

CMP4005 Parallel Computing

(Big Assignment)

Parallelizing K-Means Clustering for RGB Images and Computing silhouette Score.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Sec** | **B.N** | **Code** |
| Basma Elhoseny | 1 | 16 | 9202381 |
| Sarah Elzayat | 1 | 29 | 9202618 |

Table Of Content

[Pipeline: 3](#_Toc167001820)

[Development Steps: baby steps move mountains 😀 4](#_Toc167001821)

[Implementation: 5](#_Toc167001822)

[Kernel (1): 5](#_Toc167001823)

[Optimization (1): 5](#_Toc167001824)

[Optimization (1): Less Congestion at the global memory for K2 7](#_Toc167001825)

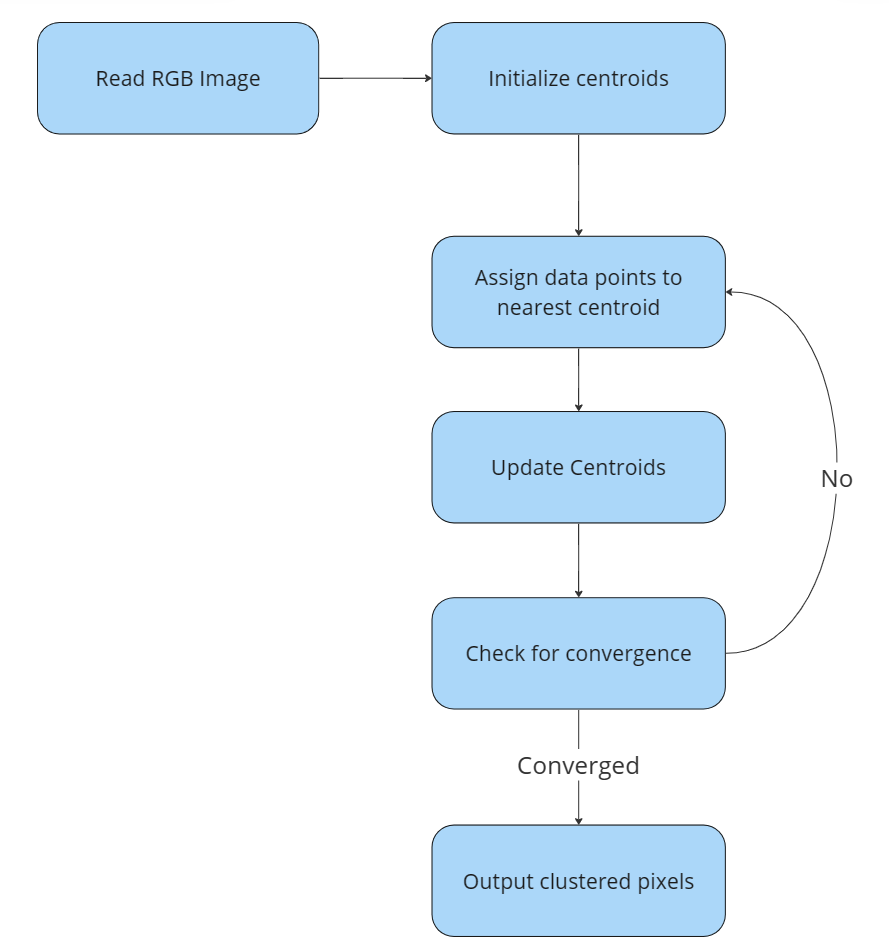
[Streaming: 8](#_Toc167001826)

[Kernel (3): 9](#_Toc167001827)

[**Performance Analysis** 10](#_Toc167001828)

[Unsuccessful Trials: 13](#_Toc167001829)

# Pipeline:

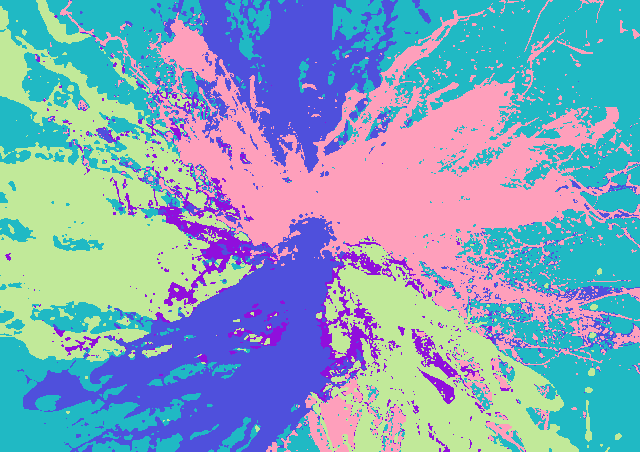


# Development Steps: baby steps move mountains 😀

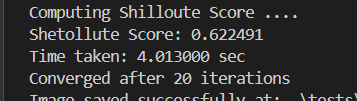
1. Build CPU version for Clustering Grey Scale Images [cpu.cu]

1. Extend to be 3 Channel [cpu\_3.cu]

1. Begin with Basic GPU parallel Implementation [gpu\_3.cu]
2. Enhancement in Kernel 1 (Assign Data Points to the nearest Centroid) [gpu\_3\_1.cu]
3. Enhancement in Kernel 2 (Compute New Centroids [gpu\_3\_2.cu] (VERY EFFECTIVE)
4. Streaming for Kernel 1 (Assign Data Points to the nearest Centroid) gpu\_3\_stream\_0.cu]
5. Trying to Compute Silhouette Score for clustered Images (CPU/GPU)



# Implementation:

### Kernel (1):

Main Task of this kernel is getting nearest centroid to each data point

d\_data\_points

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |

d\_clutser\_assigment

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |

d\_centroids

|  |  |  |
| --- | --- | --- |
|  |  |  |

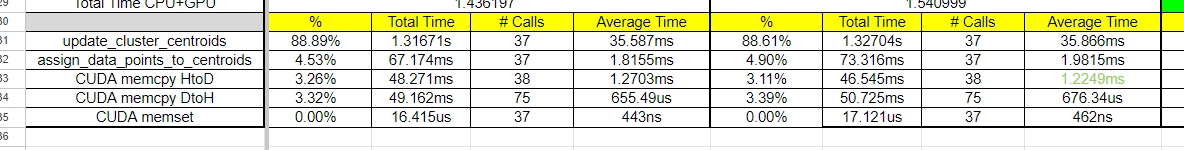
#### Optimization (1):

Centroids are shared between all threads in all blocks, Make thread put centroids in shared memory to be used by other threads in the same block [reduce read from global]

Sh\_centroids

|  |  |  |
| --- | --- | --- |
|  |  |  |

No Significant enhancement ?! Due to small no of centroids 😀



A slight increase in the avg time for kernel 1 due sync between threads within block for centroids load in shared memory.

Kernel (2):

Main task for this kernel is to compute summation of all points within same cluster then average to get new centroids

d\_data\_pointsA black background with a black square

Description automatically generated with medium confidenceA black background with a black square

Description automatically generated with medium confidence

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | A black background with a black square  Description automatically generated with medium confidence | A black background with a black square  Description automatically generated with medium confidence |  |  |  |  |  |  |

sh\_data\_points

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |

d\_clutser\_assigment

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |

sh\_clutser\_assigment

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |

Each Thread loads a data point to the shared Memory + The corresponding cluster index to the shared memory.

Then we need to wait till all threads finish loading <3

Thread 0: (Is responsible for ALL data points loaded by other threads in the block)

Data\_points\_sum (per cluster)

|  |  |  |
| --- | --- | --- |
|  |  |  |

Cluster\_sizes (per cluster)

|  |  |  |
| --- | --- | --- |
|  |  |  |

Add this local reduction result to the global memory reduction result (Atomic Add)

d\_centroids

|  |  |  |
| --- | --- | --- |
|  |  |  |

A black background with a black square

Description automatically generated with medium confidenceA black background with a black square

Description automatically generated with medium confidenced\_cluster\_sizes (per cluster)

|  |  |  |
| --- | --- | --- |
|  |  |  |

#### Optimization (1): Less Congestion at the global memory for K2

Instead of having all thread have race condition on adding centroids and cluster sizes to global memory d\_cluster\_sizes & d\_centroids they will have race condition within their block in the shared Memory (Near and faster to access) and only the first thread in each block only try to write the global Memory [Less Race conditions]

Sh\_data\_points\_sum

|  |  |  |
| --- | --- | --- |
|  |  |  |

Sh\_cluster\_sizes

|  |  |  |
| --- | --- | --- |
|  |  |  |

d\_centroids [By Th0 only 😊]

|  |  |  |
| --- | --- | --- |
| A black background with a black square  Description automatically generated with medium confidence |  |  |

d\_cluster\_sizes (per cluster) [By Th0 only 😊]

|  |  |  |
| --- | --- | --- |
|  |  |  |

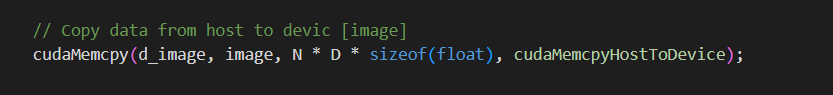
A screenshot of a computer

Description automatically generated

## Streaming:

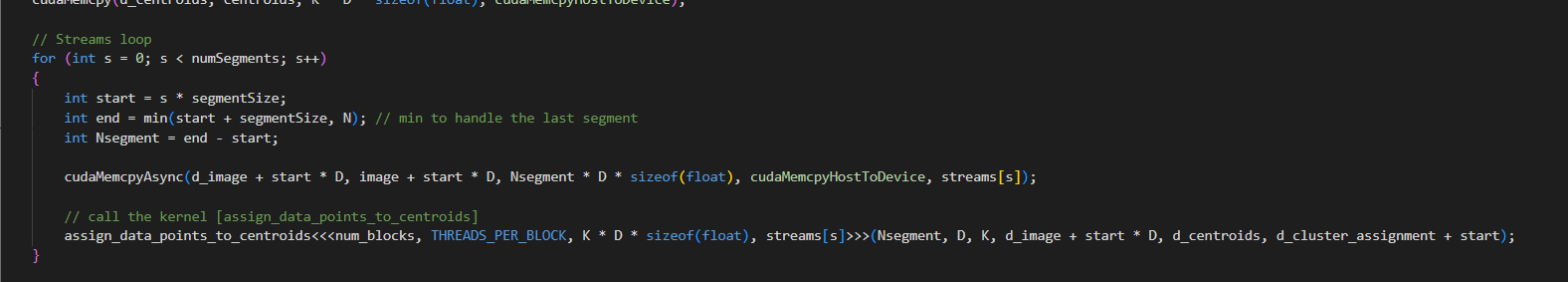
we only copy between kernel calls in the iteration on Kmeans algorithm are just centroids so copying here isn’t a critical point to be optimized.

The only Thing we can optimize is the first copy of datapoints to the cuda memory at the very first beginning of the algorithm



Then we got an idea ?!

We will always perform first iteration in the loop of Centroids update and assignment then steam copying of the data points with the execution of K1 (Overlap data engine with kernel engine)



Then continue as previous cases with the loop of iterations in which we don’t have copy for the data points, they reside in the cuda memory reused between iterations.

### Kernel (3):

After Finishing our optimization for K1 & K2 we need a metric to describe the clustering, We will use silhouette score to describe efficiency of clusters.

But silhouette is very expensive.

Computing Silhouette using CPU version of our code took 30 mins on Colab for image 640x452.

For 4K image using this method to compute silhouette 🡺 We don’t have time to keep wanting beside this lazy ideal algorithm .😑

We could reach 1.8seconds for clustering image to 5 clusters + computing silhouette on Colab

A screenshot of a computer

Description automatically generated

😲 We are good PC engineers :D

Instead of having the CPU compute inter/intra distances between each point and all other points O(2N^2) parallelize this part by making each thread be reusable for 1 data compute and keeps computing silhouette score of it with other data points.

Then TH0 within each block compute summation of scores for all threads the only in CPU we just add partial summations compute by each block and divide by N ( to get silhouette score for all points) .

# **Performance Analysis**

* **Theoretical benchmarks**

Key Factors Affecting Performance:

* Dataset Size:

Number of data points (n).

Number of features (d).

* Number of Clusters (k):

More clusters typically increase computational complexity.

* Number of Iterations (T):

More iterations lead to higher accuracy but increase computation time.

* Implementation:

Different libraries and optimizations can significantly affect performance.

CPU Benchmarks:

Intel Xeon v4 (18 cores, 2.30 GHz)

Dataset: 1,000,000 data points, 100 features

5 Clusters: ~12 seconds

15 Clusters: ~36 seconds

GPU Benchmarks:

NVIDIA Tesla T4 GPU

Dataset: 1,000,000 data points, 100 features

5 Clusters: ~2 seconds

15 Clusters: ~6 seconds

* **Theoretical speedup**

Factors to Consider:

* Parallelism: GPUs are highly parallel, with thousands of cores designed for parallel execution of tasks, whereas CPUs have fewer cores optimized for sequential processing.
* Memory Bandwidth: GPUs typically have higher memory bandwidth, which can accelerate operations involving large datasets like image processing.
* Floating Point Operations per Second (FLOPS): GPUs generally have much higher FLOPS compared to CPUs.

For NVIDIA Tesla T4 GPU and Intel Xeon E5-2686 v4 (Used on Google Colab)

*NVIDIA Tesla T4 GPU*

* CUDA Cores: 2560
* Single Precision Performance: ~8.1 TFLOPS
* Memory Bandwidth: 320 GB/s
* Memory: 16 GB GDDR6

*Google Colab's CPU (Typical Intel Xeon)*

* Cores: 2 to 4 cores (Intel Xeon E5-2686 v4)
* Single Precision Performance: ~0.5 TFLOPS (for the entire CPU, considering all cores)
* Memory Bandwidth: ~68 GB/s (varies depending on the specific CPU model and configuration)
* Memory: Typically 13 GB RAM available for Colab free tier

Single Precision Performance (FLOPS):

* Tesla T4 GPU: ~8.1 TFLOPS
* Google Colab CPU: ~0.5 TFLOPS
* The Tesla T4 GPU provides significantly higher computational power, approximately 16 times more than the CPU in terms of FLOPS.

Parallelism:

* Tesla T4 GPU: 2560 CUDA cores allow for massive parallel processing, ideal for tasks like K-means clustering.
* Google Colab CPU: 2 to 4 cores, which are optimized for sequential and some parallel tasks but not to the extent of a GPU.

Memory Bandwidth:

* Tesla T4 GPU: 320 GB/s
* Google Colab CPU: ~68 GB/s
* The GPU has much higher memory bandwidth, allowing for faster data transfer between memory and processing cores, which is crucial for data-intensive tasks.

Memory:

* Tesla T4 GPU: 16 GB GDDR6, optimized for high throughput.
* Google Colab CPU: Typically 13 GB RAM available, shared with the system and other processes.

Theoretical Performance: The Tesla T4 GPU is approximately **16** times faster in terms of FLOPS compared to the Google Colab CPU.

* **Calculated speedup**

For the same kernel (gpu\_3\_1) and number of clusters (5) for different images, we’ll calculate the speedup as the average of them all

Which is roughly *15* times faster (within the range theoretical range)

* **Room for improvement**

To improve the speedup, methods like tiling and streams can be used to achieve higher GPU utilization.

The calculated speedup for the kernel that uses streams is *26* times faster

* **Peers comparison**

|  |  |  |  |
| --- | --- | --- | --- |
| CPU | Input Size | #Clusters | Average Total Time |
| PyTorch | 3200\*5600\*3 | 5 | 30.868s |
|  |  | 15 | 129.135s |
|  | 512\*512\*3 | 5 | 0.0457s |
|  |  | 15 | 0.0686s |
| Our Kernel | 3200\*5600\*3 | 5 | 40.5s |
|  |  | 15 | 415.5s |
|  | 512\*512\*3 | 5 | 0.156s |
|  |  | 15 | 0.53s |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **GPU** |  | **Input Size** | **#Clusters** | **Average Total Time** |
| PyTorch |  | 3200\*5600\*3 | 5 | 3.079s |
|  |  |  | 15 | 3.628s |
|  |  | 512\*512\*3 | 5 | 0.011s |
|  |  |  | 15 | 0.018s |
| Our Kernel | gpu\_3\_2 | 3200\*5600\*3 | 5 | 0.819s |
|  |  |  | 15 | 3.24s |
|  |  | 512\*512\*3 | 5 | 0.0205s |
|  |  |  | 15 | 0.071s |
|  | gpu\_3\_1 | 3200\*5600\*3 | 5 | 1.544s |
|  |  |  | 15 | 4.847s |
|  |  | 512\*512\*3 | 5 | 0.0205s |
|  |  |  | 15 | 0.071s |
|  | gpu\_3\_stream | 3200\*5600\*3 | 5 | 0.793s |
|  |  |  | 15 | 2.556s |
|  |  | 512\*512\*3 | 5 | 0.0408s |
|  |  |  | 15 | 0.0709s |

# Unsuccessful Trials:

* Trying to Parallelize Execution of inter and intra distance computation [gpu\_3\_stream\_0\_sihouette\_3.cu]
* Tiling in K3 Worse Performance
* Reduction Sum for k3