bit operation as

```
b = ~(a >> 31) & a;
```

4. Q4 - Extra Credit

We submitted the assignment on time.

References

[1] "Arm neoverse v1 cpu specification." (2022), [Online]. Available: https://www.arm.com/products/silicon-ip-cpu/neoverse/neoverse-v1.

[2] K. Goto and R. A. v. d. Geijn, "Anatomy of high-performance matrix multiplication," *ACM Transactions on Mathematical Software (TOMS)*, vol. 34, no. 3, pp. 1–25, 2008.

[3] T. M. Smith, R. A. Geijn, M. Smelyanskiy, J. R. Hammond, and F. G. V. Zee, "Anatomy of high-performance many-threaded matrix multiplication," *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pp. 1049–1059, 2014.