PA1 Report Template

Here is a general rubric/template. You may get credit for additional things not on the rubric/template (e.g. extra graphs, introduction, other stuff we haven't explicitly thought of so think of this as a general guide. Depth of analysis and clarity are key objectives.

***** Your report must clearly label question numbers given here so we can easily find the different sections. Reports that require extensive time for us to read will be penalized. Please try to keep the page count around 10 pages (up to -10 pts if the report is not organized well).**

The Analysis for the report should address the following questions based on the optimization you do in the program. You are free to add extra insights regarding the assignment.

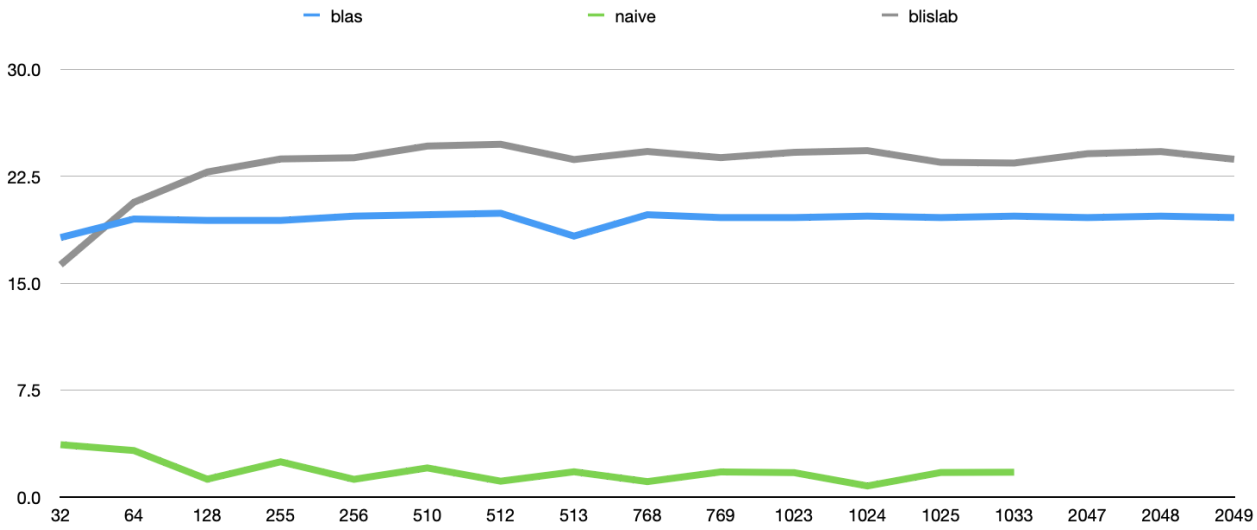
Q1. Results - 15 pts

In this section you will do a performance study for 17 values of N from 32 to 2049 (see Q1a) on your optimized code.

Q1.a. Show performance of your optimized code for the following numbers (fill out the table) and in the file data.txt (see "what to submit->data file in the instructions for the specific format. Points may be deducted for not following the format):

N	Peak GF
32	16.29
64	20.67
128	22.795
255	23.71
256	23.795
510	24.615
512	24.465
513	24.74
768	23.67
769	24.235
1023	23.805
1024	24.175
1025	24.295
1033	23.475
2047	23.425
2048	24.08
2049	24.23

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



Q2. Analysis - 33 pts

Clearly Describe:

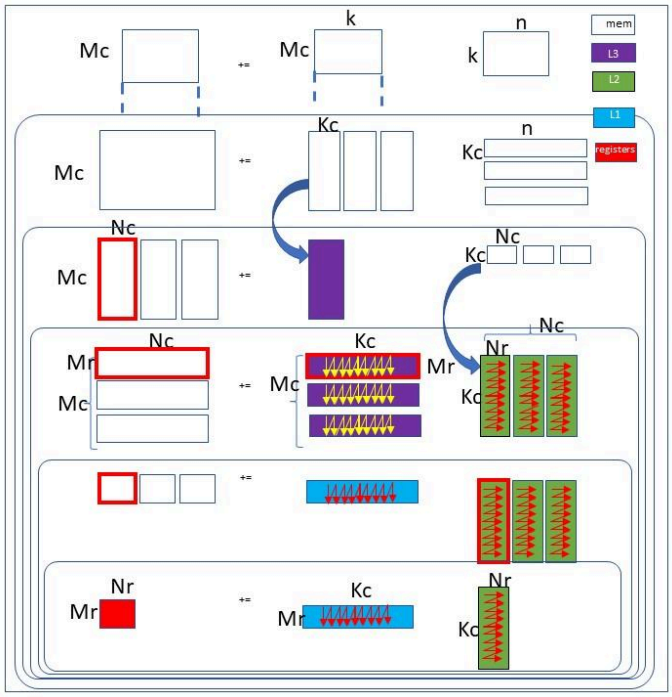
Q2.a. How does the program work - **don't include the entire source code**, instead describe it in prose, flow chart, pseudo-code, appropriately sized code snippets etc.

Answer.

In order to optimize the matrix multiplication computation, the program is designed to separate the input matrices A and B into chunks in size of MR x KC and KC x NR respectively so that the MR x NR output matrix can be small enough to fit into the registers and thus reduce the communication cost. Meanwhile, the computation is optimized to perform SIMD parallel processing with SVE intrinsics.

To realize the idea mentioned above, the program involves five loops, which could be found in *my_dgemm.c*. Considering an M x N output matrix C and input matrices A and B with size of M x K and K x N respectively, the outermost (fifth) loop first divides A into blocks(Aj) with size of MC x K. Then, the fourth loop further divides Ajs into blocks(Ap) in the size MC x KC and B into blocks(Bp) in the size of KC x N. At the same time, Aps should be packed into MR x KC subpanels in the order that the cache locality advantage would be utilized later in the microkernel computation. The third loop then divides Bp into KC x NC blocks (Bj) and packs Bjs into KC x NR subpanels for the same reason. After the data preparation, the program starts the macrokernel, which includes the 2 loops remaining. The second loop goes through MC with a step size of MR and the innermost loop goes through NC with a step size of NR. Below is a visualization and pseudo-code from the assignment instruction.

Finally, it's the microkernel that performs the computation in an SVE approach. The implementation could be found in *bl_dgemm_ukr.c* and the broadcast method is adopted. The microkernel first loads the output matrix into the SVE registers. Elements from input matrix B are loaded into the registers in order and could be used repetitively. On the other hand, elements from input matrix A need to be loaded one at a time and then be duplicated. More loads on A are needed. Take an example of MR and NR are both 4, since the vector length is 256 bits and a double floating point variable takes 64 bits, a vector can take 4 doubles at a time, which means that the MR x NR output matrix can be represented in 4 registers: c0x, c1x, c2x, c3x, each representing a row. Then the program does KC times of A[0, k]_A[0, k]_A[0, k]_A[0, k] multiplies B[k, 1]_B[k, 1]_B[k, 1]_B[k, 1] and adding them on to c0x. Similar computations are done on c1x, c2x, and c3x.




```
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 1

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 or results for the optimizations implemented. One thing we would like to see is the performance of only implementing packing routines (i.e., before implementing SVE kernel) and comparison between it and the performance of using SVE intrinsics.

> **It is required to use your github checkins to show your development process (e.g. commit logs). Refer to PA1 instructions - Part 3 Grading - analysis for more details.**

Answer.

The packing routines are implemented first. After that, the performance measured is similar to the program using a naive approach, which is around 2 GFLOPS. The reason that the performance is not improved is that the microkernel is not modified yet. The packing is just a preparation for the SIMD implementation implemented later.

After the SVE implementation, we get a much better performance at about 17 to 18 GFLOPS. The number is close to the performance of using BLAS library. The idea that is mentioned in the previous question is utilized by the program successfully.

In order to get a much more optimized performance, we further increase the number of MR and NR to utilize the hardware resource as much as we can. Besides the adjustment on the

numbers, the microkernel implementation should be modified correspondingly. By carefully dealing with the broadcast process, we are able to get a better result of around 23 to 24 GFLOPS for N>=128.

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.

Answer.

We observe obvious performance drops when N=513, 1025, 2049, compared to N=512, 1024, 2048 respectively. It is a fair result since the 256-bit vector could contain at most four 64-bit doubles at a time. In our implementation, we use two vectors for each row and thus become eight doubles. Therefore, when the matrix size is not a multiple of eight, we could not fully utilize the benefits from vectorization. Especially when the remainder is one, there are more spaces that become wasted. This irregularity can be found both in our implementation and the OpenBLAS library.

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 (caution : may or may not work with arm SVE) and knowledge of the machine's micro-architecture to support your theory. On AWS, 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 different way from the BLISlab tutorial.

Answer.

During the process of tuning parameters, we basically multiply the size by two each time until the performance degrades. Finally, we set our final parameter configuration at MC=512; KC=256; NC=128; MR=14; NR=8. Among these parameters, MR and NR are determined by the number of registers we could use and the microkernel implementation. On the other hand, the other three parameters are expected to be constrained by the cache size. For example, we utilize around 29 KiB (MRxKCx8byte) of the 64 KiB L1 cache and 262 KiB (KCxNCx8byte) out of the 1000 KiB L2 cache. Both of them could be doubled and still fit in the cache. However, we find that if we double the size, we get a better performance when the matrix size is a multiple of eight but we also get a larger penalty when encountering a matrix of size that is not a multiple of eight. We also find that L3 cache may not be critical, since increasing MC over 512 doesn't help the performance. All of these observations let us believe that we have to reserve some spaces for other data in these cache memories.

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

Answer.

In our implementation, we only explore the broadcast method and thus need more loads on the input matrix A. If we have more time, the butterfly method utilizing permutations and shufflings is also an option. It can reduce the number of data loads.

Another thing that could definitely be improved is the choice of parameters. Since we are using manual tuning, it is not feasible to go through a greater search space. There might be more ways to determine which parameters we should pick, such as writing a script or program to search with brute force or even developing a neural network. The way that we implement the broadcast is also not suitable for each case, and needs adjustment if not using the same parameter configuration. Hence, how to develop a more generalized implementation is also something which could be improved in the future.

Q2.f. (Optional) Any additional insight or optimizations that you tried implementing and how it affected the performance.

Q3. References - 2 pts

List all your references here.

1. PA1 Instructions
2. BLISlab tutorial: https://github.com/flame/blislab
3. Course material:
    a. Lecture w1_3: BLAS
    b. Lecture w2_4: SVE
    c. Lecture w2_5: Broadcast and Butterfly

Q4. Extra Credit - 5 pts

Extra Credit is 5 pts MAX (even if you do everything)

Our TA team might not be able to answer your questions on extra credits

(5 pts) Q4.a. After you have followed the instructions, you should be able to compile the edited code. But very likely (because you wrote some SVE microkernel which gives you performance good enough to start this EC), the computation becomes wrong.

(i) (2 pts) Modify your microkernel, and possibly function args when calling it, to make the computation correct.

(ii) (3 pts) Run the performance test on the same set of matrix sizes as Q1.a. Draw a plot (can be any kind) to illustrate the performance difference. Your Google Cloud performance doesn't have to surpass or be as good as the AWS performance. Instead, explain the reason for the difference. (We won't be strict on this, as long as it makes some sense.)

(5 pts) Q4.b. Implement your algorithm (including the SIMD optimization) on Arm Neon. It is another SIMD architecture that is supported on common Arm processors, such as most android phones.