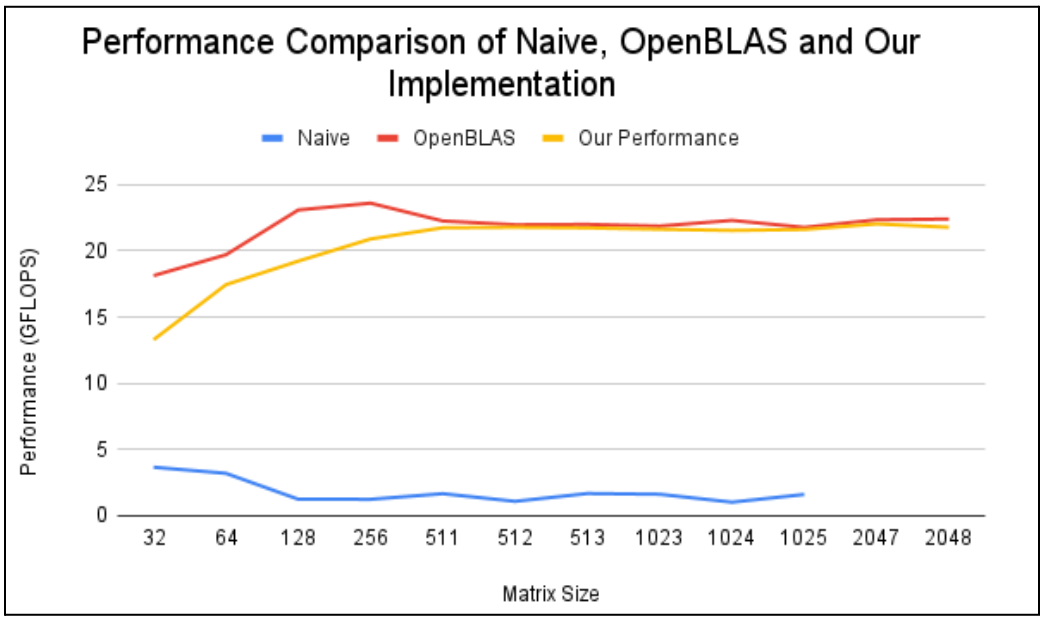
PA1 Report - Matrix Multiplication

Q1 Results

Q1a

Matrix Size	Peak GF achieved in our Implementation
32	13.295
64	17.445
128	19.225
256	20.895
511	21.745
512	21.78
513	21.735
1023	21.63
1024	21.54
1025	21.615
2047	22.035
2048	21.78

Q1b



Q2 Analysis

Q2a Working of Program

The following sequence of steps describe our code workflow:

1. bl_config.h file defines the values for Mc, Nc, Kc, Mr, and Nr parameters that are going to be used for our matrix multiplication. This file also declares BL_MICRO_KERNEL variable to dgemm_14x4x4, where dgemm_14x4x4 is the name of the microkernel that we have implemented for matrix multiplication.
2. bl_dgemm_kernel.h file declares all the different size microkernels (16x4x4, 14x4x4, 12x4x4, 10x4x4, 8x4x4, 6x4x4, 4x4x4, 2x4x4) that we have implemented as a part of this assignment.
3. bl_dgemm_ukr.c file contains definitions of all the different size microkernels that we have implemented as a part of this assignment. Function name dgemm_14x4x4 corresponds to the microkernel that performs outer product of 14 x 4 (Mr x Kc) subpanel of matrix A with a 4 x 4 (Kc x Nr) subpanel of matric B.
4. my_dgemm.c file contains the code for actual matrix multiplication. square_dgemm() function invokes bl_dgemm() function which contains three loops followed by invocation of bl_macro_kernel().
    a. The 5th loop around microkernel (as commented in code) partitions matrix A and C into submatrices of size Mc x n.
    b. The 4th loop around microkernel (as commented in code) further partitions A into panels of size Mc x Kc, and partitions matrix B into submatrix of size Kc x n.
        i. This loop also contains a loop to pack subpanels of size Mr x Kc into each panel of size Mc x Kc.
        ii. Each Mr x Kc subpanel is accessed in column major order and elements are saved in that order in an array named packA.
    c. The 3rd loop around microkernel (as commented in code) further partitions B into panels of size Kc x Nc.
        i. This loop also contains a loop to pack subpanels of size Kc x Nr into each panel of size Kc x Nc.
        ii. Each Kc x Nr subpanel is accessed in row major order and elements are saved in that order in an array named packB.
    d. The bl_macro_kernel() contains the 2nd and 1st loop around microkernel (as commented in code) that passes a Mr x Kc subpanel of A and Kc x Nr subpanel of B to microkernel.
    e. Finally, the dgemm_14x4x4() microkernel reads 14 values stored in Mr x Nr subpanel in matrix C, and one by one performs an outer product of a column of 14 x 4 (Mr x Kc) subpanel of matrix A with a row of 4 x 4 (Kc x Nr) subpanel of matric B. It makes use of SVE instructions to multiply an element of matrix A with four elements in row of matrix B in a single instruction.

Q2b Development Process

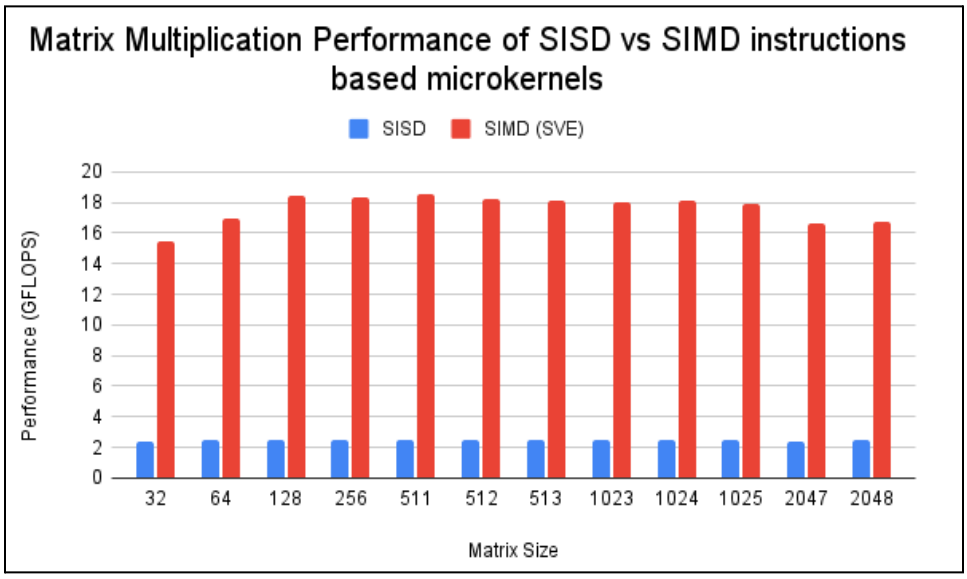
This section describes the seven optimizations that we tried for this assignment. The first four optimizations yielded us performance gain while the rest three optimizations did not give us any performance gain.

Optimization 1: Implementing a microkernel using scalable vector registers and SVE instructions

Commit Message	Commit Link
Finished packing for naive microkernel	https://github.com/cse260-fa22/pa1-hgondali-shananth/commit/9b6ba86c74ee9f0e2aebd94163ff12a6cdaf6edf
Added 4x4 microkernel	https://github.com/cse260-fa22/pa1-hgondali-shananth/commit/811619117ae7f8e82d2380070d7ecd5e5607e624

Methodology and Reasoning: We implemented a 4x4 (Mr x Nr) microkernel that performs matrix multiplication of Mr x Kc subpanel of A with Kc x Nr subpanel of B using SIMD-based SVE instructions. This

microkernel is compared against a naive microkernel that performs the same computation using SISD-based instructions. These microkernels were used to perform matrix multiplications for different matrix sizes and their results are compared.



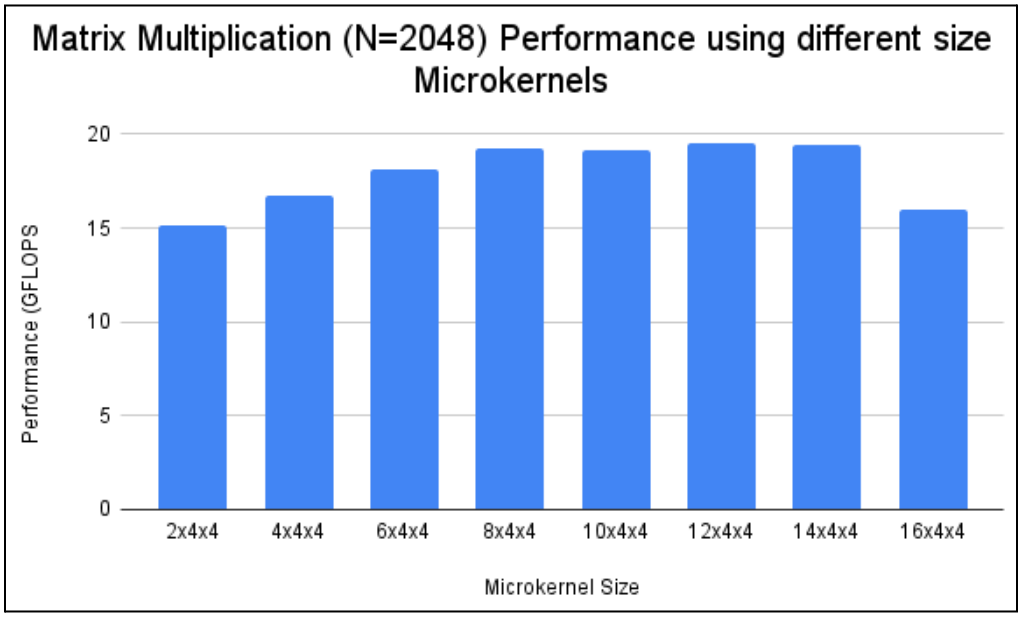
Results Analysis: In the case of SISD instructions based microkernel, a single instruction can perform a single multiplication operation for a single set of operands. On the other hand, SVE instructions adopt a SIMD approach where four multiplication operation can be performed at the same time for four independent sets of operands. Thus, SVE instruction based microkernel can perform the same number of operations in significantly lower time than the naive kernel. Due to this reason, SVE instruction based microkernel gives a noticeable performance enhancement of 14-16 Gflops than the naive microkernel.

Conclusion: Using SVE instructions (i.e. SIMD instructions) in microkernel gave a **performance enhancement of about 14 to 16 Gflops** for square matrix multiplication of different matrix sizes**.**

Optimization 2: Choosing a micorkernel size

Commit Message	Commit Link
Added 4x4 microkernel	https://github.com/cse260-fa22/pa1-hgondali-shananth/commit/811619117ae7f8e82d2380070d7ecd5e5607e624
Added 10x4 8x4 6x4 2x4 micorkernel	https://github.com/cse260-fa22/pa1-hgondali-shananth/commit/c63204394f742363e30f657ea33b02ffcc51a223
Added 12x4 microkernel	https://github.com/cse260-fa22/pa1-hgondali-shananth/commit/c5c3ee8be237f1ef9df9a8d6d919713b1b600d7f
Added 14x4 and 16x4 micorkernel	https://github.com/cse260-fa22/pa1-hgondali-shananth/commit/da439cbfdaf76066826f8ab1dd5ae67645fad251
Added dumps for 12x4 14x4 16x4 kernel	https://github.com/cse260-fa22/pa1-hgondali-shananth/commit/d9227aa2e7046212ba7e485a83fedd856139c5cb

Methodology and Reasoning:
Our microkernel is responsible for calculating multiplication results for Mr x Nr block of matrix C. Throughout our experiment, we have kept Nr value as constant (i.e. equal to 4). This is with the insight that the vector length of Neoverse V1 is 256 bits [2]. Thus, by setting Nr value to 4, the four 64-bit double precision numbers will utilize vector width to its full capacity.
We could have not tried Nr values as 8, 12, 16, etc., but we reasoned out that there is more merit in using the available SVE registers for getting more depth along Mr dimension. This is because elements of Mr x Kc block are going to be accessed more frequently and thus will be in L1 cache while the elements of Kc x Nr subpanel are going to be accessed relatively less frequently and will be in the L2 cache.
Hence, this this experiment we implemented different microkernels by varying Mr size and keeping Nr as constant.



Results Analysis:
For Mr value 2 to 14, we see an **increase** in performance for matrix multiplication for square matrices of size 2048. This is because register access costs least latency when compared to other forms of memory. Thus, as we increase Mr value, relatively more operands are read from registers which results in lesser time to fetch data. This is the reason for improvement in performance.
As we increase Mr value from 14 to 16, we see a **decrease** in performance. This is because SVE instruction set only has 32 scalable vector registers. When Mr value is set to 16, we run out of registers and thus need to access L1 cache to fetch some operand values. Since access to L1 cache causes more latency than access to registers, the performance for microkernel with Mr value as 16 is lesser than the micorkernel with Mr value as 14.
We verified our reasoning by checking the assembly code for microkernels [1] with Mr value 12, 14, and 16. We clearly see that the microkernel with Mr value 16 has used up all 32 registers and **needs to use register z0.d twice (line 55 and 72 in 16x4.txt at [1]) for loading values**. Thus, in each iteration there will be one L1 cache access to get the required operand.

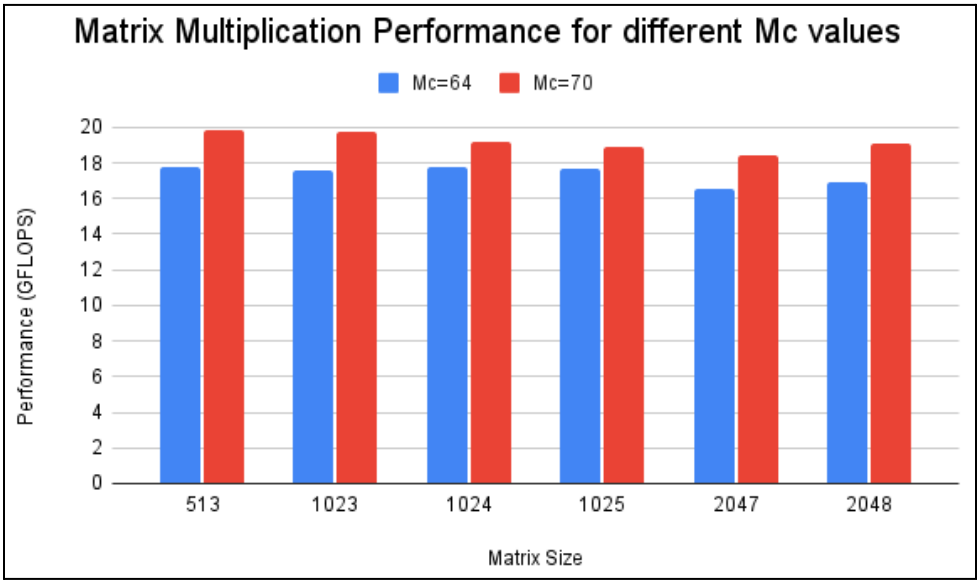
Conclusion: With all the results and analysis shared above, **we chose 14 x 4 (Mr = 14, Nr = 4) as our micro-kernel size**

Optimization 3: Choosing Mc and Nc values that are multiples of Mr and Nr respectively

Commit Message	Commit Link

Results for Mc value 70 and 64 for 14x4 microkernel	https://github.com/cse260-fa22/pa1-hgondali-shanantha/commit/815be8656ee8b2ed61fe8464e444d35d1dd47048
Results for Nc value 64 and 70 for 14x4 microkernel	https://github.com/cse260-fa22/pa1-hgondali-shanantha/commit/c2753b6059b82adab916e53afa9f2ffb117815b1

Methodology and Reasoning: In our experiments, we came across this observation that we must choose Mc and Nc that are multiples of our Mr and Nr respectively (i.e. 14 and 4 respectively). This is because if we choose values that are not a multiple of Mr and Nr then each Mr x Kc and Kc x Nr subpanel will have padding and we will be performing unrequired computations in each subpanel. By choosing Mc and Nc that are multiples of our Mr and Nr respectively, we ensure that padding only needs to be done in subpanels that fall along the bottomost rows and rightmost columns of the matrix A and B.



Results Analysis: We notice a performance **difference of around 0.1 to 2 Gflops** between experiments where Mc and Nc values are multiples of Mr and Nr respectively and the case where they are not a multiple of Mr and Nr respectively. Due to space constraints, only plot for Mc value variation is shown.

Conclusion: The chosen value of **Mc and Nc must be a multiple of Mr and Nr** respectively.

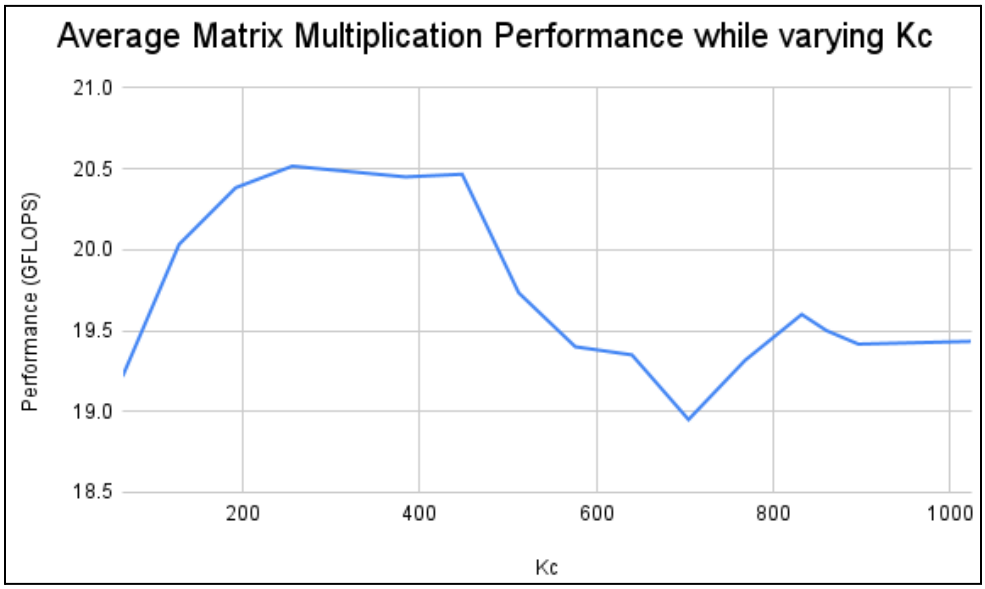
Optimization 4: Searching for best possible (fine-tuned) values of Mc, Nc, and Kc

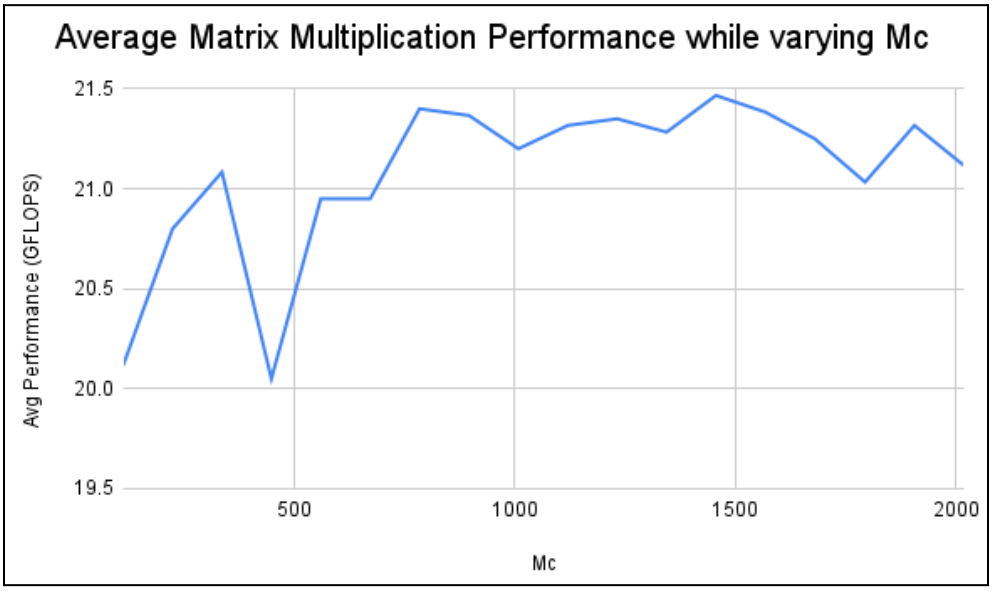
Commit Message	Commit Link
Results while varying Kc	https://github.com/cse260-fa22/pa1-hgondali-shanantha/commit/3c0da5dfebb711f1256505fd1659db5abad2543b
Results while varying Mc	https://github.com/cse260-fa22/pa1-hgondali-shanantha/commit/d135ea15672e7636647d8686d5b3b655f48cf3d2
Results while varying Nc	https://github.com/cse260-fa22/pa1-hgondali-shanantha/commit/8b0dce5f2f54343f52829c80a3494b5d9d0f040d

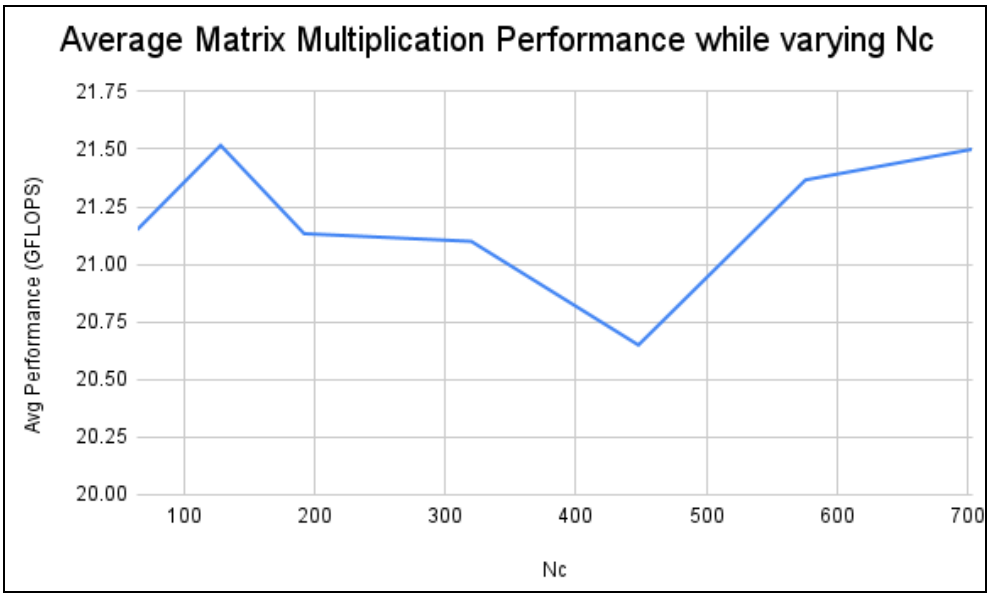
Perf counters while varying Mc, Nc, Kc	https://github.com/cse260-fa22/pa1-hgondali-shananth/commit/ee08f528485990591a6be11f0d0112a3a7c9361a

Methodology and Reasoning: Given the paucity of time, we figured out the best set of Kc, Mc, and Nc values for a 14 x 4 (Mr x Nr) size microkernel in the following manner:

1. Kept Mc = 70 (multiple of 14, i.e. Mr value), Nc = 64 (default value), and varied Kc value from 64 to 1024 in increments of 64. Found that Kc value of 448 gives us good performance results and hence used it further in experiments to find the best Mc and Nc value.
2. Kept Kc = 448, Nc = 64 (default value), and varied Mc value from 112 to 1232 in increments of 112 (multiple of 14, i.e. Mr value). Found that Mc value of 784 gives us good performance results and hence used it further in experiments to find the best Nc value.
3. Kept Mc = 784, Kc = 448, and varied Nc value from 64 to 704 in increments of 64. Found that Nc value of 128 gives us good performance results.







Results Analysis:

Varying Kc value: The hardware brings entire cachelines worth of data to L1 cache when any one of its data element is accessed. Moreover, hardware and software prefetchers also bring adjacent cache lines into cache to take advantage of spatial locality.

Hence, if we do not have a larger Kc value then we would not be capitalizing upon the data (in the same cacheline and one in the prefetched cacheline) that is brought into cache. On the other hand, by having larger Kc value, our performance will improve because more number of data accesses done in the Mr x Kc subpanel will be satisfied in the L1 cache itself. Our measurement results from the Linux Perf Tool [4] validate our reasoning by revealing that the **L1d cache miss rate decreases by 1.65%** as we move from Kc value 64 to 448.

Secondly, product of 448 (Kc), 14 (Mr), 8 (double-precision size) roughly equals to 50 KB which can be accommodate entirely in the 64 KB L1-d cache.

Kc Value	L1-d Cache Miss Rate
64	3.65
448	2.00

Varying Mc value: As we increase the value of Mc, more number of Mr x Kc panels could fit into the Mc x Kc panel. As per the access pattern, Mc x Kc panel resides in L3 cache and by increasing Mc size, more number of Mr x Kc subpanel could be accommodated in the panel. Hence, when these subpanels are packed together, we can again take advantage of the cacheline worth of data that is brought in altogether while performing operations. The cost of bring Mc x Kc panel into L3 cache is amortized over multiple Mr x Kc subpanels that settle the reads for operations at the L1 cache itself. This leads to an increase in performance. Our measurement results from the Linux Perf Tool [4] validate our reasoning by revealing that the **L1d cache miss rate decreases by 0.56%** as we move from Mc value 112 to 784.

Mc Value	L1-d Cache Miss Rate
112	1.59
784	1.03

Varying Nc value: Increasing Nc value did not give us any significant improvement in performance. We suspect this is because increasing Nc value increases the number of Kc x Nr subpanels that could fit into Kc x Nc panel. Since Kc x Nr subpanel is accessed less frequently and is always pushed down to L2 cache as per the access pattern, there is no significant performance improvement by varying Nc.

Conclusion: Mc = 784, Kc = 448, Nc = 128 values gives us best possible performance enhancement. Finally, with all the four optimizations shared above, we are able to reach an average performance of **~21 Gflops**.

Optimization 5: Using alternative SVE instructions

Commit Message	Commit Link
Using svmla_f64_x instruction in microkernel	https://github.com/cse260-fa22/pa1-hgondali-shananth/commit/7177bea95efef2725d94eca6f88df954bc7d67f9
Using svmla_f64_z instruction in microkernel	https://github.com/cse260-fa22/pa1-hgondali-shananth/commit/7177bea95efef2725d94eca6f88df954bc7d67f9

Instead of svmla_f64_m, we tried using svmla_f64_x and svmla_f64_z instructions to perform multiply, load, and add operations. This did not give any performance enhancement. Rather, implementation using svmla_f64_z instruction **decreased the performance by around 1-2 Gflops**. This is because svmla_f64_z instruction cost more CPU cycles than the other two instructions.

Optimization 6: Rearranging instructions within micro-kernel

Commit Message	Commit Link
Access B followed by A	https://github.com/cse260-fa22/pa1-hgondali-shananth/commit/db3e1de2bd7bc47448e614998e019427d26695a0

Inside the microkernel, we tried loading all values from matrix A first followed by B, and in other approach tried loading value of matrix B first followed by loading all values of matrix A. This did not give any performance enhancement because objdump revealed that compiler generated same assembly code for both the approaches.

Optimization 7: Using addition in place of multiplication to compute load address

Commit Message	Commit Link
Use add in place of multiply to compute load address	https://github.com/cse260-fa22/pa1-hgondali-shananth/commit/9a0a82ff927371f334a35c9e32c1c88ab727c774

Instead of svld1_f64(npred, C+(ldc*3)), we tried using svld1_f64(npred, C+(ldc+ldc+ldc)) for all such load instructions. This did not give any performance enhancement because objdump revealed that compiler generated same assembly code for both the approaches.

Q2c Explanation of irregularities seen in graph shown in Q1b

Our implementation's performance increases steeply as we move from matrix size 32 to 512. This is because smaller the matrix size (lesser computations), the lesser we can take advantage of cacheline worth of data (64B) that has been brought into cache when any one of the data stored in cacheline is read. As a result, we are not amortizing the cost of data packing (bring data to caches) over enough computations and hence performance is lower.

Our implementation performs noticeably better than naive approach over all matrix sizes due to the data packing and SIMD-based vectorization approach that amortizes the cost of data access over multiple computations.

Q2d Analysis of Cache Behavior and Difference in Implementation from BLISLab Tutorial

Note: Graphs, Performance Counter Readings, and Calculations referenced in this subsection are available in "Optimization 4" subsection in Q2b

We observe that increasing the value of Kc helps us take advantage of the cacheline worth of data that is brought into L1 cache for Mr x Kc subpanel. As Kc value increases, we are able to amortize the cost of bringing a cacheline worth of data over more number of computations and hence we see an increase in performance. The performance counters showed clear decrease in L1 cache miss rate with increasing Kc because more cachelines (including prefetched ones) are used in larger size Mr x Kc subpanel. The calculations and graph for average performance while varying Kc clearly shows that when Kc is value is more than 448 we are almost at the full capacity of L1 cache (64KB) and hence increasing Kc beyond that value leads to a decrease in performance.

Increasing Mc value helps us accommodate more Mr x Kc subpanels in a single Mc x Kc panel. Hence, cost of bringing Mc x Kc panel into L3 cache is amortized over multiple Mr x Kc subpanels that handle the reads for operations at the L1 cache itself. Performance counter readings clearly show a decrease in L1-d cache miss rate when Mc value is increased.

Increasing Nc value did not give us any significant improvement in performance because Kc x Nr subpanel is accessed less frequently and is always pushed down to L2 cache as per the access pattern. Thus, there is no significant performance improvement by varying Nc and thereby fitting in more Kc x Nr subpanels in Kc x Nc panel.

Our implementation is based on row-major storage order while BLISLab tutorial is based on column-major storage order. For a column major order based implementation, its better to put Kc x Nc subpanels of B into L3 cache (contiguous addresses within columns) rather than Mc x Kc subpanel of A (non-contiguous addresses within rows). Similarly, for a row-major order based implementation, its better to put Mc x Kc subpanel of A (contiguous addresses within rows) rather than Kc x Nc subpanels of B into L3 cache (non-contiguous addresses within columns). Based on this, to take advantage of storage order during packing and further computation, arrangement of loops around micokernel differs in our and BLISLab implementation.

Q2e Future Work

1. If we had more time, then we would like to implement the butterfly multiplication method described in BLISLab tutorial and see how much performance enhancement that can bring.
2. Our performance results are around 0.3 to 0.9 Gflops less than that of OpenBLAS implementation for different matrix sizes. We would like to do a more fine-grained parameter tuning to see if we can achieve better performance.

Q3 References

[1] Assembly Code for 12x4, 14x4, and 16x4 microkernels.
https://github.com/cse260-fa22/pa1-hgondali-shananth/tree/main/dumps

[2] Neoverse V1 Specification
https://en.wikichip.org/wiki/arm_holdings/microarchitectures/neoverse_v1

[3] SVE Architecture Fundamentals
https://developer.arm.com/documentation/102476/0100/SVE-architecture-fundamentals

[4] Linux Perf Tool
https://perf.wiki.kernel.org/index.php/Main_Page

[5] Intel Intrinsics Guide
https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

Q4 Extra Credit Work - Matrix Multiplication using AVX2 intrinsics

Commit Message	Commit Link
Compared our implementation with OpenBLAS and Naive	https://github.com/cse260-fa22/pa1-hgondali-shananth/commit/0bc4029788597fb265be560df30714f2bbcf4f33

We implemented the matrix multiplication using AVX2 intrinsics on a x86 Ubuntu machine on Amazon EC2, and made use of the t2.micro instance type. The Makefile was modified to replace the SVE and ARM related flags with x86 and AVX2 flags. We modified the microkernel to make use of AVX2 intrinsics instead of SVE.

In order to optimize the use of registers, we use a 7x4 microkernel that makes use of exactly 16 registers. We experimented with microkernels of size 2x4, 4x4 and 8x4. There was a gradual performance improvement as our Mr value increased from 2 to 7 followed by a sudden drop when Mr is 8. This is because we have completely exchaused the registers when Mr is 7 which then causes data to be picked up from the L1 cache when Mr increases to 8.

We also experimented with the values of Kc, Mc and Nc and found that Kc = 224, Mc = 700, Nc = 120 to give us the best result. This can be attributed to cache hierarchy - the sizes of L3, L2 and L1 caches.

Unlike SVE, AVX2 does not make use of predicates and so our initial implementation worked only when the matrix size is a power of 2. To overcome this, we made use of a mask that changes according to the size of the leading dimension of the second matrix (Matrix B) that is input to the microkernel. Subsequently, a different form of the AVX2 intrinsic was chosen to perform masked loads and stores. The table below shows a comparison of the performances of the naive, OpenBLAS and our implementation of matrix multiplication. We achieve an average performance of **22.2 Gflops** for matrix sizes greater than 512 - which is around **65%** of the OpenBLAS implementation.

Matrix Size	Naive	OpenBLAS	Peak GF achieved in our Implementation
513	1.3525	31.805	22.01
1023	1.04145	34.1	22.135
1024	0.12865	34.75	22.65
1025	0.9514	33.645	22.075
2047	NA	35.17	22.51
2048	NA	35.125	21.78