
DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

ECE415 - HIGH PERFORMANCE COMPUTING SYSTEMS

# Lab 2 - K-Means parallelization using OpenMP

*Students:*
Karageorgos Nikolaos 02528, nkarageorgos@uth.gr
Lamprinos Isidoros 02551, ilamprinos@uth.gr

November 2021

# Contents

# 1 Introduction

K-means clustering is a method of vector quantization, originally from signal processing, that aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean (cluster centers or cluster centroid), serving as a prototype of the cluster. This process stops when an observation is no longer needed to change cluster. It is perceived for this algorithm that as the number of clusters grows so does the runtime. This makes it a good application for parallelizations. This assignment's subject matter was to parallelize an implementation of k-means clustering algorithm using OpenMP API.
Firstly, we run profiling in the given code to check where the most time was spend. After that we added OpenMP directives to achieve better performance using parallelization.

# 2 System Specifications

The inf-mars1(10.64.82.31) is a system with:

- one i7-4820K processor
- each processor is 4-way multicore
- with each core 2-way SMT (Intel Hyperthread)

The profiling of the original and the development of the parallelized code were made on the inf-mars1 system.

The csl-artemis(10.64.82.65) is a system with:

- two Intel Xeon E5-2695 processors
- each processor 16-way multicore
- with each core is 2-way SMT( Intel hyperthread).

The final measurements were made on the csl-atremis system. We tried to have these final results while the system did not have any other users running experiments on it.

# 3 Profiling on the sequential version

In order to find the points of the program that consumed the most time we run profiling using Intel tool Vtune Amplifier. The profiling testing was made with 8 threads and texture17965.bin as input (as it was suggested). By running hotspot analysis the below results were extracted:

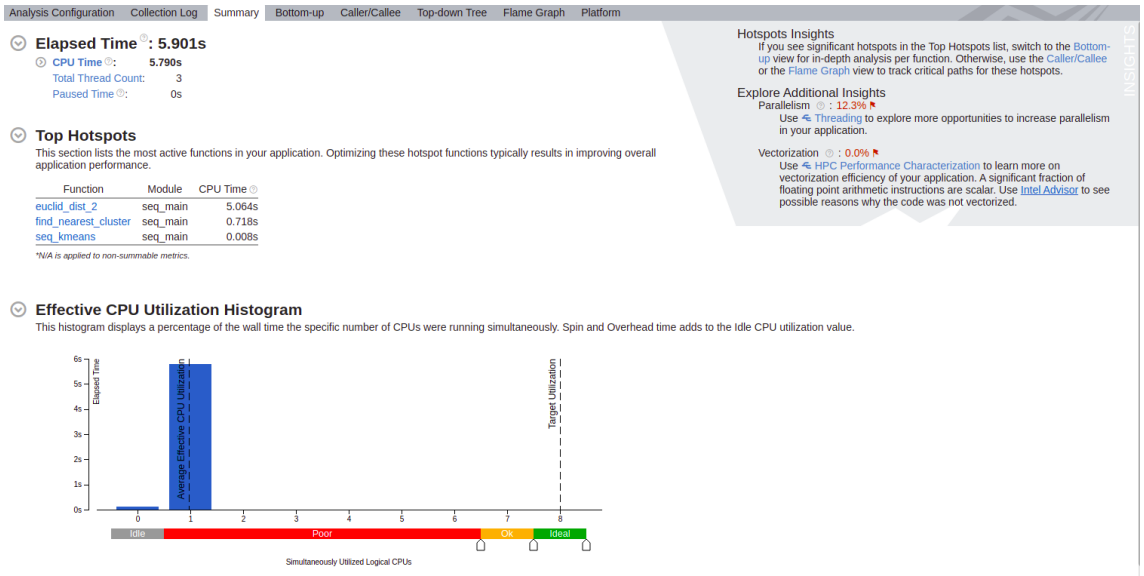


Figure 1: Vtune Amplifier summary

It is observed that the main hotspot is the function euclid_dist_2 which called by function find_nearest_cluster.

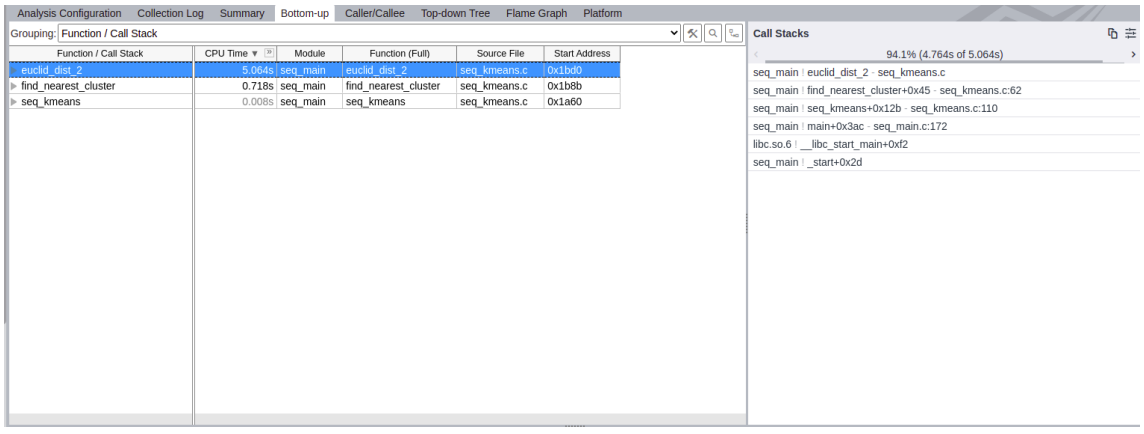


Figure 2: Vtune Amplifier bottom-up

# 4 Parallelization attempts

At this point having as guide the profiling results , several parallelizations attempts were experimented with, aiming to achieve better performance. The whole development of the parallel code version was made using mostly 8 threads and as input the image texture17965.bin

## I.

Initially, we went with the first suggestion of the profiling and we tried to parallelize the for loop inside function euclid_dist_2. We used OpenMP directive 'omp parallel for reduction(+:variable)'. Reduction was used for variable 'ans' as it was used for storing a total sum and operations needed to be made atomically.

```
float euclid_dist_2(int    numdims,  /* no. dimensions */
                    float *coord1,   /* [numdims] */
                    float *coord2)   /* [numdims] */
{
    int i;
    float ans=0.0;

    #pragma omp parallel for schedule(guided) reduction(+:ans)
    for (i=0; i<numdims; i++)
        ans += (coord1[i]-coord2[i]) * (coord1[i]-coord2[i]);

    return(ans);
}
```

However, it was observed that runtime was increased a lot (more than 1 min). This was caused maybe because euclid_dist_2 is called numClusters times by find_nearest_cluster creating a big overhead. So this parallelization was unsuccessful.

## II.

Secondly, we turned to next point of the code that profiling suggested and went to parallelize the for loop inside function find_nearest_cluster.We used directive 'omp parallel for private(variable)' for variable 'dist' and directive 'omp parallel critical' for the part of the if statement.

```
    #pragma omp parallel for private(dist)
    for (i=1; i<numClusters; i++) {
        dist = euclid_dist_2(numCoords, object, clusters[i]);
        /* no need square root */

        #pragma omp critical
        {
          if (dist < min_dist) { /* find the min and its array index */
              min_dist = dist;
              index    = i;
          }
        }
    }
    return(index);
}
```

Again this parallelization did not work as it damaged performance a lot (execution time more than 1 min) and so it was not kept.

## III.

At this point, having the first two attempts failed lead us to try parallelizing function seq_kmeans which calls the other two functions.
At first, we tried to parallelize the two 'for' loops in the beginning of the function that initialize matrices: 'membership' and 'cluster'. The first one was simple as we just used directive 'pragma omp parellel for'. However the second one cannot be parallelized the way it is because there are data dependencies. To make it parallelizable we can alter the instruction:

$$newClusters[i] = newClusters[i-1] + numCoords;$$

and write it as:

$$newClusters[i] = newClusters[0] + i * numCoords;$$

These two expressions produce the same result as the (i-1)-th element of matrix newCluster increased by numCoords is practically equal to the first element of the matrix (newClusters[0]) increased by i*numCoords. In this way we can parallelize this for loop too. Having parallelized the two for loops, we thought that placing them inside a parallel region would be even more beneficial as this would prevent an overhead from creating and destroying new threads that could be used for both for both loops' work. Due to this fusion we used 'nowait' in directive 'omp parallel for' because there is dependence between the two for loops and in this way each thread can wait for the others only in the barrier in the end of the parallel region.

```
#pragma omp parallel
  {
    #pragma omp for schedule(static) nowait
    for (i=0; i<numObjs; i++) membership[i] = -1;

    #pragma omp for schedule(static) nowait
    for (i=1; i<numClusters; i++)
    newClusters[i] = newClusters[0] + i*numCoords;
}
```

This parallelization did not have a big impact on performance managing to drop execution time by little. However we decided to keep it anyway. After experimenting with scheduling types we chose 'static'.

## IV.

Next, we attempted to parallelize the first nested 'for' loop. We parallelized the outside for loop and made variables 'j' and 'index' private. Additionally, we used 'reduction' for variable 'delta' to ensure that operations would be made atomically. Moreover, we had to make sure that operations to newCluster[index] and newCluster[index][j] would be made atomically and so we used directive 'pragma omp atomic' for both of them.

```
do {
    delta = 0.0;
    #pragma omp parallel for  private(index,j) schedule(guided) reduction(+:delta)
    for (i=0; i<numObjs; i++) {
        /* find the array index of nestest cluster center */
        index = find_nearest_cluster(numClusters, numCoords, objects[i],
                                     clusters);

        /* if membership changes, increase delta by 1 */
        if (membership[i] != index) delta += 1.0;

        /* assign the membership to object i */
        membership[i] = index;

        /* update new cluster center : sum of objects located within */
        #pragma omp atomic
        newClusterSize[index]++;
        for (j=0; j<numCoords; j++)
            #pragma omp atomic
            newClusters[index][j] += objects[i][j];
    }
```

This parallelization attempt was the most beneficial so far as it improved performance rapidly (dropping time from 5.53 sec to 1.33 sec). After that we tried various types for schedule and the best without a significant difference was 'guided' so we kept it.

## V.

We also tried to parallelize the second nested 'for' loop that takes the average of the sum and replaces old cluster center with newClusters. We declared variable 'j' as private.

```
    /* average the sum and replace old cluster center with newClusters */
    #pragma omp parallel for  private(j) schedule(guided)
    for (i=0; i<numClusters; i++) {
        for (j=0; j<numCoords; j++) {
            if (newClusterSize[i] > 0)
                clusters[i][j] = newClusters[i][j] / newClusterSize[i];
            newClusters[i][j] = 0.0;   /* set back to 0 */
        }
        newClusterSize[i] = 0;   /* set back to 0 */
    }
```

However, this change did not have any impact on the performance as it remained the same and so we decided not to keep it.

**VI.**
After that we attempted placing the do-while body (both nested for loops parallelized) inside a big parallel region.

```
do {
    delta = 0.0;
    #pragma omp parallel
    {
        #pragma omp parallel for  private(index,j) schedule(guided) nowait reduction(+:delta)
        for (i=0; i<numObjs; i++) {
            /* find the array index of nestest cluster center */
            index = find_nearest_cluster(numClusters, numCoords, objects[i],
                                         clusters);

            /* if membership changes, increase delta by 1 */
            if (membership[i] != index) delta += 1.0;

            /* assign the membership to object i */
            membership[i] = index;

            /* update new cluster center : sum of objects located within */
            #pragma omp atomic
            newClusterSize[index]++;
            for (j=0; j<numCoords; j++)
                #pragma omp atomic
                newClusters[index][j] += objects[i][j];
        }

        /* average the sum and replace old cluster center with newClusters */
        #pragma omp parallel for  private(j) schedule(guided) nowait
        for (i=0; i<numClusters; i++) {
            for (j=0; j<numCoords; j++) {
                if (newClusterSize[i] > 0)
                    clusters[i][j] = newClusters[i][j] / newClusterSize[i];
                newClusters[i][j] = 0.0;   /* set back to 0 */
            }
            newClusterSize[i] = 0;   /* set back to 0 */
         }
    }
    delta /= numObjs;
} while (delta > threshold && loop++ < 500);
```

This attempt did not work as it dropped execution time, so we did not keep it.

**VII.**
Lastly, we thought of parallelizing again function find_nearest_cluster now that the part that calls it has been also parallelized. This time it is not necessary to declare 'dist' as private or place a critical region as we have declared 'index' as private before in function seq_kmeans, so the code remains efficient. This change worked as it dropped execution time from 1.33 sec to 1.30 sec, so we kept it.

So the final code version contains the (III), (V), (VII) parallelization attempts while the rest of them were discarded. We checked that the code remained efficient after these changes by comparing the clusters membership results of the sequential and the parallel code version.

# 5 Compiler flags and environmental variables

For the purpose of this assignment we used the icc compiler with flags: -fast -g -qopemp -DNDBEBUG. We also tried some other flags for faster mathematical calculations like: -fast-transcendentals and -ffast-math. However, these did not make any difference.

From environmental variables we used OMP_PROC_BIND which controls the thread affinity policy and whether OpenMP threads can be moved between places and we set its value to TRUE which binds the threads to places. We also used OMP_DYNAMIC which enables or disables dynamic adjustment of the number of threads available for running parallel regions and set its value to FALSE which disables dynamic adjustment.

# 6 Performance Diagrams

In the following diagrams we graphically present the results of our experiments, mean CPU time and standard deviation as the number of threads increases. We run each experiment 12 times and excluded from the final results the best and the worst time. As input were used images color17695.bin, edge17695.bin and texture17695.bin with number of clusters 2000.

- color17695.bin

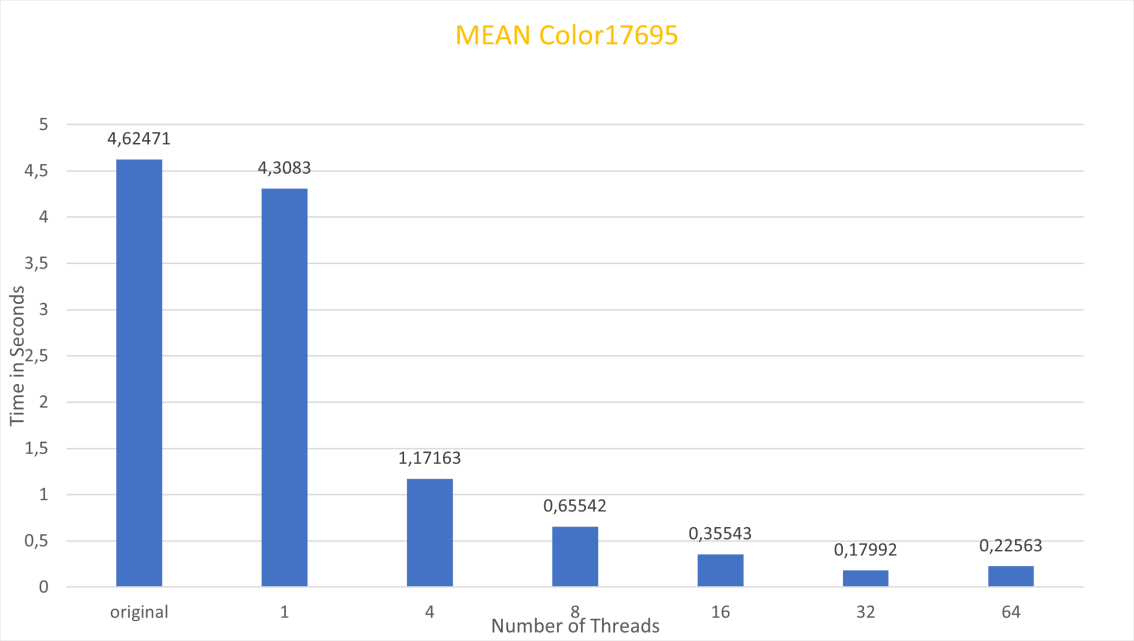


Figure 3: Mean time for color17695.bin

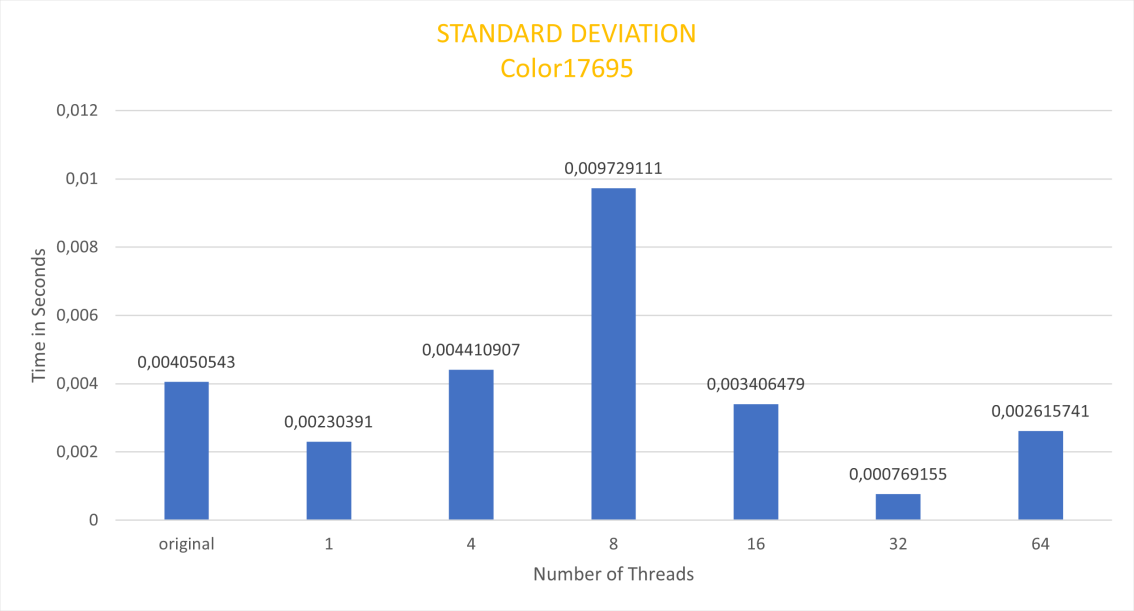


Figure 4: Standard Deviation for color17695.bin

- edge17695.bin

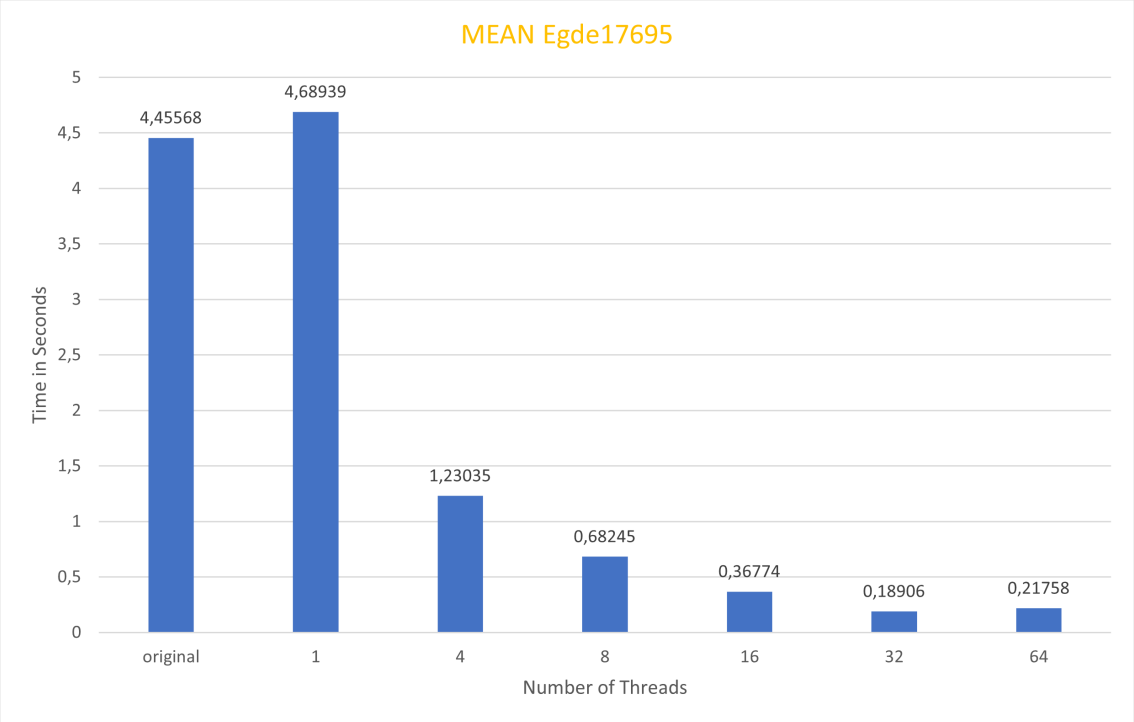


Figure 5: Mean time for edge17695.bin

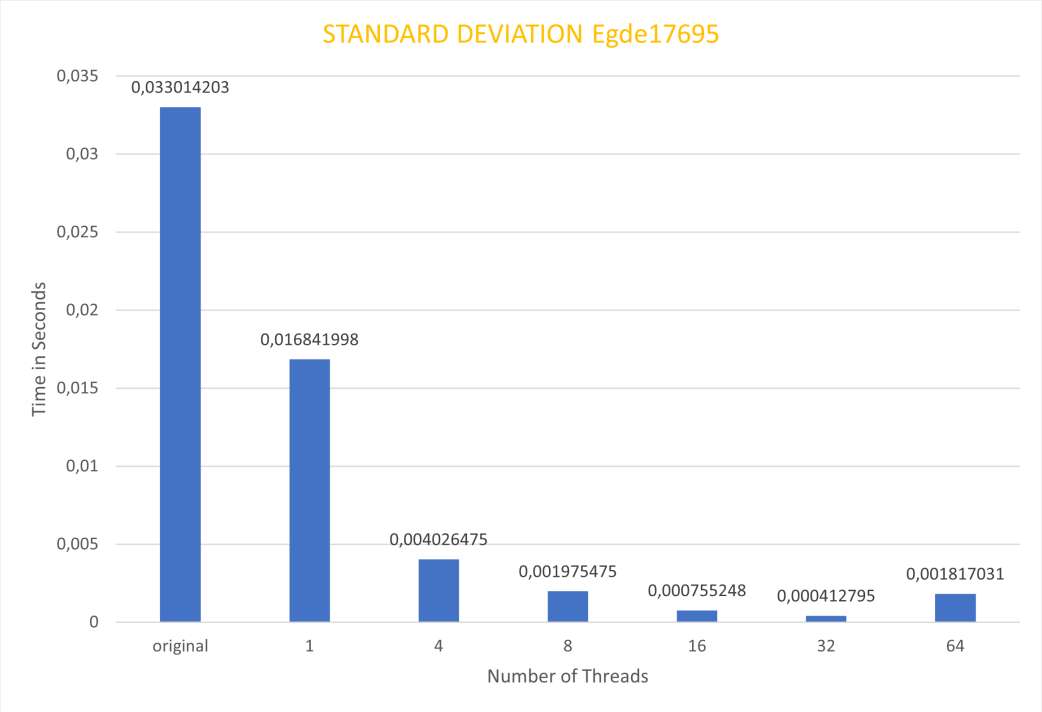


Figure 6: Standard Deviation for edge17695.bin

- texture17695.bin

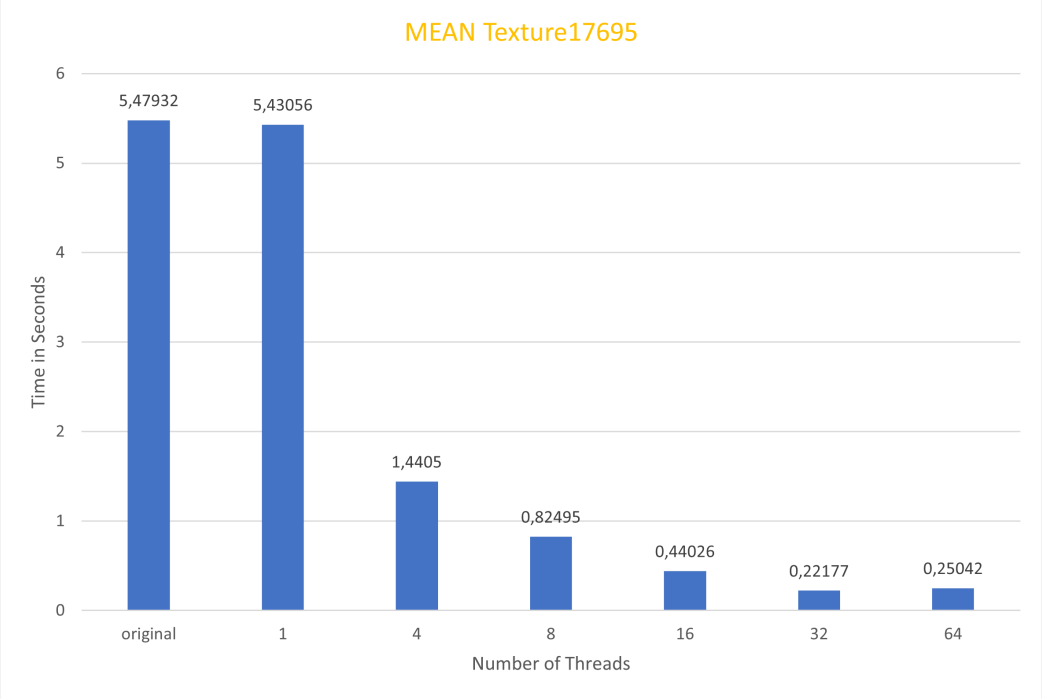


Figure 7: Mean time for texture17695.bin

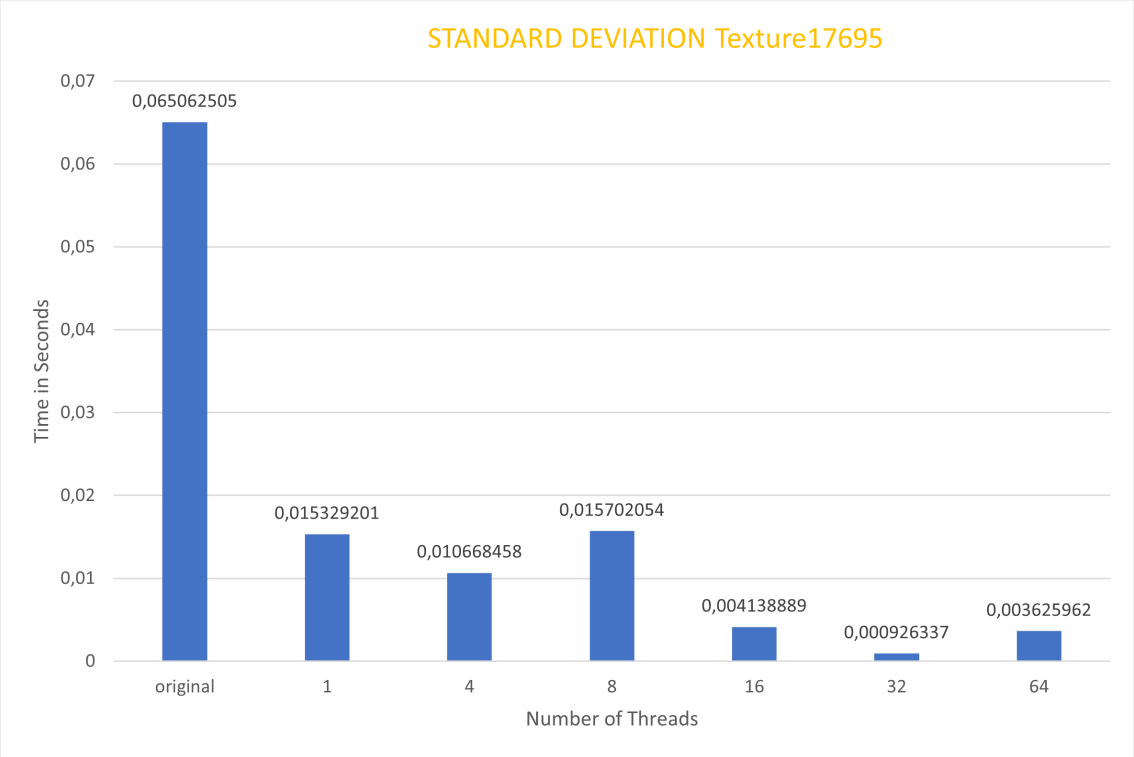


Figure 8: Standard Deviation for texture17695.bin

We observe that for all input images generally that mean CPU time drops as the number of threads increases except when using 64 threads the mean time slightly increases from the value that it had with 32 threads.