Cartoonizer: report
High Performance Computing Project

Emilio Garzia [mat. 0124/314]
Luigi Marino [mat. 0124/320]

2025

Contents

Contents **2**

0 Introduction **3**

1 Theoretical Background **5**
1.1 K-Means algorithm 5
1.1.1 Advantages and Limitations 6
1.1.2 Applications 7
1.1.3 Conclusion 7
1.2 Color Quantization 7
1.2.1 Purpose and Applications 7
1.2.2 Quantization Process 8
1.2.3 Challenges in Color Quantization 8
1.2.4 Conclusion 9

2 Technologies **10**
2.0.1 CUDA 10
2.0.2 OpenCV 10

3 Technical implementation **12**
3.1 Sequential cartoonizer 12
3.1.1 Euclidean Distance Function 12
3.1.2 Update Centroids Function 13
3.1.3 K-means Function 13
3.1.4 Main Function 15
3.1.5 Conclusion 16
3.2 Parallel cartoonizer 16

Chapter 0

Introduction

Cartoonizer aims primarily to create a graphic effect in "cartoon" style applied to digital images. This effect is achieved by reducing the number of colors in the image using the **k-means clustering** method. This algorithm groups the colors of the image into a predefined number of clusters, allowing each pixel to be represented by the color of the centroid of the cluster it belongs to. The result is a simplified image, with sharper color transitions and a visually appealing appearance similar to an animated drawing. The implementation of the Cartoonizer was developed using **CUDA** (*Compute Unified Device Architecture*) technology, designed to leverage the parallel computing power of **GPUs** (*Graphics Processing Units*), with the primary goal of significantly improving performance compared to a sequential version of the algorithm, enabling the processing of large images in reduced time. A comprehensive evaluation will be presented in the report, showcasing the performance of the algorithm across various **NVIDIA** devices. The analysis will include comparisons based on different parameters, such as thread numbers and device configurations, providing detailed insights into the scalability and optimization of the algorithm. This section will delve into how varying these factors influences both execution time and overall performance. The results will be meticulously detailed, illustrating the impact of parallelization and hardware-specific optimizations in real-world scenarios. The idea for this project stems from two foundational research papers that inspired its development. The first paper discusses various methods to generate cartoonized painterly effects on grayscale and colored images. It introduces the concept of vector quantization to achieve the painterly effect and compares algorithms such as LBG, KPE, and KMCG based on their processing time and visual results. These methods have applications in fields such as movie-to-comic conversions and digital art software [1]. The second paper focuses specifically on the use of k-means for color quantization. While k-means has historically been viewed as computationally expensive and sensitive to initialization, the paper demonstrates that with efficient implementations and appropriate initialization strategies, k-means can serve as a highly effective color quantization method. The experiments conducted on diverse images highlight the algorithm's per-

formance and its potential in image processing [2]. In the following chapters and sections, we will examine in detail the approach adopted, starting from the description of the problem and the techniques used, moving through the implementation in CUDA, and finally evaluating the performance achieved. The results highlight the benefits of parallelization, demonstrating how GPU computing can be utilized for creative and high-impact visual applications.

Chapter 1

Theoretical Background

In this chapter, we delve into the theoretical foundations that underpin the development and implementation of the Cartoonizer project. Specifically, we explore the **K-means algorithm**, a core clustering technique employed for data partitioning, and its critical role in **color quantization**, a process that simplifies the color representation of images.

The K-means algorithm is essential for grouping image pixels based on their color properties, enabling efficient clustering and compression of color information. By leveraging this technique, we achieve the desired artistic effect of cartoonization through reduced and simplified color palettes. The subsequent sections provide a detailed overview of these concepts, discussing their principles, challenges, and applications, and laying the groundwork for the project's computational and visual objectives.

1.1 K-Means algorithm

K-means clustering is a fundamental algorithm in unsupervised machine learning, widely employed for partitioning a dataset into a predefined number of clusters, K, based on feature similarity. The algorithm is iterative and aims to minimize the variance within each cluster, leading to compact and well-separated groups of data points. The K-means algorithm operates by alternating between assignment and update steps until convergence. The process can be described as follows:

1. **Initialization**: Select K initial centroids randomly from the dataset.

2. **Assignment Step**: Assign each data point to the nearest centroid using a distance metric, commonly the Euclidean distance.

3. **Update Step**: Calculate the new centroids by taking the mean of all data points assigned to each cluster.

4. **Convergence**: Repeat the assignment and update steps until the centroids stabilize, or a maximum number of iterations is reached.

The objective of K-means is to minimize the sum of squared distances between each data point and the centroid of its assigned cluster. Mathematically, this is represented as:

$$J = \sum_{i=1}^{n} \sum_{k=1}^{K} \mathbf{1}(y_i = k) \|\mathbf{x_i} - \mu_k\|^2$$

Where:

- J: The within-cluster sum of squared distances (WCSSD).
- $\mathbf{x_i}$: The i-th data point.
- μ_k: The centroid of the k-th cluster.
- $\mathbf{1}(y_i = k)$: An indicator function that equals 1 if data point $\mathbf{x_i}$ is assigned to cluster k, and 0 otherwise.

The algorithm alternates between minimizing J by assigning data points to their closest centroids and recalculating centroids based on these assignments.

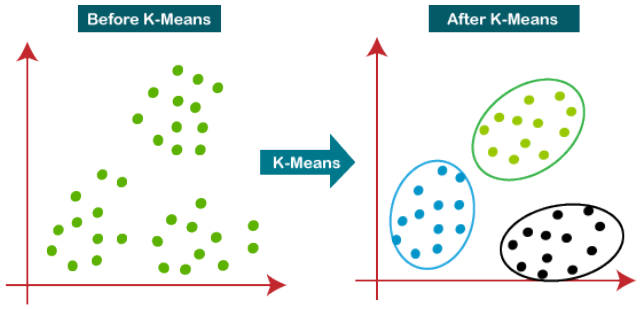

Figure 1.1: Set of data before and after the application of the K-Means algorithm

1.1.1 Advantages and Limitations

K-means is valued for its simplicity, efficiency, and scalability:

- **Simplicity**: Easy to implement and interpret.
- **Efficiency**: Computationally efficient for moderate values of K and large datasets.
- **Versatility**: Applicable to a wide range of domains, including image compression and customer segmentation.

Despite its strengths, K-means has several limitations:

- **Choice of K**: Requires prior knowledge of the number of clusters.
- **Initialization Sensitivity**: Poor initialization can lead to suboptimal clustering.
- **Cluster Shape Assumption**: Assumes clusters are spherical and of equal size, which may not hold for all datasets.
- **Outliers**: Sensitive to outliers, which can distort cluster centroids.

1.1.2 Applications

K-means clustering has numerous applications, including:

- **Image Processing**: Color quantization and compression by reducing the number of colors in an image.
- **Market Segmentation**: Grouping customers based on purchasing behavior for targeted marketing.
- **Data Summarization**: Simplifying datasets by grouping similar data points.
- **Anomaly Detection**: Identifying outliers as data points that do not fit into any cluster well.

1.1.3 Conclusion

K-means clustering remains a foundational algorithm in machine learning due to its simplicity and effectiveness in solving clustering problems. While it has limitations, various extensions and initialization strategies, such as K-means++, have been developed to address these issues and improve the algorithm's robustness.

1.2 Color Quantization

Color quantization is a process used in image processing to reduce the number of distinct colors in an image while preserving its visual quality as much as possible. This is achieved by mapping the colors of the image to a smaller set of representative colors, known as a color palette. The result is a compressed image that retains its key visual features while using fewer colors.

1.2.1 Purpose and Applications

The main purpose of color quantization is to optimize image storage, processing, and transmission by reducing the complexity of the color information. It is widely used in several fields, including:

- **Image Compression**: Reducing storage requirements for digital images by representing them with fewer colors.
- **Computer Graphics**: Rendering images efficiently on devices with limited color display capabilities.
- **Printing**: Mapping colors to a specific printer color gamut to ensure accurate reproduction.
- **Artistic Filters**: Creating stylized effects, such as cartoonization or posterization, by limiting the number of colors.

1.2.2 Quantization Process

The process of color quantization typically involves the following steps:

1. **Color Space Selection**: Choose a color space (e.g., RGB, HSV, or CIE-Lab) for representing the image's pixel values. The choice of color space can influence the effectiveness of quantization, as some spaces better capture perceptual differences between colors.
2. **Clustering of Colors**: Group similar colors into clusters. Each cluster represents a single color in the reduced palette. Clustering algorithms, such as K-means, are commonly used for this purpose. The goal is to minimize the perceptual difference between the original image and the quantized image.
3. **Color Mapping**: Replace the original colors of the image with the nearest colors from the reduced palette. This step assigns each pixel in the image to the color of its cluster centroid.

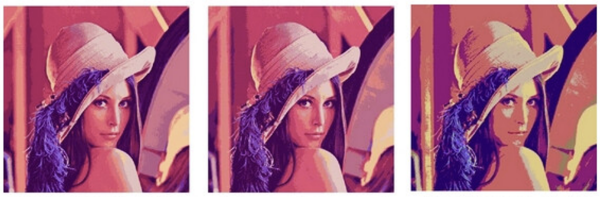

Figure 1.2: Different color quantization algorithms applied to the same image

1.2.3 Challenges in Color Quantization

Color quantization is not without its challenges, which include:

- **Perceptual Quality**: Reducing the number of colors can lead to visible artifacts or loss of detail, especially in images with subtle gradients.

- **Optimal Palette Selection**: Determining the optimal number and placement of colors in the palette is critical for balancing compression and visual fidelity.
- **Computational Cost**: Algorithms like K-means can be computationally expensive for high-resolution images, particularly with a large number of clusters.
- **Handling Outliers**: Outlier colors that do not belong to any major cluster can significantly affect the quality of the quantized image.

1.2.4 Conclusion

Color quantization is a crucial step in many image processing applications, balancing the trade-off between reducing storage requirements and maintaining visual quality. Techniques such as K-means clustering and median cut have been instrumental in achieving effective quantization, though challenges remain in optimizing perceptual quality and computational efficiency. In this work, color quantization plays a key role in achieving the desired cartoon-style effect by reducing the complexity of the image while preserving its essential features.

Chapter 2

Technologies

2.0.1 CUDA

CUDA (Compute Unified Device Architecture) is a parallel computing platform and application programming interface (API) developed by NVIDIA. It enables developers to utilize the power of NVIDIA GPUs for general-purpose computing tasks. By leveraging CUDA, applications can perform computations faster by distributing tasks across thousands of GPU cores. It is widely used in fields such as scientific computing, machine learning, and real-time rendering.

NVIDIA Nsight Compute

NVIDIA Nsight Compute is a performance analysis tool designed for CUDA applications. It provides detailed metrics and insights into GPU kernel execution, enabling developers to identify bottlenecks and optimize their code. This tool is essential for understanding and improving the performance of GPU-accelerated applications.

NVIDIA Nsight Systems

NVIDIA Nsight Systems is a system-wide performance analysis tool that helps developers identify performance issues across the CPU and GPU. It provides a holistic view of application behavior, enabling optimization at both the system and application levels.

In the results section, we will analyze our findings using these tools to evaluate the performance of our CUDA-based implementations.

2.0.2 OpenCV

OpenCV (Open Source Computer Vision Library) is an open-source library designed for real-time computer vision and image processing tasks. It provides a wide range of tools and functions for image manipulation, feature extraction, object detection, and more. OpenCV is widely adopted in industries such as

robotics, medical imaging, and autonomous vehicles due to its versatility and robust performance.

Some key features of OpenCV include:

- **Wide Functionality:** OpenCV supports tasks such as image filtering, geometric transformations, and object recognition.
- **Cross-Platform Support:** OpenCV works on multiple platforms, including Windows, Linux, macOS, and Android.
- **Hardware Acceleration:** OpenCV integrates with CUDA and OpenCL to accelerate processing on compatible hardware.

OpenCV is written in C++ but provides bindings for Python, Java, and other languages, making it accessible to a wide range of developers. With its comprehensive functionality and active community, OpenCV remains a cornerstone of computer vision development.

Chapter 3

Technical implementation

In this chapter, we present an overview of our work on implementing k-means clustering algorithms for image processing. Two distinct approaches have been developed, each tailored to different computational paradigms and performance goals:

- **Sequential Algorithm:** A straightforward implementation in C++ that processes the input image using the k-means clustering technique. This version, while simple and easy to understand, is optimized for single-threaded execution and serves as a baseline for performance comparisons.
- **Parallel Algorithm with CUDA:** A high-performance version of the k-means algorithm designed to utilize NVIDIA GPUs. This approach leverages CUDA to perform computations in parallel across thousands of GPU cores, providing significant speedups for large-scale image data.

These implementations highlight the trade-offs between simplicity and performance, and they serve as a foundation for evaluating the impact of parallelization on clustering algorithms. The following sections will delve into the specifics of each implementation, detailing their design, execution, and performance considerations.

3.1 Sequential cartoonizer

We will now break down and analyze different parts of the code, explaining how each segment works and contributes to the overall functionality of the cartoonizer.

3.1.1 Euclidean Distance Function

This function computes the Euclidean distance between two colors, which is essential for finding the nearest cluster centroid during the K-means algorithm.

```cpp
// Compute the Euclidean distance between two colors
float euclidean_distance(const Color& a, const Color& b) {
    return std::sqrt((a.r - b.r) * (a.r - b.r) + (a.g - b.g) * (a.g - b
        .g) + (a.b - b.b) * (a.b - b.b));
}
```

Explanation: This function calculates the squared differences in the RGB values between two colors and returns the square root of the sum of those squared differences. This distance metric is used to compare how close two colors are in RGB space.

3.1.2 Update Centroids Function

After assigning each pixel to the nearest centroid, the centroids need to be updated by averaging the pixel colors assigned to each cluster.

```cpp
// Update centroids based on pixel assignments
void update_centroids(const std::vector<Color>& pixels, const std::
    vector<int>& assignments, std::vector<Color>& centroids, int k) {
    std::vector<int> cluster_sizes(k, 0);
    std::vector<Color> new_centroids(k, {0.0f, 0.0f, 0.0f});

    for (size_t i = 0; i < pixels.size(); i++) {
        int cluster_idx = assignments[i];
        new_centroids[cluster_idx].r += pixels[i].r;
        new_centroids[cluster_idx].g += pixels[i].g;
        new_centroids[cluster_idx].b += pixels[i].b;
        cluster_sizes[cluster_idx]++;
    }

    for (int i = 0; i < k; i++) {
        if (cluster_sizes[i] > 0) {
            new_centroids[i].r /= cluster_sizes[i];
            new_centroids[i].g /= cluster_sizes[i];
            new_centroids[i].b /= cluster_sizes[i];
        }
    }

    centroids = new_centroids;
}
```

Explanation: This function iterates over each pixel and updates the centroids by averaging the colors of all pixels assigned to each cluster. It ensures that the new centroids represent the average color of each cluster, which will be used in the next iteration.

3.1.3 K-means Function

The main function implementing the K-means algorithm for color quantization. It converts the image to a set of color values, applies the K-means clustering

algorithm, and then generates the output image with reduced colors.

```
// K-means algorithm for color quantization
void kmeans_cpu(const cv::Mat& image, int k, int max_iter, cv::Mat&
    output_image,int seed) {
    int width = image.cols;
    int height = image.rows;
    int num_pixels = width * height;

    // Convert image to a vector of Color structs
    std::vector<Color> pixels(num_pixels);
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            cv::Vec3b color = image.at<cv::Vec3b>(i, j);
            pixels[i * width + j] = { color[2] / 255.0f, color[1] /
                255.0f, color[0] / 255.0f };
        }
    }

    srand(seed);

    // Randomly initialize centroids
    std::vector<Color> centroids(k);
    for (int i = 0; i < k; i++) {
        centroids[i] = { float(rand()) / RAND_MAX, float(rand()) /
            RAND_MAX, float(rand()) / RAND_MAX };
    }

    std::vector<int> assignments(num_pixels, 0);
    for (int iter = 0; iter < max_iter; iter++) {
        // Step 1: Assign each pixel to the closest centroid
        for (int i = 0; i < num_pixels; i++) {
            float min_dist = std::numeric_limits<float>::max();
            for (int j = 0; j < k; j++) {
                float dist = euclidean_distance(pixels[i], centroids[j
                    ]);
                if (dist < min_dist) {
                    min_dist = dist;
                    assignments[i] = j;
                }
            }
        }

        // Step 2: Update centroids based on the pixel assignments
        update_centroids(pixels, assignments, centroids, k);
    }

    // Step 3: Create the output image
    output_image = image.clone();
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            int cluster_idx = assignments[i * width + j];
            cv::Vec3b new_color(
                static_cast<unsigned char>(centroids[cluster_idx].b *
                    255),
                static_cast<unsigned char>(centroids[cluster_idx].g *
                    255),
                static_cast<unsigned char>(centroids[cluster_idx].r *
```

```
                    255)
            );
            output_image.at<cv::Vec3b>(i, j) = new_color;
        }
    }
}
```

Explanation: - The function first converts the input image into a vector of ‘Color‘ structs. Each pixel’s RGB values are normalized between 0 and 1. - It randomly initializes the centroids, assigns each pixel to the nearest centroid, and updates the centroids in each iteration. - After max_iter iterations, the final output image is generated where each pixel is replaced by the corresponding centroid color.

3.1.4 Main Function

The main driver code calls the functions, loads the image, applies K-means, and saves the output image.

```
int main(int argc, char* argv[]) {
    int clusters = DEFAULT_CLUSTERS;
    int iterations = DEFAULT_ITERATIONS;
    int seed = DEFAULT_SEED;
    int threads_per_block = DEFAULT_THREADS;
    std::string input_image_path = "images/image.jpg";
    std::string output_image_path = "images/cartoon_cpu.jpg";

    arg_parser(argc, argv, clusters, iterations, seed,
        threads_per_block, input_image_path, output_image_path);

    // Load the image
    cv::Mat image = cv::imread(input_image_path);
    if (image.empty()) {
        std::cerr << "Error loading image!" << std::endl;
        return -1;
    }

    cv::Mat output_image;
    kmeans_cpu(image, clusters, iterations, output_image, seed);

    // Save the output
    cv::imwrite(output_image_path, output_image);
    std::cout << "CPU output saved!" << std::endl;
    return 0;
}
```

Explanation: The main function reads the input image, applies the K-means algorithm, and saves the result as the output image.

3.1.5 Conclusion

The K-means algorithm used in this code provides an effective way to reduce the color complexity of an image, giving it a cartoon-like appearance. By analyzing and applying clustering to pixel colors, the image becomes more abstract and simplified, which enhances the cartoon effect.

3.2 Parallel cartoonizer

Bibliography

[1] Archana B. Patankar, Purnima A. Kubde, and Ankita Karia. "Image cartoonization methods". In: *2016 International Conference on Computing Communication Control and automation (ICCUBEA)*. 2016, pp. 1–7. DOI: 10.1109/ICCUBEA.2016.7860045.

[2] M. Emre Celebi. "Improving the performance of k-means for color quantization". In: *Image and Vision Computing* 29.4 (2011), pp. 260–271. ISSN: 0262-8856. DOI: https://doi.org/10.1016/j.imavis.2010.10.002. URL: https://www.sciencedirect.com/science/article/pii/S0262885610001411.