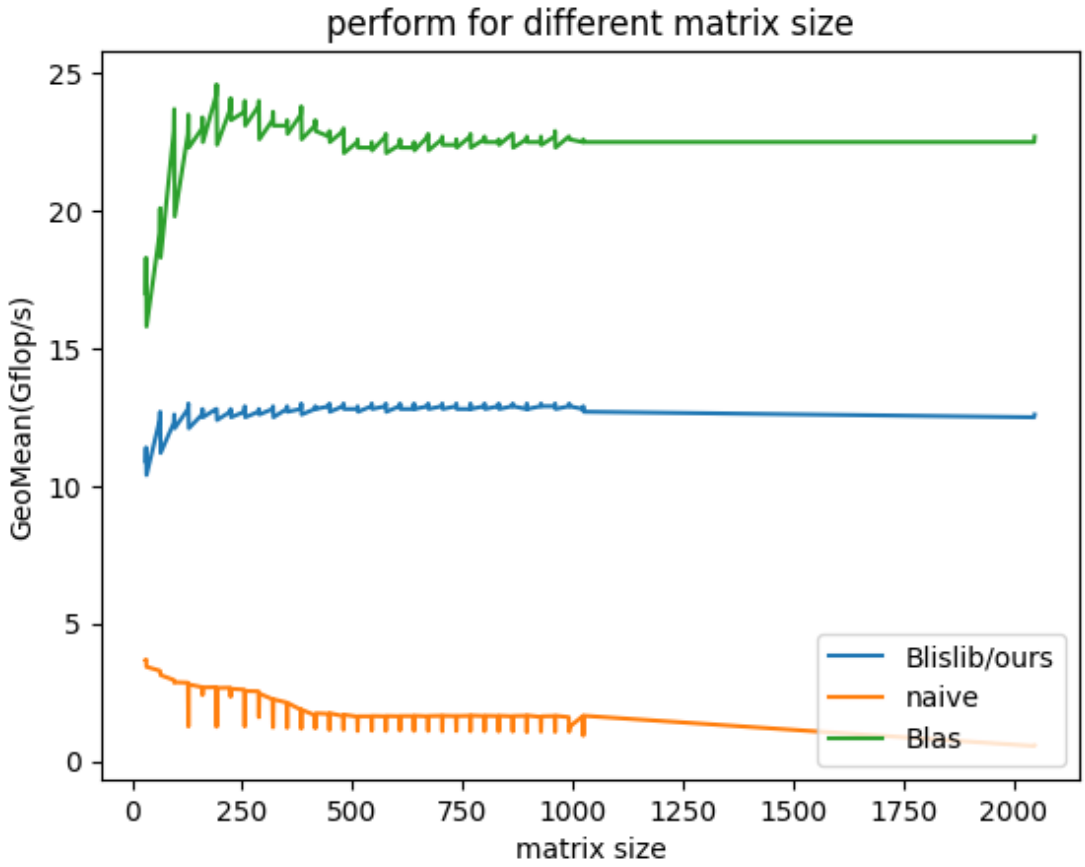
Q1. Results - 15 pts
Give a performance study for a few values (about 12 different values) of N from 32 to 2048 on your optimized code both in Q1.a. and in the file data.txt (see "what to submit->data file" for specific format. You will lose points if you do not follow this format.)

Q1.a. Show performance of your optimized code for the following numbers (fill out the table):

N	Peak GF
32	7.2465
64	11.73
128	12.55
256	12.56
511	12.645
512	12.695
513	12.395
1023	12.61
1024	12.705
1025	12.56
2047	12.265
2048	12.28

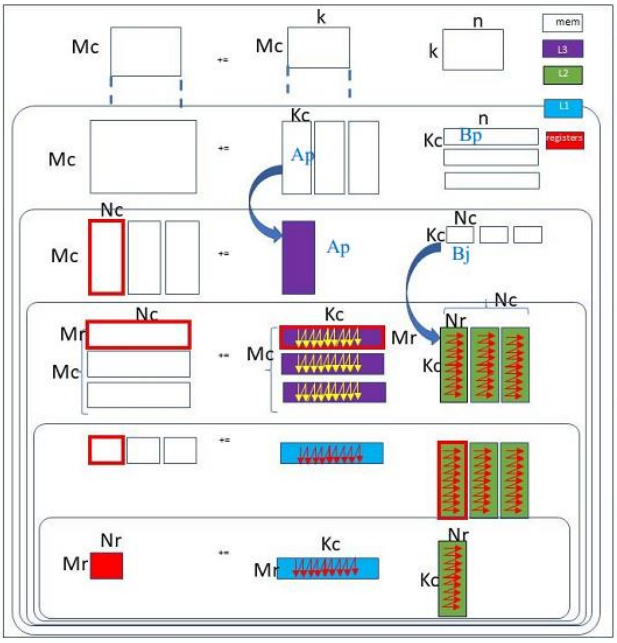
Q1.b. Make a plot of the performance of the three versions of code: the naive code, the OpenBLAS code[1], and your optimized code. OpenBLAS and your optimized code should include all N values from the table. The naive code only has to include N <= 1025.



Q2. Analysis - 33 pts

Clearly Describe:

Q2.a. How does the program work - don't include the source code, instead describe it in prose, flow chart, pseudo-code, etc.




```
Loop5: for ic = 0; ic < m; ic += Mc
         partition C into panels Cj of Mc x n
         partition A into blocks of Mc x n
Loop4:   for pc = 0; pc < k; pc += Kc
           part. A (Mc x n) -> Mc x Kc (Ap)
           pack Ap into subpanels Mr x Kc
           part. B into panels of Kc x n (Bp)
Loop3:     for jc = 0; jc < n; jc += Nc
             pack Bp into subpanels Kc x Nr
             part. Cj into panels of Mr x Nc
             // macrokernel
Loop2:       for ir = 0; ir < Mc; ic += Mr


Loop1:         for jr =0; jr < Nc; jr += Nr


                 // microkernel
Loop0:           for kr=0; kr < Kc; kr++
```

Based on J. Huang, R van de Geijn, "BLISLab: A Sandbox for Optimizing GEMM", FLAM Working Note #80, 8/31/2016

The detail flow chart is as above. In general, first partition A in to Ap and pack it in packA as above, then partition B into Bp and pack it in packB. Because we just pack them, thus their in the cache. Then take a piece of panel of A and a piece of panel of B and use the microkernel to compute it. In the microkernel, we use SIMD instructions on Arm (SVE)[2-4] to accelerate the performance. Because there are only 1 core and 1 thread for the given CPU in AWS, we did not use OpenMP.

Q2.b. Development process: What did you try , what worked, what didn't work, theories on why . Negative results are sometimes as illuminating as positive results, so try to explain as best as you can. Include the necessary graphs for the optimizations implemented. Use your github checkins to show your development process (e.g. commit logs).

In general, we first read all the code to understand the general idea and set the environment so that the Naïve and Blas version can work properly in aws. Then we try to pack A and B, but we found that we must rewrite microkernel to accommodate packA and packB at the same time. Otherwise, it did not work correctly. So we write a naïve version of microkernel without SVE. After debugging and making all of them work correctly, we try to use SVE, but found the performance can only reach about 5 Gflop/s, which is below the blas library too much. Thus, we try to optimize the microkernel, and separated and amortized the loading and saving of C block, and as a result, the performance reach about 12.76~13 Gflop/s.
Then we try to search for the best MC,NC, KC. In order to run smoothly, we write a bash script to automatically change the bl_config.h file so as to change the MC, NC, KC and collect the result to summary.txt. In order to fit the cache size, all the value of MC, NC, KC is the power of 2. Then we write a python program to extract the data and analyze the data. In this way, we plot the graph and get the best parameters for the MC, NC, KC.

We implement 2 ideas aside from the suggestions on tutorial.

1. Attempted optimization on PackA(failed).
    a) Problem statement
        In our initial version of the packing function for A, the packed A, Ã, is constructed in the order of the physical address of Ã. Specifically, with the pointer “*pack” serves as the starting address, we fill *(pack+i) (where i=0,1…) with corresponding elements of A. The issue is that the two consecutive elements in Ã, say a1 and a2, don’t reside consecutively in A. In fact, The distance between the addresses of a1 and a2 in A is ‘LdA’, which is the length of a row of A. As the size of A grows larger, in particular the number of columns of A, we can expect that a2 will not be stored in cache as a1 is accessed.

        Note: This is not an issue for packing B. According to the packing mechanism, B is accessed consecutively.

    b) Solution
        We tackled the aforementioned problem by packing A into Ã in the order of the physical address of A, instead of Ã. Specifically, we copy the element resides at ‘*(A+i)’ to the corresponding address of Ã.

c) Result and Analysis

To verify the effectiveness of our optimization, we printed the time spent on packing A for optimized and unoptimized functions respectively. Unfortunately, the time they took for packing A is roughly the same.

We investigate the problem and hereby address a few points that potentially lead to the result.

1) Both read and write calls involve moving data to cache. Due to the correspondence of A and Ã, we only choose to READ consecutively or WRITE consecutively but not both. As a result, two functions lead to the same result.

2) Each packing B call is followed by multiple packing A calls. These packing A calls disperse both spatially and temporally, which is against the nature of cache.

2. Restrict keyword on pointers

a) Problem Statement

restrict' is the keyword for a pointer of which the object it refers to will not be referred to by any other pointers. This keyword can be identified by the compiler and some optimization techniques will be implemented. In the kernel, we need to constantly load elements of packed A and B to the vector registers. The address of these elements is indicated by 2 pointers. We add 'restrict' keyword to these pointers with expectation that the program can be accelerated.

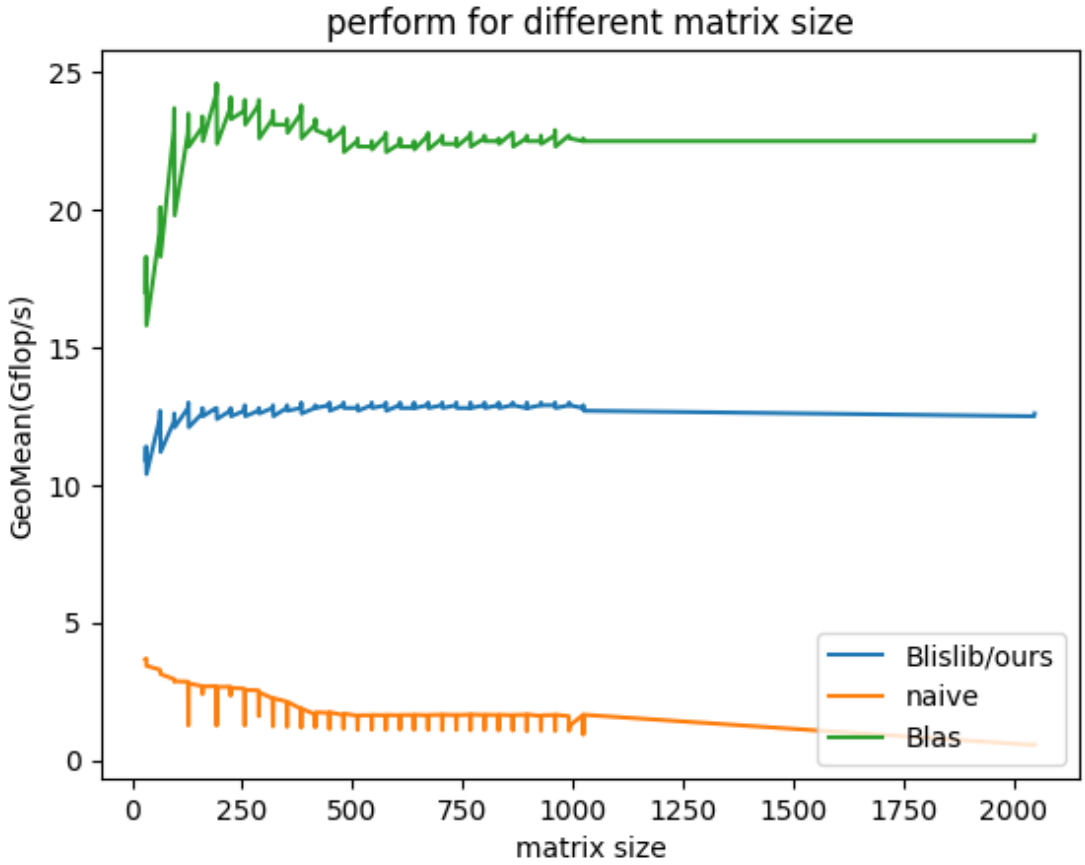
b) Solution

Add strict keyword on pointers

c) Result and Analysis

We implement this optimization and on average, the performance is improved by 0.02Gflops/s. The improvement is minor. Probably because, within the kernel, we have very few variables that will be put on registers of ALU(arithmetic and logic units), which performs scalar computation. We are doing most computation on vector registers. That may explain the performance improvement is minor.

Q2.c. Point out and explain at a high-level irregularities in the data (Places where performance scales in a non-linear way) - referring to your graph in Q1.b.



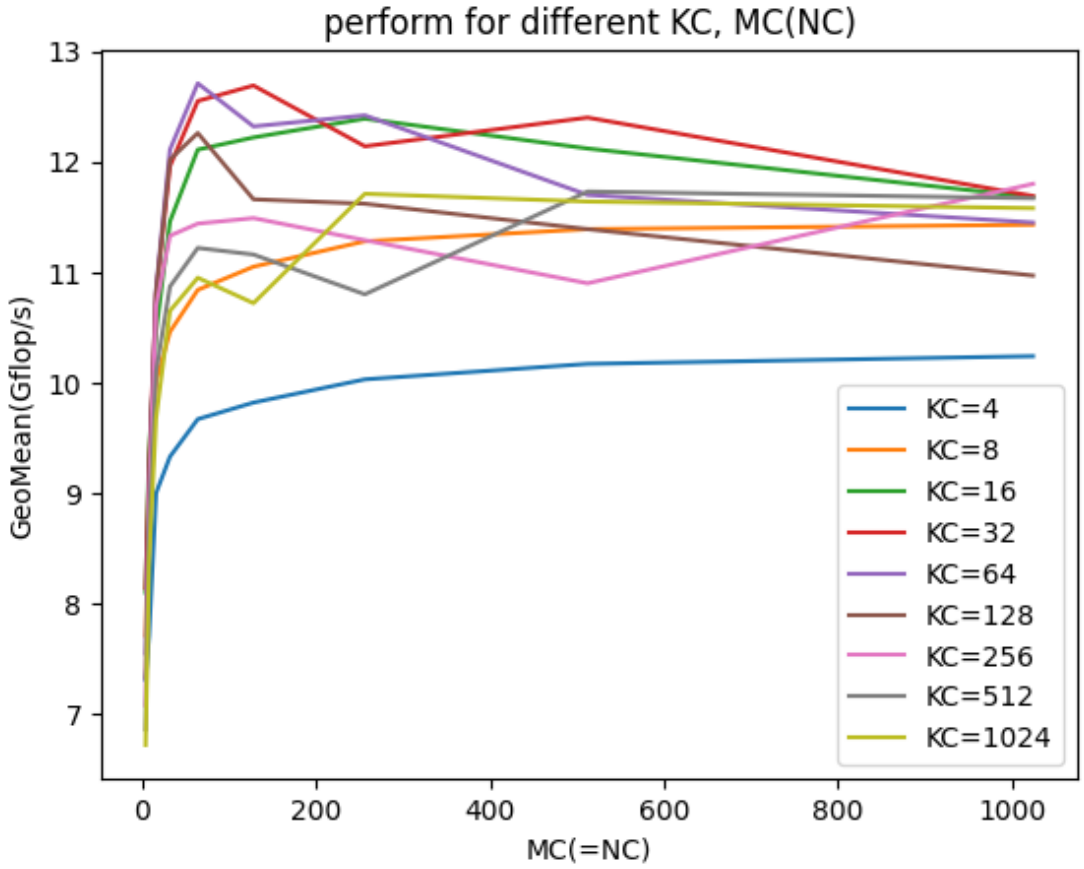
1. Analysis on the size of data

We implemented our algorithm on data with different sizes. We can notice the dramatic improvement from 32 - 256. The reason is that there exists overhead which cannot benefit from parallelism taking a fixed amount of time. Since the time doesn't grow with the size of data. We can expect the improvement as more data comes in and finally reaches its peak.

However, the performance doesn't improve monotonically with the size of data. An irregularity can be observed from the plot where there are points protruding periodically.
What is worth noting is that the data size of these protruding points is some power of 2.

Let's first consider the cases of which the corresponding data size is not power of 2. We can expect that we will be dealing with blocks, in particular those lying in the fringe, with fewer rows or columns than the vast majority. Since there exists overhead taking a fixed amount of time for each block, whenever we have these irregular blocks, we can expect the dropping of our performance.

1. Q2.d. Supporting data - e.g. analysis of cache behavior, parametric searches, or whatever will support your conclusions. Feel free to use tools such as cachegrind and knowledge of the machine's micro-architecture to support your theory . On A WS, you should have access to perf , which lets you use ARM performance counters. Note: cachegrind is slow therefore it is ok that you only measure a subset of n. Explain why we organized our skeleton code in a diff erent way from the BLISlab tutorial. Analysis on the size of block/panel



According to the graph as shown above, there are always some fluctuations of performance with different KC, MC, NC. It seems that some block size does not fit in the cache, which causes much cache miss.

Also, when the block size is too small, we did not make use of the cache and the data have to be read from memory again and again, that is why the performance is poor. At the same time, when the block size is too huge, some data will be driven out of the cache, and, thus, that is why the performance become poor again.

In particular, we find that the performance reaches its best when we choose KC = 64, MC = 64, NC = 64.

Q2.e. Future work - what could you do if you had more time?

- Our attempt over optimization of packing A failed. Our guess is that both read and write will affect the cache. Conceptually, however, only the read function will load data to memory. It is still possible to accelerate the packing process as long as the packing function is not automatically optimized by the compiler.

- Because of the fact that we are working on a processor with a single CPU and single physical thread, we didn't implement multi-thread parallelism such as OpenMP. But we can still work on that which makes our program run faster on multi-core processors.

- Till now our kernel is doing a 4-by-4 rank-1 computation. The later '4' is determined by the length of the vector register, which is 256bit therefore at most contains 4 double float numbers. A tricky issue is that we cannot attain better performance when we let the kernel do 8-by-4 rank-1 computation.

Q3. References - 2 pts

[1] https://github.com/flame/blislab - Jianyu Huang, Robert A. van de Guijn, BLISlab: A Sandbox for Optimizing GEMM, August 31, 2016
[2] Brian V an Straalen's notes on vectorizing with SSE intrinsics
[3] Interactive ARM SIMD reference pages https://developer.arm.com/architectures/instruction-sets/intrinsics/
[4] Overview of SVE: https://developer.arm.com/documentation/100891/0612/coding-considerations/using-sve-intrinsics-directly-in-your-c-code