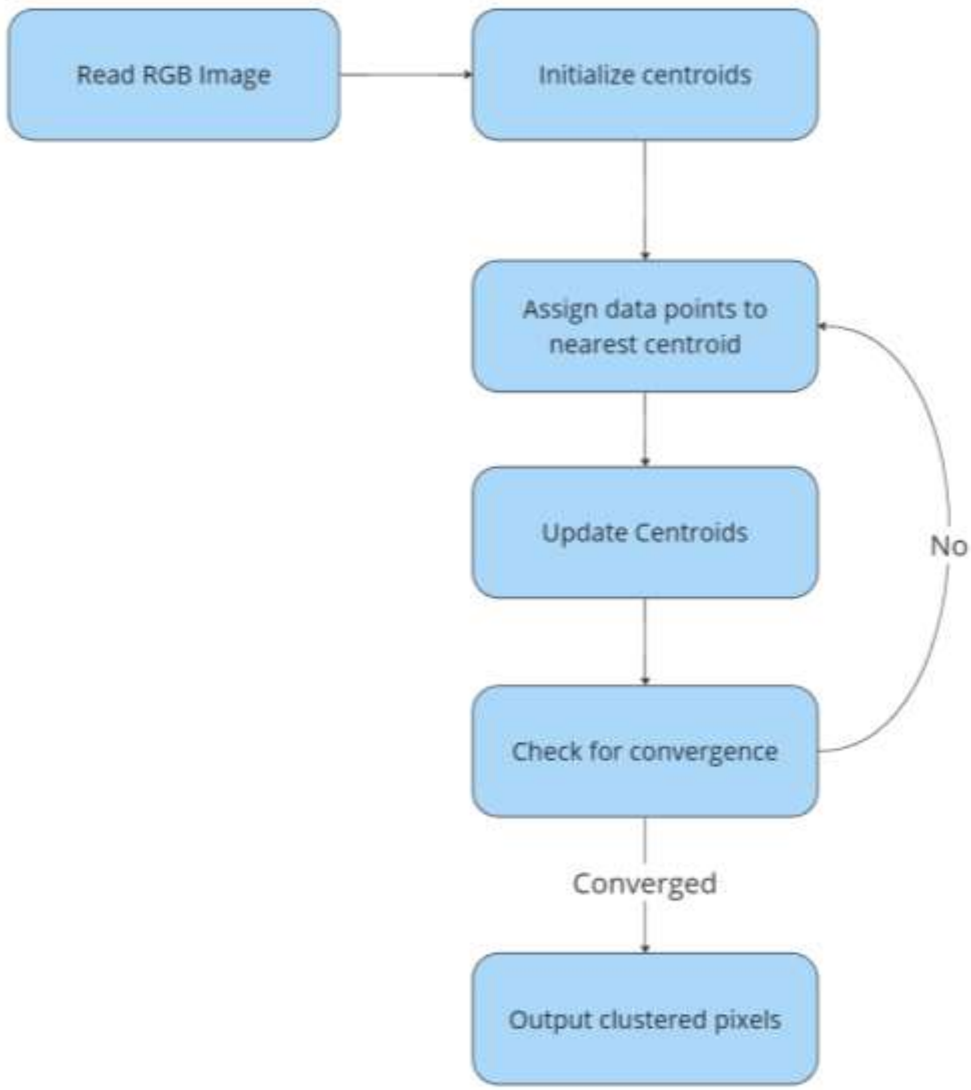
CMP4005 Parallel Computing
(Big Assignment)

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:

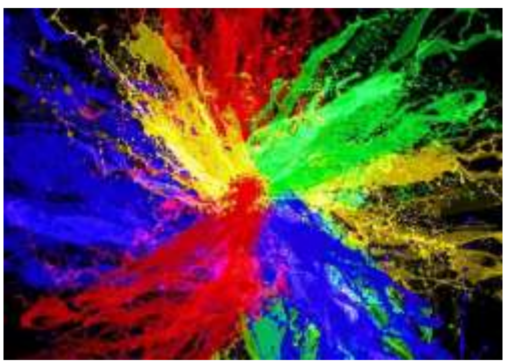
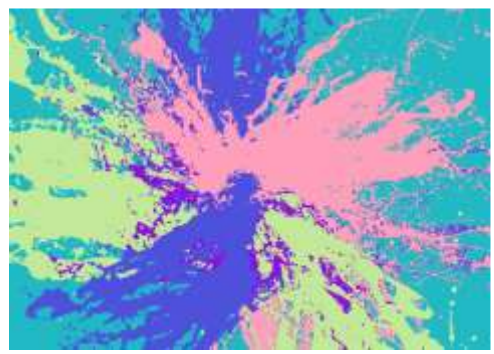


Read RGB Image
Initialize centroids
Assign data points to nearest centroid
Update Centroids
No
Check for convergence
Converged
Output clustered pixels

Development Steps: baby steps move mountains 😀

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


2. Extend to be 3 Channel [cpu_3.cu]

3. Begin with Basic GPU parallel Implementation [gpu_3.cu]
4. Enhancement in Kernel 1 (Assign Data Points to the nearest Centroid) [gpu_3_1.cu]
5. Enhancement in Kernel 2 (Compute New Centroids [gpu_3_2.cu] (VERY EFFECTIVE)
6. Streaming for Kernel 1 (Assign Data Points to the nearest Centroid) gpu_3_stream_0.cu]
7. Trying to Compute Silhouette Score for clustered Images (CPU/GPU)

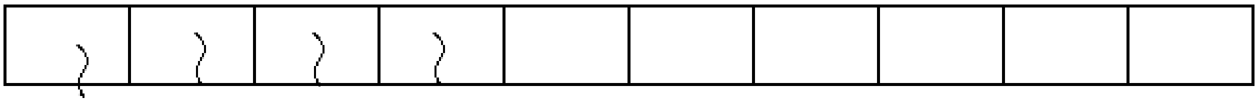
```
Computing Shilloute Score ....
Shetollute Score: 0.622491
Time taken: 4.013000 sec
Converged after 20 iterations
```

Implementation:

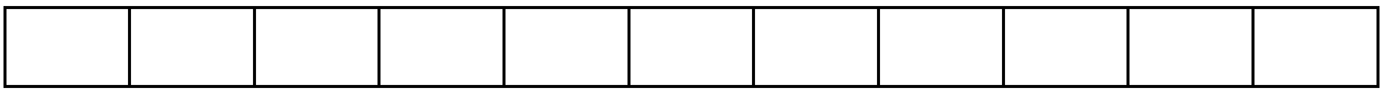
Kernel (1):

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

d_data_points



d_clutser_assigment

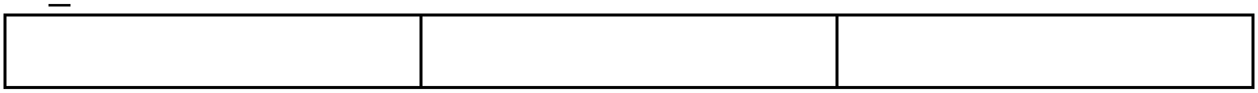


d_centroids

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 😀

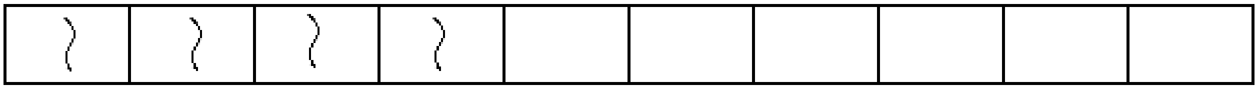
Total Time CPU+GPU	1.436197				1.540999			
	%	Total Time	# Calls	Average Time	%	Total Time	# Calls	Average Time
update_cluster_centroids	88.89%	1.31671s	37	35.587ms	88.61%	1.32704s	37	35.868ms
assign_data_points_to_centroids	4.53%	67.174ms	37	1.8155ms	4.90%	73.316ms	37	1.9815ms
CUDA memcpy HtoD	3.26%	48.271ms	38	1.2703ms	3.11%	46.545ms	38	[illegible]
CUDA memcpy DtoH	3.32%	49.162ms	75	655.49us	3.39%	50.725ms	75	676.34us
CUDA memset	0.00%	16.415us	37	443ns	0.00%	17.121us	37	462ns

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

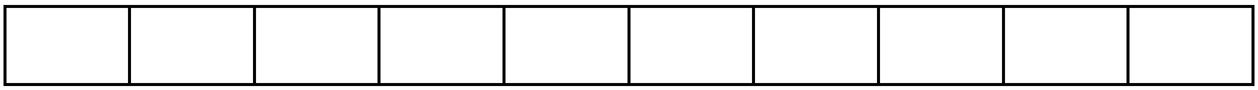
Kernel (2):

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

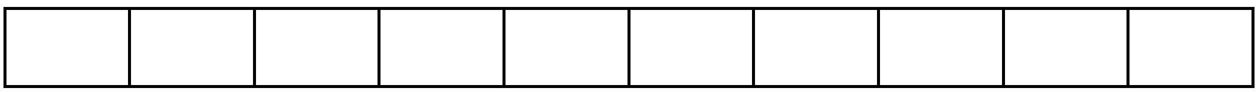
d_data_points



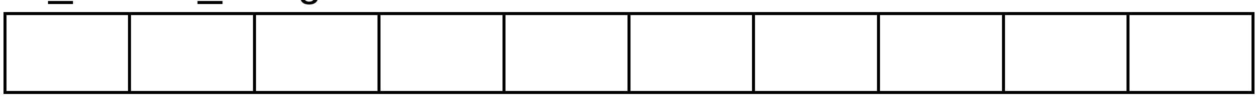
sh_data_points



d_clutser_assigment

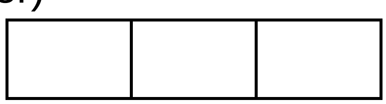


sh_clutser_assigment

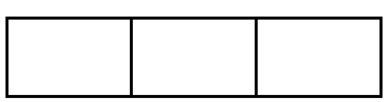


Each Thread loads a data point to the shared Memory + The corresponding cluster index to the shared memory.
Then we need to wait till all threads finish loading <3

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



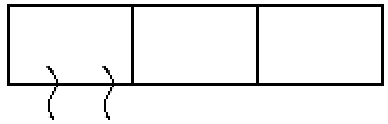
Cluster_sizes (per cluster)



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

d_centroids

d_cluster_sizes (per cluster)



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

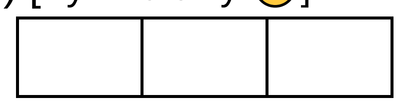
Instead of having all thread have race condition on adding centroids and cluster sizes to global memory d_cluster_sizes & d_centroids they will have race condition within their block in the shared Memory (Near and faster to access) and only the first thread in each block only try to write the global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

Sh_data_points_sum

Sh_cluster_sizes

d_centroids   [By Th0 only 😊]

d_cluster_sizes (per cluster) [By Th0 only 😊]



Version	GPU version(1)				GPU version(2)			
File	gpu_3_1.cu				gpu_3_2.cu			
Optimization	Added Shared Memory Centroids for K1				All Thraeds In Kernel(2) Compute sum instead of only thread 1 Only First K threads in the block write to the global Memory(d_centroids & d_cluster_sizes)			
Converge after	37				37			
Total Time CPU+GPU	1.540999				0.836880s			
	%	Total Time	# Calls	Average Time	%	Total Time	# Calls	Average Time
update_cluster_centroids	88.61%	1.32704s	37	35.866ms	78.77%	646.35ms	37	17.469ms
assign_data_points_to_centroids	4.90%	73.316ms	37	1.9815ms	9.15%	75.057ms	37	2.0286ms
CUDA memcpy HtoD	3.11%	46.545ms	38	1.2249ms	5.62%	46.126ms	38	1.2138ms
CUDA memcpy DtoH	3.39%	50.725ms	75	676.34us	6.46%	52.999ms	75	706.66us
CUDA memset	0.00%	17.121us	37	462ns	0.00%	35.104us	74	474ns

All Threads Compute

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

```
// Copy data from host to devic [image]
cudaMemcpy(d_image, image, N * D * sizeof(float), cudaMemcpyHostToDevice);
```

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)

```
// Streams loop
for (int s = 0; s < numSegments; s++)
{
    int start = s * segmentSize;
    int end = min(start + segmentSize, N); // min to handle the last segment
    int Nsegment = end - start;

    cudaMemcpyAsync(d_image + start * D, image + start * D, Nsegment * D * sizeof(float), cudaMemcpyHostToDevice, streams[s]);

    // call the kernel [assign_data_points_to_centroids]
    assign_data_points_to_centroids<<<num_blocks, THREADS_PER_BLOCK, K * D * sizeof(float), streams[s]>>>(Nsegment, D, K, d_image + start * D, d_centroids, d_cluster_assignment + start);
}
```

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

```
Computing Shilloute Score ....
Shetollute Score: 0.542568
Time taken: 1.876532 sec
Converged after 36 iterations
Image saved successfully at: ./tests/image_3_outpu
```

😲 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 k different images, we'll calculate the speedup as the average of them all

$$\frac{1}{k}\sum_{1}^{k} \frac{CPU_k}{GPU_k}$$

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