Report for Lab 4

Task 1 – OpenMP parallel for

```
#pragma omp parallel
{
    int thread_id = omp_get_thread_num();
    int tcount = omp_get_num_threads();

    std::cout << "Thread ID " << thread_id << std::endl;
    if (thread_id == 0) std::cout << "Total threads: " << tcount << std::endl;

    #pragma omp for
    for ( ... ) {
        ...
    }
}
```

1. The thread id inside the parallel region is printed out with the aid of the `omp_get_thread_num()` function.
2. The total number of threads inside the parallel region is printed out with the aid of the `omp_get_num_threads()` function. It is printed only if `thread_id == 0`. Therefore, it is output only once.

Task 2 – Parallel performance gains

1. We use `time` command to time the total execution time of the program and use `omp_get_wtime()` to time the parallel execution.

   ```
   double total_time_in_parallel = 0; // variable in global namespace
   double start = omp_get_wtime();
   ... parallel code ...
   double end = omp_get_wtime();
   total_time_in_parallel += end - start;
   ```

   Regardless of number of threads, the time spent on serial execution is nearly fixed: about 2.55s. The parallel execution time decreases when increasing number of threads, Table blow shows the results.

2. The table below shows the execution time and efficiency:

No of Threads:	1	2	4
Serial:	2.5528 s	2.5203 s	2.5802 s
Parallel:	64.6382 s	33.3797 s	17.8018 s
Efficiency:	1	0.968226	0.907748

Table 1: Efficiency for different number of threads

Task 3 – OpenMP scheduling policies

	(no param)	1	16	128
schedule(static, <param>)	4.6989 s	7.6719 s	4.7186 s	6.2623 s
schedule(dynamic, <param>)	10.9646 s	10.8904 s	4.8008 s	6.3657 s
schedule(guided, <param>)	4.8178 s	4.8473 s	4.7503 s	6.3849 s

Table 2: Total running times for all loop scheduling strategies.

1. We have included all three scheduling strategies, namely `static`, `dynamic`, and `guided`, in our experiments.
2. Table 2 shows the running times of all three strategies with either no parameter specified, a chunk size of 1, 16, or 128.
3. The optimal policy is `schedule(static)`. On the one hand, since all loop iterations of the GEMM loops take more or less the same time, there is no benefit of using `dynamic` scheduling because the extra time spend for scheduling does not improve the distribution of work significantly. On the other hand, `schedule(static)` is more cache-locality preserving than the `guided` scheduling strategy wherefore the `static` policy wins the comparison.

Task 4 – Which loop to parallelize?

All task 4 is done using 4 threads.

1. j loop(column) parallelism implementation:

```
for( int row = 0; row < m1_rows; ++row ) {
   #pragma omp parallel for shared(row) //task 4.1
    for( int col = 0; col < m2_columns; ++col ) {
        for( int k = 0; k < m1_columns; ++k ) {
            output[ row * m2_columns + col ] += m1[ row * m1_columns + k ] * m2[ k
         }
```

2. For k loop(entry) parallelism implementation, each output vector element will be written simultaneously by all threads, therefore we decided to use reduction on a local variable and copy the value to each vector element when parallel execution is done:

```
for( int row = 0; row < m1_rows; ++row ) {
    for( int col = 0; col < m2_columns; ++col ) {
        float sum = 0;
        #pragma omp parallel for shared(row, col) reduction(+: sum)
        for( int k = 0; k < m1_columns; ++k ) {
            sum += m1[ row * m1_columns + k ] * m2[ k * m2_columns + col ];
        }
       output[ row * m2_columns + col ] = sum; }}
```

3. Table below shows the parallel execution time of the 3 strategies.

   Apparently, parallelizing the k loop is the worst strategy because the critical region basically eliminates the benefits of multithreading, and even worse, the thread synchronization added extra cost.

   Parallelizing i loop(row) slightly outperforms parallelizing j loop(column), which can be explained by cache false sharing. Compared with i loop parallelism, j loop parallelism means output vector accesses by each thread is more interleaved, i.e. higher chance of false sharing.

Parallelized loop:	i	j	k
Parallel execution time:	17.8018 s	19.7789 s	186.638 s

Table 3: performance for different strategies

Task 5 – Exploring SIMD

1. Without any loop permutations, there is not any vectorizing happening for any of the loops in the GEMM function.

2. With the loop permutation from Lab 2, the optimization flags `-O1` and `-O2` yield also no vectorization of the loops. However, using the optimization flag `-O3` vectorizes the inner-most loop (column loop; `src/vector_ops.cpp:242:39: note: loop vectorized`). The second inner-most loop (*k* loop), however, is not vectorized (`src/vector_ops.cpp:241:31: note: bad data references.`). Also, the outer-most loop (row loop) is not vectorized either (`src/vector_ops.cpp:240:31: note: not vectorized: multiple nested loops.`).

3. With no vectorized GEMM loops (optimization flag `-O2`), the program has the following running time.

```
Total time in dot: 30.82s
Total time (all): 33.5521s
Percentage in dot: 91.8574%
```

   With the vectorized inner-most loop (optimization flag `-O3`), the program's running time improves as seen below.

```
Total time in dot: 10.9205s
Total time (all): 13.4652s
Percentage in dot: 81.1018%
```

   The total speedup is $S = \frac{33.5521\,\mathrm{s}}{13.4652\,\mathrm{s}} = 2.4918$.