Name: Chitwan Goel
Roll No. 210295

CS610 Assignment 3

1 Problem 1

Both aligned and unaligned variants for `SSE` and `AVX` intrinsics were implemented. `ikj` variant was used in order to vectorise the matrix multiplication code. Below table shows the results obtained on `csews172`.

Version	Time (ms)
Sequential	855
Unaligned AVX	79
Aligned AVX	59
Unaligned SSE	114
Aligned SSE	81

Table 1: Performance Comparison of various versions (Data taken on `csews172`)

Running the Code

The directory `problem1-dir` has the code for this part. There is a file `problem1.cpp` and script `run.sh` inside the directory. Executing `run.sh` will generate `problem1.out`. `problem1.out` will require one argument `IS_ALIGNED` depending on whether aligned or unaligned variant is to be run. If aligned variant has to be run, then the argument should be 1, else it should be 0.
In the `problem1-dir` directory,

```
chmod +x run.sh
./run.sh
./problem1.out <IS_ALIGNED>
```

2 Problem 2

Below is the approach for vecotrisation of prefix sum using `AVX` intrinsics:
For each window of size 8 integers starting from `i`, we do the following:

- Load `[h,g,f,e,d,c,b,a]` into vector register `x`, where a is at source[i].
- `tmp0 = [g,f,e,d,c,b,a,0]` (shifted by 4 bytes)
- `tmp1 = x + tmp0 = [g+h, f+g, e+f, d+e, c+d,b+c,a+b,a]`
- Shift `tmp1` left by 8 bytes while shifting in zeros and store the shifted result in `tmp0`.
- `tmp2 = tmp1 + tmp0 = [e+f+g+h, d+e+f+g, e+f+c+d, d+e+b+c, c+d+a+b, b+c+a,a+b,a]`
- Shift `tmp2` by 16 bytes, store in `tmp0` and store `out` as `tmp0 + tmp2 = [a..h, a..g, a..f, a..e, a..d, a..c, a..b, a]`
- Then add the `offset` from previous window into `out`.
- For next window, the offset should be calculated as the biggest sum broadcasted in the `offset` register.

The directory `problem2-dir` has the code for this part. There is a file `problem2.cpp` and script `run.sh` inside the directory. Executing `run.sh` will generate `problem2.out`.
In the `problem2-dir` directory,

```
chmod +x run.sh
./run.sh
./problem2.out
```

The results of timing for this problem were highly random across different runs (this was discussed with Sir). For most of the runs, even the serial version was performing way better than the `omp` version (which was given beforehand in the assignment). Also, `sse` and `avx` versions have comparable times, which is performing better changes with every run. Sometimes, `avx` performs better, while `sse` performs better otherwise. Following table summarises the results taken on `csews1`:

Version	**Time** (μs)
Serial	943500
OMP	863967
SSE	734405
AVX	749450

Table 2: Performance Comparison of various versions (Data taken on `csews1`)

3 Problem 3

The directory `problem3-dir` has the code for this part. There is a file `problem3.cpp` and script `run.sh` inside the directory. Executing `run.sh` will generate `problem3.out`.
In the `problem3-dir` directory,

```
chmod +x run.sh
./run.sh
./problem3.out -inp=<input_path> -thr=<num_producers> -lns=<lines_per_thread>
-buf=<shared_buffer_size> -out=<output_path>
```

4 Problem 4

The directory `problem4-dir` has the code for this part. There are 3 files `problem4-v0.c`, `210295-prob4-v1.c`, `210295-prob4-v2.c` and script `run.sh` inside the directory. Executing `run.sh` will generate `.out` files for the 3 versions.
In the `problem4-dir` directory,

```
chmod +x run.sh
./run.sh
```

After that any of `problem4-v0.out`, `210295-prob4-v1.out`, `210295-prob4-v2.out` can be run as per requirement. Running `problem4-v<i>.out` will generate the text file `results-v<i>.txt` which contains the results for corresponding versions. Also, timing will be printed on the console. Before running, make sure that `grid.txt` and `disp.txt` are present in the same directory (`problem4-dir`).

Version 1 (Serial but optimised)
There were a lot of redundant computations in the given code. For example, consider the line of code

```
q1 = fabs(c11 * x1 + c12 * x2 + c13 * x3 + c14 * x4 + c15 * x5 + c16 * x6 + c17 * x7 + c18 *
x8 + c19 * x9 + c110 * x10 - d1);
```

In this statement, the computations involving `x1` to `x9` can be computed outside the `for` loop involving `r10`. I removed all these redundant computations in the code and got around `2X` speedup. Again, I also tried to add loop unrolling for inner as well as outer loops, but unrolling degraded the performance. The time taken by the code increased highly and so I removed it from the code. There seemed less scope of loop permutation since all the loops are equivalent. The table presented has the results for unrolled versions also, but the file contains only `LICM` variant (since it was the most optimal). Keeping only unrolling without `LICM` made the code slower even than serial version.

Version 2 (Parallelised using OpenMP)

The `for` loops given in the code were brought to parallelisable form. `collapse` clause was used to parallelise the code. I tried quite a few values of the parameter in `collapse` clause, the most optimal performance was obtained when using 8 as the value. In addition to this, `schedule(dynamic)` was also used. When I tried without using `schedule(dynamic)`, the performance was not sufficiently good. One possible reason for this could be that the workload is not equally balanced for all the threads. Some of the threads have to perform relatively higher number of write operations to the file (if more iterations for those threads satisfy the constraints). Also, a sufficiently large number was used with the `schedule(dynamic)` clause. Furthermore, in incrementing the `pnts` variable, `#pragma omp atomic` was used since it is efficient than `#pragma omp critical`. Also, since the writes to file were required to be in the same order as the sequential version, `#pragma omp ordered` was used in writing to the file (and so the `ordered` clause in the pragma above the starting of the loop). Moreover, the 10 `fprintf` statements were clubbed into a single `fprintf` statement, in order to prevent overhead of multiple write operations.

The table below summarises the results obtained and the speedups (Various runs were done and the averages of the time is written in the table):

Version	**Time (s)**	**Speedup**
V0	372.81	1
V1 (Sequential but optimised)	172.8	2.15
V1 with 1 Loop unrolling	220.4	1.69
V1 with 2 Loop unrolling	286.4	1.30
V2 (Parallelised using OpenMP)	42.9	8.69

Table 3: Performance Comparison of various versions (Data taken on `csews1`)