Report for Lab 2

Task 1

For timing GEMM kernel, we use a scoped profiler to calculate the elapsed time of executing dot function and accumulate the execution time. The total time spent in dot function is printed out when the program exits.

```
Run: time make run_pthreads

dot exec time: 63889ms
real    1m6.346s
user    1m6.294s
sys     0m0.032s
```

Percentage of dot execution: 63.889/66.346 = 96.2967%

Task 2

a) We have applied the loop permutation technique to achieve a more locality preserving access. To be more specific, we have reordered the loops in the following way.

```
for (int row = 0; row < m1_rows; ++row)
  for (int k = 0; k < m1_columns; ++k)
    for (int col = 0; col < m2_columns; ++col)
      output[row * m2_columns + col] += ...;
```

b) Before the transformation, the total running time is 67.508 550 s, whereby the running time of the `dot` function is 64.991 598 s (96.271 654 % of total running time). After the transformation, the total running time is 10.498 188 s, whereby the running of the `dot` function is 7.994 576 s (76.151 964 % of total running time). The amount of running time that is spent in the `dot` function has been reduced significant from 96.271 654 % to 76.151 964 % (difference 20.119 69 %).

Task 3

a) The cache line size of the machine is 64 bytes by "getconf -a | grep CACHE". Theoretically, we shall get the best performance if BLOCK_SIZE * sizeof(float) equals to a cache line size, i.e. const int block_size = 64 / sizeof(float);

b) Our implementation of loop tiling:

```
int N = m1_rows;
int M = m1_columns;
int K = m2_columns;
for (int i0 = 0; i0 < N; i0 += block_size) {
    int imax = i0 + block_size > N ? N : i0 + block_size;
    for (int j0 = 0; j0 < M; j0 += block_size) {
        int jmax = j0 + block_size > M ? M : j0 + block_size;
```

```
            for (int k0 = 0; k0 < K; k0 += block_size) {
                int kmax = k0 + block_size > K ? K : k0 + block_size;
                for (int j1 = j0; j1 < jmax; ++j1) {
                    int sj = K * j1;
                    for (int i1 = i0; i1 < imax; ++i1) {
                        int mi = M * i1;
                        int ki = K * i1;
                        for (int k1 = k0; k1 < kmax; ++k1) {
                            output[ki + k1] += m1[mi + j1] * m2[sj + k1];
    }}}}}}
```

c) We have observed that BLOCK_SIZE < 64 is not as good as BLOCK_SIZE $= 64$. However, if we use task2 result as baseline, loop tiling didn´t significantly improve the cache hit. This could be explained by that the float vector is not aligned with cache line size, which causes extra cache miss even with looping technique. Meanwhile, extra loop instructions also increase the total execution time. In summary, we got better cache hits, but the total execution time increased.

Below is the result of baseline and loop tiling with BLOCK_SIZE $= 64$:

```
Run: perf stat --repeat 10 -e LLC-load-misses ./nnetwork

 Performance counter stats for './nnetwork' (10 runs):
      Baseline:
           441,399      LLC-load-misses:u
          10.47281 +- 0.00513 seconds time elapsed  ( +-  0.05% )

      Loop tiling:
           415,132      LLC-load-misses:u
           16.6134 +- 0.0106 seconds time elapsed  ( +-  0.06% )
```

Task 4

a) The following snippet shows our implementation of the parallel version of the GEMM kernel.

```
...

const int chunk = m1_rows / num_partitions;
const int remainder = m1_rows % num_partitions;

pthread_t threads[num_partitions];

for (int i = 0; i < num_partitions; ++i)
{
  gemm_thread_args *args = new gemm_thread_args;
```

```
  args->m1 = &m1;
  args->m1_columns = m1_columns;
  args->m1_rows = m1_rows;
  args->m2 = &m2;
  args->m2_columns = m2_columns;
  args->output = &output;

  args->row_start = i * chunk + std::min(i, remainder);
  args->row_end = (i + 1) * chunk + std::min(i + 1, remainder);

  pthread_create(&threads[i], NULL, &dot_block, args);
}

for (int i = 0; i < num_partitions; ++i)
  pthread_join(threads[i], NULL);

...
```

b) The running times for different number of threads are given by the table below.

Number of threads	Total running time (s)
1	133.080 872
2	70.213 904
4	38.302 974
8	38.577 283
16	39.085 463
32	40.197 144
64	42.522 234

c) Since the used computer has 4 processor cores, the usage of more than 4 threads does not yield any improvement. Using more than 4 threads leads to an increasing total running time because the overhead for the thread management becomes larger and the maximum number of threads that can be executed in parallel stays constant.

Task 5

a) Besides the multi-threading implementation, we also did loop interchange to improve the performance.

When the program run with 1 processor, total execution time:

11.786s = 2.605(sequential) + 9.181(parallelizable)

If assume linear speedup, considering the machine has 4 cores, the theoretical maximum speedup can be achieved is: 11.786/(2.605 + 9.181/4) = 2.41

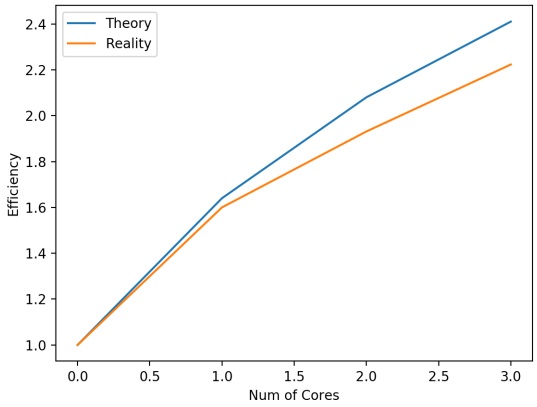
b) We used 1,2,3,4 threads to train the network respectively.

Sequential part:

2.605s (1 thread), 2.558s (2 threads), 2.605s (3 threads), 2.591s (4 threads)

Parallel part:

9.181s (1 thread), 4.777s (2 threads), 3.499s (3 threads), 2.711s (4 threads)



c) The efficiency that we achieved with 4 threads is : 11.786/(2.591 + 4*2.711) = 0.877