CuNeuQuant: A CUDA Implementation of the NeuQuant Image Quantization Algorithm

David Bottisti[1], Liuva Mendez[1], and Damian Dechev[1,2]
[1]Department of Computer Science, University of Central Florida, Orlando, FL, USA
[2]Scalable and Secure Systems R&D Department, Sandia National Laboratories, Livermore, CA, USA

Abstract— *Color quantization is an often performed pre-step in many image processing and computer vision applications. Quantization is defined as the process of selecting a palette of representative colors P which can replace the original colors C in an image such that $|P| \ll |C|$ and the perceptual distortion of the reduced color image is minimized. It is well known that the quantization process is an NP-complete problem and as such, many competing heuristic algorithms exist. One high-quality quantization algorithm is NeuQuant due to Dekker. In this paper, we describe a GPU based parallel implementation of the NeuQuant algorithm. Our GPU-based approach demonstrated a speedup by a factor of 5 or more in the performance evaluation we have performed. The details of the NeuQuant algorithm present unique difficulties to implementing a parallel version due to the sequential dependencies present when training the underlying neural network.*

Keywords: Kohonen Neural Network, CUDA, Image Quantization

1. Introduction

Natural images often contain several hundred thousand colors in order to represent their subjects. While necessary to best preserve visual details for human observers, these additional colors can make computer processing of an image very difficult. A large number of colors can also increase the storage requirements for an image. For these reasons a frequently preformed pre-processing step on images is to reduce the number of unique colors to a more manageable set. This process is called color quantization [1].

Consider an original image whose colors are represented as a set C. We denote the carnality of a set using the notation $|\cdot|$. Thus, the number of original colors in the image is $|C|$. The goal of quantization is to find an optimal palette P such that $|P| \ll |C|$. Here optimal refers to minimizing some perceptual error between the original image and the resulting quantized image. Common error measurements include root mean squared error (RMSE) and signal to noise ratio (PSNR)[2].

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

There is a wide variety of color quantizations algorithms available in the literature today, each with their own strengths and weaknesses (see [1] for an overview of common color quantization algorithms). One such algorithm is the NeuQuant algorithm which uses an advanced neural network training step to produce high-quality quantization at the expense of algorithmic complexity and speed [3]. Figure 1 compares an original full-color image with a version quantized to 256 colors using the NeuQuant algorithm. Notice that the NeuQuant algorithm creates very little banding or other quantization artifacts. However, one of the downsides of the NeuQuant algorithm is that training the neural network is very slow.

In this paper, we describe a GPU based implementation of the NeuQuant algorithm leveraging the massively parallel stream processor technology available in nVidia's newest graphics processing units (GPUs) [4]. We chose NeuQuant due to its high quality quantization in subjective evaluations as well as the fact that no current GPU implementations are believed to exist. The rest of this paper is organized as follows: Section 2 discusses a small subset of the many image quantization algorithms present in the literature. Following this, Section 3 describes the original NeuQuant algorithm. We then discuss implementation details for implementing NeuQuant using nVidia's GPU development environment, CUDA (Compute Unified Device Architecture), in Section 4. In Section 5 we present our comparative timing and quality results and finally conclude in Section 6.

2. Related Work

As discussed previously, performing high-quality image quantization is very important in image processing and compression. Due to the fact that it is of such high importance and yet NP-hard, there are various proposed heuristic algorithms to perform image quantization. In general, the approach to image quantization is to divide the color space of an image into disjoint regions of similar colors (called clusters). A representative color is then determined from each region forming what is called the colormap (this is a smaller set of colors that is going to be used to represent the image).

The Median Cut algorithm [5] repeatedly divides the color space into rectangular boxes until the desired number of

(a) Original, Full-Color Image

(b) Image Quantized to 256 Colors using NeuQuant

Fig. 1: NeuQuant Example

colors is obtained. The algorithm involves the following three steps (while using an RGB color space):

1) Sampling the three original (red, green, and blue) colors for color statistics in order to create a color frequency histogram;
2) Choosing a new color space based on the gathered statistics (by successively subdividing the color space so that each of the new colors is represented by an equal number of pixels from the original image);
3) Mapping the original colors to the nearest neighbors in the new color space.

Median-cut is very simple and easy to implement. The results from running this algorithm show that the time it takes to produce a colormap scales almost linearly with the image size and logarithmically with the colormap size. However, due to the fact that each color space is divided equally, some important but infrequently occurring colors may fail to be represented in the colormap (i.e., a red ladybug against a swath of green leaves.)

The K-means algorithm is classified as a partitional clustering algorithm since it assigns a set of objects into K pre-specified clusters [6], [7]. The K-means algorithm seeks an optimal partition of the data by minimizing the sum-of-squared-error criterion with an iterative optimization procedure [8]. It starts with an initial colormap (initialized to K cluster centers), then each color pixel is assigned to the closest color in the colormap (or cluster centers). The colors in the colormap are recomputed as the cluster centers of the resulting clusters. This process is then repeated until convergence. The K-means algorithm has been proven to converge to a local optimum [9] (versus a global optimum). The basic steps of K-means are summarized as follows:

1) Choose K initial clusters either randomly or based on some prior knowledge;
2) Assign each object in the data set to the nearest cluster;
3) Recalculate the cluster prototypes based on the current partition;
4) Repeat steps 2-3 until there is no change for each cluster.

The procedure of the K-means algorithm is very simple and straightforward, and it can be easily implemented. The time complexity of K-means is approximately linear, which makes it a good selection for large-scale data applications. The major disadvantages of K-means are dependence on the initial partitions, convergence problems, and sensitivity to noise and outliers. To identify the number of clusters K in advance is also a big challenge.

The Octree quantization algorithm [10] uses an octree hierarchical data structure (a tree structure in which each node has up to eight children). The idea behind the octree algorithm is to sequentially read in the image, then store every color in an octree of depth 8 (where every leaf at depth 8 represents a distinct color). While executing the algorithm, a limit of K (desired number of colors the original image is being reduced to, i.e. $K = 256$) leaves are placed on the tree. The algorithm has two stages; first the insertion of colors in the tree (creating the tree as a representation of the original image), and then merging the nodes until the desired number of colors (colormap) is reached.

Insertion of a color in the tree can result in two outcomes:

1) If there are less than K leaves then the color is filtered down the tree until either it reaches some leaf node that has an associated representative color or it reaches the leaf node representing its unique color.
2) If there are greater than K leaves in the tree some set of leaves in the tree must be merged (their representative colors averaged) together and a new representative color stored in their parent.

Gervautz et al. [10] propose two possible criteria to be used in the selection of leaves to be merged:

1) Reducible nodes that have the largest depth in the tree should be chosen first. They represent colors that lie closest together.
2) If there is more than one group of leaves at the maximum depth the algorithm could:
   a) Merge the leaves that represent the fewest number of pixels. This will help keep the error small.
   b) Reduce the leaves that represent the most pixels. In this case large areas will be uniformly filled in a slightly wrong color while maintaining detailed shadings.

Once the entire image has been processed in this manner, the colormap consists of the representative colors of the leaf nodes in the tree. The index of the colormap is then stored at that leaf, and the process of quantizing the image is done by filtering each color down the tree until a leaf is found. Because a limit is placed on the number of leaves in the tree, this algorithm has a modest memory complexity, $O(K)$, compared to median-cut. Gervautz and Purgathofer [10] cite the search phase as being linear with respect to the number of pixels in the image. While the reduced memory complexity and faster run-time are advantages of the Octree algorithm, it is still susceptible to mis-quantization when infrequent but important colors are present in an image.

The spatial color quantization algorithm (scolorq) [11] combines color quantization with dithering (dithering is a technique used to create the illusion of color depth in images with a limited colormap, in which colors not available in the map are approximated by a diffusion of colored pixels from within the available colormap). Spatial color quantization is a technique used for decreasing the color depth of an image [11]; it combines palette selection and dithering with a simple perceptual model of human vision. The authorsÕ implementation was never made public, and the algorithm is relatively complex, discouraging others from implementing it.

Scolorq uses less memory and is faster compared to median-cut. The scolorq algorithm implemented after the algorithm described in the paper [11] produces richer colors than median-cut and Octree, especially when reducing to very low color depths (4, 8, and 16 colors). The major disadvantage of using the scolorq algorithm is when it is used for large photographs and other large continuous-tone (smooth) images. It should also not be used when the target number of colors is relatively larger, i.e., on the order of 256 or more, since the algorithm becomes extremely slow for large numbers of colors and does not outperform the standard algorithms available.

NeuQuant approaches the problem of color quantization much differently than these statistical methods. Using the original colors of the image, NeuQuant trains a Kohonen Neural Network, also known as a Self-Organizing Map (SOM), to represent the palette of colors. By restricting the network to a size less than the number of input colors, the network represents the quantized color palette. This approach creates a robust color palette that does not suffer from under-representation of important colors in the image. We describe the NeuQuant algorithm more completely in the following section.

3. NeuQuant

We now present a brief introduction to the NeuQuant algorithm which we base on Kohonen's SOM. For a more detailed explanation of the algorithm, and for original implementation details, the reader is directed to [3].

Consider a $3 \times k$ matrix N representing our neural network. In this matrix, each row contains a 3-element vector representing a color which we refer to as a node, and k is the number of desired output colors (usually 256). Originally, the k colors (rows) in the network are initialized to gray values (red = green = blue) equally spaced to span the achromatic axis of a colorspace from black to white.

Network training proceeds as each pixel of the input image is presented to the network. For each training pixel, the L_1 distance between it and each node in the network is computed. The node with the minimum distance is designated the winning node. This winning node and its neighbors are modified so that the network is stretched towards the input pixel by a factor α. The radius of the network, as well as the amount of movement, decay exponentially over time. In [3] the authors provide a detailed explanation of the radius and decay rates.

We describe the order in which the input pixels are presented to the network. Since the network is updated for every input pixel, and these updates affect the decision for a winning node on the next iteration, the order of execution greatly affects the network. For this reason, it is undesirable to train the network in sequential order. For example, consider an image of a landscape with a blue sky and green grass. Sequential training will provide a large number of blue pixels to the network before any green pixels are presented. This biases the network towards blue and, due to the exponential decay of the radius, very little of the network will be used to represent the green grass.

To avoid the shortcomings associated with sequential training while still remaining deterministic, the algorithm presents points to the network in a quasi-random fashion. We first choose a prime number p close to 500 that is not a factor of the number of pixels in the image. We then present every p^{th} pixel to the network, wrapping around the image as necessary. After five subsequent wraps around the image, the neighborhood radius and network updating factor are decreased. This choice of p means that every pixel is presented to the network after about 500 such wraps around the image and the decaying parameters will have been updated no more than 101 times.

After we present a pixel to the network, the training phase is complete and the mapping phase can begin. In the mapping phase, every pixel is presented again to the network. As the network does not change in the mapping phase, it is safe to present the pixels in sequential order. For each pixel, the node closest to the pixel is again identified and the pixel in the original image is mapped to the value of the winning node. Since there are a fixed number k of network nodes, this guarantees that the resultant image will be mapped to a palette containing at most k colors.

One possible adjustment that may be made by the user to reduce the training time is to limit the network training to a subset of the input colors. This is accomplished through the selection of an integer parameter $n \geq 1$ by the user indicating that every n^{th} color in the original image is to be used for training. A value of $n = 1$ allows for training to occur on the entire input set. Large values of n create noticeable decreases in quantization quality [3]. Throughout this work we set $n = 1$.

4. CuNeuQuant

With the recent proliferation of consumer multiprocessors, there has been increased effort to port many common sequential algorithms to their parallel equivalents. Image quantization is no exception. Focusing specifically on NeuQuant, one may (incorrectly) assume that the network can be trained in parallel using separate regions of the input image. However, as mentioned before, the network changes after being presented with each data point, so the networks being trained by each thread are likely to become divergent. An attempt to overcome this by intelligently partitioning the input pixels so as to not overlap in the network also proves futile since there is no way to know the outcome of the network a priori.

Despite these challenges in the parallelization of the NeuQuant algorithm, there are still opportunities we can explore. Several of the operations performed during training and mapping lend themselves well to a parallel implementation. Specifically, the identification of the winning node, as well as updating the network, can be implemented in parallel. Both of these tasks involve iteration over the network, performing single instruction, multiple data (SIMD) operations on each node. For this reason, the massively parallel stream processor architecture present on a GPU is well suited to this task. These operations are the primary differences between the original NeuQuant and our parallel CuNeuQuant implementation.

Our GPU implementation of NeuQuant uses nVidia's GPU technology [4]. nVidia provides a development environment known as CUDA to assist programmers when developing for their GPUs. One advantage of the CUDA programming language is that it allows the programmer to design code to run natively on the hardware directly rather than conforming to a graphics-specific API such as OpenGL. CUDA is an extension to the C/C++ language and allows for the mixing of CPU and GPU code within the same implementation file.

In order to ensure a fair comparison between the sequential and parallel versions of NeuQuant, we implemented CuNeuQuant to perform quantization to 256 colors. Since the symmetric multiprocessors (SM) on the GPU have no direct access to any memory not residing on the GPU itself, we must transfer the image to the GPU global memory. Furthermore, the entire training network is contained within the device's global memory. To avoid an additional memory transfer, the network is initialized directly on the device in parallel. To do this, 256 threads, one for each color, are spawned, each one responsible for initializing one node of the network to its thread index. Algorithm 1 shows the pseudo code for this initialization.

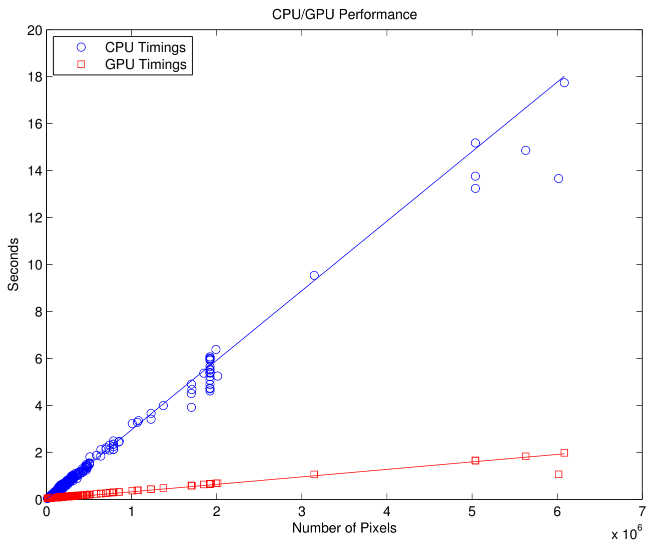
Algorithm 1 Neural Network Initialization

1: idx ← [thread index]
2: networkRed[idx] ← idx
3: networkGreen[idx] ← idx
4: networkBlue[idx] ← idx

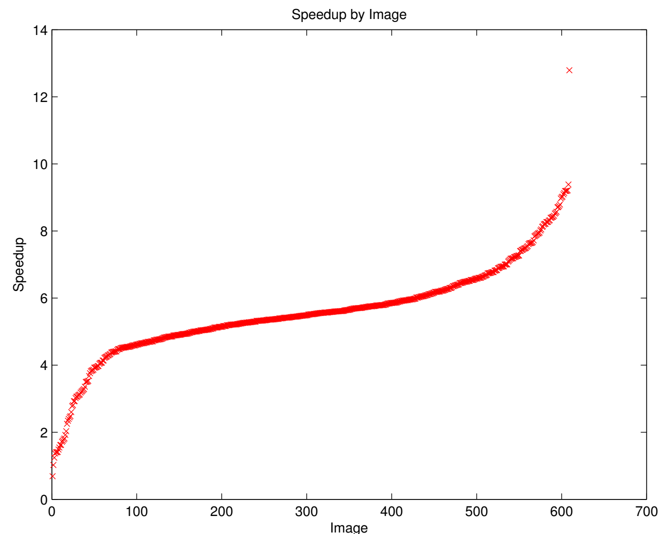
Following network initialization, we proceed with training. For this purpose, we arrange our threads into blocks of 256 threads each, corresponding to the nodes of our neural network, and a grid of blocks with the same dimensions as our input image. Within each GPU thread, the thread index indicates the index of the network node and the grid indices (x, y) indicate the spatial location of the pixel. The choice of the width and height for grid size is not necessary as spatial location of pixels is ignored during training. However, to ensure the SM are fully utilized, the dimensions of the grid should be factors of the number of training points; the image width and height conveniently providing such a factorization. CUDA also imposes a limit of 65535 for each grid dimension, and a one-dimensional ordering could easily exceed this limit.

Within each thread we compute a linear grid index. The index of the training point is then computed by multiplying the grid index by our step-size p modulo $|C|$. The grid index is also used to obtain the iteration number for the purposes of computing our training weight factor α and radius as defined within the original NeuQuant algorithm.

Following identification of the index of the training point, we then copy this point to the local memory space of the thread. Unfortunately, performing an uncoalesced memory transfer is extremely expensive, and such a transfer is even more detrimental when performed on every thread. However, since each thread within a block operates on the same image point, and a read-only copy of the point is sufficient, we instead make a single shared copy of the image point. The memory transfer is performed only by the thread with an index of 0 while all other threads wait at a barrier to ensure



(a) Sequential and Parallel Timings



(b) Per-Image Speedup

Fig. 2: Experimental Results

the point is available prior to proceeding. This reduces the number of uncoalesced memory transfers by a factor of 255.

Following this memory transfer, each thread then proceeds to compute the distance from the training point to its network node. This is done using an L_1 metric (sum of absolute differences) and the result is stored in a shared memory location. Since each thread t is accessing the tth item in the network, this data transfer is coalesced, avoiding the memory hit described previously. In addition, we initialize an array of best indices so that index t contains the value t. In Algorithm 2 we show the pseudo-code for computing the network-to-point distance, with the initialization of r in line 6 as in [3].

Algorithm 2 Network-to-Point Distance Computation

1: $n \leftarrow$ Number of pixels in image
2: $linearPtIdx \leftarrow$ Linear point index
3: $pointIdx \leftarrow (linearPtIdx \times p) \bmod n$
4: $t \leftarrow$ Thread index
5: $i \leftarrow linearPtIdx \div$ Points Per Iteration
6: $r \leftarrow \lfloor 32e^{-0.0325 \times i} \rfloor$
7: **if** $t = 0$ **then**
8: $\quad point \leftarrow points[pointIdx]$
9: **end if**
10: barrier()
11: $dist[t] \leftarrow |point - network[t]|$
12: $bestIdx[t] \leftarrow t$

After computing these network-to-point distances, we perform another thread synchronization operation to ensure that all distances are computed prior to proceeding. This is necessary since the next operation is a reduction used to identify the network node that is the closest to the training point, i.e., has the minimum distance. In Algorithm 3 we show the pseudo-code for the reduction operation. To perform this reduction, we create a stride variable s which begins at half the network size (128) and is halved at each iteration down to 1, inclusive, as show in line 2. For each iteration, thread $t < s$ compares its distance to the distance at location $t+s$. If the distance at index $t+s$ is less than the distance at t, then distance t is overwritten with distance $t+s$ (line 4), and index $t+s$ is stored in the best indices array at location t (line 5). After all iterations have completed, the value in index 0 of the distance array contains the smallest distance, and the value in index 0 of the best indices array contains the network index generating that distance (line 9).

Algorithm 3 Data Reduction

1: barrier()
2: **for** $s = 128 \rightarrow 1$ multiply by $\frac{1}{2}$ at each iteration **do**
3: $\quad$ **if** $t < s$ and $\text{dist}[t+s] < \text{dist}[t]$ **then**
4: $\quad\quad \text{dist}[t] \leftarrow \text{dist}[t+s]$
5: $\quad\quad \text{bestIdx}[t] \leftarrow \text{bestIdx}[t+s]$
6: $\quad$ **end if**
7: $\quad$ barrier()
8: **end for**
9: $w \leftarrow \text{bestIdx}[0]$

Once the winning node has been identified, each affected node within the network needs to be updated. This update

Table 1: Average Timings, Parallel vs. Sequential

	Time Per Pixel (seconds)
Sequential	2.9610×10^{-6}
Parallel	5.8899×10^{-7}
Speedup	5.0272

procedure is illustrated in Algorithm 4, with α and ρ defined as in [3]. An "affected node" is identified as a node whose index $i < |w - r|$ where w is the index of the winning node, and r is the current radius. The update is then performed for each network node within the bounds.

Algorithm 4 Network Refinement

1: **if** $|w - t| < r$ **then**
2: $\quad \alpha \leftarrow e^{-0.03i}$
3: $\quad \rho \leftarrow 1 - (\frac{|w-t|}{r})^2$
4: $\quad \text{node}[t] \leftarrow \alpha\rho \times \text{trainingPoint} + (1 - \alpha\rho) \times \text{node}[t]$
5: **end if**

Algorithms 2, 3, and 4 together make up the training kernel of our code. After training has completed, we move into the mapping phase. For this phase, we execute a second kernel which is structured very similarly to the training kernel. We intentionally did not combine these two kernels since we need to ensure that every block has completed training prior to mapping, and there is no mechanism within CUDA to achieve this. However, since each kernel call is performed sequentially, making two separate kernels gives us a form of block-level synchronization.

For the mapping kernel, we arrange the grids and blocks the same way as within the training kernel; that is, each block contains 256 threads and the grid dimensions are equal to the dimensions of the image. We then perform the network-to-point distance computation and reductions (Algorithms 2 and 3) to determine the closest trained node to the input point. Next we assign to the point (now serving as our output) the value within the winning node.

5. Experimental Results

For our timing results, we ran sequential and parallel versions of the NeuQuant algorithm on a subset of images obtained from the ImageNet image database [12]. The sequential metrics were gathered by running Dekker's original C implementation on a single core of an Intel i7 processor with 8 GB of RAM. The parallel metrics were gathered using an nVidia GeForce 9800 GTX+ GPU attached to the same machine, so as to not introduce any bias by different memory, bus, or processor speeds.

Figure 2a shows the result of the sequential and parallel timings. Circles represent timings performed for individual images with the lines indicating the average trend, with the x-axis representing pixels in millions, and the y-axis representing time in seconds. It can be seen from this figure that the sequential algorithm takes between four and six seconds to quantize a two-megapixel image. It can also be seen from this figure that the GPU implementation performed markedly faster and more predictably, with only one obvious outlier. Notice that a two-megapixel image is quantized on the GPU in about 0.7 seconds, a speedup of about 7x on average. Also note that a six-megapixel image is quantized in under 2 seconds compared to almost 18 seconds with the sequential algorithm. See Table 1 for a comparison of the quantization time per pixel, averaged over all images, for both the sequential and parallel algorithms. Using this data we also compute the average speedup of the parallel algorithm verses the sequential.

This speedup is primarily due to the use of the stream processors within the GPU. By performing the nearest-neighbor search (Algorithms 2 and 3) and network updates (Algorithm 4) on the GPU, we parallelize these operations across all 128 CUDA cores available on the GeForce 9800 GTX+. Since these operations would require 256 iterations on a sequential CPU implementation, we are able to realize a speedup of 128x for these operations alone (neglecting memory transfers to and from the GPU). Since CuNeuQuant still contains sequential components, we realize a lesser speedup of about 5x to 7x, due to Amdahl's law.

In Figure 2b we calculate the per-image speedup and sort these in increasing order. In this figure, the x-axis represents an index number of the images and the y-axis represents the speedup. It can be seen that for the majority of images, we achieved a speedup between 5x and 7x.

6. Conclusion

In this work we presented a GPU-based parallel algorithm based upon the NeuQuant algorithm by [3]. We showed that for a majority of the representative images supplied to the algorithms, a speedup of 4x or more is possible. Furthermore, we showed that timings performed on the GPU hardware are more consistent than running on the CPU with in the sequential algorithm. CuNeuQuant achieves this speedup by utilizing multiple CUDA cores for the nearest-neighbor search and neural network updates.

We have released our implementation of the CuNeuQuant algorithm under the Limited GNU Public License (LGPL). This code is hosted on Google Code and can be downloaded at `https://code.google.com/p/dali-neuquant/`.

From our results, it can be seen that the NeuQuant algorithm is sped up significantly by the parallelism provided by a GPU. It is our hope that these speed improvements can help make CuNeuQuant a viable competitor in the high-quality image quantization arena.

References

[1] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 3rd ed. Prentice Hall, 2008.

[2] Z. Wang, A. Bovik, H. Sheikh, and E. Simoncelli, "Image quality assessment: from error visibility to structural similarity," *Image Processing, IEEE Transactions on*, vol. 13, no. 4, pp. 600 –612, april 2004.

[3] A. H. Dekker, "Kohonen neural networks for optimal colour quantization," *Network: Computation in Neural Systems*, vol. 5, no. 3, pp. 351–367, 1994.

[4] *CUDA C Programming Guide*. NVIDIA Corporation, 2011.

[5] P. Heckbert, "Color image quantization for frame buffer display," *SIGGRAPH Comput. Graph.*, vol. 16, pp. 297–307, July 1982.

[6] E. Forgy, "Cluster analysis of multivariate data: efficiency versus interpretability of classifications," *Biometrics*, vol. 21, pp. 768–780, 1965.

[7] 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. 281-297. California, USA, 1967, p. 14.

[8] J. A. Hartigan and M. A. Wong, "Algorithm as 136: A k-means clustering algorithm," *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, vol. 28, no. 1, pp. pp. 100–108, 1979.

[9] B. Freisleben and A. Schrader, "An evolutionary approach to color image quantization," in *Evolutionary Computation, 1997., IEEE International Conference on*, apr 1997, pp. 459 –464.

[10] M. Gervautz and W. Purgathofer, "A simple method for color quantization: octree quantization," in *Graphics Gems*, A. S. Glassner, Ed. San Diego, CA, USA: Academic Press Professional, Inc., 1990, pp. 287–293.

[11] J. Puzicha, M. Held, J. Ketterer, J. M. Buhmann, and D. Fellner, "On spatial quantization of color images," *in Proceedings of the European Conference on Computer Vision*, vol. 9, pp. 563–577, 1998.

[12] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *CVPR09*, 2009.