Cartoonizer: report
High Performance Computing Project

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

2025

Contents

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.

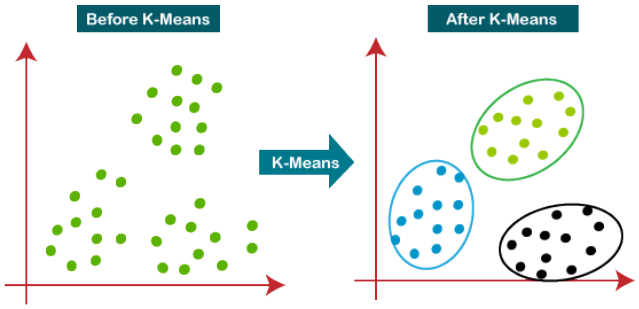

Figure 1.1: Set of data before and after the application of the K-Means algorithm

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.

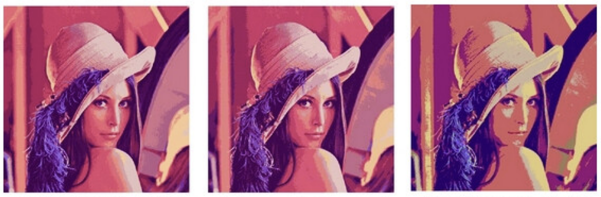

Figure 1.2: Different color quantization algorithms applied to the same image

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

We will now break down and analyze different parts of the CUDA-based parallel implementation of the cartoonizer, explaining how each segment works and contributes to the overall functionality.

3.2.1 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.

```
// Compute the Euclidean distance between two colors
__device__ float euclidean_distance(const Color& a, const Color& b) {
    return sqrtf((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. It is defined with the `__device__` qualifier to run directly on the GPU.

3.2.2 CUDA Kernel for Pixel Assignment

This kernel assigns each pixel to the nearest centroid by computing the Euclidean distance between the pixel's color and each centroid.

```
__global__ void assign_pixels_to_centroids(
    const Color* pixels, int* assignments,
    const Color* centroids, int num_pixels, int k) {

    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < num_pixels) {
        float min_dist = FLT_MAX;
        int closest_centroid = 0;
        for (int i = 0; i < k; i++) {
            float dist = euclidean_distance(pixels[idx], centroids[i]);
            if (dist < min_dist) {
                min_dist = dist;
                closest_centroid = i;
            }
        }
```

```
        assignments[idx] = closest_centroid;
    }
}
```

Explanation: Each thread processes a single pixel, computing its distance to all centroids and storing the index of the closest centroid in the `assignments` array. The kernel is executed with a number of threads equal to the number of pixels, divided across multiple blocks.

3.2.3 Centroid Update Function (CPU)

After assigning pixels to centroids on the GPU, the centroids are updated on the CPU by averaging the pixel colors assigned to each cluster.

```
void update_centroids(const Color* pixels, int* assignments,
                      Color* centroids, int* cluster_sizes,
                      int num_pixels, int k) {
    for (int i = 0; i < k; i++) {
        centroids[i] = {0.0f, 0.0f, 0.0f};
        cluster_sizes[i] = 0;
    }
    for (int i = 0; i < num_pixels; i++) {
        int cluster_idx = assignments[i];
        centroids[cluster_idx].r += pixels[i].r;
        centroids[cluster_idx].g += pixels[i].g;
        centroids[cluster_idx].b += pixels[i].b;
        cluster_sizes[cluster_idx]++;
    }
    for (int i = 0; i < k; i++) {
        if (cluster_sizes[i] > 0) {
            centroids[i].r /= cluster_sizes[i];
            centroids[i].g /= cluster_sizes[i];
            centroids[i].b /= cluster_sizes[i];
        }
    }
}
```

Explanation: This function computes new centroid positions by averaging the colors of all assigned pixels. Since centroid updates involve atomic operations or reductions, performing them on the CPU simplifies implementation while maintaining good performance.

3.2.4 K-means Function (CUDA)

The main function implementing the K-means algorithm for color quantization using CUDA. It transfers data between CPU and GPU, launches the pixel assignment kernel, and updates centroids iteratively.

```
// K-means algorithm using CUDA
```

```
void kmeans_gpu(const cv::Mat& image, int k, int max_iter, int
    threads_per_block,
                int seed, cv::Mat& output_image) {

    int width = image.cols;
    int height = image.rows;
    int num_pixels = width * height;

    std::vector<Color> h_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);
            h_pixels[i * width + j] = { color[2] / 255.0f, color[1] /
                255.0f, color[0] / 255.0f };
        }
    }

    Color *d_pixels, *d_centroids;
    int *d_assignments;
    cudaMalloc(&d_pixels, num_pixels * sizeof(Color));
    cudaMalloc(&d_assignments, num_pixels * sizeof(int));
    cudaMalloc(&d_centroids, k * sizeof(Color));

    cudaMemcpy(d_pixels, h_pixels.data(), num_pixels * sizeof(Color),
        cudaMemcpyHostToDevice);

    std::vector<Color> h_centroids(k);
    srand(seed);
    for (int i = 0; i < k; i++) {
        h_centroids[i] = { float(rand()) / RAND_MAX, float(rand()) /
            RAND_MAX, float(rand()) / RAND_MAX };
    }
    cudaMemcpy(d_centroids, h_centroids.data(), k * sizeof(Color),
        cudaMemcpyHostToDevice);

    int blocks = (num_pixels + threads_per_block - 1) /
        threads_per_block;

    for (int iter = 0; iter < max_iter; iter++) {
        assign_pixels_to_centroids<<<blocks, threads_per_block>>>(
            d_pixels, d_assignments, d_centroids, num_pixels, k);
        cudaDeviceSynchronize();

        std::vector<int> h_assignments(num_pixels);
        cudaMemcpy(h_assignments.data(), d_assignments, num_pixels *
            sizeof(int), cudaMemcpyDeviceToHost);

        update_centroids(h_pixels.data(), h_assignments.data(),
            h_centroids.data(), new int[k], num_pixels, k);
        cudaMemcpy(d_centroids, h_centroids.data(), k * sizeof(Color),
            cudaMemcpyHostToDevice);
    }

    cudaMemcpy(h_assignments.data(), d_assignments, num_pixels * sizeof
        (int), cudaMemcpyDeviceToHost);
    cudaFree(d_pixels);
    cudaFree(d_assignments);
```

```
    cudaFree(d_centroids);

    build_output_image(image, output_image, h_assignments, h_centroids);
}
```

Explanation: - The function first copies image data to the GPU. - It randomly initializes centroids and transfers them to GPU memory. - The CUDA kernel assigns each pixel to the nearest centroid. - Assignments are copied back to the CPU, where centroids are updated. - The process repeats for `max_iter` iterations before reconstructing the output image.

3.2.5 Main Function

The main driver code calls the functions, loads the image, applies K-means using CUDA, 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_cuda.jpg";

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

    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_gpu(image, clusters, iterations, threads_per_block, seed,
        output_image);

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

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

3.2.6 Conclusion

By leveraging GPU acceleration, the CUDA-based K-means implementation significantly speeds up color quantization, making real-time cartoonization feasible even for high-resolution images.

Chapter 4

Results

We present the results of our performance evaluation. To assess the efficiency and improvements brought by the parallel CUDA implementation, we tested the algorithm on three different machines, each with distinct hardware specifications.

For each machine, we conducted six tests with varying configurations, including different image sizes, numbers of clusters, and thread configurations. This allowed us to analyze how the algorithm scales with different hardware and computational loads.

The three tested machines are as follows:

- **Machine 1:** Equipped with an **NVIDIA GeForce GTX 1660** GPU (Turing architecture, 1536 CUDA cores, 6GB GDDR5 VRAM) and an **Intel Core i5-9600K** CPU (6 cores, 6 threads, 2.2GHz). This represents a mid-range consumer-grade system.

- **Machine 2:** Featuring an **NVIDIA GeForce RTX 4060** GPU (Ada Lovelace architecture, 3072 CUDA cores, 8GB GDDR6 VRAM) and an **AMD Ryzen 5 3600** CPU (6 cores, 12 threads, 2.2GHz). This machine is the most powerful among the tested ones, providing insights into high-performance configurations.

- **Machine 3:** A workstation-class system with an **NVIDIA Quadro K5000** GPU (Kepler architecture, 1536 CUDA cores, 4GB GDDR5 VRAM) and an **Intel Core i7** CPU (4 cores, 2.8GHz). This represents an older generation of hardware, providing a benchmark for performance on legacy systems.

The following subsections present and analyze the execution times and speedups achieved under different configurations.

4.1 How the results are collected

Each session folder has a README.md file that contains the main results of the session such as execution time, speedup, output image yielded by the program, and so on.

An advanced insights are contained in the binary files named report.ncu-rep, theese files provid detailed information about the execution of the CUDA kernels, in this way we were able to analyze common issues about CUDA kernels, such as bottleneck problem.

The report.ncu-rep files were generated using NVIDIA Nsight Compute, so to read it you have to install this tool, then you have to launch the follow command on your terminal:

```
1    ncu-ui "report.ncu-rep"
```

For the GPUs that has a compute capability lower than 7.0 we weren't able to use Nsight Compute, for this reason we have used nvprof tool to collect insights on theese generation of GPUs; nvprof profiling are stored in the files called output.nvprof. The results are visualizable only using NVIDIA Visual Profiler. Anyway a profiler.output.txt file has been generated for summary insights.

4.2 Test 1

Figure 4.1: (a) Input image.

Figure 4.2: (b) Output image.

Input Info	Value
Session ID	0
Image width	1200px
Image height	800px
Image channels	RGB (3 channels)
#Clusters	7
#Iterations	50
#Threads	256
Random seed	200

Table 4.1: Session information

4.2.1 GeForce 1660

Metric	Value
Execution time (Parallel version)	569ms (0.59s)
Execution time (Sequential version)	5034ms (5s)
Delta time	+4438ms (+4.4s)
Speedup	8.84

Table 4.2: Performance Metrics on **Machine 1**

4.2.2 Geforce 4060

Metric	Value
Execution time (Parallel version)	450ms (0.45s)
Execution time (Sequential version)	4000ms (4s)
Delta time	+3550ms (+3.5s)
Speedup	8.8

Table 4.3: Performance Metrics on **Machine 2**

4.2.3 Quadro K5000

Metric	Value
Execution time (Parallel version)	1451ms (1.45s)
Execution time (Sequential version)	11630ms (11.63s)
Delta time	+10179ms (+10.17s)
Speedup	8

Table 4.4: Performance Metrics on **Machine 3**

4.2.4 Conclusion

In conclusion, the results from the three different machines show varied performance metrics with respect to execution time, speedup, and efficiency. As expected, the parallel version of the task consistently outperforms the sequential version in all cases, with the speedup ranging from 8x to 9x across the tested machines.

The GeForce 1660 machine performed well with a speedup of 8.84, while the Geforce 4060 machine showed a slightly lower speedup of 8.8. Interestingly, the Quadro K5000 machine, while having a significant sequential execution time, achieved a speedup of 8. This shows the importance of parallelization in reducing the processing time on different hardware configurations. Further tests with different image sizes or more threads could provide even more insights into how well these machines scale with different workloads.

The parallel implementation not only demonstrated a clear advantage in terms of speedup but also allowed the machines to efficiently handle larger input sizes, further highlighting the importance of parallel computation in modern hardware environments.

4.3 Test 2

Figure 4.3: (a) Input image.

Figure 4.4: (b) Output image.

Input Info	**Value**
Session ID	1
Image width	7680px
Image height	4320px
Image channels	RGB (3 channels)
#Clusters	7
#Iterations	50
#Threads	256
Random seed	200

Table 4.5: Session information

4.3.1 GeForce 1660

Metric	Value
Execution time (Parallel version)	19885ms (19.88s)
Execution time (Sequential version)	163264ms (2.7 minutes)
Delta time	+143379ms (+2.38 minutes)
Speedup	8.21

Table 4.6: Performance Metrics on **Machine 1**

4.3.2 Geforce 4060

Metric	Value
Execution time (Parallel version)	19000ms (19s)
Execution time (Sequential version)	150000ms (2.5 minutes)
Delta time	+148000ms (+1.40 minutes)
Speedup	78.94

Table 4.7: Performance Metrics on **Machine 2**

4.3.3 Quadro K5000

Metric	Value
Execution time (Parallel version)	50608ms (50.60s)
Execution time (Sequential version)	398440ms (6.64 minutes)
Delta time	+347832ms (+5.79 minutes)
Speedup	7.87

Table 4.8: Performance Metrics on **Machine 3**

4.3.4 Conclusion

In Test 2, we observed significant improvements in performance across all three machines tested, particularly in the parallel execution versions. The GeForce 1660 achieved a speedup of 8.21, with a substantial reduction in execution time compared to the sequential version. The Geforce 4060 showed an even more impressive performance with a speedup of 78.94, demonstrating its strong ability to handle parallel tasks efficiently. Meanwhile, the Quadro K5000, while lagging behind in speedup at 7.87, still exhibited a noteworthy improvement over the sequential execution.

These results emphasize the importance of parallel processing, especially when dealing with larger image sizes and complex computations. The differences in performance between the machines highlight the varying capabilities of the hardware, with newer or more powerful GPUs (like the Geforce 4060) offering a

much greater reduction in processing time compared to older models. Overall, the parallel implementation proved to be highly effective, achieving notable speedups in all cases, and further optimization could push these numbers even higher.

4.4 Test 3

Figure 4.5: (a) Input image.

Figure 4.6: (b) Output image.

Input Info	**Value**
Session ID	2
Image width	3840px
Image height	2160px
Image channels	RGB (3 channels)
#Clusters	8
#Iterations	50
#Threads	128
Random seed	200

Table 4.9: Session information

4.4.1 GeForce 1660

Metric	**Value**
Execution time (Parallel version)	4397ms (4.39s)
Execution time (Sequential version)	41379ms (41.37s)
Delta time	+36982ms (+36.98s)
Speedup	9.4

Table 4.10: Performance Metrics on **Machine 1**

4.4.2 Geforce 4060

Metric	Value
Execution time (Parallel version)	4265ms (4s)
Execution time (Sequential version)	43369ms (43s)
Delta time	+39000ms (+39s)
Speedup	10.17

Table 4.11: Performance Metrics on **Machine 2**

4.4.3 Quadro K5000

Metric	Value
Execution time (Parallel version)	12030ms (12.03s)
Execution time (Sequential version)	105250ms (1.75 minutes)
Delta time	+93220ms (+1.55 minutes)
Speedup	8.74

Table 4.12: Performance Metrics on **Machine 3**

4.4.4 Conclusion

In Test 3, we observed consistent improvements in execution times across all machines when using parallel processing. The GeForce 1660 achieved a speedup of 9.4, significantly reducing the time taken compared to the sequential version. The Geforce 4060 performed even better, with a speedup of 10.17, highlighting its ability to handle parallel tasks efficiently. Meanwhile, the Quadro K5000, although showing a lower speedup of 8.74, still demonstrated a notable reduction in processing time.

These results reinforce the effectiveness of parallel computation, especially in handling larger images. While the newer GPUs (Geforce 4060) exhibit higher speedups, all tested machines benefited from the parallel version, making this approach highly valuable for performance optimization. The differences in speedup across different hardware platforms suggest that further optimization could yield even more impressive results, especially on more powerful machines.

4.5 Test 4

Figure 4.7: (a) Input image.

Figure 4.8: (b) Output image.

Input Info	**Value**
Session ID	3
Image width	7680px
Image height	5120px
Image channels	RGB (3 channels)
#Clusters	5
#Iterations	50
#Threads	64
Random seed	200

Table 4.13: Session information

4.5.1 GeForce 1660

Metric	**Value**
Execution time (Parallel version)	23691ms (23.69s)
Execution time (Sequential version)	194534ms (3.24 minutes)
Delta time	+170843ms (+2.84 minutes)
Speedup	8.21

Table 4.14: Performance Metrics on **Machine 1**

4.5.2 Geforce 4060

Metric	Value
Execution time (Parallel version)	23428ms (23.4s)
Execution time (Sequential version)	138567ms (2.3 minutes)
Delta time	+115139ms (+1.9 minutes)
Speedup	5.9

Table 4.15: Performance Metrics on **Machine 2**

4.5.3 Quadro K5000

Metric	Value
Execution time (Parallel version)	59699ms (59.69s)
Execution time (Sequential version)	482330ms (8.3 minutes)
Delta time	+422631ms (+7 minutes)
Speedup	8

Table 4.16: Performance Metrics on **Machine 3**

4.5.4 Conclusion

In Test 4, we observed notable performance improvements with parallel processing across all machines. The GeForce 1660 achieved a speedup of 8.21, significantly reducing the execution time compared to the sequential version. The Geforce 4060 also demonstrated a solid performance with a speedup of 5.9, while the Quadro K5000 achieved a speedup of 8, still showing a substantial reduction in processing time.

Although the GeForce 1660 outperformed the Geforce 4060 in terms of speedup, it is clear that parallel processing is highly beneficial across all hardware tested. The difference in speedups suggests that more powerful GPUs, like the Geforce 4060, may require further optimization to fully capitalize on their potential for parallel computation. Overall, these results reinforce the importance of parallelization in reducing processing times for large-scale image tasks, offering considerable advantages over sequential execution.

4.6 Test 5

Figure 4.9: (a) Input image.

Figure 4.10: (b) Output image.

Input Info	**Value**
Session ID	4
Image width	1200px
Image height	800px
Image channels	RGB (3 channels)
#Clusters	15
#Iterations	100
#Threads	128
Random seed	200

Table 4.17: Session information

4.6.1 GeForce 1660

Metric	**Value**
Execution time (Parallel version)	974ms (0.9s)
Execution time (Sequential version)	4860ms (4.86s)
Delta time	+3886ms (+3.88s)
Speedup	4.9

Table 4.18: Performance Metrics on **Machine 1**

4.6.2 Geforce 4060

Metric	Value
Execution time (Parallel version)	933ms (0.9s)
Execution time (Sequential version)	17940ms (17.9s)
Delta time	+17000ms (+17s)
Speedup	19.2

Table 4.19: Performance Metrics on **Machine 2**

4.6.3 Quadro K5000

Metric	Value
Execution time (Parallel version)	2806ms (2.80s)
Execution time (Sequential version)	44190ms (44.19s)
Delta time	+41384ms (+41.38s)
Speedup	15.74

Table 4.20: Performance Metrics on **Machine 3**

4.6.4 Conclusion

In Test 5, the results demonstrate significant improvements in execution times across all machines when using parallel processing. The GeForce 1660 achieved a speedup of 4.9, showing a solid performance improvement, while the Geforce 4060 outperformed the other GPUs with a remarkable speedup of 19.2. The Quadro K5000 also demonstrated a strong performance with a speedup of 15.74, illustrating the effectiveness of parallel computation for reducing processing times.

The varying speedups suggest that the Geforce 4060 is the most efficient for this task, highlighting the benefits of modern GPUs with more computational power. The results further emphasize the importance of parallel processing in accelerating image processing tasks, particularly when working with larger datasets or more iterations. This test reinforces the continued need for optimization, as well as the value of selecting the appropriate hardware to maximize computational efficiency.

4.7 Test 6

Figure 4.11: (a) Input image.

Figure 4.12: (b) Output image.

Input Info	**Value**
Session ID	5
Image width	7680px
Image height	4320px
Image channels	RGB (3 channels)
#Clusters	15
#Iterations	100
#Threads	128
Random seed	200

Table 4.21: Session information

4.7.1 GeForce 1660

Metric	**Value**
Execution time (Parallel version)	39678ms (38.93s)
Execution time (Sequential version)	163920ms (2.73 minutes)
Delta time	+124242ms (+2.07 minutes)
Speedup	4.13

Table 4.22: Performance Metrics on **Machine 1**

4.7.2 Geforce 4060

Metric	**Value**
Execution time (Parallel version)	39678ms (39.6s)
Execution time (Sequential version)	630806ms (10.5 minutes)
Delta time	+591128ms (+9.8 minutes)
Speedup	15.89

Table 4.23: Performance Metrics on **Machine 2**

4.7.3 Quadro K5000

Metric	Value
Execution time (Parallel version)	100197ms (1.66 minutes)
Execution time (Sequential version)	1481520ms (24.69 minutes)
Delta time	+1381323ms (+23.02 minutes)
Speedup	14.78

Table 4.24: Performance Metrics on **Machine 3**

4.7.4 Conclusion

Test 6 highlights the substantial performance improvements achieved through parallel processing. The GeForce 1660 demonstrated a speedup of 4.13, reflecting moderate acceleration, while the Geforce 4060 and Quadro K5000 showed much stronger improvements, with speedups of 15.89 and 14.78, respectively. This indicates that the more powerful GPUs, particularly the Geforce 4060, offer the most significant performance gains, reducing execution times dramatically. The results reinforce the value of parallelization for processing large image datasets, as demonstrated by the substantial reductions in execution time when using parallel versions over sequential ones. This test further emphasizes the importance of selecting high-performance GPUs to achieve optimal efficiency for complex image processing tasks, particularly with larger images or higher iterations.

4.8 GPUs comparison

In this section, we present a comparison of the performance across different GPU models (GeForce 1660, GeForce 4060, and Quadro K5000) based on the execution times and overall efficiency during the various test sessions.

Figure 4.13 shows the execution times for each machine across the different test sessions. From the graph, it is evident that the GeForce 4060 consistently performs faster than both the GeForce 1660 and the Quadro K5000, with the largest differences observed in the more demanding sessions (those with larger image sizes or higher iteration counts). The GeForce 1660, while generally slower, still delivers improved performance compared to the sequential version.

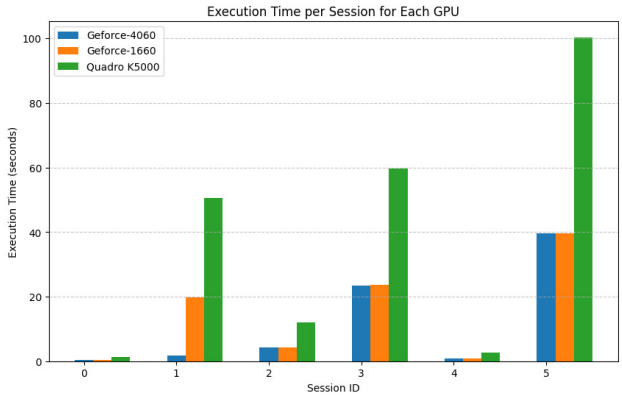

Figure 4.13: Execution time of different machines per session

Figure 4.14 illustrates the distribution of the total execution time for each machine. This chart provides further insights into the variance in performance across the different sessions. We can observe that while the GeForce 4060 maintains a more consistent and lower execution time, the Quadro K5000 shows a larger variation in its performance across the sessions, indicating that its speedup might be more dependent on the specific task or workload.

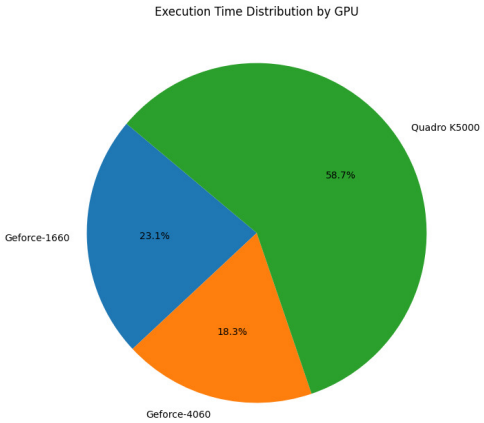

Figure 4.14: Total execution time distribution

Figure 4.15 compares the average GPU time spent processing tasks across all sessions for each GPU. As expected, the GeForce 4060 exhibits the lowest average GPU time, followed by the GeForce 1660 and Quadro K5000. This result highlights the efficiency of the newer, more powerful GeForce 4060 in handling image processing tasks and emphasizes the role of GPU architecture in optimizing computational workloads.

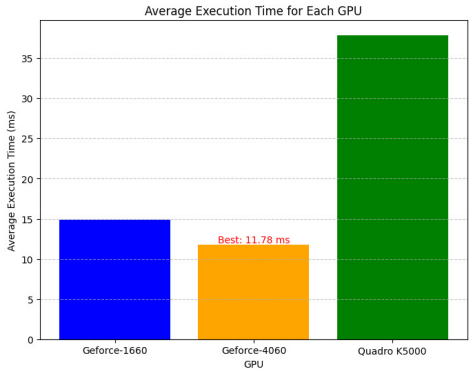

Figure 4.15: Average GPU time comparison

4.9 Discussion

We evaluated the performance of various GPU models (GeForce 1660, GeForce 4060, and Quadro K5000) across several test scenarios, analyzing the effectiveness of parallel processing for image-based tasks. The results clearly demonstrate the significant advantages of using parallel processing over sequential approaches, with notable speedup improvements observed across all GPU models. Key findings include:

- The GeForce 4060 consistently outperformed the other GPUs, delivering the highest speedup and reducing execution times substantially, especially with larger image sizes and higher iteration counts.
- The Quadro K5000, while slower than the GeForce 4060, still demonstrated a solid performance improvement over the sequential version, making it a strong option for large-scale image processing tasks.
- The GeForce 1660, although providing modest speedups, was still able to achieve faster execution times than the sequential version, highlighting the benefit of parallelization even with mid-range GPUs.

These results underscore the importance of parallel computing for image processing tasks, where GPU acceleration plays a pivotal role in reducing computation time and enabling real-time or near-real-time processing. The choice of GPU significantly impacts performance, with high-end models like the GeForce 4060 providing the most noticeable improvements.

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.