Roll: 241110406

Assignment 3

Problem 1:

SSE4 and AVX2 Implementation:
Basic idea is the same as the sequential version, only difference being that using 128-bit vectors used by SSE4 intrinsics and 256-bit vectors in AVX2 intrinsics, allows operation on 4 and 8 floating point values simultaneously respectively.

Blocked Matrix Multiplication implementation:
Blocks of size 32 were used.

Worse or equivalent performance than sequential version is seen for sse4 versions when not specifying any optimization levels.

Compile command: g++ -mavx2 -o problem1 problem1.cpp

Timings:

Implementation	Time taken
Sequential	971 ms
SSE4	732 ms
SSE4 aligned	738 ms
SSE4 blocked	252 ms
AVX2	638 ms
AVX2 aligned	588 ms
AVX2 blocked	170 ms

Problem 2:

Implementation is based on the SSE4 version. One key thing to note here is that SSE4 has an api *_mm_slli_si128(__m128i vector, int n)* which can shift all the contents of a __m128i vector by n bits. A similar api *_mm256_slli_si256(__m256i vector, int n)* exists for AVX2 vectors but the difference is it shift the bits in each of the two 128-bit lanes, making it look like a concatenation of two *_mm_slli_si128* call on two separate vectors.

Suppose the original __m256i vector x = [1,2,3,4,5,6,7,8]
Let tmp0 = *_mm256_slli_si256(x, 4)* = [0,1,2,3,0,5,6,7]
Let tmp1 = _mm256_add_epi32(x, tmp0) = [1,3,5,7,5,11,13,15]
Let tmp2 = _mm256_slli_si256(tmp1, 8) = [0,0,1,3,0,0,5,11]
Let tmp3 = _mm256_add_epi32(tmp1, tmp2) = [1,3,6,10,5,11,18,26]

tmp3 contains two different prefix sums: tmp[0:3] has prefix sum of 1 to 4 and tmp[4:7] has prefix sum of 5 to 8.
To make it a complete prefix sum, prefix sum of tmp[0:3] is added to each element of tmp[4:7] making tmp3 = [1,3,6,10,15,21,28,36]

Compile command: g++ -O3 -mavx2 -fopenmp problem2.cpp -o problem2

Timings:
Array of size N = 2^30 is used, timings in microseconds.

Implementation	Time taken
Sequential	462880
SSE4	430797
AVX2	421456
OpenMP	422285

Problem 4:

Part 1:

Optimizations applied:

- **grid.txt** contains 3 columns, each row (1 to 10) first and second columns are start and end limits of x_i values (where i range from 1 to 10, i.e. x_1, x_2, x_3, …, x_10) and rows of third column contain the step size for each x_i. Each level of loop nest has additional calculations to generate all possible values for a single x_i. And for each value of x_i, the next level of loop calculates all possible values of x_(i+1) using additional calculations. Removing additional calculations and generating values of x_i in the loop itself using **dd_i** variables themselves shaves off around 15 seconds from the runtime of the original code.
- The loop nest generates all possible combinations of different values of x_i (x1 to x10) generating a vector X = <x1, x2, x3, …, x9, x10>. **disp.txt** can be thought of as a table containing factors (first 10 columns, 'c' variables in code), biases (11th column, 'd' variables in code), and comparison references(12th column, 'ey' variables in code which generate 'e' variables which are used for comparison). Factors from ith column are multiplied with x_i for all values of i and summed together, a bias 'd_i' is subtracted and is compared with 'e_i'. Every possible variation of vector X = <x1, x2, x3, …, x9, x10> is not useful and thus need not be calculated. Wasteful variations of X can be detected in early loops and thus skipped to improve efficiency. Each level of loop is now transformed to calculate the following expression:

$$temp_{ki} = \sum_{j=1}^{k} (c_{ij} * x_j) - d_i$$

  Where i varies from 1 to 10, and k is the level of loop control is on. The final condition check in the original code also calculates the same value 'q_i' but for k = 10.

$$q_i = abs(\sum_{j=1}^{10} (c_{ij} * x_j) - d_i)$$

  and is compared with the values in 'e_i' such that all the values of q_i are less than or equal to corresponding values of e_i.

```
if ((q1 <= e1) && (q2 <= e2) && (q3 <= e3) && (q4 <= e4) && (q5 <= e5) &&
                    (q6 <= e6) && (q7 <= e7) && (q8 <= e8) && (q9 <= e9) && (q10 <= e10)) {
                pnts = pnts + 1;}
```

  Assuming all the factors (c_ij) and biases (d_i) are always non-negative (*Also confirmed by the instructor),* we can conclude that the value of 'temp_ki' can only grow in positive direction as the value of k increases when we go to deeper levels of loop. So if at any value of k, 'temp_ki' crosses the value of 'e_i', it is not going to reduce in lower levels of loop and we know that the above condition is going to fail. So we skip calculations for deeper levels of the loop nest (because the vector X it is going to generate will be a wasteful vector) and move for the next iteration of x_k in the kth level of the loop nest.

Optimization achieved: Applying this kind of transformation reduces the runtime of code to **under 1 second**. Such efficiency is achieved because we are skipping the deeper levels of the nested loop where the time spent on calculation would be more when we know the condition would fail.

Tested on CSEWS9 system:
Time for sequential version = 392.62 seconds
Time for loop transformed version = 0.82 seconds
Time for OpenMP version = 0.75 seconds