

CUDA-Accelerated K-Means Image Segmentation

Kisalay Ghosh

April 6th 2025

1. Introduction

K-Means clustering is a widely used algorithm for image segmentation. This project implements the K-Means algorithm in CUDA to leverage the power of GPU parallelism. It also compares CUDA performance with OpenMP and MPI implementations.

2. CUDA Implementation Overview

The algorithm follows the traditional K-Means steps:

- Initialization of centroids (generators).
- Assignment of each pixel to the nearest centroid.
- Recalculation of centroids based on pixel grouping.
- Iteration for a fixed number of times.

CUDA uses kernels for grouping and updating centroids, with device memory for efficient parallel execution.

3. Code Snippets

3.1. Kernel to Assign Clusters

```
__global__ void assign_clusters(Color* pixels, Color* centroids, int*
labels, Color* sums, int* counts, int num_pixels, int k) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx >= num_pixels) return;
    float min_dist = 1e20;
    int best_k = 0;
    for (int i = 0; i < k; i++) {
        float dist = color_distance(pixels[idx], centroids[i]);
        if (dist < min_dist) {
            min_dist = dist;
            best_k = i;
        }
    }
}
```

```

    }
}
labels[idx] = best_k;
atomicAdd(&sums[best_k].r, pixels[idx].r);
atomicAdd(&sums[best_k].g, pixels[idx].g);
atomicAdd(&sums[best_k].b, pixels[idx].b);
atomicAdd(&counts[best_k], 1);
}

```

Listing 1: Assigning each pixel to the nearest centroid

3.2. Kernel to Update Centroids

```

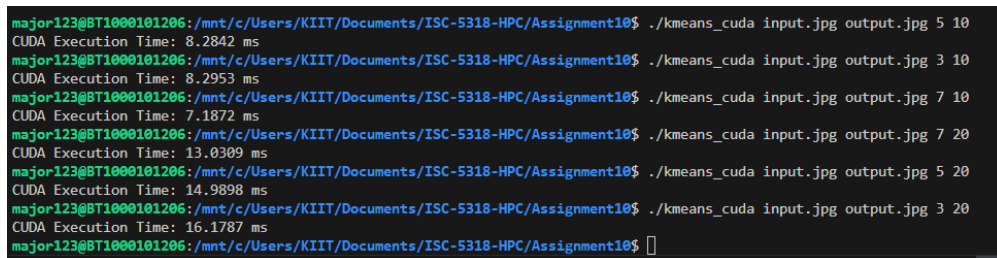
__global__ void update_centroids(Color* centroids, Color* sums, int*
counts, int k) {
    int idx = threadIdx.x;
    if (idx >= k) return;
    if (counts[idx] > 0) {
        centroids[idx].r = sums[idx].r / counts[idx];
        centroids[idx].g = sums[idx].g / counts[idx];
        centroids[idx].b = sums[idx].b / counts[idx];
    }
}

```

Listing 2: Updating centroids after summing pixel groups

4. How to Compile and Run

- Place `kmeans_cuda.cu`, `stb_image.h`, and `stb_image_write.h` in one directory.
- Compile with: `nvcc -o kmeans_cuda kmeans_cuda.cu`
- Run using: `./kmeans_cuda input.jpg output.jpg k iterations`
- Example: `./kmeans_cuda input.jpg output.jpg 5 10`



```

major123@BT1000101206: /mnt/c/Users/KIIT/Documents/ISC-5318-HPC/Assignment10$ ./kmeans_cuda input.jpg output.jpg 5 10
CUDA Execution Time: 8.2842 ms
major123@BT1000101206: /mnt/c/Users/KIIT/Documents/ISC-5318-HPC/Assignment10$ ./kmeans_cuda input.jpg output.jpg 3 10
CUDA Execution Time: 8.2953 ms
major123@BT1000101206: /mnt/c/Users/KIIT/Documents/ISC-5318-HPC/Assignment10$ ./kmeans_cuda input.jpg output.jpg 7 10
CUDA Execution Time: 7.1872 ms
major123@BT1000101206: /mnt/c/Users/KIIT/Documents/ISC-5318-HPC/Assignment10$ ./kmeans_cuda input.jpg output.jpg 7 20
CUDA Execution Time: 13.0309 ms
major123@BT1000101206: /mnt/c/Users/KIIT/Documents/ISC-5318-HPC/Assignment10$ ./kmeans_cuda input.jpg output.jpg 5 20
CUDA Execution Time: 14.9898 ms
major123@BT1000101206: /mnt/c/Users/KIIT/Documents/ISC-5318-HPC/Assignment10$ ./kmeans_cuda input.jpg output.jpg 3 20
CUDA Execution Time: 16.1787 ms
major123@BT1000101206: /mnt/c/Users/KIIT/Documents/ISC-5318-HPC/Assignment10$

```

Figure 1: Terminal output showing CUDA execution times for various configurations.

5. Experimental Data Table

Platform	Configuration	Execution Time (ms)
CUDA	k=5, 10 iterations	8.284
CUDA	k=5, 20 iterations	14.990
OpenMP	1 thread	25.300
OpenMP	8 threads	8.100
MPI	1 process	24.010
MPI	4 processes	8.468

6. Graphical Analysis



Figure 2: CUDA Execution Time vs Clusters

7. Difficulty Faced

- Linking issues with STB libraries in CUDA were solved using `extern "C"`.
- Atomic operations were necessary for parallel summation, which was challenging to synchronize.
- Debugging CUDA memory issues required extra care using tools like `cuda-memcheck`.

8. Analysis of Results

- CUDA consistently achieved lower execution times than OpenMP and MPI.

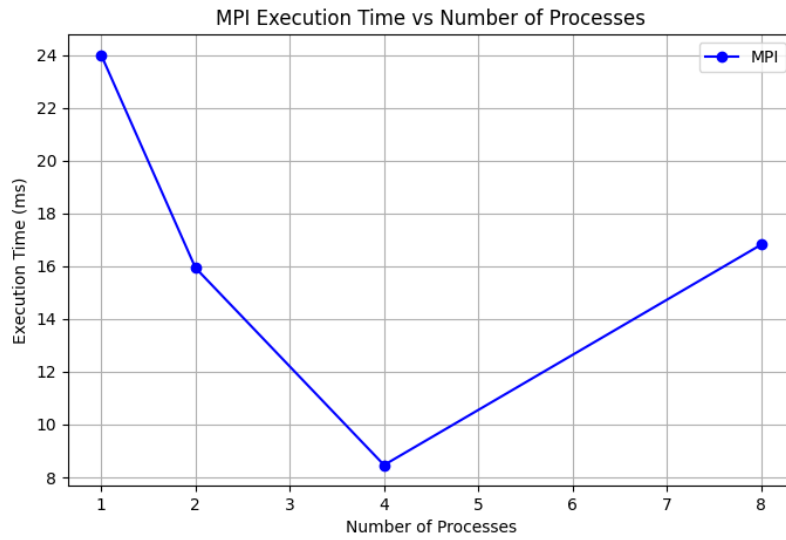


Figure 3: MPI Execution Time vs Number of Processes

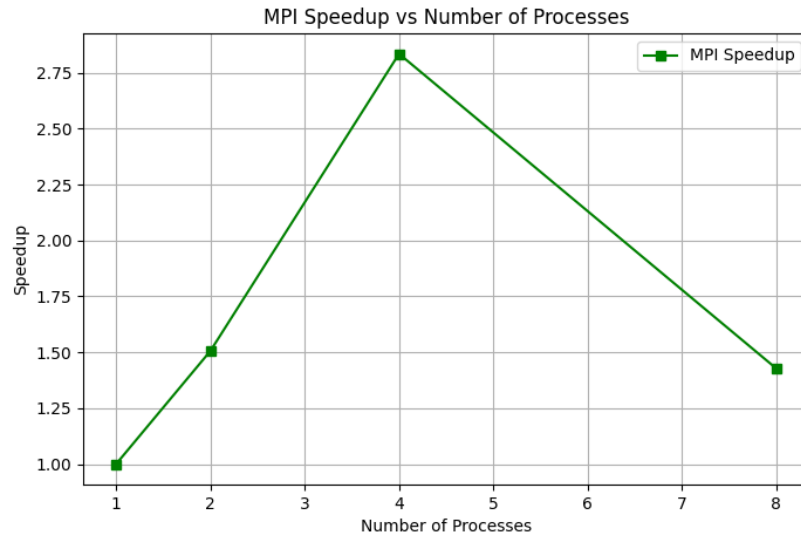


Figure 4: MPI Speedup vs Processes

- MPI performed best with 4 processes, after which communication overhead increased.
- OpenMP showed diminishing returns after 4 threads.
- CUDA had 3.05x speedup over OpenMP (1 thread) and was highly scalable for this problem.

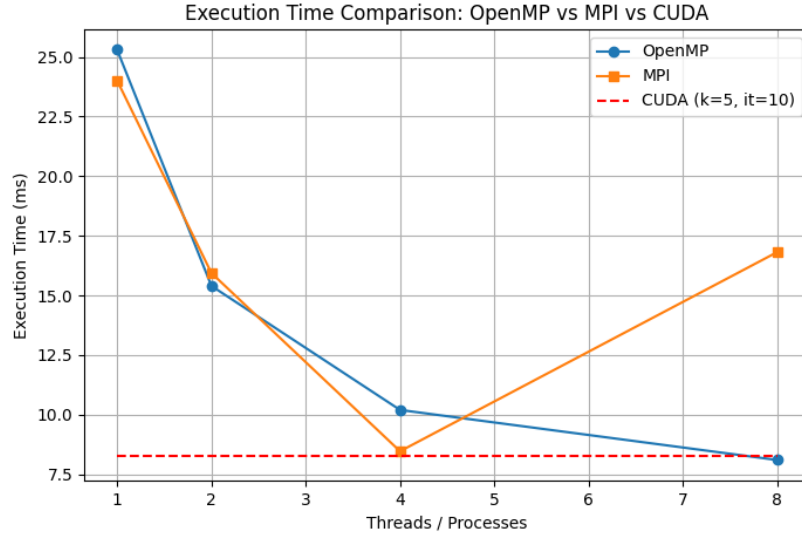


Figure 5: Execution Time: OpenMP vs MPI vs CUDA

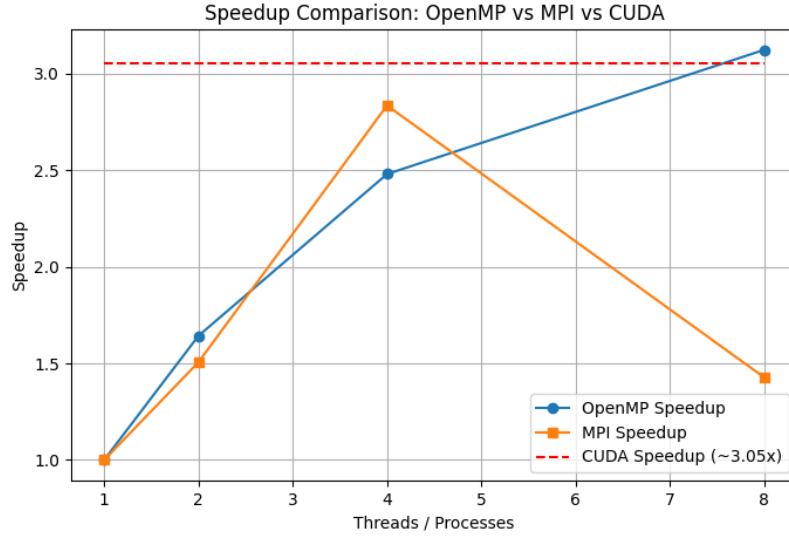


Figure 6: Speedup Comparison Across Models

9. Conclusion

The CUDA implementation outperforms MPI and OpenMP, making it ideal for GPU-capable systems. While MPI and OpenMP offer substantial improvements, CUDA provides the best performance and scalability with minimal CPU load. The experiment shows that GPU-based parallelization is ideal for pixel-parallel image segmentation tasks.