Q1. Results - 15 pts

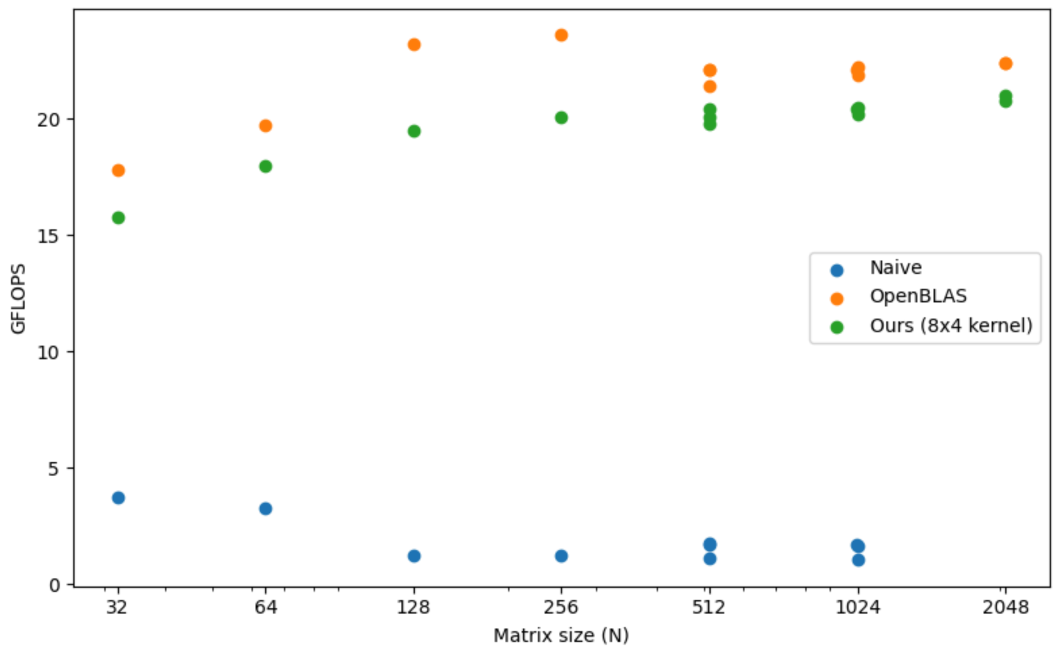
Give a performance study for a few values (about 12 different values) of N from 32 to 2048 on your optimized code both in Q1.a. and in the file data.txt (see "what to submit->data file"   for specific format. You will lose points if you do not follow this format.)

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

N	Peak GF
32	15.8
64	18.0
128	19.5
256	20.1
511	20.1
512	20.4
513	19.8
1023	20.4
1024	20.5
1025	20.2
2047	20.8
2048	21.0

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 N values from the table. The naive code only has to include N <= 1025.

The performance of the 3 implementations are shown in the figure below. We also report the values in a table format as the figure does not clearly differentiate similar Ns.



N	Peak GF, naive	Peak GF, OpenBLAS	Peak GF, ours (8x4 kernel)
32	3.73	17.8	15.8
64	3.27	19.7	18.0
128	1.25	23.2	19.5
256	1.26	23.6	20.1
511	1.76	22.1	20.1
512	1.10	22.1	20.4
513	1.69	21.4	19.8
1023	1.71	22.1	20.4
1024	1.05	22.2	20.5
1025	1.67	21.9	20.2
2047		22.4	20.8
2048		22.4	21.0

Q2. Analysis - 33 pts

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

The program implements 5 levels of loops that break down the matrices into smaller submatrices or panels. This allows us to put different submatrices in different levels of cache to speed up data access and improve the overall efficiency.

- Loop 5: This loop partitions matrix C into stripes of Mc*N and matrix A into stripes of Mc*N. Each stripe of C is the product of a stripe of A and the matrix B
- Loop 4: This loop partitions a Mc*N stripe of A into panels of Mc*Kc and B into stripes of Kc*N. An Mc*Kc panel of A is then packed into subpanels of Mr x Kc. This makes the panel continuous in memory (and in the right memory order for the micro kernel).
- Loop 3: This loop splits a Kc*N stripe of B into Kc*Nc panels. Each panel is then packed into Kc*Nr subpanels and multiplied with the Mc*Kc panel of A.
- Loop 1&2 iterates over Kc*Nr subpanels of B and Mr*Kc subpanels of A to compute a Mc*Nc block of C. This is the macro kernel.
- Loop0: This is the optimized microkernel that multiplies a subpanel of A with a subpanel of B by taking Kc outer products.

In the ideal case this setup puts different parts of matrix A, B, and C in different levels of the memory hierarchy: one Mr * Nr chunk of matric C is stored in the register file; a Mr*Kc subpanel of matrix A is put in L1 cache; a Kc * Nc panel of matrix A is put in L2 cache; and a Mc*Kc panel of matrix B is stored in the L3 cache.

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. 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.

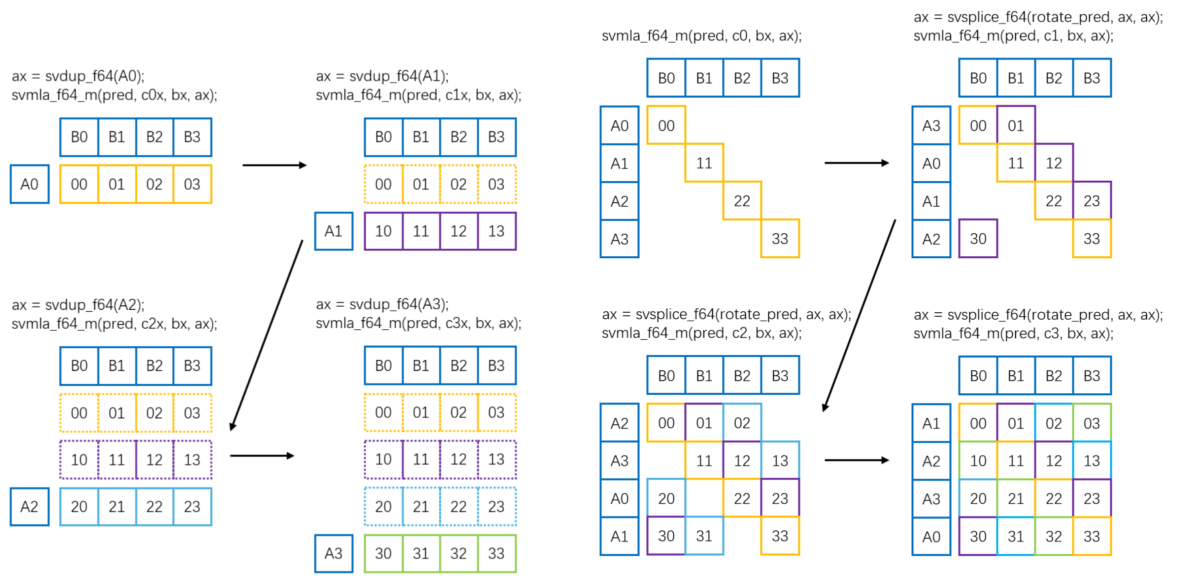
We first implemented the packing routine (commit 33af18a). As shown in the table below, compared to the skeleton code, packing did not increase performance and in fact made it worse for small values of matrix size. This makes sense because for small N, the packing overhead accounts for a larger fraction of the overall run time (see Q2.d.1). But for larger sizes like 2047 and 2048, it increased the Gflops a little bit.

N	32	64	128	256	511	512	513	1023	1024	1025	2047	2048
Starter code	2.5	2.6	2.5	2.4	2.5	2.4	2.5	2.5	2.4	2.5	2.3	2.3

+ packing	2.3	2.4	2.5	2.5	2.5	2.4	2.5	2.5	2.5	2.4	2.4	2.5
+ 4x4 SVE	14.6	17.9	18.4	18.2	18.0	18.0	17.8	18.5	18.3	17.8	18.1	18.1

Average performances of the starter code, after implementing packing, and using SVE intrinsics.

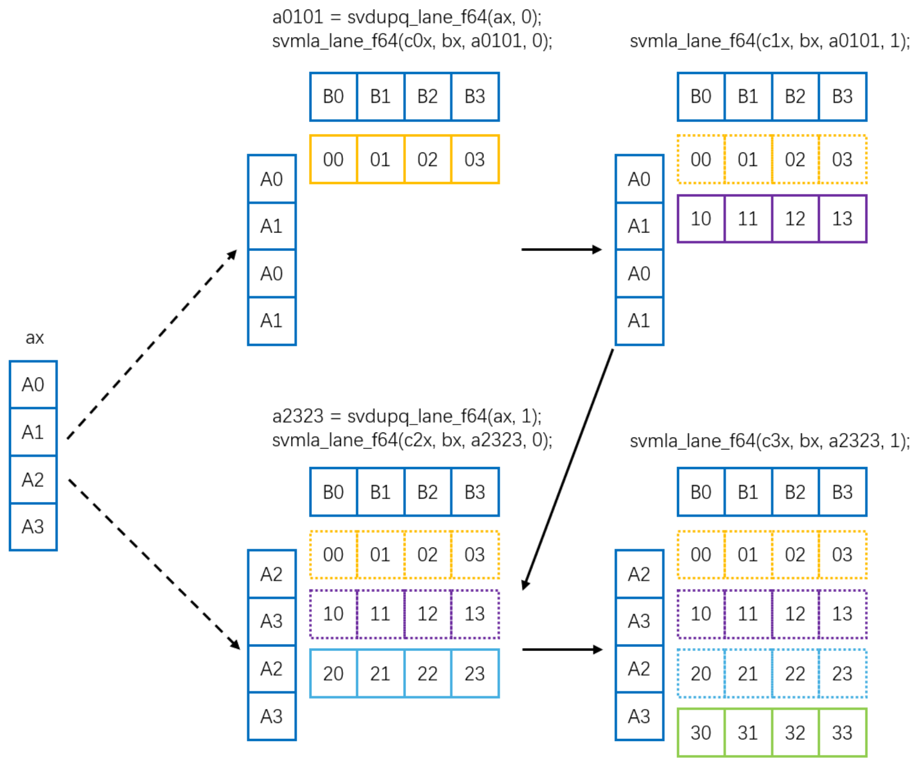
Next, we implemented microkernels with SVE intrinsics using the `svdup` + `svmla` method covered during the lecture. We tried several kernel sizes, including 2x4 (commit 65f01f8), 4x4, 8x4 (commit ee77578), 4x8, 4x12 (commit 502f18a).


```
ax = svdup_f64(A0);
svmla_f64_m(pred, c0x, bx, ax);

ax = svdup_f64(A1);
svmla_f64_m(pred, c1x, bx, ax);

ax = svdup_f64(A2);
svmla_f64_m(pred, c2x, bx, ax);

ax = svdup_f64(A3);
svmla_f64_m(pred, c3x, bx, ax);

svmla_f64_m(pred, c0, bx, ax);

ax = svsplice_f64(rotate_pred, ax, ax);
svmla_f64_m(pred, c1, bx, ax);

ax = svsplice_f64(rotate_pred, ax, ax);
svmla_f64_m(pred, c2, bx, ax);

ax = svsplice_f64(rotate_pred, ax, ax);
svmla_f64_m(pred, c3, bx, ax);
```

Left: “normal” 4x4 kernel, right: modified butterfly kernel

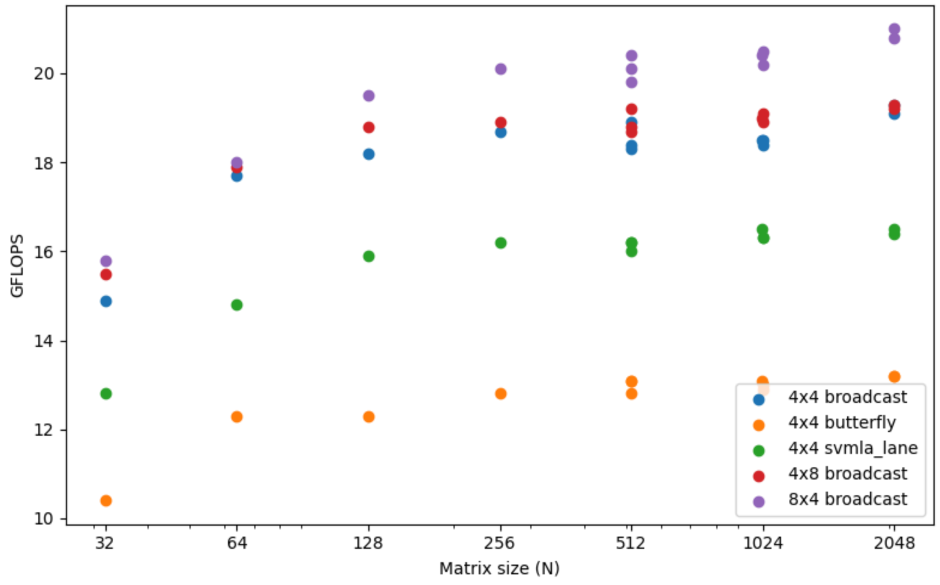
For a m*n microkernel (where m and n are multiples of 4), we use m*n/4 registers to hold the m*n block of matrix C. We first load n elements of B into n/4 registers (i.e., 1 register if n=4, 2 registers if n=8). Next, we load and broadcast one element of A into register `ax`, and use `svmla` to update the corresponding C register. This is repeated m times until we multiplied m elements of A with n elements of B. We use predicates to mask off invalid elements when the actual input is smaller than m*n. The above figure shows a 4x4 example. As shown in the table above, by taking advantage of the SIMD hardware to increase the number of operations per cycle, we achieve significantly better performance.

In addition to the `svdup` + `svmla` microkernel, we also implemented a modified butterfly kernel (commit d970ab4). Instead of using `svrev` and `svsplice` to permute the A register, we repeat `svsplice` 3 times to rotate register A using a “rotation predicate” (the highest bit is set, while the rest are cleared). This allows us to load A and B only once for each 4x4 update. However, as the elements of C are not in memory order, we need to use `svld1_gather_u64offset_f64` and `svst1_scatter_u64offset_f64` to load and store the results. Note this requires a bit extra work to set up the offset register and the predicate (in case the input is smaller than 4x4). We only implemented a 4x4 version of this kernel, and the rotation process is shown in the above figure.


a0101 = svdupq_lane_f64(ax, 0);
svmla_lane_f64(c0x, bx, a0101, 0);
svmla_lane_f64(c1x, bx, a0101, 1);
a2323 = svdupq_lane_f64(ax, 1);
svmla_lane_f64(c2x, bx, a2323, 0);
svmla_lane_f64(c3x, bx, a2323, 1);
ax
A0
A1
A2
A3
B0
B1
B2
B3
00 01 02 03
10 11 12 13
20 21 22 23
30 31 32 33

Lastly, inspired by the kernel shown in SVE and Neon coding compared [1], we implemented another kernel with `svmla_lane_f64` (commit 4649102). The overall flow is shown in the above figure. Since `svmla_lane_f64` indexes elements within each 128b segment [2], we first tile the low & high 128b segments of register A using `svdupq_lane_f64`. Then, we can use `svmla_lane_f64` to update C registers by indexing the tiled registers (i.e., `a0101` and `a2323`). Compared to the broadcasting version, this kernel has the theoretical benefit that the register A only needs to be loaded from memory once per 4x4 update (similar to the butterfly kernel). Additionally, since the C values are already in memory order, loads and stores will be more efficient.

The below figure shows the performance of different microkernels. Overall, the 8x4 kernel that broadcasts A (`svdup` + `svmla`) performs the best. Compared to the 8x4 broadcast kernel, 4x4 and 4x8 versions perform slightly worse. Surprisingly, the `svmla_lane` 4x4 kernel performs significantly worse than the 4x4 broadcast kernel, despite only loading A register once per 4x4 update. Lastly, the 4x4 butterfly kernel performs the worst.



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.

Our implementation has performance dips when N is not a multiple of 8. This is especially noticeable when N is just a little over a multiple of 8, and N is small (e.g., N=513). This is because even if a Mr * Nr block of matrix C is smaller than the size of the 8x4 kernel, it still takes a single microkernel call. This effectively pads the matrices to the nearest multiple of 8 and 4 (e.g., for N=513, the effective matrix size is 520x516). These wasted flops in the padded region leads to the reduced performance. Additionally, note the area of the padded region over the entire matrix is O(N / N^2) = O(1/N), so the effect is more pronounced for small Ns.

This behavior is shown in the figure (and the table) in Q1.b as a significant dip for N=513 and N=1025. For N=511, N=1023, and N=2047, as we only need to pad 1 in each direction (compared to 7 in the N=2^k + 1 cases), less padding is needed and the dips are less noticeable.

Q2.d. Supporting data - e.g. analysis of <u>cache behavior</u>, 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 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.

1. **Why does our 8x4 kernel (and all kernels in general) perform worse with smaller Ns?**

We hypothesize that this is because the time complexity of packing A is O(N^2), while the complexity of GEMM is O(N^3). As a result, when N is small, the packing overhead can have more impact on performance.

To verify this hypothesis, we use `gprof` to profile `benchmark-blislab` with N=32 and N=2048. To take more accurate measurements, we increase the timeout to 5 seconds. The below table shows the profiling results.

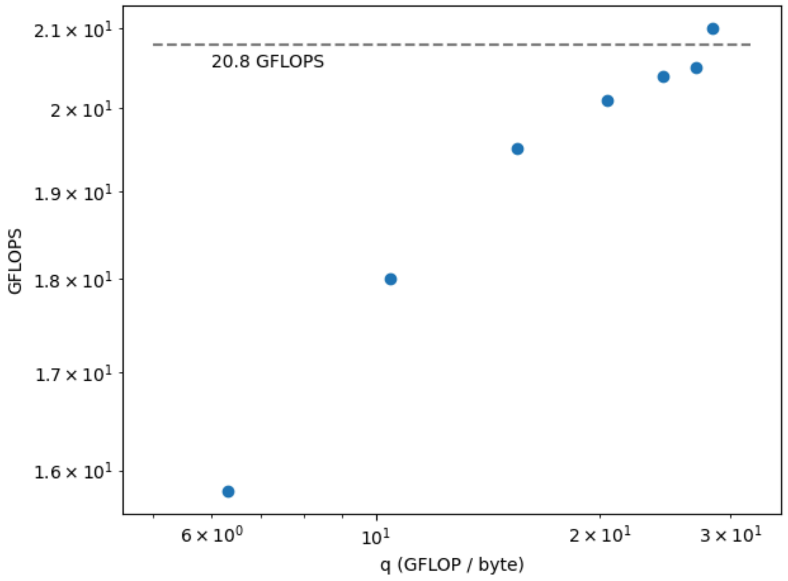
Function name	% time, N=32	% time, N=2048
bl_dgemm_ukr_sve_8x4	73.96	88.81
bl_dgemm	20.91	1.09

Note, the packing routine is inlined in `bl_dgemm`. The results show, when N is small, the packing overhead is much more noticeable (>20%). When N is large, the packing overhead becomes negligible (~1%) and our implementation spends most of the time in the microkernel. This observation supports the hypothesis that packing (and thus data movement) becomes a bottleneck for multiplying small matrices. Also note, since the microkernel needs to load and store C elements, `bl_dgemm_ukr_sve_8x4` also involves data movements, so the % time ratio between the two functions is not an appropriate approximation of q (flops / bytes).

Next, to quantitatively analyze the relation between varying arithmetic intensity (as a result of varying matrix size N) and GFLOPS, we perform a roofline analysis. Assuming the only data movement we consider is 1) packing panels of matrix A and B, and 2) loading and storing blocks of matrix C in the microkernel, based on the analysis of Q2.a, we can derive the arithmetic intensity as a function of matrix size N, and blocking parameters Mc and Kc.

$$q = \frac{2N^3}{N^2 + \frac{N^3}{Mc} + \frac{2N^3}{Kc}}$$

Using the above formula to estimate the arithmetic intensity of our 8x4 implementation for different matrix sizes (N) produces the following roofline plot. Note in practice we multiply the denominator by `sizeof(double)=8` so the unit of q is GFLOP / byte.



We estimate the peak single core double precision FLOPS of the Graviton 3 processor based on the measurements of Graviton 3: First Impressions. Assuming Graviton 3 has four 128-bit FP pipes (and thus can do 1 double precision fused multiply-add per cycle) [3], the peak FLOPS is given by 2.6 GHz * 4 op/s * 2 (multiply-add) = 20.8 GFLOPS. This number is shown in the figure as the gray dashed line. One thing to note is in our experiment OpenBLAS achieves peak GFLOPS above 20.8. We believe this could be a result of OpenBLAS using a kernel with time complexity below O(N^3), and thus the reported GFLOPS is above the actual number of operations.

Theoretically, the linear portion of the roofline plot reflects the relation $FLOPS = BW \times q$ (and as a result in a log-log plot the y-intercept should be log(BW)), where BW is the main memory bandwidth [5]. However, fitting a curve using our data yields the relation $FLOPS = 11.85\, q^{0.17}$. Additionally, the `mbw` utility indicates the bandwidth of the main memory is around 24 GB/s, which also doesn't match our fitted curve. We hypothesize this mismatch is a result of our naive method of estimating q. First, our calculation of data movements between slow and fast memory may not be accurate due to the complex caching and prefetching behavior of modern processors. Second, naively calculating an “average” q ignores the fact that our GEMM implementation is a mix of two very different workloads: the matrix packing phase has a very low q (i.e., near zero), while the microkernel has a very high q outside of the sloped region of the roofline plot (as Kc >> Mr, Nr). As a result, the formula we used to calculate q may not reflect the effective q in practice.

2. Parametric search

Next, we perform parametric search to tune the GEMM parameters using the 8x4 microkernel. We fix Mr=8, Nr=4 to fit the geometry of the kernel, and try Kc, Nc in (256, 512, 1024, 2048); nc in (64, 128, 256, 512). Additionally, considering the performance impact of matrix packing especially for small matrices, we tried manually unrolling the packing routing (commit 000723d; the "unroll" column in the table below). The best configuration for each matrix size is shown in the table below. Note the effective values of the parameters are limited by the size of the matrices (e.g., for N=32 and Kc=256, the effective Kc is 32). We mark values larger than the matrix size with a *.

N	Kc	Mc	Nc	Manual unroll	Peak GF
32	256 *	256 *	128 *	TRUE	16.2
64	1024 *	1024 *	128 *	TRUE	18.7
128	256 *	2048 *	64	FALSE	19.8
256	256	1024 *	64	FALSE	20.2
511	512 *	512 *	512 *	FALSE	20.9
512	256	2048 *	512	FALSE	21
513	2048 *	1024 *	512	FALSE	20.6
1023	512	512	128	FALSE	21
1024	256	1024	512	FALSE	20.9
1025	256	2048 *	512	FALSE	20.6
2047	512	2048 *	512	FALSE	21.2
2048	256	2048	512	FALSE	20.9

Best configurations for each matrix size N.

AWS Graviton3 has 64 kiB L1 data cache (per core), 1 MiB L2 cache (per core), and 32 MiB L3 cache (shared) [4]. In theory, the best performance is obtained when 1) Mr * Nr fits in registers, 2) Mr * Kc fits in L1 cache, 3) Kc * Nc fits in L2 cache, and 4) Mc * Kc fits in the L3 cache. Since Mr=8 and Nr=4, point 1 is always true in our case. For point 2, Kc=256 and Kc=512 corresponds to Mr * Kc * `sizeof(double)` = 16 kiB and 32 kiB respectively, which are just under the size of the L1 data cache. For point 3, our most popular combination, 256 * 512 * `sizeof(double)`, is exactly 1 MiB, the size of the L2 cache. Lastly, for point 4, the best configurations for the large matrices generally use between 2 - 4 MiB. Considering L3 cache is shared between 64 cores, 2 - 4 MiB seems reasonable. In conclusion, our parametric search yields configurations that agree with the cache hierarchy of the underlying hardware.

We also observe, for smaller matrices (N less than or equal to 64), it is desirable to use the manually unrolled matrix packing routine. For larger matrices, manual unrolling is not helpful. We hypothesize that this is because loop unrolling increases the size of the working set and might cause register spilling. Additionally, it increases code size and could put more pressure on the L1 instruction cache.

Useless stated otherwise, we use Kc=256, Mc=2048, Nc=512 in the other sections.

3. Why is 8*4 broadcasting kernel the best

To investigate why the 8x4 broadcast kernel has the best performance, we measure the number of L1/L2 cache reads and misses using `perf` (we weren't able to get accurate L3 measurements as `perf` always returns 0), and report the miss rate in the table below.

Kernel	L1 cache miss ‰	L2 cache miss ‰
4x4 broadcast	4.93	4.57
4x4 butterfly	10.90	4.44
4x4 svmla_lane	13.28	6.47
4x8 broadcast	9.38	7.64
8x4 broadcast	3.90	6.78
4x12 broadcast	8.86	7.33

The best performing 8x4 broadcasting kernel has the lowest L1 cache miss rate among broadcasting kernels of all sizes. Interestingly, the 8x4 and 4x8 variants have drastically different cache behavior. We speculate this is because 1) rows and columns are not symmetric in our implementation due to the row major memory layout (Mr and Nr affect how A and B are packed), and 2) characterics of the underlying hardware, for example, loading into a general purpose register might be different from loading into a scalable vector register (in terms of latency and the number of concurrent loads).

Additionally, we note the 4x4 broadcasting kernel works better than the butterfly and `svmla_lane` kernels. While the broadcasting kernel needs to load elements from matrix A one at a time, most of these loads likely result in L1 cache hits and thus a very small latency penalty. Furthermore, in the case of the broadcasting kernels, the SMID unit can start fused multiply-addition as soon as one element of A and a vector of B is loaded, so it is possible to hide latency through out-of-order execution. The butterfly kernel and the `svmla_lane` kernel, however, need to wait for the entire vector of A to be loaded due to data dependency on the values A. Thus in the case where only elements of A are split between 2 cache lines and only some are cached, the broadcasting kernel might better hide the cache miss latency.

4. **Why we organized our skeleton code in a different way from the BLISlab tutorial**

   The main reason is that the BLISlab tutorial uses column major matrices, while we use row major matrices (i.e., rows are continuous in memory). Our code organization has better spatial locality properties for row major matrices compared to BLISlab's setup intended for column major matrices. Specifically, when splitting A into Mc x N strips, and B into Kc x N panels, these strips and panels are aligned with the rows (since Mc, Kc << N), which reduces the rate of cache misses.

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

In our experiments we observed that different matrix sizes require different blocking parameters. Additionally, small matrices benefit from manually unrolled packing routines, while large matrices do not. Thus, one interesting idea to explore is to select the best configuration for each call to the GEMM routine based on the size of the matrix. This could be implemented in a just-in-time search fashion, or in the compilation phase for compile-once-run-many type workload, similar to the hardware dependent operator kernel search in JAX or TVM.

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

We explored manually unrolling the packing routine (see the "parametric search" subsection in Q2.d). We find it helpful for small matrices, but hurts performance when the matrices are large. Please see Q2.d for a more detailed discussion.

Q3. References - 2 pts

[1] SVE and Neon coding compared, https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Learn%20the%20Architecture/102131_0100_01_SVE_and_Neon_coding_compared.pdf?revision=feaaf72e-a941-461c-bd92-0d960d0f8615

[2] A64 SIMD Instruction List: SVE Instructions, https://dougallj.github.io/asil/index.html

[3] Graviton 3: First Impressions, https://chipsandcheese.com/2022/05/29/graviton-3-first-impressions/

[4] WikiChip - AWS Graviton3, https://en.wikichip.org/wiki/annapurna_labs/graviton/graviton3

[5] Understanding Roofline Charts, https://www.telesens.co/2018/07/26/understanding-roofline-charts/