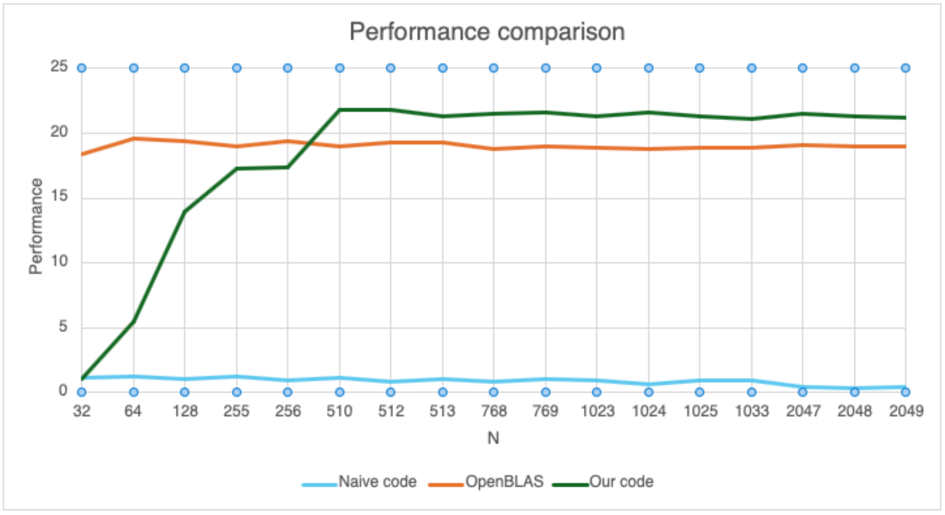
Names: Felicia Tao, Sean Ko GitHub: here (https://github.com/cse260-fa25/cse260-fa25-hw1-f2tao-s7ko)

Q1. Results - 15 pts

N	Naive code	OpenBLAS	Your code
32	1.1273	18.4333	1.0169
64	1.2607	19.6317	5.4575
128	1.0501	19.3544	13.9487
255	1.2062	19.0461	17.2601
256	0.9927	19.3859	17.4159
510	1.1353	19.0144	21.8404
512	0.8224	19.3235	21.8401
513	1.0417	19.2848	21.3285
768	0.8333	18.8171	21.4718
769	1.0653	19.0239	21.6106
1023	0.9851	18.9383	21.2751
1024	0.6942	18.7637	21.6620
1025	0.9710	18.9238	21.3569
1033	0.9953	18.9118	21.1060
2047	0.4732	19.0553	21.5346
2048	0.3922	19.0444	21.3345
2049	0.4638	18.9689	21.2543

Q1.b.



Q2. Analysis - 33 pts

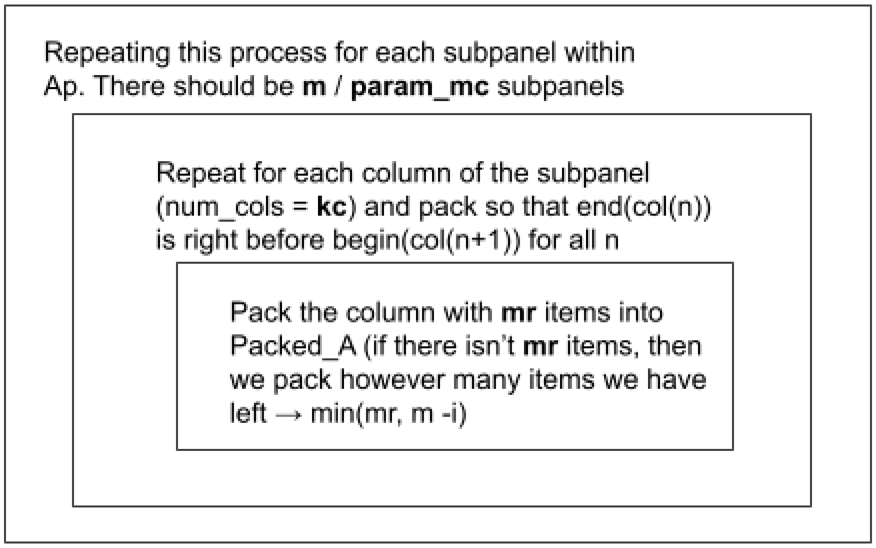
Q2.a - 0-2pts - team evaluation filled out for each team member (teameval)

Completed

Q2.b How does the program work

1. Packing

Packing A: Our approach



Repeating this process for each subpanel within Ap. There should be **m / param_mc** subpanels
Repeat for each column of the subpanel (num_cols = **kc**) and pack so that end(col(n)) is right before begin(col(n+1)) for all n
Pack the column with **mr** items into Packed_A (if there isn't **mr** items, then we pack however many items we have left → min(mr, m -i)

Above is a diagram on how we approached the packing logic for A matrix.

```
for (int i = 0; i < m; i += param_mr) { // iterating through the Mr subpanels of Ap
        int true_row = std::min(param_mr, m - i);
     for (int j = 0; j < k; j++) { // iterating through each column in Kc of Ap subpanel
        for (int l = 0; l < true_row; l++) { // iterating through each row in the column of subpanel
              *packed_A++ = A[(i + l) * lda + j];}
        }   }
```

In our code, m = **mc** and k = **kc**. A similar packing logic applies to packed B but with **kc, nc, param_nr**. Another difference is we pack columns for A matrix but for B matrix we pack by row.

2. Memory alignment

We wanted to align the memory of packed_A and packed_ B because that can efficiently use cache line/SIMD loads by avoiding instances of misaligned vector loads. So when we packed we used **posix_memalign** to align our pointers to start at the 64 byte aligned boundary. Below is how we memory aligned packed_A:

```
size_t packed_A_size = (size_t)param_mc * (size_t)param_kc;
```

```
double* packed_A = nullptr;
// call posix_memalign((void**)&packed_A, 64, packed_A_size * sizeof(double))
```

3. SIMD

We did SIMD Broadcasting by writing cases when 0 <= mr <= 12. We set in our parameters for mr to be 12 and nr to be 4, however in cases when subpanel Ap is not divisible by 12, we need to handle that with every possible case of Mr. Here is the case when mr == 2:

```
case 2: {
            //initialize result vectors
           svfloat64_t c0x = svld1_f64(npred, &c[0 * ldc + col_offset]);
           svfloat64_t c1x = svld1_f64(npred, &c[1 * ldc + col_offset]);
           for (int j = 0; j < kc; j++) {
              const double* b_row = &b[j * nr]; // pointer to current row in B
              svfloat64_t b_vec = svld1_f64(npred, b_row + col_offset); // load B row vector
              const double* a_col = &a[j * mr]; // pointer to current column in A
              svfloat64_t a0 = svdup_f64(a_col[0]); // broadcast A values
              svfloat64_t a1 = svdup_f64(a_col[1]);
              c0x = svmla_f64_m(npred, c0x, b_vec, a0); // multiply-accumulate
              c1x = svmla_f64_m(npred, c1x, b_vec, a1); }
            svst1_f64(npred, &c[0 * ldc + col_offset], c0x);// store back to C
           svst1_f64(npred, &c[1 * ldc + col_offset], c1x);
```

The microkernel broadcasts each scalar in the column of A into a vector register and does a multiply-add with a loaded vector of B. Then, these results are added into the result vectors for C. The process is repeated for every kc, which we chose to be 512.

4. Loop Unrolling (Packing)
   The loop before iterates to true col by making 1 step increments.
   To parallelize this to increase performance, we packed B with 4 sized step increments instead

```
for (int l = 0; l + 3 < true_col; l+=4) {

           *packed_B++ = B[j * ldb + i + l + 0];

           *packed_B++ = B[j * ldb + i + l + 1];

           *packed_B++ = B[j * ldb + i + l + 2];

           *packed_B++ = B[j * ldb + i + l + 3];         }
```

However if true_col is not divisible by 4, that will lead to the edge of the subpanel to be not packed. So to account for this we created a 2nd loop that steps by 1 and packs the remainder.

```
for (int l = true_col - (true_col % 4); l < true_col; l++) {
          *packed_B++ = B[j * ldb + i + l];   }
```

We applied this same logic to A as well because the packing logic only differs from row/column major.

Q2.c It is required to use your github checkins to show your development process (e.g. commit logs). Refer to HW1 instructions - Part 3 Grading - analysis for more details.

- Packing

  We implemented packing first to improve performance. The starter code performance was around 6.2 and with packing, our performance increases marginally to around 6.8.

- Initial SIMD Broadcasting

  At first, we kept the matrix sizes within the microkernel small. We implemented a SIMD broadcasting up to 4 x kc x 4 matrix. The performance was around 18, a pretty decent performance with limited lines of code.

- Generalizing SIMD

  We knew that we weren't taking full advantage of the registers and wanted to increase **mr** to use more registers. Our first approach was to try to generalize the logic of a mr x kc x 4 microkernel.

  We thought, instead of manually creating all the c0x, c1x,..., a0x, a1x vector registers, we could use for- loops to initialize the vector registers for C and A and hold them in an array. This would make it so that we could use the same code for any arbitrary mr. However, we ran into problems with storing vector registers in the array, as it seems the vector registers went out of scope once the for loop in which they were created was done. This made our program run into errors. After some research and testing different methods, we realized that it was kind of a dead end and instead decided to hard code logic for the different mr values.

```
        svfloat64_t c_vecs[param_mr];
    for (int j = 0; j < mr; j++) {
        svfloat64_t c_vec = svld1_f64(npred, &c[j * ldc + col_offset]);
        c_vecs[j] = &c_vec;
    }

    for (int j = 0; j < kc; j++) {
        const double* b_row = &b[j * nr]; // pointer to current row in B
        svfloat64_t b_vec = svld1_f64(npred, b_row + col_offset); // load B row vector

        const double* a_col = &a[j * mr]; // pointer to current column in A
        svfloat64_t* a_vals[param_mr];
        for (int k = 0; k < mr; k++) {
            svfloat64_t a_val = svdup_f64(a_col[k]); // broadcast A values
            a_vals[k] = &a_val;
            *c_vecs[k] = svmla_f64_m(npred, *c_vecs[k], b_vec, *a_vals[k]); //     multiply-accumulate
            }
    }
    for (int j = 0; j < mr; j++) {
        svst1_f64(npred, &c[j * ldc + col_offset], *c_vecs[j]); // store results back to C
    }
```

- Hard Coded Cases for SIMD

  We decided to hard code cases for values of mr up to 15. We decided on 15 because we start out with 32 registers. One register needs to be used for b_vec, which means 31 registers to be used between storing values for broadcast vectors ax and result vectors cx. Since every ax requires a cx, we get 31 // 2 = 15 vector registers for ax and cx each. We predicted that maximizing the usage of the vector registers would increase performance the most. We did indeed see improvements, getting an average of 20.1 and 20.4 with loop unrolling in packing.

- Parameter tuning nr

  We had kept nr to be 4 up until now because that is how many doubles a vector register holds with one load. However, we considered the possibility of increasing the number, thus doing more loops/computations in one call of the microkernel. We began designing this by calculating how many vector chunks we would need to split nr into to cover all columns by doing a ceiling of nr / VL. Then, in a for loop, we would loop over the chunks and have the mr cases inside. In addition, there would be a column offset variable to account for the column offset for each chunk and a predicate used to mask vector lanes when the chunk of nr is not a multiple of VL. However, when we tested numbers greater than 4, both multiples and not multiples, we found that performance decreased in both cases. Therefore, we did not explore further with changing nr and kept it at 4.

```
const int VL = svcntd(); // get number of doubles that can fit per vector
  const int num_vecs = (nr + VL - 1) / VL; // number of vectors to cover nr columns
  for (int i = 0; i < num_vecs; i++) {
     int col_offset = i * VL;
     svbool_t npred = svwhilelt_b64((uint64_t)col_offset, (uint64_t)nr);
```

- Parameter tuning mr

  We tried taking full advantage of scaling mr but however we found that maximizing the use of registers to store vectors of A and C was not as efficient as dialing mr back to 12. It was a difference between average perf of 20.5 and 21.5. We hypothesize that filling up all the registers with vectors coming from A, B, C doesn't give room to store intermediate or index variables such as **i** in our for loop in fast memory. Relieving some register pressure improved our performance.

- Parameter tuning kc

  We found that increasing kc from the default of 128 to 512 consistently increased performance by around 0.1. We think that this increase was the result of using each packed block more times, which helps to cover the overhead for packing and all the loop setup. However, going beyond 512 resulted in decreased performance, likely due to restraints of cache size.

- Tried Open MP

  This wasn't in any commits but we tried using pragma open mp parallel for to parallelize our packing . However, we realized that our EC2 instance did not support OpenMP

- More loop rolling for packed

  We tried to unroll the loop by step size 8 instead of 4 but saw a decrease in performance. This may be due to the diminishing return of having larger step sizes and more work done per loop. This diminishing return, we hypothesize, might come from the clean up **for** loop that at worst case needs to loop 7 times instead of 3 times before. It may also come from an increase in instruction cache misses.

Q2.d 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.

In your explanation, include information that supports your conclusion. For example analysis of **cache behavior**, parametric searches, etc.

N	IPC	Cache Miss Rate (%)	dTLB Miss Rate (%)
32	1.23	1.57	4.7
64	1.3	1.7	6.37
128	1.5	1.58	2.54
255	2.15	0.92	1.22
256	2.19	1.25	1.08
510	2.57	0.75	0.61
512	2.59	1.06	0.61
513	2.53	0.94	0.83
768	2.64	0.93	0.52
769	2.64	0.76	0.65
1023	2.61	0.75	0.58
1024	2.65	1.1	0.51
1025	2.64	1.16	0.61
1033	2.6	0.8	0.7
2047	2.66	1.03	0.55
2048	2.64	1.18	0.5
2049	2.64	1.22	0.6

Cache line size for L1, L2, LLC – 64 bytes which can hold 8 doubles.

We see a general trend of performance not scaling in a linear way when size **n** is not divisible by 8 due to an additional cache line that needs to be loaded (which comes at a cost of a cold miss as well) when we need to pack the panels of the matrices and also perform multiple-adds to subpanels. For example, when n = 1024, performance is 21.66. Because 1024 % 8 = 0, every load from memory to cache has a full cache line, all with values we need. However, when n = 1025, the computer is required to load an extra cache line from memory which is an additional miss penalty, this slows down our computation to a performance of 21.35.

Looking at our perf stats, our cache miss rate and dTLB miss rate varies quite a bit from n = 32 to n = 512 but volatility is much lower as n increases after that. Referring to our table, we see not only is there more cache misses when n is not divisible by 8 but an even stronger trend is that dTLB miss rate increases when n % 8 != 0 and especially n % 512 != 0. A page is 4096 bytes which means it can hold 512 doubles. So when we need to store an additional double, we need to access another page which may increase the chance of a page fault. That is why we see a huge increase of TLB miss rate between n = 512 and 513 , 1024/1025, etc. TLB miss penalties for a page fault are more expensive than cache misses and that furthers the explanation of the decrease in performance when n is not a multiple of 8. We want to point out this is a trend that doesn't hold all the time because we see inconsistent behavior at n = 768 and n = 769.

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

1. Butterfly method

   We did not test the performance of butterfly against SIMD broadcasting because we didn't have time to implement the logic to calculate a row of C.

2. Using one more register to hold B in microkernel

   We could test what performance would be if we double nr to 8 to handle a bigger subpanel of Bp every call of microkernel. So instead of having one register that handles a row of B, we would assign 2. We know that using **all** registers actually is not optimal however because we scaled mr down to 12, we have some breathing room to scale nr a little higher.

Q2.f Schedule your interview(s) - one per partner. Write down when they are or if you were not able to find a suitable time.

Felicia - During OH, probably 10/28 1:30 - 2:30

Sean - 10/30 class 12:30 - 1:00

Q2.g – None

Q3. References - 2 pts

Blislab Git Repo
HW1 Instruction
Posix_memalign doc https://man7.org/linux/man-pages/man3/posix_memalign.3.html
SIMD Instrinsics
https://developer.arm.com/architectures/instruction-sets/intrinsics/#f:@navigationhierarchiessimdisa=%5BSVE%5D

Q4. Extra Credit - 5 pts

Project Proposal

We think that it would be helpful to make some enhancements to the makefile to automate/simplify remote development. We're thinking of three main features:

1. make upload: upload the current homework directory to the remote aws instance
2. make remote <command>: run make targets on the remote host, like make remote, make remote clean, etc. There is also the possibility of adding some more make commands for commonly executed things, like make remote grade to run ./grade.sh and so on.
3. make command <command>: can run any arbitrary command

The commands should also automatically pipe any output back to the local machine.

All that needs to be customized is the ssh configuration login to remote and execute all the commands.

We think that this would be a great help because it eliminates the need to ssh into AWS every single time simple testing needs to be done. It also removes the difficulty of having to connect VS Code to AWS for easier development or having to configure GitHub on AWS to push and pull.