Parallel K-Means Clustering for Image Colour Quantization

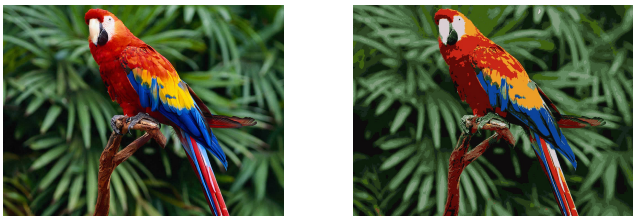
Mandikal Vikram, Aditya Anantharaman, Suhas B S, Rakshith Gopalakrishna
Department of Information Technology
National Institute of Technology Karnataka
Surathkal, Mangaluru 575025
Email: 15it217.vikram@nitk.edu.in,15it201.aditya.a@nitk.edu.in
15it110.suhas@nitk.edu.in, 15it134.rakshith@nitk.edu.in

***Abstract*—Image colour quantization is an important problem in Computer Graphics and Image Processing. This reduces the number of colours in the image and facilitates the display, storage, and transmission of images. This is a necessary prepossessing to make images compatible low-end hardware device which can display a limited number of devices. This work shows how this quantization can be achieved by the means of k-Means clustering. The algorithm is implemented in parallel for efficient execution and we provide a performance comparison of various platforms such as OpenMP, CUDA, and MPI. We find that CUDA outperforms other platforms by a large margin due to fine-grained parallelism and a higher degree of concurrency.**

***Keywords*—Colour quantization, K-means clustering, Parallel Programming, OpenMP, CUDA, MPI**

I. Introduction

Colour quantization refers to reducing the number of distinct colours in an image while trying to maintain the visual appearance of the image as shown in figure 1. It finds its application in many tasks in graphics and image processing such as compression[1], segmentation[2], colour-texture analysis[3] and watermarking[4]. Hence, it is important to achieve this in an efficient manner for which we use parallel programming techniques. The proposed algorithm is independent of the image and does not require any image specific information such as the variance-based method[5], binary splitting[6] and greedy orthogonal bipartitioning[7].



(a) Original Image (b) Colour Quantized Image

Fig. 1: Original and colour quantized images

Each pixel in an image stores three values - its respective red, blue and green intensities. When we quantize the colours i.e. limit the colours to a few fixed values, each pixel will have to only store one value which indicates which quantum it belongs to. The common colour intensities of each quantum is separately stored which will require minimal space compared to storing the intensities of every pixel.

Our method identifies these quanta by clustering pixels with similar colour intensities. The similarity between two pixels is measured by using the euclidean distance between the RGB intensities. The clustering is done using the K-Means algorithm which ensures that there is maximum intra-cluster similarity and maximum inter-cluster dissimilarity. The mean RGB intensities of each of these clusters are stored and each pixel in the quantized image belongs to one of the cluster and will display the mean intensities of this cluster which is stored.

In this work we explore various parallel programming techniques and platforms to achieve a higher efficiency. This algorithm is highly suitable for parallel implementation as each pixel can be independently be mapped to its cluster center. This mapping can be theoretically performed simultaneously for all the pixels but it is limited by the number of processors for OpenMP and MPI. Larger number of cores in the GPU allows CUDA to achieve a higher degree of concurrency. Highlights of this work includes an average compression factor of 2.2 and an average speedup of 1500 for CUDA over serial implemntation.

The paper is further organized as follows : II Related Work, III Proposed Approach, IV Results and Discussion, V Conclusion and VI References.

II. Related Work

This section describes briefly two important works from which the proposed approach draws inspiration.

The first work is a work on k-means algorithm for colour quantization by M. Emre Celebi[8]. This work explores the k-means algorithm from the colour quantization perspective. It describes various experiments to show that the k-means algorithm can be used for effective colour quantization. The work also mentions that the k-means clustering is easy to implement and is fast when applied to image colour quantization.

The second work is based on parallelising the k-means algorithm by Reza Farivar et al[9]. This work shows that

the k-means algorithm is suitable for parallel implementation and can be easily accelerated by using larger number of processors. They show that with p cores, the first phase of k-means clustering which includes mapping of the data points to the cluster centers can be accelerated to $\Theta(nk/p)$ from $\Theta(nk)$ where n is the number of data points and k is the number of clusters. This is because mapping of one data point is independent of other data points and can be executed in parallel. If the number of processors equal the number of data points this phase can be completed in a single phase.

We incorporate the parallel k-means algorithm for colour quantization as the number of data points i.e. the number of pixels is large for an image and hence we can obtain a large speedup if we parallelise the algorithm. We also parallelise the second phase of the k-means clustering which was not been done by [9] . This phase involves re-calculating the cluster centers to an extent, however, this phase cannot be fully parallelised due to data dependency.

III. Proposed ALgorithm

First, we describe the sequential k-means algorithm and present its analysis before moving to the parallel implementations.

A. Sequential K-Means Algorithm

The k-means algorithm as described in [10] assigns a given set of data points to k clusters. In the update step, a given data point is assigned to its closest cluster center (μ) (see equation 1). In case more than one cluster center has the same distance to a data point, the cluster center is chosen randomly.

$$S_i^{(t)} = \left\{x_p : \left\|x_p - \mu_i^{(t)}\right\|^2 \leq \left\|x_p - \mu_j^{(t)}\right\|^2 \forall j, 1 \leq j \leq k\right\} \quad (1)$$

This is followed by reinitializing the cluster centers to the mean of the data points assigned to each cluster (see equation 2).

$$\mu_i^{(t+1)} = \frac{1}{|S_i^{(t)}|} \sum_{x_j \in S_i^{(t)}} x_j \quad (2)$$

This process is repeated until there is no change in cluster centers. The k-means algorithm tries to optimize the objective function 3.

$$J = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} ||x_n - \mu_k||^2 \quad (3)$$

$$\text{with } r_{nk} = \begin{cases} 1 & x_n \in S_k \\ 0 & \text{otherwise} \end{cases}$$

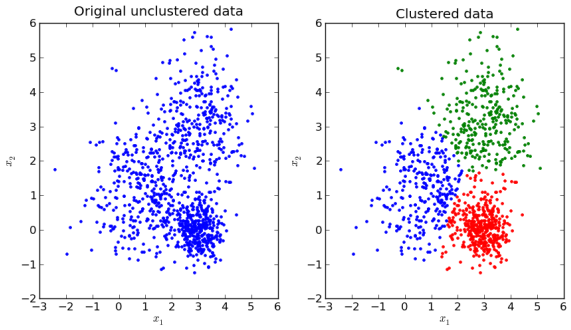


Fig. 2: Clusters formed by kmeans algorithm

Algorithm 1 Sequential kmeans algorithm

Input $X = \{x_1, x_2, x_3 ... x_N\}$ (N Data Points), k (Number of Clusters)
Output $C = \{c_1, c_2, c_3 ... c_K\}$ (k Clusters)

Select initial set of cluster centers as a random subset C of X

while *cluster centers donot change* **do**
  Phase 1: Map points to cluster centers
  for $j = 1; j <= N; j = j + 1$ **do**
    $m[j] = argmin_{k \in \{1,2..k\}}\ dis(x_j, c_k)$
  end
  Phase 2: Recalculate the cluster centers
  for $j = 1; j <= N; j = j + 1$ **do**
    $c_k = \frac{1}{|S_k|} \sum_{x_j \in S_k} x_j$
  end
end

The sequential k-means algorithm is described by Algorithm 1. The formations of cluster by this algorithm can be visualized in Figure 2. In the context of colour quantization, the data points are the individual pixels which store the values of their RGB intensities. The number of clusters k equals the number of colours desired in the colour quantized image. The value $dis(x_i, c_k)$ in Algorithm 1, is calculated using the euclidean distance of the colour intensities as given by equation 4 where r_{xi} , g_{xi}, b_{xi} refer to the red, green and blue intensities of the pixel and r_{cj} , g_{cj}, b_{cj} refer to the intensities of the cluster center. Random pixels from the image are initially set as the cluster centers.

$$dis(x_i, c_j) = \sqrt{(r_{xi} - r_{cj})^2 + (g_{xi} - g_{cj})^2 + (b_{xi} - b_{cj})^2} \quad (4)$$

B. Parallelising the algorithm

We analyze the first phase and second phase in Algorithm 1 separately to see how they can be parallelised. These phases

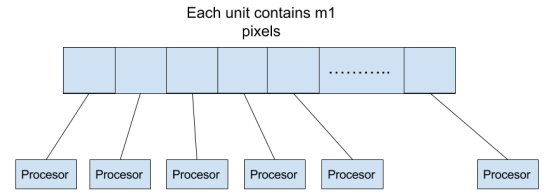
are iterated repeatedly in the algorithm. Also, phase 1 can occur only after phase 2 has occurred in the previous iteration, and, phase 2 can occur only after phase 1 has been completed in the current iteration. Hence, it is not possible to parallelise the outermost loop in Algorithm 1.

1) Phase 1:

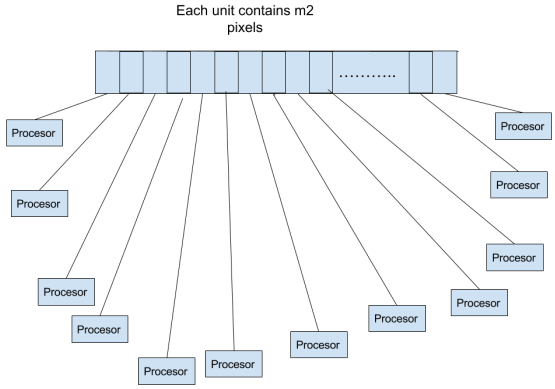
- Input - data points and cluster centers.
- Output - mapping of data points to cluster centers.
- The mapping of a given pixel to one of the cluster centers is independent of other pixels. It only depends on its own pixel intensities and that of the cluster centers.
- The cluster centers remain constant in this phase and hence the output is naturally a function of the input without any dependencies. Hence, **input data decomposition** is possible.
- Each thread can be assigned a group of pixels, for which it will calculate the mapping.
- A more granular decomposition will increase the speedup of this phase. If the number of processors permit to assign one thread to each pixel, this phase can be completed in a single step.
- Serial complexity of this phase : $\Theta(nk)$, where n is the number of pixels and k is the number of cluster centers.
- With p processors, the complexity will be reduced to $\frac{(nk)}{p}$.

2) Phase 2:

- Input - Cluster labels for each pixel and individual pixel intensities.
- Output - New cluster centers.
- The new cluster centers are calculated as the mean of the clusters.
- Since, the calculation of mean of one cluster depends on all the points in that cluster, data dependency exists.
- Thus, it is not possible to completely parallelise this phase as in phase 1.
- We parallelise this phase in two different ways:
    - Calculate the mean for each cluster using reduction i.e. in parallel. This, is done in a sequential way for each cluster. This technique is used in our OpenMP implementation.
    - Calculate the mean for all clusters in parallel, but for a given cluster the calculation is done sequentially. This is done in our CUDA implementation.



(a) Lower Granularity of Decomposition



(b) Higher Granularity of Decomposition

Fig. 3: This shows the decomposition of the pixels to processors in Phase 1. Here, $m1 > m2$, hence, sub-figure (a) has a coarser granularity compared to sub-figure (b). Since, the output mapping is dependent only on the input pixel, a finer granularity increases the speed of this phase.

C. OpenMP

Here, phase 1 is parallelised by using a parallel for loop with static scheduling. Each thread receives p/t pixels where p is the number of pixels and t is the number of threads. Each thread then calculates the nearest cluster for all the pixels assigned to it. This assignment is done in a critical section to form a map from the cluster number to the pixels assigned to the cluster, this enables the calculation of cluster mean in phase 2 to be done using reduction. Mean calculation for a particular cluster is done in parallel, which is repeated sequentially for all clusters.

D. MPI

Here, the master process initially assigns p/t pixels to each processor where p is the number of pixels and t is the number of processors. The slave processors only have local information about the pixels assigned to them and do not have any global information, hence the master processor coordinates with the slave processors. Each slave processor calculates the nearest cluster for all the pixels assigned to it in phase 1 and sends the sum and count of pixels assigned to each cluster to the master processor. The master processor upon receiving all sums and counts from the slave processors calculates the new

cluster centroids which completes phase 2. These recalculated centroids are broadcasted back to the slave processors.

E. CUDA

There are specific kernels designated to both phase 1 and phase 2. In phase 1, a thread is assigned to each pixel which simply assigns the nearest cluster to the pixel. Larger number of cores in the GPU felicitates finer granularity in phase 1, thus reducing the overall time complexity and accelerating the performance. In phase 2, the mean calculation for all clusters takes place in parallel,but for a given cluster it is done sequentially.

F. Parallel Models Used

- Data parallel model - In phase 1, each thread works on a subsection of the image and calculates nearest centroid in parallel. This model is used in OpenMP and CUDA implementations.
- Master Slave model - In MPI, the master processor assigns the task of mapping pixels to clusters and the slave processors complete this task and return the sum and count values for each cluster back to the master processor.

IV. Results and Discussion

All experiments were conducted on a system with 56 Intel Xeon CPU E52680 cores and Nvidia Tesla M40 GPU with 3072 Nvidia CUDA cores and 24 GB of GDDR5 video memory.

The details of the images used for conducting our experiments is summarized in table I. The time taken on different platforms for these images are compiled in table II

TABLE I: Image Details

Image	Dimensions	Number of Colours
Parrot	1024 × 768	153063
Bird	730 × 487	107257
Flower	960 × 563	123137
Colosseo	1000 × 1000	231382
Bike	1024 × 256	99272

TABLE II: Time taken in seconds for different images with k=16

Platform	Parrot	Bird	Flower	Colosseo	Bike
CUDA	0.129	0.131	0.06	0.181	0.209
MPI	4.781724	2.694147	3.465208	6.45931	4.140578
OpenMP	53.0696	57.8544	19.1501	53.7564	64.7001
Sequential	216.988	227.768	85.4472	250.747	300.264

The above table suggests that CUDA performs exceedingly well on all the images. MPI performs better than OpenMP whereas both MPI and OpenMP are much faster than the sequential implementation. The speedup for CUDA over sequential was 1522.83, it was 64.14 for MPI and 5.55 for

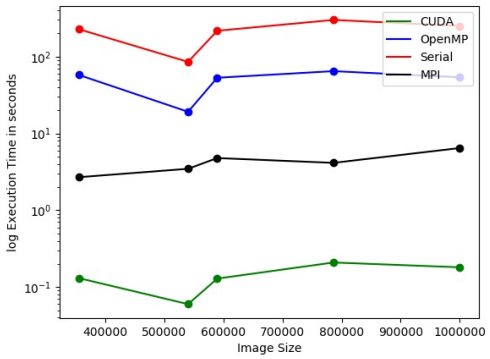


Fig. 4: Time Comparison on different platforms

OpenMP. These times are plotted in figure 4, where the y-axis is in logarithmic scale due to the large difference in the range of values for CUDA platform and serial execution. The x-axis represents the number of pixels in the different images. As seen in the graph, increase in the number of pixels does not necessary lead to increase in time taken, this is because early convergence of the k-means algorithm will lead to a lesser time. For example, a larger image with very few colours will converge in a lesser number of iterations and hence may take lesser time when compared to a smaller image with a large number of colours which does not converge easily. The number of iterations depends on the convergence of the algorithm, this cannot be improved by parallel programming - only the computations which occur in each iteration can be made more efficient by executing them in parallel. As seen in table II, although 'bird' is a smaller image than 'parrot', it takes more time because the distinct colours in the parrot image allow it converge in a few iterations. We can see that this trend where the larger 'parrot' image takes converges faster than the smaller 'bird' image is consistent in most of the platforms.

As seen in figure 5, when we increase the number of clusters, the quality of the image also improves. There is very minimal loss of quality by reducing the image with 231382 colours to just 256 colours. This concept is used to make images compatible with devices which can display only a few limited colours.

Image Compression

Image compression is an application of colour quantization. In a standard image, each pixel stores its own RGB intensities. Since, each colour intensity can take a value between 0-256, the intensity would occupy $8 \times 3 = 24$ bits. Thus, an image with dimensions $H \times W$ would occupy $H \times W \times 24$ bits. When the colours are limited to 256 values, a map data structure is used to store the intensity values of the 256 colours - the key is the colour number and the value is the RGB intensities. This dictionary would occupy 256×24 bits. In the image, at each pixel only the colour number is stored which will occupy 8 bits (since there are 256 colour numbers). Thus, the new size of the compressed image would be $(H \times W \times 8) + 256 \times 24$ bits which is much lesser when compared to the initial $H \times W \times 24$ bits.

(a) 4 colours

(b) 8 colours

(c) 16 colours

(d) 32 colours

(e) 64 colours

(f) 128 colours

(g) 256 colours

(h) Original Image with 231382 colours

Fig. 5: Images obtained by changing the number of clusters i.e k

This compression factor is formalised by equation 5 where K is the number of clusters.

$$compressionfactor = \frac{H \times W \times 24}{(H \times W \times \log_2 K) + K \times 24} \quad (5)$$

Thus, we expect a theoretical compression of a factor of 6 for an image of dimension 750×750 with 16 colours. However, when we serialized C++ data structures using the boost package we obtained an average compression factor of **2.2**.

A. *Comparison of the three platforms*

A detailed comparison of the time taken in various parallel programming platforms is recorded in table III. The same is visualized in figure 6. It is evident that CUDA outperforms other platforms by a large margin, the reasons contributing to this are as follows :

- A much larger number of cores on the GPU (3072) compared to the CPU (56) allows a higher degree of concurrency.
- No communication overhead which is present in MPI.

TABLE III: Detailed Comparison of Various Parallel Programming Platforms
Time taken in seconds

K	Platform	Parrot	Bird	Flower	Colosseo	Bike	Average
4	CUDA	0.046879	0.037211	0.024972	0.06177	0.013609	0.0368882
4	OpenMP	19.6825	12.9806	12.0248	25.0436	6.5882	15.26394
4	MPI	4.352257	1.580914	1.897685	3.284286	2.353469	2.6937222
8	CUDA	0.047922	0.064404	0.031161	0.117236	0.107063	0.0735572
8	OpenMP	18.5444	22.2507	11.1587	34.9771	36.5382	24.69382
8	MPI	4.015927	1.870726	3.573816	4.936909	3.456555	3.5707866
16	CUDA	0.129847	0.13187	0.062558	0.203582	0.219095	0.1493904
16	OpenMP	53.0696	57.8544	19.1501	19.1501	53.7564	64.7001
16	MPI	4.781724	2.694147	3.465208	6.45931	4.140578	4.3081934
32	CUDA	0.277212	0.18513	0.306853	0.232024	0.186148	0.2374734
32	OpenMP	107.008	52.9384	95.001	65.3133	105.526	85.15734
32	MPI	8.403226	3.196628	5.164831	8.678829	5.876272	6.2639572

Each thread performs the computation in Phase 1 for a designated pixel.

- The means of various clusters can be calculated in parallel.

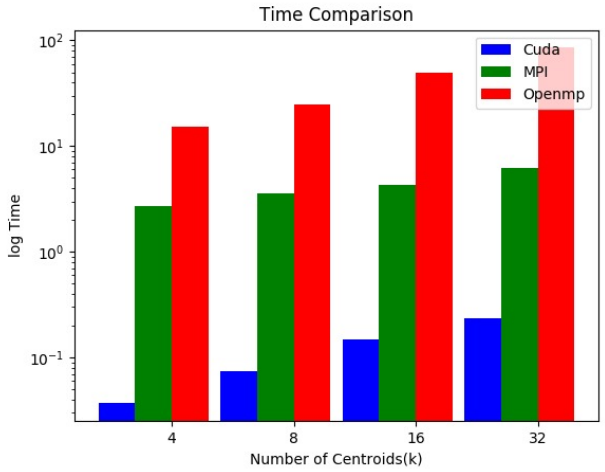


Fig. 6: Time plotted against Number of Centroids

V. Conclusion

In this work we show how clustering can be used as a means for colour quantization. Since the number of pixels in any image is large, the clustering should be efficient and hence we explore parallel techniques for achieving that. We describe the parallel model used and present comparison of performance using three different platforms - CUDA, MPI and OpenMp. We also describe how this colour quantization technique can be used for image compression. We show that CUDA outperforms other platforms by a large margin due to a higher degree of concurrency.

References

[1] C.-K. Yang and W.-H. Tsai, "Color image compression using quantization, thresholding, and edge detection techniques all based on the moment-preserving principle," *Pattern Recognition Letters*, vol. 19, no. 2, pp. 205–215, 1998.

[2] Y. Deng and B. Manjunath, "Unsupervised segmentation of color-texture regions in images and video," *IEEE transactions on pattern analysis and machine intelligence*, vol. 23, no. 8, pp. 800–810, 2001.

[3] O. Sertel, J. Kong, U. V. Catalyurek, G. Lozanski, J. H. Saltz, and M. N. Gurcan, "Histopathological image analysis using model-based intermediate representations and color texture: Follicular lymphoma grading," *Journal of Signal Processing Systems*, vol. 55, no. 1-3, p. 169, 2009.

[4] C.-T. Kuo and S.-C. Cheng, "Fusion of color edge detection and color quantization for color image watermarking using principal axes analysis," *Pattern Recognition*, vol. 40, no. 12, pp. 3691–3704, 2007.

[5] S. Wan, P. Prusinkiewicz, and S. Wong, "Variance-based color image quantization for frame buffer display," *Color Research & Application*, vol. 15, no. 1, pp. 52–58, 1990.

[6] M. T. Orchard and C. A. Bouman, "Color quantization of images," *IEEE transactions on signal processing*, vol. 39, no. 12, pp. 2677–2690, 1991.

[7] X. Wu, "Efficient statistical computations for optimal color quantization," *Graphics gems*, vol. 2, pp. 126–133, 1991.

[8] M. E. Celebi, "Effective initialization of k-means for color quantization," in *Image Processing (ICIP), 2009 16th IEEE International Conference on*. IEEE, 2009, pp. 1649–1652.

[9] R. Farivar, D. Rebolledo, E. Chan, and R. H. Campbell, "A parallel implementation of k-means clustering on gpus." in *Pdpta*, vol. 13, no. 2, 2008, pp. 212–312.

[10] J. MacQueen *et al.*, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, vol. 1, no. 14. Oakland, CA, USA., 1967, pp. 281–297.