Assignment 1

Q1) We perform DAXPY operation with formula X[i] = a*X[i] + Y[i]. The size of the vector is 65536. The code compares the time taken to complete this operation from 2 to 256 threads. I also made the code do 10000 repetition for additional verification.

16 and 32 threads give the fastest output for single iterations and 32 threads give the fastest value for 10000 repetition.

If the number of threads is increased after a certain point, the performance degrades because the operation is memory bound not compute bound due to the read of 2 vars and write of 1 var. After a certain number of threads, additional threads just hinder the memory management and slow down the process.

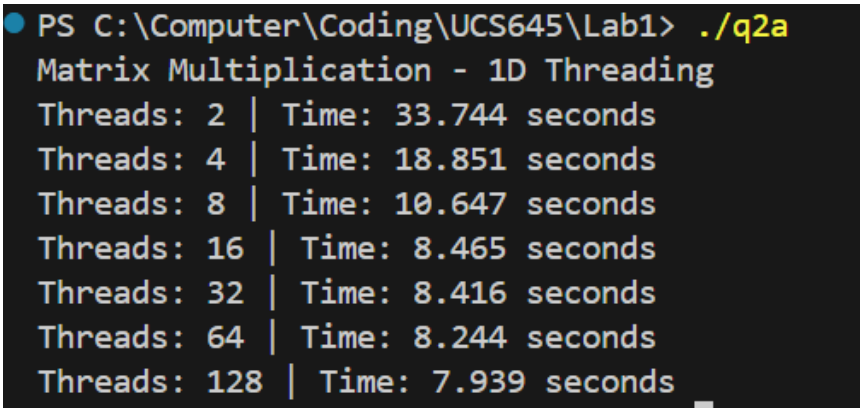
```
DAXPY OpenMP Performance
Vector size: 65536

Threads: 2 | Time: 0.000999928 seconds | Speedup: 1
Threads: 4 | Time: 0.000999928 seconds | Speedup: 1
Threads: 8 | Time: 0.000999928 seconds | Speedup: 1
Threads: 16 | Time: 0.00300002 seconds | Speedup: 0.333307
Threads: 32 | Time: 0.00300002 seconds | Speedup: 0.333307
Threads: 64 | Time: 0.00500011 seconds | Speedup: 0.199981
Threads: 128 | Time: 0.0079999 seconds | Speedup: 0.124993
Threads: 256 | Time: 0.0189998 seconds | Speedup: 0.0526283

Maximum Speedup: 1 achieved with 2 threads.
```

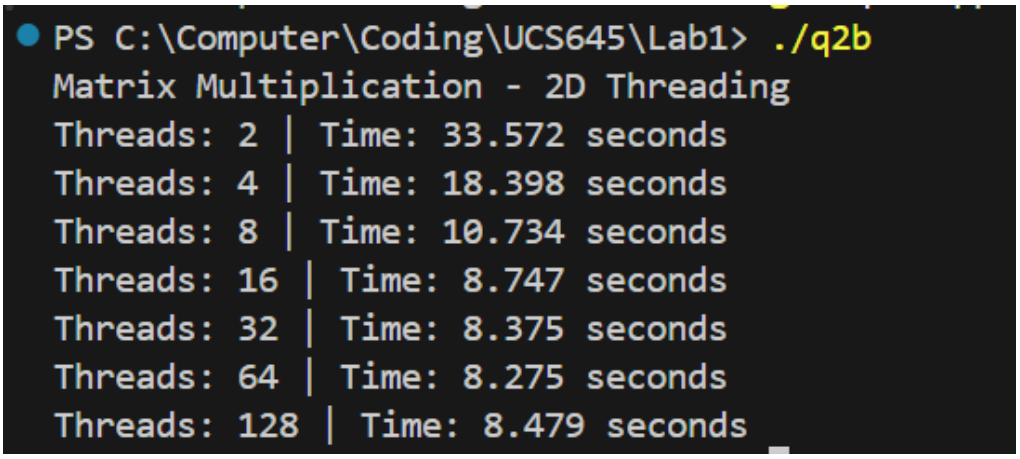
```
DAXPY OpenMP Performance
Vector size: 65536
Repeat count: 10000

Threads: 2 | Time: 5.258 seconds | Speedup: 1
Threads: 4 | Time: 3.409 seconds | Speedup: 1.54239
Threads: 8 | Time: 1.817 seconds | Speedup: 2.89378
Threads: 16 | Time: 1.572 seconds | Speedup: 3.34478
Threads: 32 | Time: 1.548 seconds | Speedup: 3.39664
Threads: 64 | Time: 1.712 seconds | Speedup: 3.07126
Threads: 128 | Time: 1.768 seconds | Speedup: 2.97398
Threads: 256 | Time: 1.63 seconds | Speedup: 3.22577

Maximum Speedup: 3.39664 achieved with 32 threads.
```

Q2) The code initializes three vectors with 1000*1000 elements each. These are the matrices for the operation. It repeats the multiplication process 5 times for every thread count. The test begins with 2 threads and doubles the count until it reaches 128.

In 1D matrix, Pragma omp for handles the distribution of the outermost loop iterations. Omp_get_wtime function captures the start and end times.

For 2D matrix, pragma omp parallel for collapse(2) directive merges the two outer loops into a single parallel region. This method is known as 2D threading.





Q3) This code approximates the value of pi through the numerical integration of the function $\pi = \int_0^1 \frac{4.0}{1+x^2} dx$ over the interval [0, 1]. The algorithm performs 100000000 iterations to calculate the area of rectangles under the curve.
The code measures the execution time for thread configurations from 2 to 256. It uses #pragma omp parallel for reduction(+:sum) to manage the sum of values across threads. This ensures data integrity for parallel updates to the sum variable. The program then outputs the final result and the elapsed time for each thread level.

```
PS C:\Computer\Coding\UCS645\Lab1> ./q3
Threads: 2 | Pi: 3.14159 | Time: 0.586 seconds
Threads: 4 | Pi: 3.14159 | Time: 0.303 seconds
Threads: 8 | Pi: 3.14159 | Time: 0.158 seconds
Threads: 16 | Pi: 3.14159 | Time: 0.132 seconds
Threads: 32 | Pi: 3.14159 | Time: 0.132 seconds
Threads: 64 | Pi: 3.14159 | Time: 0.123 seconds
Threads: 128 | Pi: 3.14159 | Time: 0.12 seconds
Threads: 256 | Pi: 3.14159 | Time: 0.127 seconds
```