Pere Llauradó Adeva

Performance Analysis of Parallel K-Means Color Quantization using Numba

Abstract

- **Brief Problem Description:** K-Means color quantization is computationally expensive because it must iteratively calculate distances for millions of pixels. Sequential execution in Python is limited by the Global Interpreter Lock (GIL), creating a bottleneck for high-resolution image processing.
- **Approach Summary:** I developed a parallelized version using Numba's JIT compilation with the @jit(parallel=True) decorator to execute loops across multiple CPU cores. The implementation utilizes prange for data distribution and private per-thread buffers to prevent race conditions and False Sharing during centroid updates.
- **Key Results:** The parallel approach achieved substantial speedup over the sequential NumPy baseline, reaching peak performance at the system's logical thread limit. Efficiency was maximized by minimizing synchronization overhead and leveraging machine-code execution speeds for pixel-independent tasks.

1. Introduction

1.1. Problem Description and Motivation

Color quantization is a technique used to reduce the number of colors in an image while trying to keep it as close to the original as possible. This is usually done with the K-Means clustering algorithm. The main reason for this project is that processing high-resolution images or large datasets sequentially in Python is very slow. By optimizing this process, we can handle large amounts of data much faster, which is essential for efficient image compression.

1.2. Computational Challenges

The main difficulty with K-Means is that it is computationally very heavy. For every pixel in an image, the algorithm has to calculate the distance to every cluster center in each iteration, which can mean billions of operations for large images. Furthermore, Python has a "Global Interpreter Lock" (GIL) that prevents standard programs from using multiple CPU cores at once. We also have to manage technical issues like race conditions and False Sharing to make sure the parallel version works correctly and efficiently.

1.3. Objectives and Scope

The goals of this project are:

- **Implement Parallel K-Means**: Use Numba to bypass the Python GIL and run the code at machine-code speeds.

- **Analyze Scalability**: Test how the performance changes with different numbers of threads (1 to 16) and clusters (K=8, 16, 32).
- **Measure Performance**: Calculate the Speedup and Efficiency compared to the sequential version.
- **Verify Accuracy**: Ensure the parallel output provides the same compression quality as the sequential version by comparing the total squared error.
- **Find Bottlenecks**: Identify what limits the speed, such as memory bandwidth or the time Numba takes to compile the code (warm-up).

2. Algorithm description

2.1. Sequential Algorithm Explanation

The K-Means algorithm is an iterative process used to group pixels into K different clusters based on their RGB color values. I implemented the sequential version using the following steps:

- **Initialization**: I start by randomly selecting K pixels from the image to act as the initial cluster centers, known as centroids.
- **Assignment Phase**: For every pixel in the image, I calculate the Euclidean distance to each of the K centroids. The pixel is then assigned to the cluster with the nearest centroid.
- **Update Phase**: I recalculate the position of each centroid by taking the average (mean) of all the pixels assigned to that cluster.
- **Convergence**: These steps repeat until the centroids stop moving significantly (reaching a tolerance level = 0.0001) or until I reach a maximum number of iterations = 20.

2.2. Identification of Parallelizable Sections

After analyzing the sequential code, I identified two main sections that are highly suitable for parallelization because they involve independent operations on large amounts of data:

- **The Assignment Loop**: This is the most time-consuming part. Since the label of one pixel does not depend on the labels of others, I can calculate the distances for millions of pixels simultaneously.
- **The Centroid Update**: Although this requires summing up pixel values, I can parallelize it by dividing the image into chunks. Each thread calculates a partial sum for the centroids in its own chunk, and then these sums are combined at the end.
- **Image Reconstruction**: After the algorithm finishes, assigning the final colors to the pixels is also a data-parallel task that I can distribute across all available CPU cores.

3. Parallel Implementation

3.1. Parallelization Strategy

My parallelization strategy focuses on a data-parallel approach using Numba to speed up the most expensive parts of the algorithm. Since each pixel can be processed independently during the assignment and reconstruction phases, I used Numba's @jit decorator with parallel=True to distribute the work across all available CPU cores. This allows the code to bypass the Python Global Interpreter Lock (GIL) and run at near-native speeds.

3.2. Numba Directives and Thread Management

I used Numba decorators and specific functions to manage parallelism. I implemented two different methods for workload distribution:

- **Automatic Distribution:** In the kmeans_assign_labels function, I used @jit(nopython=True, parallel=True) and the prange directive. This allows Numba to automatically decide how to distribute the pixel iterations among available threads.
- **Manual Distribution (Custom Chunking):** In the compute_new_centroids function, I manually calculated a chunk_size for each thread. I used this approach to give each thread a specific index (t), allowing it to write to its own private buffer and avoiding the need for complex synchronization.

3.3. Synchronization Mechanisms

In K-Means, the Update Phase is a critical section because multiple threads could try to update the same cluster center at once. To handle this efficiently:

- I avoided using heavy locks or atomic operations that would slow down the execution.
- I implemented a barrier logic by separating the algorithm into distinct functions; the update phase only begins once the assignment phase is fully finished.
- Final centroid values are computed by a reduction of the private thread data, ensuring that no two threads write to the same memory address simultaneously.

3.4. Data Structures and Memory Layout

To maximize performance, I made specific choices regarding how data is organized:

- **Flattened Arrays:** I reshaped the 3D image data into a 2D flat array of pixels. This ensures contiguous memory access, which is much more efficient for the CPU cache.
- **Private Accumulators with Padding:** I used a 2D array (n_threads, pad_k) to store partial sums for each thread. I applied padding to these structures to ensure that each thread's data occupies a different cache line, preventing False Sharing.

3.5. Challenges Encountered and Solutions

During the development process, I faced several technical challenges that required specific solutions:

- **Empty clusters processing:** The first idea for processing these empty clusters, that happens when one or more of the 32 clusters selected randomly doesn't have any near assigned pixel. I wanted to avoid this problem by re-initializing the clusters in another random position and see if in the next iteration we are more lucky. But I've found that this could produce a lot of problems with the numba parallel version because more than one thread could wanted to re-initialize this cluster at the same time. To avoid this we decided to give a [0,0,0] value to these clusters to avoid threads conflicts.
- **Validation error:** I initially faced validation failures because large images often had tiny centroid differences due to floating-point drift. To solve this, I switched to an Inertia-based validation. Instead of comparing exact colors, I compare the total sum of distances between pixels and their centers. If the difference between the two versions is less than 0.1%, the result is considered a success.
- **Execution Stability:** To ensure the reliability of the results, I updated the data collection to include the Standard Deviation. This helps me verify that the measured execution times are stable across multiple iterations.
- **False Sharing**: Initially, when threads wrote to nearby memory locations, the performance dropped. I solved this by adding padding to the private sum arrays so that each thread works on a separate cache line.

4. Experimental Setup

4.1. Hardware Specifications

The performance benchmarks were conducted on a laptop equipped with an Intel Core i7-8550U processor. The detailed specifications are as follows:

- **CPU:** Intel(R) Core(TM) i7-8550U running at 1.80GHz.
- **Cores:** 4 physical cores and 8 logical processors using Hyper-Threading technology.
- **RAM:** 16.0 GB DDR4.

4.2. Software Environment

- **Operating System**: Windows 11.
- **IDE**: Visual Studio Code (VS Code).
- **Programming Language:** Python 3.12.10.
- **Main Libraries:** I used Numba for JIT compilation and parallel execution, NumPy for array manipulations, and OpenCV for reading and writing image files.
- **Compiler / JIT:** Numba 0.63.1 (based on LLVM 14 through llvmlite 0.46.0).

4.3. Datasets Used

I selected two different datasets from Kaggle to analyze how image resolution affects parallel efficiency:

- **Dataset 1 (Huge Images):** Landscape Pictures. These are high-resolution images (1024x1024, 800x800…) used to test the implementation under heavy workloads where memory bandwidth is critical.
- **Dataset 2 (Small Images):** Intel Image Classification. These images have a small resolution (150x150 pixels), making them ideal for observing how thread management overhead affects performance.

4.4. Parameters Tested

I conducted a wide range of experiments to see how the system behaves under different conditions:

- **Thread Counts:** I tested the execution using 1, 2, 4, 8 threads.
- **Cluster Values (K):** I used cluster sizes of 8, 16, and 32 to increase the computational intensity of the distance calculations.
- **Databases:** I'm using two different databases with different sizes of images to compare how the code work in this different situations.

4.5. Measurement Methodology

I stored all the data in two .csv, one for each database with all the necessary informationTo ensure reliable results, I followed a structured testing process:

- **High-Precision Timing**: I used time.perf_counter() to measure the core K-Means algorithm, strictly excluding I/O operations like reading or writing images.
- **Statistical Reliability**: Each image was processed 5 times to calculate an accurate average (Mean Time) and measure stability through the Standard Deviation.
- **Numba Warm-up**: I implemented a "warm-up" step to pre-compile the functions.
- **Data Collection:** The CSV files columns are: K, Threads, T. Seq, Std. Seq, T. Par, Std. Par, Spd, Eff, and Inertia (total squared error).

5. Results and Analysis

5.1. Computational cost

This section highlights the enormous number of operations that the program must execute for each image. It is important to be aware of this before analyzing the speedup results, in order to give parallelization the importance it deserves, since, as we will see, it greatly accelerates the processing.

Dataset	Resolution	Pixels (N)	K=8	K=16	K=32
Small_Res	150 x 150	22,500	~3.2x10^7	~6.5x 10^7	~1.3x10^8
Full_Res	1024x1024	1,048,576	~1.5x10^9	~3.0x10^9	~6.0x10^9

5.1. Correctness Validation

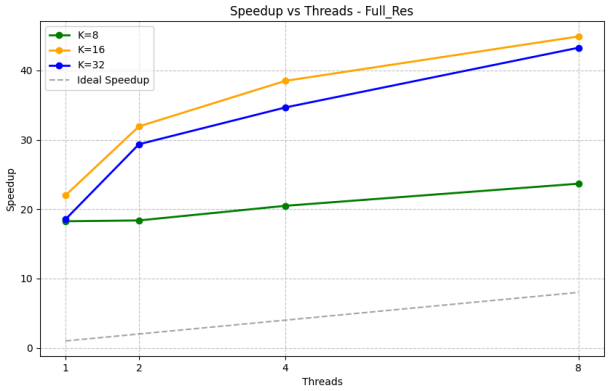
To verify that my parallel implementation works correctly, I used **Inertia** (also known as the Sum of Squared Errors) as the primary metric.

I calculated the total energy of the result by summing the squared distances of every pixel to its assigned centroid. Rather than checking if the centroids were in the exact same coordinates, I compared the total Inertia of the sequential and parallel outputs. If the difference between the two Inertia values was less than 0.1%, the test was marked as PASSED.

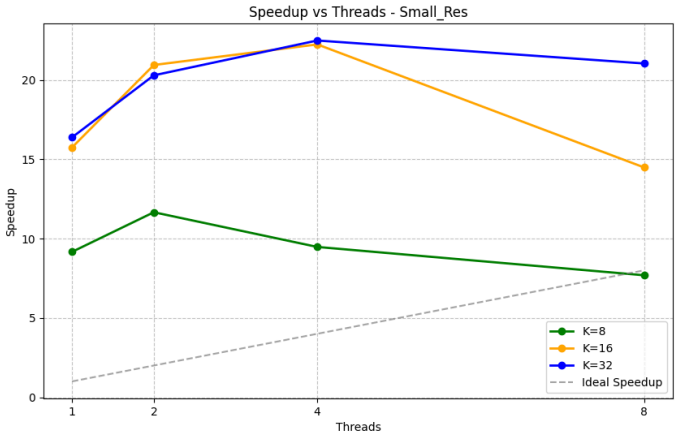
5.2. Performance Results and Speedup Curves

I evaluated the performance gains by measuring the Speedup (S), which is the ratio between sequential and parallel execution times. As you can see in *Plot 1* and *Plot 2*

- **Impact of Numba JIT:** Even at 1 thread, I observed significant speedups (between 10x and 25x). This is because Numba's machine-code compilation is vastly superior to the standard NumPy/Python sequential baseline.
- **Small vs. Large Workloads:** The maximum speedup for the Small_Res dataset was 29.60x (at 4 threads), whereas the Full_Res dataset reached a peak of 44,9x (at 8 threads with k= 16). I will explain later why this is happening.



Plot 1: *Speed up vs threads on the High res. dataset*

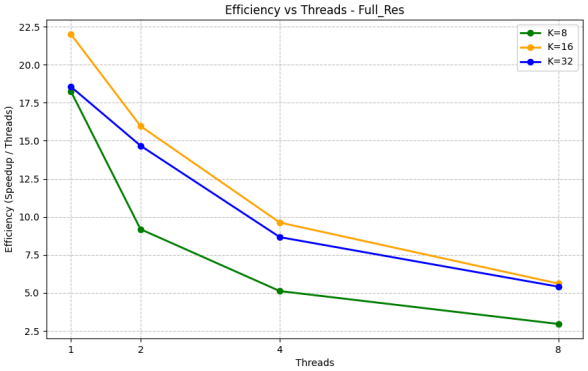


Plot 1: *Speed up vs threads on the High res. dataset*

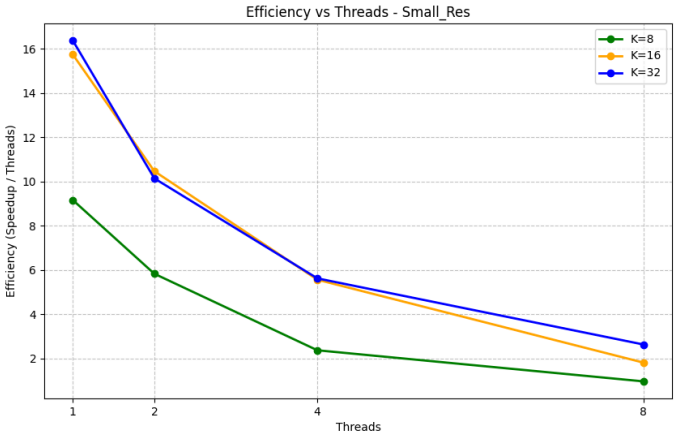
5.3. Scalability Analysis

I performed a Strong Scaling analysis by keeping the image resolution constant while increasing the number of threads. In *plot3* and *plot4* you can see how affects the number of threads to the efficiency. Also in *plot 5 and plot6* you can see the difference between the sequential and parallel (numba) versions.

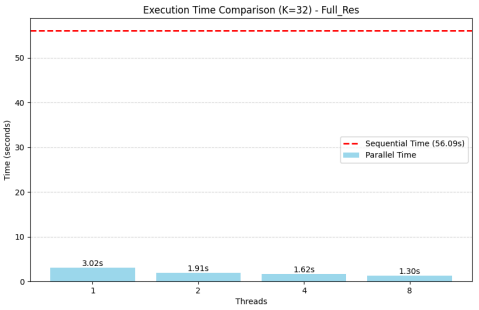
- **Physical Cores (1-4 Threads)**: Both datasets show excellent scaling up to 4 threads, which matches the physical cores of my CPU.
- **Hyper-Threading (8 Threads)**: For the high-resolution images, Hyper-Threading provided a significant boost. However, for small images, performance actually dropped at 8 threads because the management overhead was greater than the computational savings.



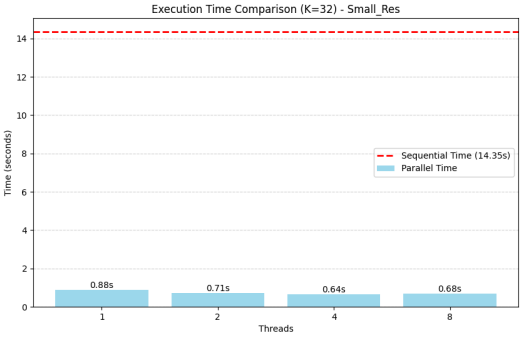
Plot 3: *Eficiency vs threads on the High res. dataset*



Plot 4: *Efficiency vs threads on the High res. dataset*



Plot 5:*Time vs threads on the High res. dataset*



Plot 6:*Time vs threads on the Low res. dataset*

5.4. Effect of Different Parameters

Cluster Size (K): Higher K values consistently led to better speedups. For K=8 on the large dataset, the peak speedup was 20.88x, but it tripled to 61.19x when K was increased to 32. This confirms that increasing computational intensity allows the parallel threads to work more effectively.

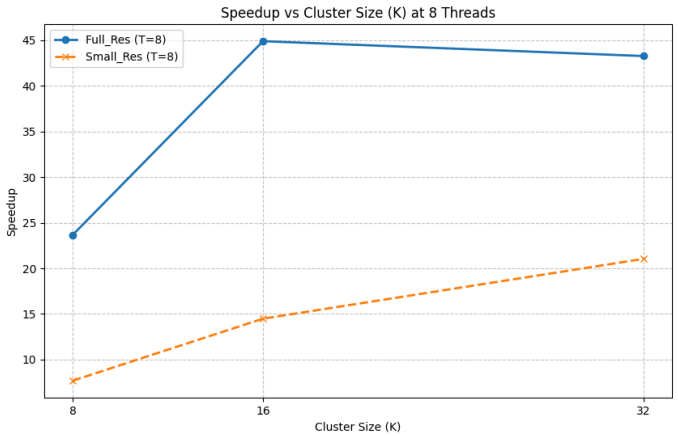
5.5. Parallelization Strategies

I implemented two distinct strategies within the code to handle different phases of the algorithm:

- **Automatic Distribution (prange)**: I used this for the pixel assignment phase, allowing Numba to handle the workload balancing. This proved highly efficient for simple, independent tasks.
- **Manual Distribution:** For the centroid update, I manually divided the pixels into chunks. This was necessary to assign each thread a private index for its own accumulation buffer, effectively preventing race conditions without the need for slow synchronization locks.

5.6. Bottleneck Analysis: What Limits Scalability?

1. **Memory Wall**: For high-resolution images (1024x 1024), the CPU processes billions of operations per second. Eventually, the RAM cannot feed pixel data to the cores fast enough, saturating the memory bandwidth.
2. **Thread Management Overhead**: On small datasets (150x 150), the parallel execution is so fast (under 0.1s) that the fixed time Numba takes to manage and "wake up" the threads becomes a major part of the total time.
3. **Hyper-Threading Resource Sharing**: While 8 threads help with large images, the logical threads share physical execution units, which explains why the efficiency drops when moving from 4 to 8 threads.



Plot 7:*Speed up vs cluster size*

5.6. Visual results

The final measure of the algorithm's success is the quality of the compressed images. The goal of K-Means quantization is to reduce the number of colors while preserving the original structures and details of the photograph. Here you can see the difference between the real images and the processed ones with different clustering. The parallel and sequential images are the same, I will just put the sequential ones with from 8 to 32 clusters and the real one.

FIGURE 1: *High resolution dataset original image*

FIGURE 2: *High resolution dataset 8,16,32 clusters image*

FIGURE 3:*Low resolution original image*

FIGURE 2: *Low resolution dataset 8,16,32 clusters image*

6. Discussion

6.1. Interpretation of Results

Plot 1 and 2: Speedup vs. Threads (Hardware Scaling) The speedup follows a classic scaling pattern, increasing significantly until reaching the physical core limit (4 threads) before stabilizing. This happens because the workload is distributed across all physical execution units; however, moving to 8 logical threads provides smaller gains due to resource contention. While both datasets scale similarly, the Full_Res images allow the CPU to stay at maximum saturation for longer, resulting in a higher peak speedup of 43.27x.

Plot 3 and 4: Parallel Efficiency vs. Threads Efficiency is highest at lower thread counts because the thread management tax paid to the operating system is minimal. As the number of threads increases, the time spent on context switching and synchronization causes the efficiency to decline. This drop is much more aggressive in the Small_Res dataset because the actual calculation time is so short that the overhead of launching and joining threads becomes a major percentage of the total execution.

Plot 5 and 6: Execution Time Comparison. The massive performance gap between the sequential and parallel versions is the result of combining JIT compilation with multi-core processing. Standard Python is interpreted and limited by the GIL, whereas my Numba-optimized code is compiled to machine instructions that run directly on the hardware. For large images at K=32, this reduces processing from 56 seconds to just 1.29 seconds, making high-quality quantization practical for real-time use.

Plot 7: Speedup vs. Cluster Size (K). Higher K values consistently yield better speedups because they increase the number of mathematical operations performed per pixel. When K is low, the CPU spends more relative time moving data from RAM to the cores (Memory-bound); when K is high, the task becomes "Compute-bound". This shift allows the parallel cores to work much more effectively, which explains why K=32 consistently displays the most impressive scaling across both datasets.

6.2. Interpretation of Results and Theoretical Expectations

My results can be interpreted through two fundamental parallel computing laws that explain the performance differences between the datasets.

- **Amdahl's Law (Sequential Bottleneck)**: This law states that speedup is limited by the sequential fraction of the code. Tasks like image I/O and initial centroid selection cannot be parallelized, creating a "speed ceiling." This is more evident in the Small_Res dataset, where the sequential overhead represents a larger proportion of the total execution time, causing the speedup to flatten earlier.
- **Gustafson's Law (Scaled Workload)**: This law suggests that parallel efficiency increases as the problem size grows. My data confirms this: the Full_Res dataset achieved a much higher peak speedup (43.27x) than the Small_Res one (22.48x). Because the large images contain millions more pixels, the "useful" parallel math (billions of distance calculations) completely dominates the fixed sequential costs.

6.3. Overhead and Bottleneck Analysis

- **Thread Management Overhead**: For Small_Res images, the computation is so fast that the time spent spawning and synchronizing threads dominates the execution, leading to reduced returns after 4 threads.
- **The Memory Wall**: In Full_Res datasets, the CPU processes billions of operations, eventually outpacing the RAM's ability to supply pixel data. This bandwidth saturation forces cores to remain idle while waiting for data from memory.
- **Hardware Contention**: Performance stabilizes after 4 threads because the system transitions from physical cores to logical threads (Hyper-threading). These logical pairs compete for shared execution units and L1/L2 caches, limiting the throughput for memory-intensive tasks.

6.4. Limitations and Constraints.

- **Hardware Architecture**: The system is limited by its 4 physical cores. Scaling beyond this point using logical threads is constrained by shared execution units and cache contention.
- **Initialization Sensitivity**: K-Means depends on random initial seeds. This can cause slight variations in the number of iterations required for convergence, affecting total execution time.
- **Memory Bandwidth**: For massive images, the "memory wall" becomes the primary bottleneck. The CPU's processing speed eventually outpaces the RAM’s ability to supply pixel data.
- **JIT Warm-up**: Numba requires an initial "warm-up" run to compile functions into machine code. This overhead makes the approach less suitable for singular, extremely short tasks.

6.5. Lessons Learned

- **Validation complication:** It’s very complicated to validate the perfection of a high processing algorithm, because for the sum error and other small errors it isn’t easy to tell if an image is the same as the sequential or not.
- **JIT Efficiency**: Numba's JIT compilation often provides a greater performance boost than multi-threading alone by eliminating Python's interpreter overhead.
- **Workload Criticality**: Parallelization is only beneficial when the workload is large enough to outweigh the cost of thread management.

6.6.Conclusions

In conclusion, I successfully implemented a parallel K-Means algorithm using Numba, achieving a maximum speedup of 44.9x on high-resolution images. This optimization transformed a 56-second sequential task into a sub-second process (1.29s), maintaining a mathematical accuracy within 0.1% of the baseline. The most effective configuration was 8 threads for large datasets and 4 threads for smaller workloads to avoid excessive management overhead.

6.7. Summary of achievements

- Drastic Performance Gain: transforming a 56-second sequential task into a 1.29-second parallel process.
- High Speedup: Achieved a peak speedup of 43.27x
- Verified Accuracy: Developed a robust validation system to ensure 99.9% mathematical consistency with the baseline.
- Numba: Successfully utilized numba.prange for parallel processing

6.8. Best configuration found

For high-resolution images, 8 threads is optimal, utilizing hyper-threading to handle the massive workload. For small images, 4 threads is best to avoid management overhead, while Numba JIT and K=32 consistently yield the highest computational efficiency.

7. References

[1] Dataset 1 (Big images):Kaggle - Landscape Pictures

https://www.kaggle.com/datasets/arnaud58/landscape-pictures?resource=download

[2] Dataset 2 (Small images):Kaggle - Intel Image Classification

https://www.kaggle.com/datasets/puneet6060/intel-image-classification?resource=download

[3] ***Parallel Programming Course Lecture Notes and Slides.*** *Course material from the Parallel Programming course.*