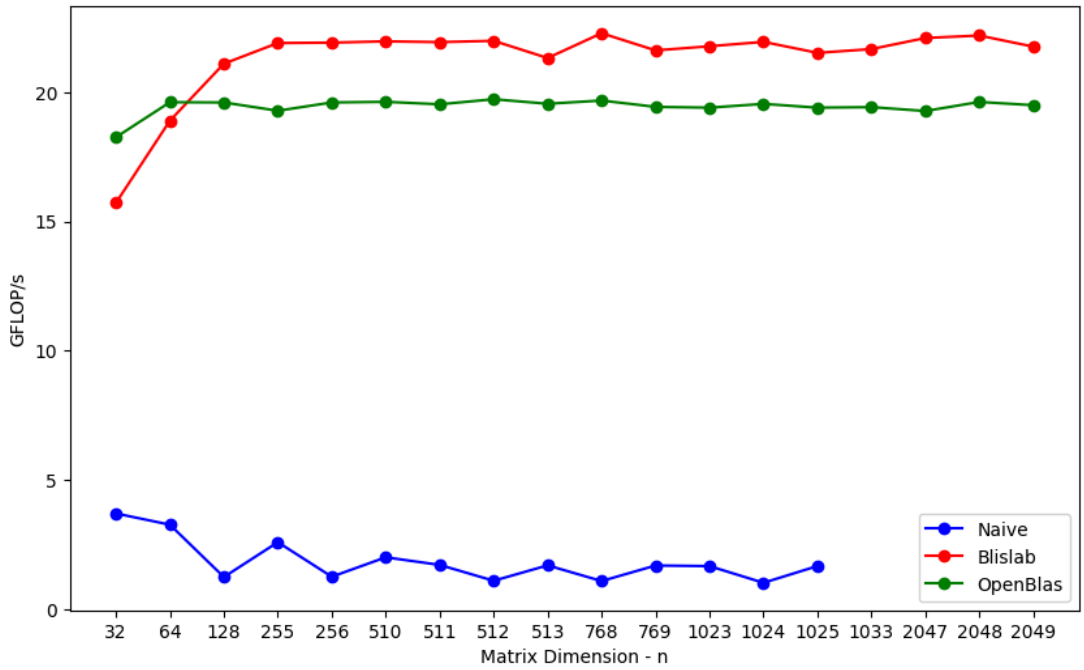
# Q1. Results - 15 pts

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 the 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 | 15.8 |
| 64 | 18.9 |
| 128 | 21.1 |
| 255 | 21.9 |
| 256 | 21.9 |
| 510 | 22 |
| 512 | 22.1 |
| 513 | 21.3 |
| 768 | 22.3 |
| 769 | 21.6 |
| 1023 | 21.8 |
| 1024 | 21.9 |
| 1025 | 21.5 |
| 1033 | 21.6 |
| 2047 | 22 |
| 2048 | 22.2 |
| 2049 | 21.8 |

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

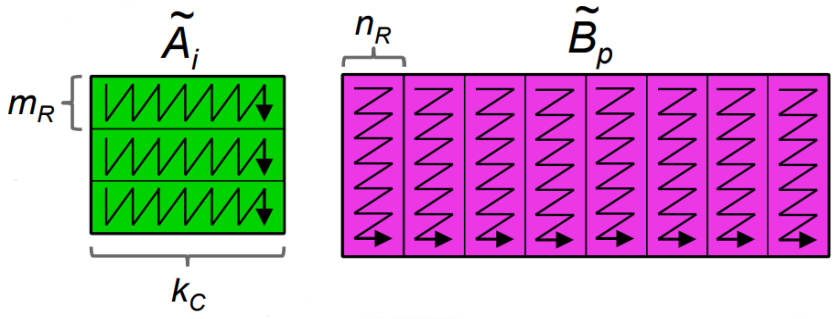
Clearly Describe:

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.

We are implementing Matrix multiplication C=A*B+C where A is an m x k matrix, B is a k x n matrix, and C is an m x n size matrix. In this code, we implemented outer product matrix multiplication. To optimize the matrix multiplication operation, we used Goto's algorithm to partition A, B, and C matrices into smaller blocks such that these smaller blocks fit in L1, L2, and L3 caches. This ensures data is accessed more efficiently from the faster caches than the slower memory.

```
Loop 5: for ic = 0:m; ic += Mc
          partition C into panels Cj of Mc x n
          partition A into blocks of Mc x k
Loop 4:   for pc = 0:k; pc += Kc
            Further partition A (Mc x k) into panels of Mc x Kc (Ap)
            pack Ap into subpanels Mr x Kc
            partition B into blocks of Kc x n (Bp)
Loop 3:     for jc = 0:n; jc += Nc
              partition Cj into panels of Mc x Nc
              partition Bp into panels of Kc x Nc (Bj)
              pack Bj into subpanels Kc x Nr
                // macrokernel
Loop 2:         for ir = 0:Mc; ic += Mr
Loop 1:           for jr =0:Nc; jr += Nr
Loop 0:             for kr=0:Kc; kr++
                      // microkernel
                      C_{Mr x Nr} += A_{Mr x Kc} x B_{Kc x Nr}
```

Loop5 partitions matrix A and matrix C along the row, into matrices of size Mc x K and Mc x N, respectively. Loop4 further partitions matrix A along the column into panels of size Mc x Kc, and matrix B along the row into a smaller matrix of size Kc x N. Loop 3 breaks matrix B along the column into panels of size Kc x Nc and matrix C along the column into panels of size Mc x Nc.




Within Loop4, the algorithm packs the Mc x Kc sized panels of A into packA Matrix of size Mr x Kc, in column-major ordering to utilize spacial locality for outer product multiplication. Within Loop3, this Kc x Nc panel of B is packed into the pack matrix of size Kc x Nr in row-major order, that is, elements along a row are stored sequentially.

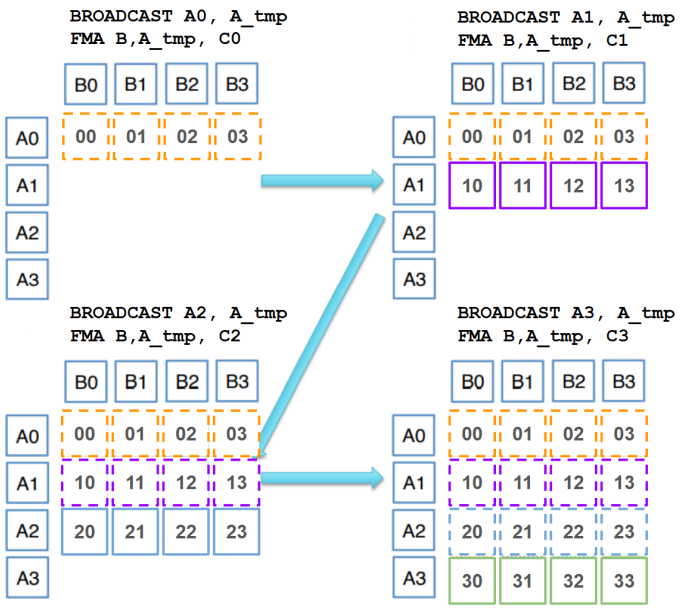
Loop1 and Loop2 - **MacroKernel** - iterates over each of the Mc x Nc panels of C, Mc x Kc panels of A, and Kc x Nc panels of B in blocks of Mr x Kc size of A and Kc x Nr size of B and Mr x Nr size of C. MacroKernel multiplies the sub-panel of A with the sub-panel of B to compute the outer product and store the result in the sub-panel of C. We want panel A of size Mc x Kc to fit in L3 and panel B of size Kc x Nc in the L2 cache due to more frequent loading calls of A compared to B.

**Microkernel:**

Finally, the innermost microkernel loop actually does the matrix multiplication of (Mr x Kc) packA with (Kc x Nr) packB to get the (Mr x Nr) C matrix by iterating over rows and columns of these sub-panels.

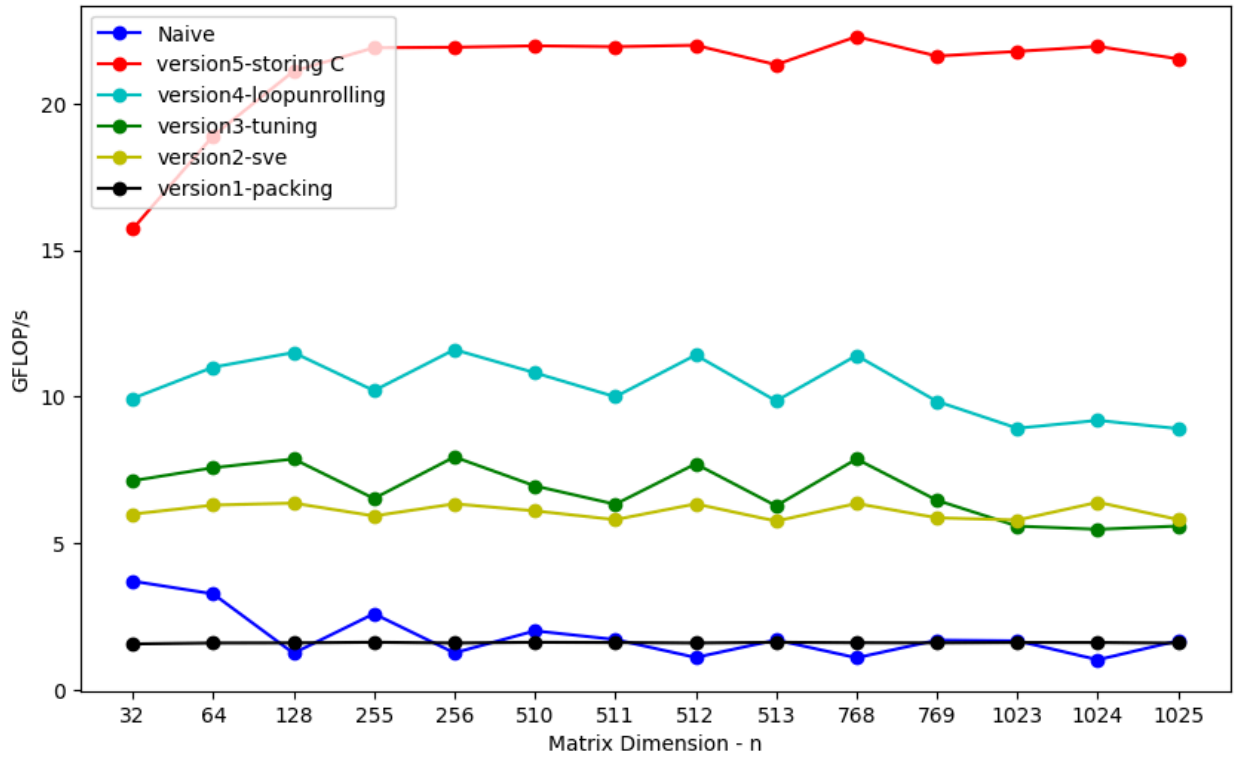
Within the Microkernel, we have implemented SVE instructions for SIMD. We have selected Mr, Nr such that C fits into the register. Subpanel A of size Mr x Kc and subpanel B of size Kc x Nr, both should fit in the L1 cache during microkernel computation.

We used broadcasting on A as described below(this is an example of a 4*4*4 kernel):




Each element of A_i(ith row) is broadcasted, and matrix B is vectorized along the registers. The broadcasted A_i values are then multiplied with the vectorized B, producing the corresponding row of matrix C_i. This is iterated over all rows of A.
Fringe cases arise when Mr doesn't divide Mc, Nr doesn't divide Nc. These are handled by allocating extra buffer spaces for packA, and packB at the start. When Mr doesn't divide Mc or Nr doesn't divide Nc, packA is allocated with a row size of (Mr/Mc+1)×Mc and packB with a column size of (Nr/Nc+1)×Nc. These extra buffers of size act as zero padding and help handle fringe cases effectively. We are also adding extra if-else conditions in the microkernel to handle fringe cases while loop unrolling(discussed in 2b).

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.




**Version 1 with packing: version1**

Packing A in column-major and B in row-major ordering initially slightly decreased the GFlops compared to Naive. This might be because of the overhead caused by packing with insignificant performance gain.

Peak GFLOP/sec: **~1.6**

**Version 2 with Sve: version2**

Implementation of SIMD increased the performance by a factor of 4. This ARM architecture in our EC2 has registers of 256 VLEN, which can hold 4 double-precision FPs. This allows for 4 FMLA operations simultaneously.

Peak GFLOP/sec: **~6.3**

**Version 3 with hyperparameter tuning: version3**

Initial given values: Mr = 4, Nr = 4, Mc=64, Kc=64, Nc=64

**Tuned values: Mr=16, Nr=4, Mc=2048, Kc=409, Nc=320 (Details in 2d)**

Performance slightly increased by 1-2 GFLOP/sec. Peak GFLOP/sec: **~7.8** GFlop/sec

**Version 4 with Loop Unrolling: version4**

We unrolled the loop i = 0 to  Mr-1(0 to 15) loop into 16 individual code blocks. This has increased the performance from 7 GFLOPS to 9 GFLOPS.

**Fringe case**: I have added if-else conditions without directly padding the C matrix with zero, since C is allocated a fixed memory of N x N. Sample case for i=15:

```
c15x = (m > 15) ? svld1_f64(npred, C+15*ldc) : svdup_f64(0.0);
aval = *(A + k*m+15);
ax =svdup_f64(aval);
bx = svld1_f64(svptrue_b64(), B + kk*n);
c15x =svmla_f64_m(npred, c15x, bx, ax);
if (m > 15) svst1_f64(npred, C+15*ldc, c15x);
```

Mr=16 and When N%16=m<15, these additional computations on the zero-initialized registers(c15 in above case) effectively act like **implicit zero padding** for the elements beyond the mth row in the microkernel. However, since these values are never stored back to `C` (due to the `if` conditions on `svst1_f64` calls), the computation on these registers is redundant and only serves to increase the compute workload without affecting the output.

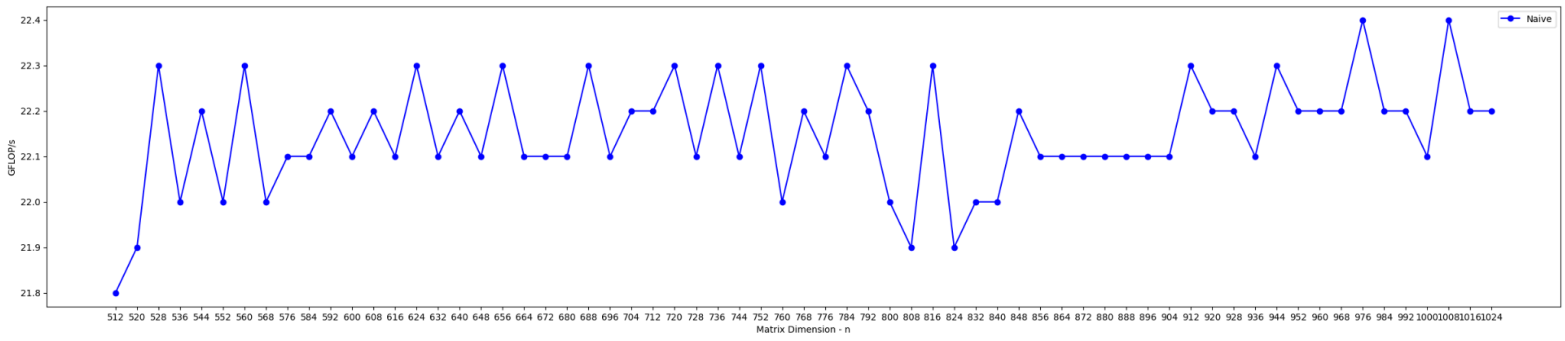
**Version 5 by storing C values after microkernel computation:** [version5]

Here we observe a significant jump in performance from 9 GFLOP/sec to 21 GFLOP/sec. In version 4, we stored matrix C from register c_ix in each iteration. Instead, if we maintained registers for different rows of C and stored these registers at the end after microkernel matrix multiplication computation, we stored them into C only Mr times. (Storing into c_ix registers is considered a negligible amount of time)

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.

From (1b), we can observe that performance drops when N is not a multiple of 4 and increases when N is a power of 4.

We can see from the above graph that when N is multiple of 8 but not 16, the performance is




still dropping. This indicates that whenever Mr doesn't divide N, the performance has been dropping. Fringe cases arise when Mr doesn't divide N and these fringe cases account for extra computations in the microkernel loop. When N%16=m<15, there are additional computations on the zero-initialized registers. However, since these values are never stored back to `C` (due to the `if` conditions on `svst1_f64` call), the computation on registers is redundant and only serves to increase the compute workload without affecting the output. This is the reason for the performance drop.

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.

## Cache Behaviour:

```
Description:    Naive, three-loop dgemm.

Size: 1024      Gflop/s: 0.944
GeoMean  = 0.94

 Performance counter stats for 'system wide':

          7160.99 msec task-clock                #    1.000 CPUs utilized
       17145945757      cycles                    #    2.394 GHz                      (18.74%)
        8278340546      instructions              #    0.48  insn per cycle             (18.74%)
         268538         branch-misses                                                 (18.76%)
        3221130006      cache-misses              #   69.99% of all cache refs        (12.51%)
        4602046493      cache-references          #  642.655 M/sec                    (12.51%)
        4649311097      L1-dcache-loads           #  649.256 M/sec                    (12.51%)
        3259043967      L1-dcache-load-misses     #   70.10% of all L1-dcache accesses  (12.51%)
        1632383357      L1-icache-loads           #  227.955 M/sec                    (12.51%)
           3709986      L1-icache-load-misses     #    0.23% of all L1-icache accesses  (12.51%)
        4648896300      l1d_cache                 #  649.198 M/sec                    (12.51%)
        3260918133      l1d_cache_lmiss_rd        #  455.373 M/sec                    (12.49%)
        1631342745      l1i_cache                 #  227.810 M/sec                    (12.49%)
           4049012      l1i_cache_lmiss           #  565.427 K/sec                    (12.49%)
        7251098065      l2d_cache                 #    1.013 G/sec                    (12.49%)
         358261650      l2d_cache_lmiss_rd        #   50.030 M/sec                    (12.49%)
        4521753613      mem_access                #  631.443 M/sec                    (12.49%)

       7.159590483 seconds time elapsed
```

**Naive**

```
Description:    Reference dgemm.

Size: 1024      Gflop/s: 18.6
GeoMean  = 18.56

 Performance counter stats for 'system wide':

           685.53 msec task-clock                #    1.001 CPUs utilized
        1653904024      cycles                    #    2.413 GHz                      (18.50%)
        4056123232      instructions              #    2.45  insn per cycle             (18.53%)
         253888         branch-misses                                                 (18.67%)
          10352007      cache-misses              #    0.87% of all cache refs         (12.55%)
        1193755937      cache-references          #    1.741 G/sec                    (12.54%)
        1179798781      L1-dcache-loads           #    1.721 G/sec                    (12.55%)
          10258666      L1-dcache-load-misses     #    0.87% of all L1-dcache accesses  (12.55%)
         684917600      L1-icache-loads           #  999.106 M/sec                    (12.55%)
            548146      L1-icache-load-misses     #    0.08% of all L1-icache accesses  (12.57%)
        1188703176      l1d_cache                 #    1.734 G/sec                    (12.55%)
           9075962      l1d_cache_lmiss_rd        #   13.239 M/sec                    (12.53%)
         685315322      l1i_cache                 #  999.686 M/sec                    (12.55%)
            613509      l1i_cache_lmiss           #  894.940 K/sec                    (12.55%)
         369041200      l2d_cache                 #  538.329 M/sec                    (12.55%)
           1721073      l2d_cache_lmiss_rd        #    2.511 M/sec                    (12.51%)
        1176582212      mem_access                #    1.716 G/sec                    (12.37%)

       0.684644289 seconds time elapsed
```

**Openblas**

```
Description:    my blislab

Size: 1024      Gflop/s: 16.8
GeoMean  = 16.85

 Performance counter stats for 'system wide':

           714.85 msec task-clock                #    1.002 CPUs utilized
        1719585950      cycles                    #    2.406 GHz                      (18.56%)
        5245145819      instructions              #    3.05  insn per cycle             (18.68%)
         448448         branch-misses                                                 (18.82%)
          27625782      cache-misses              #    1.62% of all cache refs         (12.59%)
        1704104685      cache-references          #    2.384 G/sec                    (12.59%)
        1698108881      L1-dcache-loads           #    2.375 G/sec                    (12.59%)
          27131299      L1-dcache-load-misses     #    1.60% of all L1-dcache accesses  (12.59%)
         731182576      L1-icache-loads           #    1.023 G/sec                    (12.59%)
            672357      L1-icache-load-misses     #    0.09% of all L1-icache accesses  (12.59%)
        1679026087      l1d_cache                 #    2.349 G/sec                    (12.60%)
          26436996      l1d_cache_lmiss_rd        #   36.983 M/sec                    (12.59%)
         734289546      l1i_cache                 #    1.027 G/sec                    (12.51%)
            870067      l1i_cache_lmiss           #    1.217 M/sec                    (12.38%)
         213270145      l2d_cache                 #  298.343 M/sec                    (12.32%)
           1972640      l2d_cache_lmiss_rd        #    2.760 M/sec                    (12.32%)
        1703433081      mem_access                #    2.383 G/sec                    (12.32%)

       0.713392959 seconds time elapsed
```

**Blislab - Untuned**

```
Description:    my blislab

Size: 1024      Gflop/s: 20.6
GeoMean  = 20.57

 Performance counter stats for 'system wide':

           652.97 msec task-clock                #    1.001 CPUs utilized
        1560653240      cycles                    #    2.390 GHz                      (18.44%)
        3791165380      instructions              #    2.43  insn per cycle             (18.56%)
         333754         branch-misses                                                 (18.70%)
          11409047      cache-misses              #    0.79% of all cache refs         (12.56%)
        1438360638      cache-references          #    2.203 G/sec                    (12.56%)
        1430117872      L1-dcache-loads           #    2.190 G/sec                    (12.56%)
          12881605      L1-dcache-load-misses     #    0.90% of all L1-dcache accesses  (12.56%)
         572030644      L1-icache-loads           #  876.044 M/sec                    (12.57%)
            732329      L1-icache-load-misses     #    0.13% of all L1-icache accesses  (12.56%)
        1433957512      l1d_cache                 #    2.196 G/sec                    (12.55%)
           9600981      l1d_cache_lmiss_rd        #   14.704 M/sec                    (12.56%)
         562603727      l1i_cache                 #  861.607 M/sec                    (12.56%)
            498067      l1i_cache_lmiss           #  762.771 K/sec                    (12.56%)
         263137922      l2d_cache                 #  402.986 M/sec                    (12.56%)
           1614908      l2d_cache_lmiss_rd        #    2.473 M/sec                    (12.42%)
        1418645221      mem_access                #    2.173 G/sec                    (12.27%)

       0.651993165 seconds time elapsed
```

**Blislab - Tuned**

Blislab - tuned: with fine-tuned parameters, Mc = 2048, Kc=409, Nc=320, Mr=16, Kr-4

Blislab - untuned with given parameter Mc=Nc=Kc=64 and Mr=Nr=4

| Version | Avg GFLOP/s | Cache Miss Rate(%) | L1 Cache Miss Rate(%) | L1 cache Read Misses per sec | L2 cache Read Misses per sec | Memory accesses per sec | Instructions per Cycle |
|---|---|---|---|---|---|---|---|
| **Naive** | 0.94 | 69.99 | 70.10 | 455.373 M/sec | 50.030 M/sec | 631.443 M/sec | 0.48 |
| **Blas** | 18.6 | 0.87 | 0.87 | 13.239 M/Sec | 2.511 M/sec | 1.716 G/sec | 2.45 |
| **Blislab (Untuned)** | 16.8 | 1.62 | 1.62 | 36.983 M/sec | 2.76 M/sec | 2.381 G/sec | 3.05 |
| **Blislab (Tuned)** | 20.57 | 0.79 | 0.79 | 14.704 M/sec | 2.473 M/sec | 2.173 G/sec | 2.43 |

The above table summarizes a few important data points observed when running the perf tool on different versions of the code during its development, as shown in the images above.
**Effect of Packing:** The Cache miss rate in the naive implementation is **69.99%,** and it can be observed that even the L1 Cache miss rate is **70%**. After properly packing the Matrix B into sub-panels, we invoked spacial locality, significantly reducing the Cache Miss Rate. This reduced Cache Miss Rate also shows a significant reduction in the number of Cache Read misses per second in L1. The same can be observed in L2 Cache, where the number of Cache misses per second is reduced from **50.030 M/sec** to **2.76M/sec**.
**Effect of Parameter Tuning:** After packing, we tuned the size of the sub-panels to properly align the data and fit the maximum number of doubles in the cache (increasing cache utilization). This resulted in a further reduction of the Cache Miss rate, from **1.62** to **0.79**. This Cache Reduction rate significantly reduced the number of Cache Read misses at L1 level from **36.983 M/sec** to **14.704 M/sec**.
**Effect of SIMD:** As observed in the last column of the table, the number of instructions executed per cycle shot up from **0.48** to **3.05** with the introduction of SIMD.

## Parameter Search:

The VLEN of the ARM SVE supported by the Graviton 3 Processor is of 256 Bytes, which equals 4 Doubles in C. Hence: **Nr = 4.** This eliminates the innermost loop of j =0 to Nr as it calculates four FMLA operations simultaneously.

The ARM SVE Processor has 32 SIMD Registers. We want the matrix C of size **Mr** X **Nr** to fit into these Registers:

**Mr = 16**

We used the following cache sizes of the CPU to calculate the values for the remaining parameters:

```
Caches (sum of all):
  L1d:                    64 KiB (1 instance)
  L1i:                    64 KiB (1 instance)
  L2:                     1 MiB (1 instance)
  L3:                     32 MiB (1 instance)
```

For the sub-panels of A, size Mr X Kc, and B, of size Kc X Nr, to fit into the L1 cache, we need:

- ➔ Mr*Kc + Kc*Nr = 8192 (64KiB translates into 8192 doubles)
- ➔ Kc = 409.6 (rounding off to 409)

**Kc = 409**

For the panel of B, size Kc X Nc, to fit into the L2 cache, we need:

- ➔ Kc*Nc = 131072 (1 MiB translates to 131072 doubles)

**Nc = 320**

For the panel of A, size Mc X Kc, to fit into the L3 cache, we need:

- ➔ Mc*Kc = 4194304 (32 MiB translates to 4194304 doubles)
- ➔ Mc = 10240 (upper bound on Mc is 2048 -> for matrix dimension n:512 to 2048)

**Mc = 2048**

**Different from blas: Code implemented in blas**

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

1. We tried implementing the Butterfly method which enables efficient data reuse by minimizing the need to reload vectors of A and B matrices and comparing its performance with the broadcast method. We would also like to test various unshuffling methods for matrix C to observe their impact on performance.
2. We would also code our SVE microkernel in a parameterized manner to support multiple Mr X Nr configurations instead of running for only a single fixed configuration, in our case it is 16 X 4. We would also like to explore additional SVE commands to optimize the existing code.
3. We want to explore different SIMD versions like NEON and SME and compare it to our SVE to analyze the difference in performance in using different architectures.
4. In this implementation, we have used the memory constraints to get the optimal parameters(question 2d). We can optimize the total memory access time instead, resulting in a more optimal performance for higher matrix size.

# Q3. References - 2 pts


1) https://arxiv.org/pdf/1609.00076
2) https://github.com/flame/blislab - Jianyu Huang, Robert A. van de Guijn, BLISlab: A Sandbox for Optimizing GEMM, August 31, 2016
3) Kazushige Goto and Robert A. van de Geijn. Anatomy of a high-performance matrix multiplication. ACM Trans. Math. Soft., 34(3):12, May 2008. Article 12, 25 pages